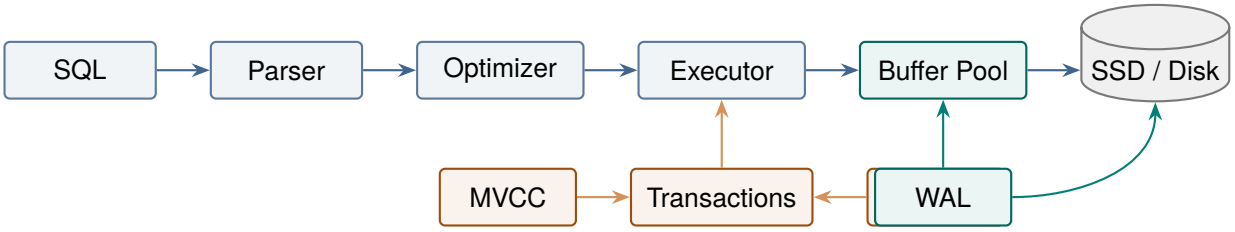
資料庫系統深入學習教材

Database Deep-Dive：從關聯模型到分散式資料庫

儲存引擎·B+Tree ·Buffer Pool ·查詢最佳化·交易·WAL ·MVCC ·鎖·ARIES ·
LSM-Tree ·行式儲存·複寫與分片·Raft/Paxos ·分散式交易·安全·動手打造 MiniDB

以 PostgreSQL 內部實作為主要案例研究，並以 C++ 逐步實作一個迷你資料庫



本教材依照「Database Deep-Dive Roadmap」十九個主題編寫，每章包含：學習目標、核心觀念、
圖解、PostgreSQL 內部實作對照、常見陷阱、練習題與延伸閱讀（含可直接查閱的網路資源）。

- ✓ 第一篇～第四篇：單機資料庫（儲存、索引、緩衝、查詢、交易、復原）
- ✓ 第五篇：LSM-Tree 與行式儲存（RocksDB、DuckDB、ClickHouse 的基礎）
- ✓ 第六篇：分散式資料庫（複寫、分片、CAP、Raft、2PC、Percolator、Spanner）
- ✓ 第七篇：安全與實作（RLS、TLS、SQL Injection；MiniDB v1–v7 C++ 專案）

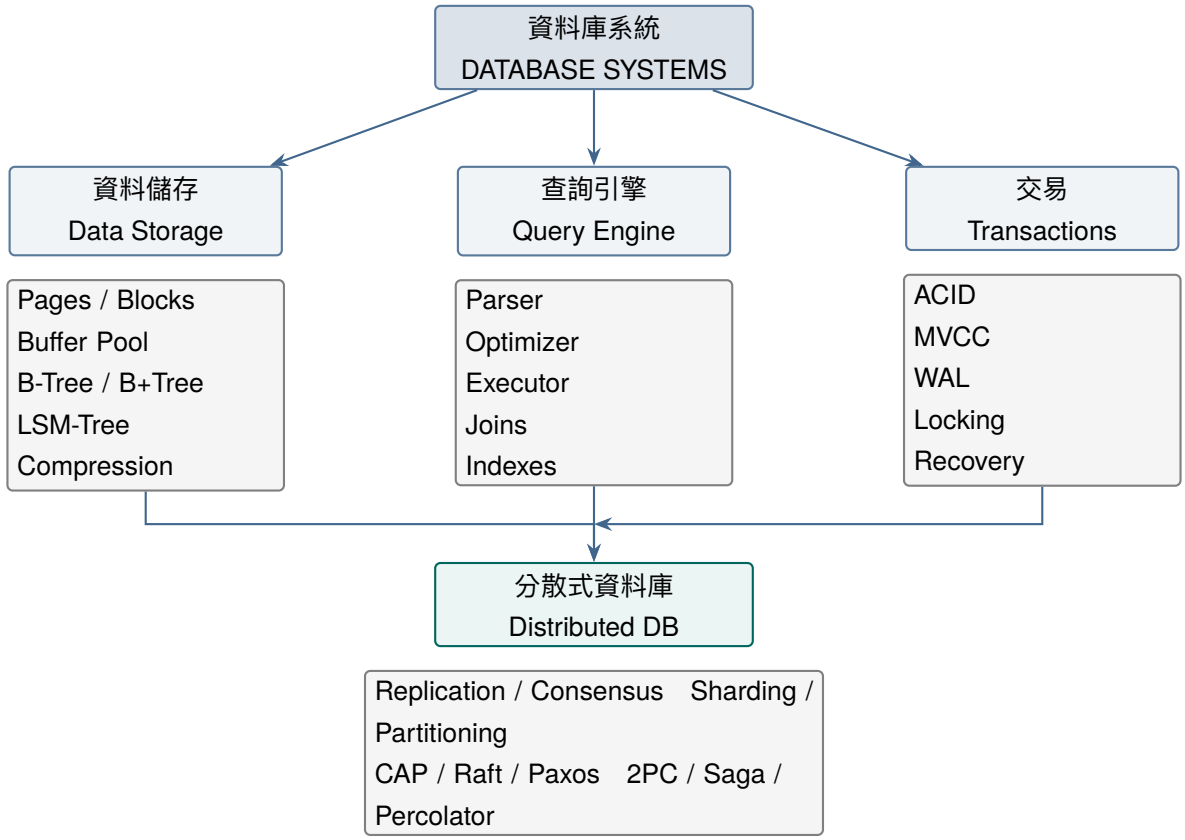
版本 1.0　2026 年 10 月以 PostgreSQL 18 文件與原始碼、CMU 15-445（Fall 2025）為主要參考

前言：如何使用本教材

本教材的目標

大多數人學資料庫，是從 SQL 語法開始，最後也停在 SQL 語法。本教材的目標不同：我們要回答的是「**資料庫到底是怎麼運作的**」——當你輸入 `SELECT * FROM users WHERE id = 123;`，從字串被解析、計畫被選出、頁面從 SSD 讀進緩衝池、MVCC 判定哪個版本可見、到交易提交時 WAL 被刷到磁碟，每一步究竟發生了什麼事。

全書依照「Database Deep-Dive Roadmap」的十九個主題編寫，將資料庫內部拆成四大支柱：



每一章的結構

每章都以相同的節奏編排，方便你規劃學習進度：

- **本章學習目標**：讀完本章後你應該能回答的問題與能做到的事。

- **核心觀念**：以圖解與範例說明機制「為什麼要這樣設計」。
- **PostgreSQL 內部實作**：對照 PostgreSQL 18 的文件、原始碼目錄與 README，讓抽象觀念落地。
- **常見陷阱 / 實務提示**：面試常問、線上事故常見的誤區。
- **MiniDB 實作連結**：說明本章觀念對應到第 19 章 C++ 迷你資料庫專案的哪一個版本。
- **本章練習**：從觀念題、計算題到動手實驗（多數可以直接在 `psql` 操作）。
- **延伸閱讀與資源**：經過查證、可直接查閱的課程、書籍、論文與官方文件連結。

建議學習順序與難度

本教材的章節順序就是建議的學習順序。難度以 ★ 到 ★★★★★ 表示，越多星代表越需要反覆閱讀與動手實驗。

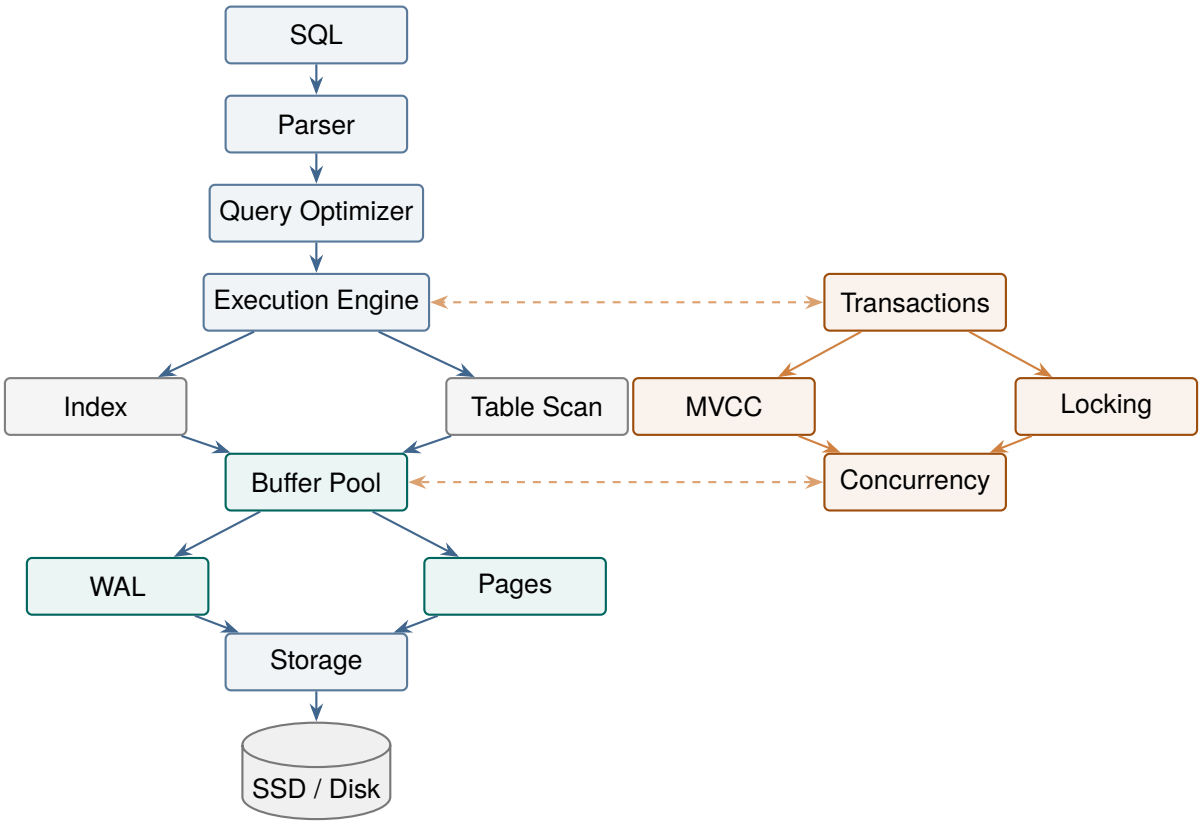
階段	主題	難度	對應章節
1	關聯模型與關聯代數	★★	第 1 章
2	資料庫儲存與頁面	★★★	第 2 章
3	B+Tree 與索引	★★★	第 3 章
4	Buffer Pool	★★★	第 4 章
5	查詢執行	★★★★	第 5 章
6	查詢最佳化	★★★★★	第 6 章
7	交易與 ACID	★★★★	第 7 章
8	WAL 與當機復原	★★★★★	第 8、11 章
9	MVCC	★★★★★	第 9 章
10	鎖與並行控制	★★★★	第 10 章
11	LSM-Tree	★★★★	第 12 章
12	行式儲存與 OLAP	★★★★	第 13 章
13	分散式資料庫	★★★★★	第 14、17 章
14	Raft / 共識	★★★★★	第 15 章
15	分散式交易	★★★★★	第 16 章
–	資料庫安全	★★	第 18 章
–	動手打造 MiniDB（C++）	★★★★★	第 19 章（貫穿全書）

如何安排時間

若每週投入 6–8 小時，單機部分（第 1–11 章）約需 10–12 週，LSM 與行式儲存 2 週，分散式部分 4–5 週。第 19 章的 MiniDB 專案建議與各章同步進行：讀完第 2 章就寫 v1 的 heap file，讀完第 3 章就加 B+Tree，依此類推。**不要只讀不寫**——「What I cannot create, I do not understand.」（Richard Feynman）

關鍵心智模型

全書要建立的，是下面這條完整管線。每一章都在填補其中一塊；讀到後面若覺得迷路，請回到這張圖：



為什麼以 PostgreSQL 為主要案例

PostgreSQL 是少數同時具備「教科書式的清晰架構」與「工業級完整度」的開源資料庫：heap 檔案與 slotted page、B-link tree 索引、clock-sweep 緩衝池、成本式最佳化器、MVCC 加上 Serializable Snapshot Isolation、物理 WAL 與 full-page write、串流與邏輯複寫……幾乎每一章的觀念都能在它的原始碼 `src/backend/` 下找到對應目錄（附錄 A 提供完整對照表）。其官方文件（`https://www.postgresql.org/docs/current/`）與 Hironobu Suzuki 的 *The Internals of PostgreSQL*（`https://www.interdb.jp/pg/`）是本教材反覆引用的兩份一手資料。

同時，我們也會對照 MySQL/InnoDB（叢集索引、undo log）、SQLite（單檔 B-tree 與 pager）、RocksDB（LSM-Tree）、DuckDB/ClickHouse（行式與向量化執行）、CockroachDB/TiDB/Spanner（分散式交易），避免把單一系統的設計誤認為唯一解。

先備知識與環境準備

- **先備知識**：基本 SQL（SELECT / JOIN / GROUP BY）、資料結構（樹、雜湊表）、作業系統概念（行程、記憶體、檔案 I/O）、C 或 C++ 基礎（第 19 章需要）。

- **PostgreSQL 18**：安裝後請同時安裝 `contrib` 模組，教材會使用 `pageinspect`、`pg_visibility`、`pg_buffercache`、`pgstattuple` 等擴充來「看見」內部結構。
- **C++ 工具鏈**：CMake 3.20+、GCC 12+ 或 Clang 15+（支援 C++20），以及 GoogleTest。
- **選用**：Docker（快速啟動多個 PostgreSQL 節點做複寫實驗）、RocksDB、etcd（觀察 Raft）。

程式碼 1: macOS 安裝與初始化範例

```
brew install postgresql@18 cmake
brew services start postgresql@18
psql postgres -c "CREATE DATABASE deepdive;"
psql deepdive -c "CREATE EXTENSION pageinspect; CREATE EXTENSION pg_visibility;
                  CREATE EXTENSION pg_buffercache; CREATE EXTENSION pgstattuple;"
```

主要參考資料

本教材所有事實性內容均對照以下一手資料撰寫，章末「延伸閱讀」會給出精確連結：

- CMU 15-445/645 *Intro to Database Systems*（Fall 2025，Andy Pavlo）：`https://15445.courses.cs.cmu.edu/fall2025/`，含全部投影片、講義、錄影與 BusTub 專案。
- Silberschatz, Korth, Sudarshan, *Database System Concepts*, 7th ed.（CMU 指定教科書）：`https://db-book.com/`。
- Alex Petrov, *Database Internals*（O'Reilly, 2019）；Martin Kleppmann & Chris Riccomini, *Designing Data-Intensive Applications*, 2nd ed.（O'Reilly, 2026）。
- Hellerstein, Stonebraker, Hamilton, *Architecture of a Database System*（2007）與 *Readings in Database Systems*, 5th ed.（`http://www.redbook.io/`）。
- PostgreSQL 18 官方文件與原始碼（`src/backend/*/README`）；*The Internals of PostgreSQL*。
- 經典論文：Codd (1970)、Selinger et al. (1979)、Mohan et al. ARIES (1992)、Berenson et al. (1995)、O'Neil LSM-Tree (1996)、Lamport *Paxos Made Simple* (2001)、Ongaro & Ousterhout Raft (2014)、Ports & Grittner SSI (2012)、Corbett et al. Spanner (2012) 等。

目錄

前言：如何使用本教材 i

第一篇　資料庫基礎 1

第 1 章　關聯模型與關聯代數 2

1.1 從檔案到關聯模型 2
1.2 關聯模型的基本元素 2
1.3 函數相依與正規化 4
1.4 關聯代數 5
1.5 關聯計算 7
1.6 SQL 的執行模型 7
1.7 OLTP 與 OLAP 8

第二篇　儲存引擎 10

第 2 章　儲存引擎與頁面佈局 11

2.1 為什麼要先懂磁碟 11
2.2 區塊與頁面 12
2.3 檔案組織：Heap File 12
2.4 頁面佈局：Slotted Page 13
2.5 記錄佈局 14
2.6 可用空間管理與插入流程 15
2.7 其他儲存模型：NSM、DSM 與 PAX 16
2.8 動手觀察：pageinspect 16

第 3 章　B-Tree、B+Tree 與索引 19

3.1 從二元搜尋樹到 B-Tree . . . 19
3.2 B-Tree 與 B+Tree . . . 20
3.3 節點結構、扇出與樹高 . . . 20
3.4 搜尋、插入、分裂、刪除與合併 . . . 21
3.5 索引並行控制：Latch Crabbing 與 B-link Tree . . . 22
3.6 索引的種類 . . . 23
3.7 索引的代價 . . . 24
3.8 B+Tree 變體速覽 . . . 24
3.9 動手觀察：pageinspect 與 EXPLAIN . . . 25

第 4 章 Buffer Pool **27**

4.1 為什麼需要 Buffer Pool . . . 27
4.2 Buffer Pool 的結構 . . . 28
4.3 核心操作 . . . 28
4.4 置換策略 . . . 29
4.5 Dirty Page 的寫回 . . . 30
4.6 預讀與非同步 I/O . . . 31
4.7 完整走一遍：`SELECT * FROM users WHERE id = 123` . . . 31
4.8 動手觀察：pg_buffercache . . . 32

第三篇 查詢處理 **34**

第 5 章 查詢執行 **35**

5.1 從 SQL 到結果的管線 . . . 35
5.2 Parser 與 AST . . . 35
5.3 邏輯計畫與實體計畫 . . . 36
5.4 處理模型 . . . 36
5.5 存取方法 . . . 38
5.6 Join 演算法 . . . 38
5.7 排序與聚合 . . . 39
5.8 其他運算子與平行執行 . . . 40
5.9 讀懂 EXPLAIN ANALYZE . . . 40

第 6 章　查詢最佳化 **43**

6.1 為什麼需要最佳化器 43
6.2 邏輯重寫：啟發式最佳化 43
6.3 統計資訊 44
6.4 選擇率與基數估計 45
6.5 成本模型 45
6.6 Join 排序 46
6.7 實體計畫的其他決策 47
6.8 診斷與修正錯誤的計畫 47

第四篇　交易、並行控制與復原 **50**

第 7 章　交易與 ACID **51**

7.1 交易是什麼 51
7.2 ACID 52
7.3 排程與可序列化 52
7.4 異常現象 53
7.5 隔離等級 53
7.6 Snapshot Isolation 與 write skew 54
7.7 並行控制的三大家族 55

第 8 章　預寫式日誌（WAL） **57**

8.1 為什麼需要日誌 57
8.2 WAL 協定 57
8.3 日誌記錄的內容 58
8.4 Full-Page Write 與 torn page 58
8.5 Checkpoint 59
8.6 Commit 的成本：fsync 與 group commit 59
8.7 PostgreSQL 的 WAL 實體佈局 60
8.8 其他系統的日誌設計 60
8.9 WAL 的其他用途 61

第 9 章　多版本並行控制（MVCC） **63**

9.1 核心想法 63
9.2 版本儲存的三種設計 64
9.3 PostgreSQL 的 tuple 版本 64
9.4 快照 65
9.5 可見性規則 65
9.6 寫入衝突 66
9.7 HOT 與 heap pruning 66
9.8 VACUUM 66
9.9 XID 回繞與凍結 67
9.10 Serializable Snapshot Isolation 68
9.11 其他系統速覽 69

第 10 章　鎖與並行控制 **71**

10.1 Lock 與 Latch 71
10.2 鎖模式與多粒度鎖 71
10.3 兩階段鎖 72
10.4 死結 72
10.5 Phantom 與述詞鎖 73
10.6 PostgreSQL 的表級鎖 74
10.7 PostgreSQL 的列級鎖 75
10.8 診斷鎖等待 75
10.9 樂觀並行控制 76

第 11 章　當機復原與 ARIES **78**

11.1 失效的類型與復原的目標 78
11.2 設計空間：steal/no-steal 與 force/no-force 78
11.3 ARIES 的三個原則 79
11.4 資料結構 79
11.5 三個階段 79
11.6 ARIES 的進階主題 81
11.7 PostgreSQL 的復原 82

11.8 其他系統 82

第五篇　替代儲存架構：LSM-Tree 與行式儲存 85

第 12 章　LSM-Tree 86

12.1 為什麼需要另一種結構 86
12.2 結構 87
12.3 讀取路徑 87
12.4 Compaction 88
12.5 刪除、tombstone 與其他實務問題 89
12.6 採用 LSM 的系統 89
12.7 LSM 還是 B+Tree？ 90

第 13 章　行式儲存與 OLAP 92

13.1 OLTP 與 OLAP 的分歧 92
13.2 列式與行式 92
13.3 壓縮編碼 93
13.4 排序、zone map 與跳塊 93
13.5 執行引擎 94
13.6 Parquet、PAX 與 Arrow 94
13.7 寫入與 HTAP 94
13.8 系統速覽 95

第六篇　分散式資料庫 98

第 14 章　分散式資料庫基礎 99

14.1 為什麼要分散，以及為什麼困難 99
14.2 複寫 99
14.3 分割（sharding / partitioning） 101
14.4 CAP 定理 102
14.5 一致性模型 103
14.6 時間與順序 104

14.7 現代分散式資料庫的架構分類 104

第 15 章　共識：Raft 與 Paxos **107**

15.1 共識問題 107
15.2 複寫狀態機 108
15.3 Paxos 108
15.4 Raft 109
15.5 Raft 與 Paxos 的比較 111
15.6 失效偵測、租約與 fencing 111
15.7 採用共識的系統 112

第 16 章　分散式交易 **114**

16.1 問題：跨節點的原子性與隔離 114
16.2 兩階段提交（2PC） 115
16.3 三階段提交（3PC） 116
16.4 以共識消除阻塞 116
16.5 跨分片的一致快照：全域時間戳 117
16.6 Saga：放棄隔離的長交易 118
16.7 選擇與實務建議 119

第 17 章　資料庫網路與分散式儲存 **121**

17.1 連線模型 121
17.2 PostgreSQL 前後端協定 122
17.3 連線池 123
17.4 延遲、RPC 與序列化 123
17.5 Timeout、重試與冪等 124
17.6 分散式儲存架構 124

第七篇　安全與動手實作 **128**

第 18 章　資料庫安全 **129**

18.1 威脅模型與縱深防禦 129
18.2 認證：pg_hba.conf 130

18.3 傳輸加密：TLS . . . 130
18.4 授權：角色與權限 . . . 131
18.5 Row-Level Security . . . 132
18.6 SQL Injection . . . 133
18.7 靜態加密 . . . 134
18.8 稽核與監控 . . . 134

第 19 章 動手打造 MiniDB **137**

19.1 為什麼要自己寫一個 . . . 137
19.2 架構總覽 . . . 138
19.3 專案佈局與工具鏈 . . . 138
19.4 v1：SQL 子集合、Heap File、Page Manager . . . 139
19.5 v2：B+Tree 索引 . . . 140
19.6 v3：Buffer Pool . . . 141
19.7 v4：交易、WAL 與復原 . . . 141
19.8 v5：MVCC . . . 142
19.9 v6：成本式最佳化器與更多運算子 . . . 142
19.10 v7：網路與複寫 . . . 143
19.11 測試策略 . . . 143
19.12 時程建議 . . . 144
19.13 參考實作 . . . 144

第八篇 附錄 **146**

附錄 A PostgreSQL 原始碼導覽地圖 **147**

A.1 取得與建置 . . . 147
A.2 目錄總覽 . . . 148
A.3 概念對應表 . . . 148
A.4 建議的閱讀路徑 . . . 151

附錄 B 完整學習資源清單 **153**

B.1 課程 . . . 153

B.2 書籍 153
B.3 論文（依章節） 154
B.4 PostgreSQL 官方與社群資源 156
B.5 工具 156
B.6 動手專案 157
B.7 保持更新 157

附錄 C 中英術語對照表 **158**

第一篇

資料庫基礎

第 1 章　關聯模型與關聯代數

本章學習目標

- 說明關聯模型（relational model）為何能把「資料怎麼存」與「資料怎麼問」分離，並解釋 Codd 的資料獨立性。
- 精確使用關聯、元組、屬性、鍵、外鍵與完整性限制等術語。
- 利用函數相依（functional dependency）判斷正規化程度，並完成無損分解。
- 以關聯代數（σ、π、$\bowtie$、$\cup$、$-$、ρ 等）改寫 SQL 查詢，為第 6 章的查詢最佳化打底。
- 區分關聯代數（程序式）與關聯計算（宣告式），並說明 SQL 的邏輯處理順序。
- 比較 OLTP 與 OLAP 工作負載，理解它們為何導致截然不同的儲存與執行設計。

1.1　從檔案到關聯模型

1970 年 E. F. Codd 在〈A Relational Model of Data for Large Shared Data Banks〉中提出：應用程式不應該知道資料在機器內部怎麼排列。當時主流的階層式（IMS）與網狀式（CODASYL）資料庫，程式必須沿著指標「導航」才能取得資料，一旦實體佈局改變，程式就得重寫。Codd 的答案是把資料抽象成**數學上的關聯**（集合論中的 relation），讓使用者只描述「要什麼」，由系統決定「怎麼拿」——這個分離正是今天查詢最佳化器存在的理由。

資料獨立性（Data Independence）

- **邏輯資料獨立性**：改變邏輯綱要（新增欄位、拆表）不影響既有查詢（透過 view 隔離）。
- **實體資料獨立性**：改變實體儲存（加索引、換儲存引擎、分割表）不改變查詢結果，只改變速度。

本教材第 2–6 章討論的一切（頁面、B+Tree、緩衝池、執行計畫）都藏在實體層；關聯模型的威力在於你不必知道它們也能寫出正確的查詢——但要寫出快的查詢，就必須知道。

1.2　關聯模型的基本元素

1.2.1　關聯、元組與屬性

- **關聯**（relation）：在數學上是若干定義域（domain）笛卡兒積的一個子集，$r \subseteq D_1 \times D_2 \times \cdots \times D_n$；在 SQL 中對應**表**（table）。
- **元組**（tuple）：關聯中的一個元素 $(v_1, \ldots, v_n)$；對應**列**（row）。
- **屬性**（attribute）：關聯綱要中的一個命名欄位及其定義域；對應**欄**（column）。
- **關聯綱要**（relation schema）：$R(A_1 : D_1, \ldots, A_n : D_n)$，是「型別」；**關聯實例**（relation

instance）是某一時刻的實際內容，是「值」。

- **度**（degree / arity）：屬性個數；**基數**（cardinality）：元組個數——後者是最佳化器估算成本的核心數字。

純數學的關聯是**集合**：沒有重複元組、沒有順序。SQL 的表則是**多重集合（bag）**，允許重複、且 `SELECT` 不加 `ORDER BY` 時順序未定義。這個差異在第 6 章討論等價規則時很重要：集合代數的某些等價式在 bag 語意下並不成立。

1.2.2　鍵

- **超鍵**（superkey）：能唯一識別元組的屬性集合。
- **候選鍵**（candidate key）：極小的超鍵（拿掉任何屬性就不再唯一）。
- **主鍵**（primary key）：從候選鍵中選出的一個，作為預設識別方式；在實務上常是人工代理鍵（surrogate key，如 `id BIGSERIAL`）。
- **外鍵**（foreign key）：關聯 R 的屬性集合，其值必須出現在關聯 S 的候選鍵中——這就是**參照完整性**。

主鍵的選擇會直接影響儲存引擎

在 MySQL/InnoDB 中，表資料**就是**依主鍵排序的 B+Tree（叢集索引，見第 3 章）；以隨機 UUIDv4 當主鍵會讓插入散佈到整棵樹，造成大量頁分裂與緩衝池抖動。PostgreSQL 的表是 heap，主鍵只是「唯一索引 + NOT NULL 限制」，隨機主鍵的代價較小，但索引本身一樣會膨脹。第 2、3 章會解釋背後的機制。

1.2.3　完整性限制

限制	SQL 寫法	意義
定義域限制	型別、`CHECK (age >= 0)`	屬性值必須屬於定義域
鍵限制	`PRIMARY KEY`、`UNIQUE`	候選鍵唯一
實體完整性	主鍵隱含 `NOT NULL`	主鍵不可為 NULL
參照完整性	`FOREIGN KEY ... REFERENCES`	外鍵值必須存在於被參照表（可搭配 `ON DELETE CASCADE`）
一般斷言	`CHECK`、觸發器、`EXCLUDE`	跨列或跨表的商業規則

程式碼 1.1: 以 PostgreSQL 語法定義含完整性限制的綱要

```
CREATE TABLE departments (
  dept_id   SERIAL PRIMARY KEY,
  name      TEXT NOT NULL UNIQUE
);
CREATE TABLE employees (
  emp_id    BIGSERIAL PRIMARY KEY,
  name      TEXT NOT NULL,
  age       INT  CHECK (age BETWEEN 16 AND 100),
  salary    NUMERIC(12,2) NOT NULL,
```

```
  dept_id   INT  NOT NULL REFERENCES departments(dept_id) ON DELETE RESTRICT
);
```

1.3 函數相依與正規化

1.3.1 函數相依

屬性集合 X **函數決定** Y（記作 $X \to Y$），意思是：在任何合法實例中，兩個元組若在 X 上相等，則在 Y 上也相等。候選鍵 K 的定義就是 $K \to R$（K 決定所有屬性）且極小。

Armstrong 公理（完備且正確）：

- 自反律（reflexivity）：若 $Y \subseteq X$，則 $X \to Y$。
- 增廣律（augmentation）：若 $X \to Y$，則 $XZ \to YZ$。
- 遞移律（transitivity）：若 $X \to Y$ 且 $Y \to Z$，則 $X \to Z$。

屬性集合 X 的**閉包** X^+ 是由 F 推得、所有 X 能決定的屬性。計算閉包是判斷鍵與正規式的基本工具：

程式碼 1.2: 屬性閉包演算法

```
closure(X, F):
  result = X
  repeat
    for each (A -> B) in F:
      if A ⊆ result then result = result ∪ B
  until result 不再改變
  return result
```

1.3.2 正規式

正規化的目的，是消除因**冗餘**造成的更新異常（update / insertion / deletion anomaly）。

正規式	條件（在 1NF 的基礎上）
1NF	每個屬性值都是不可分割的原子值（無重複群、無陣列欄位——雖然現代 SQL 允許 array/JSON，但那是刻意的反正規化）。
2NF	非主屬性不能部分相依於候選鍵（只在複合鍵時有意義）。
3NF	對每個 FD $X \to A$：X 是超鍵，或 A 是主屬性（屬於某候選鍵）。
BCNF	對每個非平凡 FD $X \to A$：X 必須是超鍵。比 3NF 嚴格，但可能無法保持相依性。
4NF / 5NF	處理多值相依與連接相依，實務上較少刻意追求。

分解一個關聯 R 為 R_1, R_2 時要檢查兩件事：

1. **無損連接**（lossless join）：$R_1 \bowtie R_2 = R$。充分條件：$(R_1 \cap R_2) \to R_1$ 或 $(R_1 \cap R_2) \to R_2$。
2. **相依保持**（dependency preserving）：原本的每個 FD 都能在單一分解後的表上檢查，不必 join。

範例：從異常到 3NF

假設 `Orders(order_id, customer_id, customer_email, product_id, product_name, qty)`，且 *customer_id* → *customer_email*、*product_id* → *product_name*。一位客戶改了 email，就得更新他所有訂單列（更新異常）；刪掉某產品的最後一筆訂單，產品名稱也消失了（刪除異常）。分解為 `Customers(customer_id, email)`、`Products(product_id, name)`、`Orders(order_id, customer_id, product_id, qty)` 後，三張表皆為 BCNF，且無損、相依保持。

正規化與效能的取捨

正規化減少冗餘、保證一致性，代價是查詢需要更多 join。OLTP 系統通常維持 3NF/BCNF；OLAP 的星狀綱要（star schema）則刻意把維度反正規化成寬表，以換取掃描效率（第 13 章）。「先正規化，再有憑有據地反正規化」是通則。

1.4　關聯代數

關聯代數是一組以關聯為輸入、關聯為輸出的運算子。它是**程序式**的——運算的順序就是執行的順序——因此查詢最佳化器實際上就是在「同一個關聯代數表達式的所有等價寫法中，挑最便宜的那個」。

1.4.1　基本運算子

符號	名稱	語意	SQL 對應
$\sigma_p(R)$	選擇 selection	保留滿足述詞 p 的元組（水平切割）	`WHERE p`
$\pi_{A_1,\dots}(R)$	投影 projection	只保留指定屬性（垂直切割），集合語意下去重	`SELECT DISTINCT A1,...`
$R \times S$	笛卡兒積	所有元組配對	`FROM R, S`（無條件）
$R \bowtie_\theta S$	連接 theta-join	$\sigma_\theta(R \times S)$	`JOIN ... ON theta`
$R \bowtie S$	自然連接	以同名屬性相等為條件並去掉重複欄	`NATURAL JOIN`
$R \cup S$、$R \cap S$、$R - S$	集合運算	需綱要相容	`UNION` / `INTERSECT` / `EXCEPT`
$\rho_{S(B_1,\dots)}(R)$	更名 rename	更改關聯或屬性名稱（自我連接必備）	`AS`
$R \div S$	除法 division	「對所有 S 都有配對」的 R 元組	以雙重 `NOT EXISTS` 表達

1.4.2　擴充運算子

- **外部連接**：$R \mathbin{⟕} S$（左）、$R \mathbin{⟖} S$（右）、$R \mathbin{⟗} S$（全），無配對的一側補 NULL。
- **半連接** $R \ltimes S$：保留 R 中「在 S 有配對」的元組（對應 `WHERE EXISTS`）；**反連接**（anti-join）對應 `NOT EXISTS`。分散式查詢常用半連接減少資料搬移（第 14 章）。
- **分組與聚合** $\gamma_{G;\,f(A)}(R)$：對應 `GROUP BY G` 搭配 `SUM/COUNT/...`。
- **排序** $\tau_A(R)$ 與 **去重** $\delta(R)$：在 bag 代數中才需要。
- **賦值** $T \leftarrow E$：把子表達式存成暫存關聯，對應 CTE（`WITH`）。

1.4.3　從 SQL 到關聯代數

```sql
SELECT e.name, d.name AS dept
FROM   employees e JOIN departments d ON e.dept_id = d.dept_id
WHERE  e.age > 30 AND d.name = 'Engineering';
```

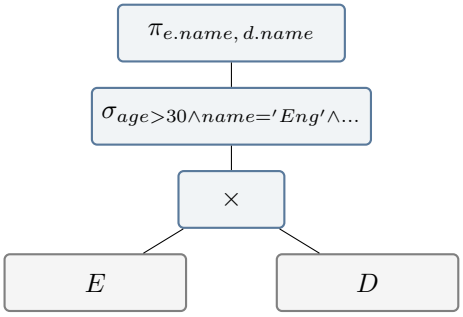
直接翻譯（先積再選再投影）：

$$\pi_{e.name,\,d.name}\Big(\sigma_{e.age>30 \,\wedge\, d.name='Engineering' \,\wedge\, e.dept_id=d.dept_id}(E \times D)\Big)$$

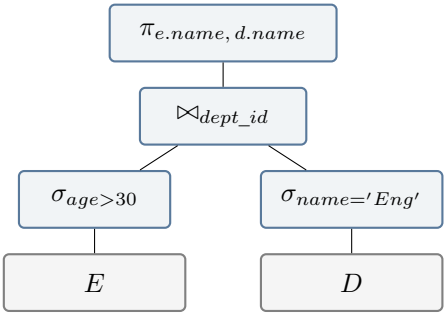
最佳化器會改寫成「先把選擇推到各表、再連接」的等價形式（**述詞下推**，第 6 章）：

$$\pi_{e.name,\,d.name}\Big(\sigma_{age>30}(E) \;\bowtie_{dept_id}\; \sigma_{name='Engineering'}(D)\Big)$$

兩者結果相同，但第二式在連接前就把 D 縮到可能只剩一列——這就是為什麼學好關聯代數對理解最佳化器至關重要。



(a) 直譯：先積再選



(b) 述詞下推後的等價計畫

1.4.4　常用等價規則（最佳化器的工具箱）

1. 選擇可拆分、可交換：$\sigma_{p_1 \wedge p_2}(R) = \sigma_{p_1}(\sigma_{p_2}(R)) = \sigma_{p_2}(\sigma_{p_1}(R))$。
2. 投影串接只留最外層：$\pi_A(\pi_B(R)) = \pi_A(R)$，若 $A \subseteq B$。
3. 選擇與連接交換（述詞只涉及一側時）：$\sigma_{p_R}(R \bowtie S) = \sigma_{p_R}(R) \bowtie S$。
4. 連接的交換律與結合律：$R \bowtie S = S \bowtie R$；$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$ ——這讓 n 個表有 $O(n!)$ 種以上的連接順序，是第 6 章「連接重排」的搜尋空間。
5. 投影下推：$\pi_A(R \bowtie_\theta S) = \pi_A(\pi_{A_R \cup \theta_R}(R) \bowtie_\theta \pi_{A_S \cup \theta_S}(S))$，提早丟掉不需要的欄位。
6. 外部連接**不**滿足一般的交換與結合律；選擇只有在「NULL 拒絕」（null-rejecting）述詞下才能推過外部連接。

1.5　關聯計算

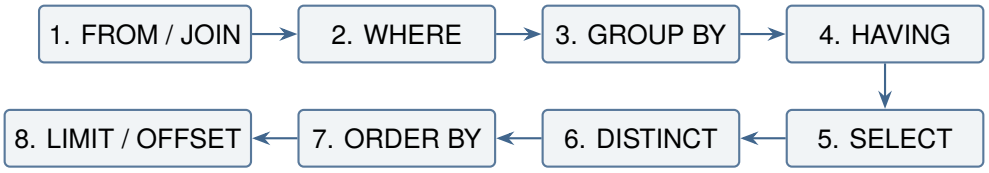
關聯計算（relational calculus）是**宣告式**的：只描述結果元組必須滿足的條件。

- **元組關聯計算（TRC）**：$\{\, t \mid \exists e \in \mathit{Employees}\ (e.age > 30 \wedge t.name = e.name) \,\}$。
- **定義域關聯計算（DRC）**：變數代表屬性值而非元組：$\{\, \langle n \rangle \mid \exists a, s, d\ (\langle _, n, a, s, d \rangle \in \mathit{Employees} \wedge a > 30) \,\}$。

Codd 定理：**安全的**（結果有限的）關聯計算與關聯代數表達能力相同。SQL 在語法上更接近 TRC（`WHERE EXISTS (SELECT ...)` 就是 $\exists$），而資料庫內部先把 SQL 轉成關聯代數樹（邏輯計畫），再做最佳化——這正是第 5 章的起點。

1.6　SQL 的執行模型

雖然 `SELECT` 寫在最前面，SQL 的**邏輯處理順序**其實是：



這解釋了幾個常見困惑：為什麼 `WHERE` 不能用聚合函數（還沒分組）、為什麼 `SELECT` 的別

名不能在 WHERE 使用（還沒投影），以及為什麼 ORDER BY 可以用別名（已經投影）。注意這只是邏輯順序；實體執行順序由最佳化器決定，只要結果等價即可。

NULL 與三值邏輯

SQL 使用三值邏輯（TRUE / FALSE / UNKNOWN）。NULL = NULL 是 UNKNOWN 而非 TRUE；WHERE 只保留 TRUE 的列；NOT IN（子查詢含 NULL）永遠不會回傳任何列；聚合函數忽略 NULL 但 COUNT(*) 不會。這些語意最佳化器都必須保留，因此許多等價規則都附帶「NULL 拒絕」的條件。

1.7　OLTP 與 OLAP

	OLTP（線上交易處理）	OLAP（線上分析處理）
典型查詢	依鍵讀寫少量列：下單、轉帳、登入	掃描數百萬列做聚合：報表、趨勢分析
每次觸及資料	幾 KB	幾 GB 到 TB
存取模式	隨機、點查詢、高並行寫入	循序、全欄掃描、少量寫入（批次載入）
重視指標	延遲、每秒交易數（TPS）、ACID	吞吐量、壓縮率、掃描速度
綱要	正規化（3NF）	星狀/雪花（反正規化）
儲存佈局	列式（row store） + B+Tree	行式（column store） + 壓縮 + 向量化
代表系統	PostgreSQL、MySQL、Oracle、SQL Server	ClickHouse、DuckDB、BigQuery、Snowflake、Redshift
基準測試	TPC-C、TPC-E	TPC-H、TPC-DS

兩種工作負載對**儲存引擎**的要求幾乎相反：OLTP 需要快速找到並修改單一列（第 2–4 章），OLAP 需要以最少 I/O 掃過單一欄位（第 13 章）。試圖同時服務兩者的系統稱為 HTAP（Hybrid Transactional/Analytical Processing），通常以「列式主儲存 + 行式副本」實現（如 TiDB 的 TiFlash、SingleStore）。

PostgreSQL 如何表示關聯

PostgreSQL 把「關聯」這個概念實作得非常徹底：表、索引、序列、view、物化 view 都是 pg_class 裡的一列（relkind 區分 r 表、i 索引、S 序列、v view、m 物化 view、t TOAST 表），屬性存在 pg_attribute，限制存在 pg_constraint，統計資訊存在 pg_statistic。系統目錄本身也是普通的 heap 表，可以直接查詢：

```
SELECT c.relname, c.relkind, c.relpages, c.reltuples
FROM   pg_class c JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE  n.nspname = 'public';
```

relpages 與 reltuples 就是第 6 章最佳化器估算掃描成本的起點。

MiniDB 連結：v1 的 Catalog

第 19 章的 MiniDB v1 需要一個最小的系統目錄：`Schema`（屬性名稱、型別、位移）與 `TableInfo`（表名 → 首頁 page id）。把它設計成「存在 page 0 的特殊表」，就能體會 PostgreSQL「系統目錄也是表」的自舉（bootstrap）哲學。

本章練習

1. 給定 $R(A,B,C,D)$ 與 $F=\{A\to B,\ B\to C,\ C\to D\}$，求 A^+ 與所有候選鍵；R 是否為 BCNF？若否，給出無損且相依保持的 3NF 分解。
2. 以關聯代數寫出：「找出所有部門都有員工年齡大於 30 的部門名稱」，並說明為何需要除法或雙重否定。
3. 證明或舉反例：$\sigma_p(R\bowtie S)=\sigma_p(R)\bowtie S$ 在 p 只涉及 R 屬性時是否恆成立？若 p 只涉及 S 屬性呢？
4. 在 bag 語意下，$\pi_A(R\cup S)=\pi_A(R)\cup\pi_A(S)$ 是否成立？$\pi_A(R\cap S)$ 呢？
5. （動手）在 PostgreSQL 建立本章的 `employees/departments`，執行 `EXPLAIN SELECT ...` 觀察最佳化器輸出的計畫樹，對照你手寫的關聯代數樹，找出被下推的述詞。
6. 說明三值邏輯如何導致 `WHERE x NOT IN (SELECT y FROM t)` 在 `t.y` 含 NULL 時回傳空結果，並改寫成 `NOT EXISTS`。

延伸閱讀與資源

- E. F. Codd, *A Relational Model of Data for Large Shared Data Banks*, CACM 1970. 原文 PDF：`https://cs-people.bu.edu/mathan/reading-groups/papers-classics/codd.pdf`
- CMU 15-445 Fall 2025, Lecture #01 *Relational Model & Algebra*、#02 *Modern SQL*：`https://15445.courses.cs.cmu.edu/fall2025/schedule.html`
- *Database System Concepts* 7th ed., Ch. 2（關聯模型）、Ch. 6（形式化查詢語言）、Ch. 7（正規化）：`https://db-book.com/`
- RelaX 線上關聯代數計算器（可輸入代數式並看到結果與樹）：`https://dbis-uibk.github.io/relax/`
- PostgreSQL 文件 Ch. 5 *Data Definition*（限制、繼承、分割）與 Ch. 7 *Queries*：`https://www.postgresql.org/docs/current/ddl.html`

第二篇

儲存引擎

第 2 章　儲存引擎與頁面佈局

本章學習目標

- 說明 HDD、SSD、NVMe 的存取特性，以及「循序 vs 隨機 I/O」為何主導資料庫的所有設計決策。
- 解釋資料庫為何以固定大小的**頁面（page）**作為 I/O 與緩衝的基本單位，並比較各系統的頁面大小。
- 畫出 slotted page 的結構，解釋 slot 陣列（line pointer）帶來的間接層有什麼好處。
- 描述一筆記錄（tuple）在頁面內的佈局：標頭、NULL 位元圖、對齊填補、變長欄位與 TOAST。
- 說明 heap file 與 free-space map 的運作方式，並完整追蹤「一列資料如何落到磁碟」。
- 使用 `pageinspect` 擴充實際觀察 PostgreSQL 的頁面內容。

2.1　為什麼要先懂磁碟

資料庫是「以磁碟為導向（disk-oriented）」的系統：資料的主要住所是非揮發性儲存裝置，記憶體只是暫存。因此所有資料結構的設計目標都是**減少磁碟 I/O 次數**，而非減少 CPU 指令數。要理解這一點，先看各層儲存的延遲差距：

層級	典型延遲	相對於 L1 快取
L1 快取	≈ 1 ns	$1\times$
L2 / L3 快取	4–40 ns	$10\times$
DRAM	≈ 100 ns	$100\times$
NVMe SSD（隨機 4 KB 讀）	10–100 μs	10^4–$10^5\times$
SATA SSD（隨機 4 KB 讀）	≈ 100 μs	$10^5\times$
HDD（隨機讀，含尋道）	5–10 ms	$10^7\times$
同一資料中心網路往返	≈ 0.5 ms	$5\times10^5\times$

2.1.1　HDD、SSD 與 NVMe

- **HDD**：機械臂移動（seek，約 4–10 ms）加上碟片旋轉等待，隨機 I/O 每秒只有約 100–200 次（IOPS），但循序讀寫可達 100–250 MB/s。「隨機比循序慢兩個數量級」是傳統 B+Tree、WAL、循序掃描設計的根源。
- **SATA SSD**：無機械部件，隨機讀 IOPS 達數萬；但 NAND flash 的寫入必須先**抹除**（以較大的 block 為單位），造成**寫入放大**（write amplification）與磨損；FTL（flash translation layer）在內部做了類似 LSM 的日誌式寫入。

- **NVMe SSD**：走 PCIe 通道、多達 64K 個佇列，隨機 IOPS 達數十萬到百萬，延遲降到 10 μs 等級。隨機與循序的差距縮小到數倍，使得 `random_page_cost` 等成本參數需要重新調整（第 6 章）。

即使在 SSD 上，循序仍然比較快

SSD 的「隨機」讀取仍需經過 FTL 查表、可能跨越多個 NAND die；而大區塊循序 I/O 可以合併請求、充分利用內部平行度。此外作業系統的 read-ahead、資料庫的 I/O 合併（PostgreSQL 18 的 `io_combine_limit`）都只對循序存取有效。因此「把相關的資料放在一起（locality）」這條原則在 SSD 時代依然成立。

2.2　區塊與頁面

作業系統以**區塊（block）**（通常 4 KB）為單位與儲存裝置溝通；資料庫在其上定義自己的**頁面（page）**：

系統	預設頁面大小	備註
PostgreSQL	8 KB	編譯期常數 `BLCKSZ`，可改為 1–32 KB
MySQL / InnoDB	16 KB	`innodb_page_size` 可設 4–64 KB
SQLite	4 KB	可為 512 B–64 KB 的 2 的冪
Oracle	8 KB	每個 tablespace 可不同
SQL Server	8 KB	8 頁組成 64 KB 的 extent
RocksDB（SST block）	4 KB	壓縮前大小，非固定邊界

頁面是三件事的共同單位：**磁碟 I/O 的單位**、**緩衝池配置的單位**（第 4 章）、**鎖定與 WAL 記錄定址的單位**（第 8 章）。頁面太小會讓 B+Tree 變高、標頭開銷比例上升；太大則浪費記憶體與 I/O 頻寬在不需要的資料上，也增加寫入放大。

撕裂寫入（torn write）

OS 區塊是 4 KB，PostgreSQL 的頁面是 8 KB；若在寫入 8 KB 頁面的過程中斷電，磁碟上可能只更新了前 4 KB，形成一個「半新半舊」的損毀頁面，WAL 記錄的差異式更新無法在其上正確重做。PostgreSQL 的解法是 `full_page_writes`：每次 checkpoint 後第一次修改某頁面時，把整頁影像寫進 WAL（第 8 章）。InnoDB 則用 doublewrite buffer 達到同樣目的。

2.3　檔案組織：Heap File

資料庫把一張表存成一個或多個檔案，檔案由頁面組成。最常見的組織是 **heap file**：頁面之間沒有順序，一筆記錄可以放在任何有空間的頁面。相對的還有依鍵排序的 sorted file（更新代價高，實務上以叢集索引取代）與以雜湊分桶的 hashed file。

要在 heap file 中快速找到「哪一頁還有空間」，有兩種作法：

1. **頁面鏈結串列**：標頭頁維護「有空間的頁面」與「已滿頁面」兩條串列，找空間可能需要走訪多頁。
2. **頁面目錄（page directory）**：專門的目錄頁記錄每個資料頁的可用空間——PostgreSQL 的 Free Space Map 即屬此類。

PostgreSQL 的檔案佈局

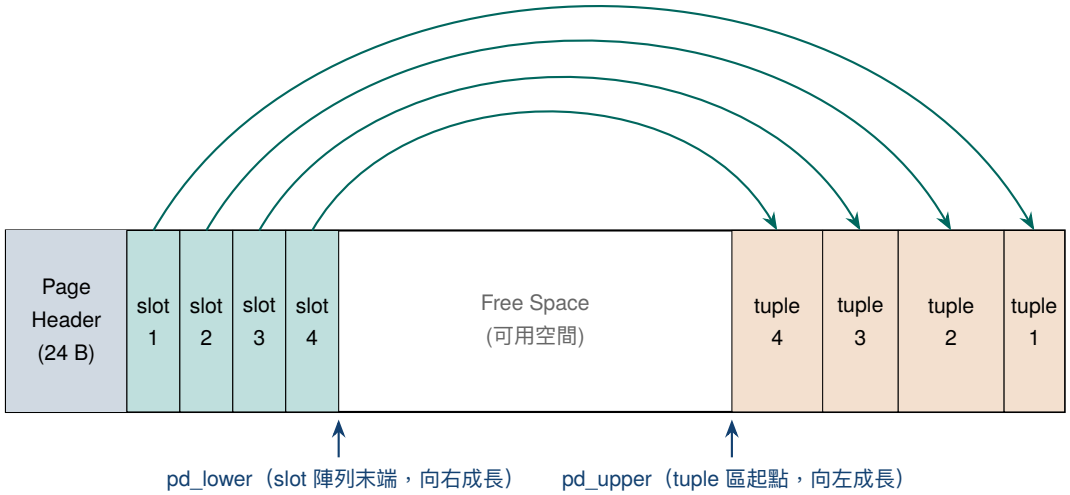
每個表或索引對應資料目錄下 `base/<db oid>/<relfilenode>` 的檔案，超過 1 GB 時切成 `.1`、`.2` 等區段（segment）。每個關聯還有多個**分支（fork）**：

- **main**：資料頁面本身。
- **fsm**（`_fsm`）：Free Space Map，以 1 byte 記錄每頁的可用空間（精度 1/256 頁），組織成一棵二元樹，讓「找一頁至少有 n bytes 空間」只需 $O(\log)$ 次頁面存取。
- **vm**（`_vm`）：Visibility Map，每頁 2 bits（all-visible、all-frozen），供 VACUUM 與 index-only scan 使用（第 9 章）。
- **init**：unlogged table 的初始化分支。

```
SELECT pg_relation_filepath('employees');          -- base/16384/16399
SELECT pg_relation_size('employees', 'main'),
       pg_relation_size('employees', 'fsm'),
       pg_relation_size('employees', 'vm');
```

2.4　頁面佈局：Slotted Page

資料庫頁面幾乎都採用 **slotted page**（又稱 slot array 或 line pointer array）佈局：



- **標頭**記錄頁面的元資料：可用空間的起訖位移、校驗和、最後修改此頁的 WAL 位置（LSN）等。
- **slot 陣列**從頁首向後成長，每個 slot 存放一筆記錄的（位移、長度、狀態）。

- **記錄**從頁尾向前成長。兩者中間就是可用空間，當 `pd_lower` 追上 `pd_upper` 頁面就滿了。

為什麼要多一層 slot 間接？

外部對記錄的參照（例如索引項）使用 **TID = (頁號, slot 號)**，而不是頁內位元組位移。這樣當記錄被更新變長、或頁面做內部整理（compaction）而移動時，只要更新 slot 的位移，索引完全不必改動。刪除記錄只需把 slot 標記為 unused/dead；空間可在稍後的整理中回收。PostgreSQL 的 HOT 更新（第 9 章）更進一步把 slot 設為「重導向（redirect）」指向同頁的新版本。

PageHeaderData 與 ItemIdData

PostgreSQL 的頁面標頭固定 24 bytes（`src/include/storage/bufpage.h`）：

欄位	大小	意義
`pd_lsn`	8 B	最後一筆修改此頁的 WAL 記錄結尾 LSN（WAL 規則與復原的關鍵，第 8 章）
`pd_checksum`	2 B	頁面校驗和（`initdb -k` 或 `pg_checksums` 啟用）
`pd_flags`	2 B	旗標：有可用 slot、全頁可見（all-visible）等
`pd_lower`	2 B	slot 陣列結尾位移（可用空間起點）
`pd_upper`	2 B	可用空間結尾位移（tuple 區起點）
`pd_special`	2 B	特殊區起點（索引頁用來放兄弟指標等；heap 頁為頁尾）
`pd_pagesize_version`	2 B	頁面大小與版本（目前為 4）
`pd_prune_xid`	4 B	頁內最舊未被修剪的 XMAX，提示是否值得做 HOT 修剪

每個 slot（`ItemIdData`）只有 4 bytes：15 bits 位移 `lp_off`、2 bits 狀態 `lp_flags`（`UNUSED` / `NORMAL` / `REDIRECT` / `DEAD`）、15 bits 長度 `lp_len`。索引頁的 special space 則放 `BTPageOpaqueData`：左右兄弟頁號、層級、旗標。

2.5　記錄佈局

一筆記錄在頁面內的典型結構為：**記錄標頭** → **NULL 位元圖** → **對齊填補** → **欄位資料**。

- **定長欄位**（`INT`、`BIGINT`、`TIMESTAMP`）直接依型別長度存放。
- **變長欄位**（`TEXT`、`VARCHAR`、`NUMERIC`、`JSONB`）以長度前綴（PostgreSQL 稱 varlena 標頭，1 或 4 bytes）帶出實際內容。
- **NULL** 不佔資料空間，只在 NULL 位元圖中設一個 bit；若整列沒有 NULL，位元圖可以省略。
- **對齊**：為了讓 CPU 直接讀取，8-byte 型別通常要對齊到 8 的倍數（PostgreSQL 的 `MAXALIGN`），因此欄位順序會影響填補量——`(bool, bigint, bool, bigint)` 比 `(bigint, bigint, bool, bool)` 多浪費 14 bytes。

程式碼 2.1: PostgreSQL 18 的 heap tuple 標頭（節錄自 `src/include/access/htup_details.h`）

```
typedef struct HeapTupleFields {
    TransactionId t_xmin;        /* 插入此版本的交易 ID */
    TransactionId t_xmax;        /* 刪除或鎖定此版本的交易 ID（0 表示仍有效） */
    union {
        CommandId     t_cid;     /* 交易內的命令序號 */
        TransactionId t_xvac;    /* 舊式 VACUUM FULL 使用 */
    } t_field3;
} HeapTupleFields;

struct HeapTupleHeaderData {
    union { HeapTupleFields t_heap; DatumTupleFields t_datum; } t_choice;
    ItemPointerData t_ctid;      /* 此版本或更新版本的 TID (block, offset) */
    uint16  t_infomask2;         /* 欄位數 + 旗標 */
    uint16  t_infomask;          /* 旗標：有 NULL、有變長欄、xmin/xmax 已提交提示位元… */
    uint8   t_hoff;              /* 使用者資料起始位移（含 NULL 位元圖與對齊） */
    /* ^ 以上固定 23 bytes ^ */
    bits8   t_bits[FLEXIBLE_ARRAY_MEMBER];   /* NULL 位元圖（變長） */
    /* 之後接欄位資料 */
};
```

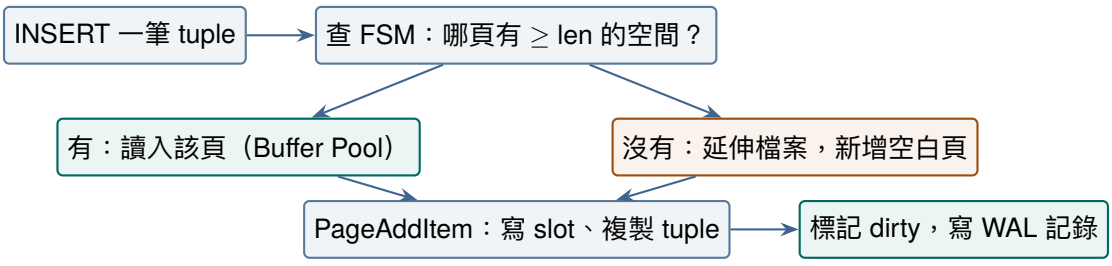
標頭中的 `t_xmin`、`t_xmax`、`t_ctid` 正是 MVCC（第 9 章）的基礎——PostgreSQL 把版本資訊直接放在每一列裡，這是它與 InnoDB（版本放在 undo log）最根本的差異之一。

TOAST：超過約 2 KB 的欄位值

一筆 tuple 不能跨頁，而 8 KB 的頁面至少要放得下 4 筆（`TOAST_TUPLE_THRESHOLD` 約 2 KB）。超過時 PostgreSQL 會依欄位的 storage 策略（`PLAIN` / `EXTENDED` / `EXTERNAL` / `MAIN`）：先嘗試以 pglz 或 LZ4 壓縮，仍太大就切成約 2 KB 的 chunk 存到關聯的 **TOAST 表**（`pg_toast.pg_toast_<oid>`），原欄位只留下 18 bytes 的指標。這就是為什麼 `SELECT *` 比 `SELECT id` 貴得多——大欄位需要額外去 TOAST 表取回並解壓。

2.6　可用空間管理與插入流程

當執行 `INSERT` 時，儲存引擎需要回答「這筆記錄該放哪一頁？」：



PostgreSQL 的 `heap_insert()`（`src/backend/access/heap/heapam.c`）流程：

1. 先試上次插入的那一頁（有空間的機率高，且維持局部性）。

2. 否則查 FSM 找一頁可用空間 ≥ tuple 長度（加上 `fillfactor` 預留的空間）。
3. 找不到就延伸檔案一頁（並在 FSM 登記）。
4. 透過緩衝池取得頁面並加排他鎖，呼叫 `PageAddItem()` 寫入 slot 與資料，更新 `pd_lower/pd_upper`。
5. 把頁面標為 dirty，寫出 WAL 記錄（`XLOG_HEAP_INSERT`），把頁面的 `pd_lsn` 設為該記錄的 LSN。
6. **此時資料頁還在記憶體**；它會在 checkpoint、background writer 或緩衝池淘汰時才真正寫回磁碟（第 4、8 章）。

fillfactor

`CREATE TABLE ... WITH (fillfactor = 70)` 讓 INSERT 只填到頁面的 70%，剩下 30% 保留給同頁的 UPDATE 新版本，提高 HOT 更新的機率（第 9 章）、減少索引膨脹。對更新頻繁的表非常有效；對只插入不更新的表則純屬浪費。

2.7　其他儲存模型：NSM、DSM 與 PAX

本章描述的「一列的所有欄位連續存放」稱為 **NSM**（N-ary Storage Model，列式）。另外兩種模型會在第 13 章深入：

- **DSM**（Decomposition Storage Model，行式）：每個欄位獨立成檔，OLAP 掃描只讀需要的欄位。
- **PAX**（Partition Attributes Across，Ailamaki et al. 2001）：頁面層級仍是列式（一頁內含整列），但頁內依欄位分組存放（mini-page），兼顧 I/O 局部性與 CPU 快取效率。Parquet、ORC 的 row group 正是這種思路。

2.8　動手觀察：pageinspect

程式碼 2.2: 用 pageinspect 看見頁面內部

```
CREATE TABLE t (id int, name text);
INSERT INTO t SELECT g, 'name_' || g FROM generate_series(1, 5) g;

-- 頁面標頭
SELECT lsn, checksum, flags, lower, upper, special, pagesize, version, prune_xid
FROM   page_header(get_raw_page('t', 0));
--    lsn     | lower | upper | special | pagesize
-- 0/1A3F2B8|   44  |  8032 |  8192   |  8192      -- 24 B 標頭 + 5 個 4 B slot = 44

-- 每個 slot 與 tuple 標頭
SELECT lp, lp_off, lp_len, t_xmin, t_xmax, t_ctid, t_infomask2, t_hoff
FROM   heap_page_items(get_raw_page('t', 0));
--  lp | lp_off | lp_len | t_xmin | t_xmax | t_ctid | t_hoff
--   1 |   8160 |     32 |   7421 |      0 | (0,1)  |   24
--   2 |   8128 |     32 |   7421 |      0 | (0,2)  |   24
```

```
-- 每一列的實體位置 (block, slot)
SELECT ctid, * FROM t;
```

觀察重點：tuple 從 8160 往前排（`lp_off` 遞減）；`t_hoff=24` 表示 23 bytes 標頭對齊到 24；`t_ctid` 指向自己，表示這是最新版本。執行一次 `UPDATE t SET name='x' WHERE id=1` 再查，會看到舊版本的 `t_xmax` 被填上、`t_ctid` 指向新版本的 slot——這就是第 9 章的起點。

MiniDB 連結：v1 的 DiskManager、Page 與 HeapFile

MiniDB v1 需要三個類別：`DiskManager`（以 `pread/pwrite` 讀寫第 n 個 4 KB 或 8 KB 頁面）、`Page`（slotted page：標頭 + slot 陣列 + 從尾端成長的 tuple 區）、`HeapFile`（走訪所有頁面、以簡單的「最後一頁優先」策略插入）。記錄序列化採「定長欄位直接複製、變長欄位以 4-byte 長度前綴」即可。第 19 章給出介面與測試建議。

本章練習

1. 一個 8 KB 的 PostgreSQL 頁面最多能放多少筆「只有一個 `INT` 欄位」的 tuple？（提示：標頭 23 bytes 對齊到 24，資料 4 bytes 對齊到 8，每筆加一個 4 bytes slot，頁標頭 24 bytes。）
2. 設計表 `(a bool, b bigint, c int, d bool, e bigint)`，計算以此順序與以最佳順序存放時每列的填補 bytes。
3. 解釋為何索引項存 TID 而非頁內位移；如果沒有 slot 間接層，`UPDATE` 一筆變長記錄會引發什麼連鎖更新？
4. 比較「頁面鏈結串列」與「頁面目錄」兩種 heap file 空間管理策略在 (a) 插入 (b) 掃描 (c) 並行寫入下的優劣。
5. （動手）建立含 `TEXT` 欄位的表，插入一筆 100 KB 的字串，用 `pg_relation_size` 與 `SELECT reltoastrelid::regclass FROM pg_class WHERE relname='t'` 找到 TOAST 表並查看 chunk 數量。
6. （動手）以 `fillfactor=50` 與預設值各建一張表，執行相同的 UPDATE 工作負載後，比較 `pg_stat_user_tables` 的 `n_tup_hot_upd` 與表大小。

延伸閱讀與資源

- CMU 15-445 Fall 2025, Lecture #03 *Database Storage I*、#05 *Database Storage II*（含講義 PDF 與錄影）：`https://15445.courses.cs.cmu.edu/fall2025/schedule.html`
- PostgreSQL 文件 Ch. 66 *Database Physical Storage*：檔案佈局、TOAST、FSM、VM、頁面佈局、HOT：`https://www.postgresql.org/docs/current/storage.html`，特別是 §66.6 *Database Page Layout*：`https://www.postgresql.org/docs/current/storage-page-layout.html`
- Hironobu Suzuki, *The Internals of PostgreSQL*, Ch. 1 *Database Cluster, Databases and Ta-*

bles：`https://www.interdb.jp/pg/pgsql01.html`

- Alex Petrov, *Database Internals*, Ch. 3 *File Formats*（slotted page、cell layout、checksumming）。
- A. Ailamaki et al., *Weaving Relations for Cache Performance*（PAX）, VLDB 2001。
- PostgreSQL 原始碼：`src/include/storage/bufpage.h`、`src/include/access/htup_details.h`、`src/backend/access/heap/heapam.c`、`src/backend/storage/freespace/README`。

第 3 章　B-Tree、B+Tree 與索引

本章學習目標

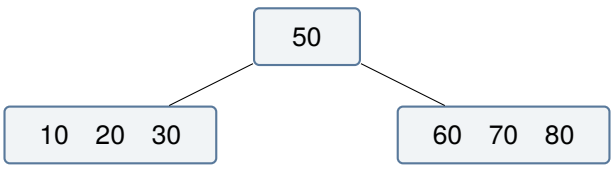
- 說明為何二元搜尋樹與平衡樹不適合磁碟，以及 B-Tree 如何用「一個節點 = 一個頁面」解決問題。
- 區分 B-Tree 與 B+Tree，並解釋為何幾乎所有資料庫索引都是 B+Tree。
- 由頁面大小與鍵長度計算扇出（fanout）與樹高，估算一次點查詢需要幾次 I/O。
- 手動執行節點分裂（split）與合併（merge），並理解為何許多系統刻意不做合併。
- 解釋 Lehman–Yao B-link tree 如何讓讀取幾乎不需要鎖，並對照 PostgreSQL 的 `nbtree`。
- 比較叢集索引與非叢集索引、涵蓋索引、複合索引的最左前綴規則，並判斷何時不該建索引。

3.1　從二元搜尋樹到 B-Tree

在記憶體中，二元搜尋樹（BST）與其平衡版本（AVL、紅黑樹）是很好的有序索引。但一旦資料在磁碟上：

- 每個節點只有一個鍵，樹高 $\log_2 N$；$N = 10^9$ 時高度約 30，意味著一次查詢可能要 30 次隨機 I/O。
- 節點很小（幾十 bytes），而 I/O 單位是 8 KB 的頁面——每次讀入一頁卻只用其中一個鍵，浪費 99%。
- 平衡操作（旋轉）牽動多個節點，每次都是額外的頁面寫入。

Bayer 與 McCreight 在 1972 年提出的 **B-Tree** 把「節點」放大成「一整個頁面」：每個節點放數百個鍵，樹高降為 $\log_f N$（f 是扇出）。$f = 400$、$N = 10^9$ 時高度只有 4。



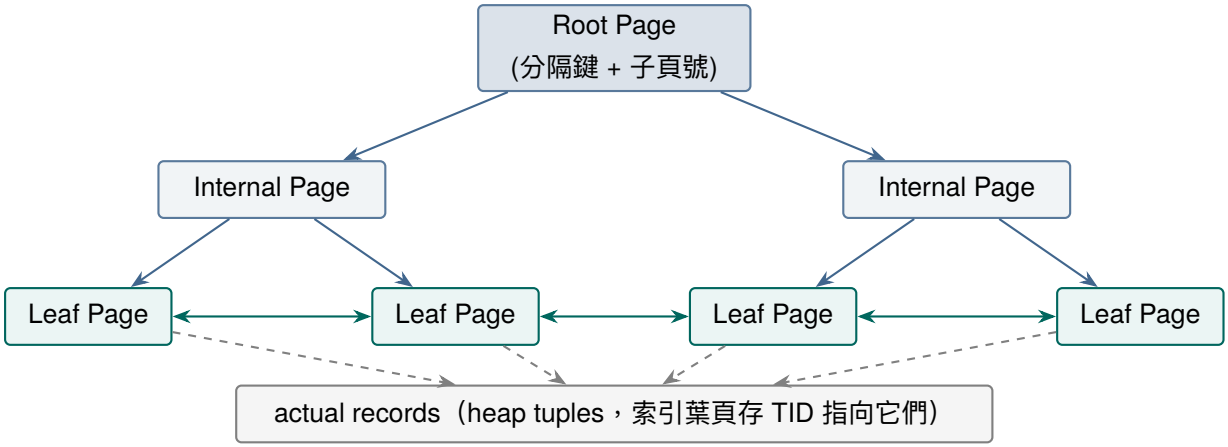
教科書式的小 B-Tree：每個節點多個鍵、多個子節點

3.2 B-Tree 與 B+Tree

- **B-Tree**：每個節點（含內部節點）都存放鍵與對應的值（或記錄指標）。查詢可能在內部節點就結束。
- **B+Tree**：內部節點只存**分隔鍵（separator key）**與子節點指標，所有值都在葉節點；葉節點之間以雙向鏈結串起來。

B+Tree 的優勢使它成為事實標準（PostgreSQL、InnoDB、SQLite、Oracle、SQL Server 都是 B+Tree，即使文件稱之為 B-tree）：

1. 內部節點不存值，能塞更多分隔鍵，**扇出更大、樹更矮**。
2. **範圍掃描**只需找到起點葉頁，然後沿兄弟鏈結走，不必回到上層。
3. 所有查詢路徑長度相同（都到葉），成本可預測。
4. 分隔鍵可以是**截斷後的最短前綴**（suffix truncation），進一步提高扇出。



高度 h 通常只有 3–4 層就能索引數十億筆；葉頁雙向鏈結 ⇒ 範圍掃描只走葉層

3.3 節點結構、扇出與樹高

3.3.1 節點內容

- **內部節點**：$\langle p_0, k_1, p_1, k_2, p_2, \ldots, k_m, p_m \rangle$。子樹 p_i 中所有鍵滿足 $k_i \le key < k_{i+1}$。
- **葉節點**：$\langle (k_1, v_1), (k_2, v_2), \ldots \rangle$ 加上左右兄弟指標。值 v_i 依索引類型是 **TID**（PostgreSQL 的 secondary index）、**主鍵值**（InnoDB secondary index）或**整列**（叢集索引葉頁）。
- 節點本身就是第 2 章的 slotted page：鍵以 slot 排序，支援頁內二分搜尋。PostgreSQL 把兄弟頁號與層級放在 special space。

3.3.2 扇出與高度估算

以 PostgreSQL 的 8 KB 頁面、`BIGINT` 鍵為例：每個索引項 = 8 bytes 索引 tuple 標頭（含 6 bytes TID）+ 8 bytes 鍵 + 4 bytes slot ≈ 20 bytes，扣掉頁標頭與 special space 後約可放 $f \approx 400$

項（預設 fillfactor 90% 時約 360）。高度 h 與可索引筆數 N 的關係：

$$N \approx f^{\,h} \quad \Longrightarrow \quad h = \lceil \log_f N \rceil$$

高度	$f = 400$ 可容納筆數	$f = 100$（長鍵，如 UUID/文字）
1（只有根）	400	100
2	160,000	10,000
3	64,000,000	1,000,000
4	25,600,000,000	100,000,000

一次點查詢實際上幾次 I/O？

根與內部節點只佔索引不到 1% 的頁面（PostgreSQL 文件：「超過 99% 的頁面是葉頁」），因此幾乎永遠在緩衝池裡。一次 `WHERE id = 123` 的典型成本是：**0–1 次索引葉頁 I/O + 1 次 heap 頁 I/O**。這就是為什麼 OLTP 系統能在 SSD 上達到每秒數十萬次點查詢。鍵越短、扇出越大、樹越矮——這是「不要用長字串當主鍵」的量化理由。

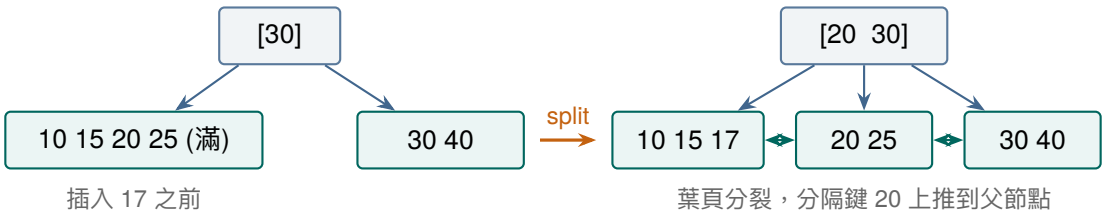
3.4　搜尋、插入、分裂、刪除與合併

3.4.1　搜尋

```
search(key):
  page = root
  while page 不是葉頁:
    i = 頁內二分搜尋，找最大的 i 使得 k_i <= key    (找不到則 i = 0)
    page = read(p_i)                                -- 透過緩衝池
  在葉頁二分搜尋 key，回傳對應的值（或「不存在」）
範圍掃描 [lo, hi]：search(lo) 到達葉頁後，沿右兄弟指標前進直到鍵 > hi
```

3.4.2　插入與分裂

1. 搜尋到應放入的葉頁 L。若 L 有空間，插入並維持排序即可（寫 1 頁）。
2. 若 L 已滿：配置新頁 L'，把 L 的鍵分成兩半（L 留前半、L' 放後半），把 L' 的最小鍵（或其截斷前綴）作為新的分隔鍵**插入父節點**，並更新兄弟鏈結。
3. 父節點若也滿了，遞迴分裂；**根分裂時樹高加一**——這是 B+Tree 唯一會長高的方式，所以它永遠平衡。



分裂點的選擇很重要：單調遞增的鍵（如 `SERIAL`、時間戳）若採 50/50 分裂，左半頁永遠不會再被填滿，空間利用率只有 50%。PostgreSQL 偵測到「插入到最右葉頁」時改用約 90/10 的分裂（並有 fastpath 直接快取最右葉頁），InnoDB 也有類似的循序插入最佳化。

3.4.3 刪除與合併

教科書演算法：刪除後若節點少於半滿，向兄弟**借**一個鍵（重新分配）或與兄弟**合併**並刪除父節點中的分隔鍵，必要時遞迴。

實務上幾乎沒人做教科書式的合併

合併需要同時鎖住兄弟與父節點，與並行插入衝突嚴重，而且刪除常常伴隨新的插入，合併後不久又得分裂。PostgreSQL、InnoDB 都採取「**延遲回收**」：刪除只是移除葉頁中的項目；只有當整頁變空時，才由 VACUUM（PostgreSQL）或 purge（InnoDB）把該頁從樹中摘除並放入可重用清單。代價是刪多插少的表會有大量半空的索引頁（**索引膨脹**），需要 `REINDEX` 或 `pg_repack` 整理。

3.5 索引並行控制：Latch Crabbing 與 B-link Tree

多個執行緒同時讀寫同一棵樹時，需要 **latch**（保護記憶體資料結構的短期鎖，與交易層級的 lock 不同，第 10 章）。

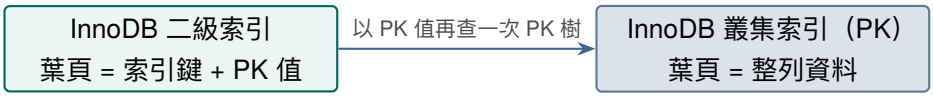
- **Latch crabbing / coupling**：由上而下，取得子節點 latch 後才釋放父節點 latch。寫入時若子節點「安全」（插入不會滿、刪除不會半空）就可釋放所有祖先的 latch。缺點是根節點成為熱點。
- **樂觀 crabbing**：先假設葉頁不會分裂，一路只拿讀 latch，到葉頁才拿寫 latch；猜錯就重來。
- **Lehman–Yao B-link tree（1981）**：每頁加**右兄弟指標**與 **high key**（頁內鍵的上界）。分裂分成兩步（先建右頁並接上右鏈結，再更新父節點），讀者若發現要找的鍵大於 high key，就「向右移動」一頁——不需要在下降時持有任何祖先的鎖。這是 PostgreSQL `nbtree` 的基礎。

nbtree：Lehman & Yao 的實作

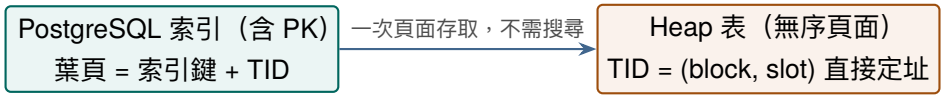
`src/backend/access/nbtree/README` 明確寫道它實作了 Lehman & Yao 的高並行演算法，並做了幾個調整：(1) 因為共享緩衝區會被其他後端修改，PostgreSQL 在讀頁面時仍短暫持有頁面層級的讀鎖（L&Y 原本假設不需要）；(2) 加上**左兄弟指標**以支援反向掃描；(3) 刪除空頁採用 Lanin & Shasha 的簡化版，分兩階段（先標記半死、再由 VACUUM 回收）。PostgreSQL 13 起還有**去重（deduplication）**：相同鍵值的多個 TID 合併成一個 posting list，大幅縮小低基數欄位的索引。

3.6　索引的種類

3.6.1　叢集索引與非叢集索引



(a) MySQL/InnoDB：表就是主鍵 B+Tree；二級索引查詢 = 兩次樹下降



(b) PostgreSQL：所有索引都是二級索引；表是 heap，主鍵不享有特殊儲存地位

- **叢集索引（clustered）**：表的資料就是按索引鍵排序存放。每張表只能有一個。依主鍵的範圍查詢極快（相鄰列在相鄰頁）；代價是二級索引必須存主鍵值作為「列指標」，主鍵越長二級索引越肥，且隨機主鍵插入會造成全樹分裂。InnoDB、SQL Server（預設 PK）、SQLite（`WITHOUT ROWID` 表）採用此法。
- **非叢集索引（non-clustered / secondary）**：索引與資料分離，葉頁存 TID。PostgreSQL 的所有索引都是如此。`CLUSTER` 命令只是「一次性」依索引重排 heap，之後的寫入不會維持順序。

3.6.2　涵蓋索引與 Index-Only Scan

若查詢需要的所有欄位都在索引中，就不必回表——這稱為 **index-only scan**，索引稱為**涵蓋索引（covering index）**。PostgreSQL 11 起可用 `INCLUDE` 把非鍵欄位附在葉頁：

```
CREATE INDEX idx_orders_cust ON orders (customer_id) INCLUDE (total, status);
EXPLAIN SELECT total FROM orders WHERE customer_id = 42;
--  Index Only Scan using idx_orders_cust on orders  (cost=0.43..8.45 rows=1 width=8)
--    Heap Fetches: 0
```

PostgreSQL 的 index-only scan 有一個額外條件

索引項沒有版本可見性資訊（第 9 章），因此執行器必須查 Visibility Map：若該 heap 頁被標為 all-visible，就能跳過回表；否則仍需讀 heap 頁檢查可見性（`EXPLAIN ANALYZE` 會顯示 `Heap Fetches: N`）。VM 位元只由 VACUUM 設定、任何修改都會清除，所以寫入頻繁、VACUUM 不及的表，index-only scan 會退化成一般 index scan。

3.6.3　複合索引與最左前綴

索引 `(a, b, c)` 的鍵以字典序排列，因此能有效支援 `WHERE a = ?`、`a = ? AND b = ?`、`a = ? AND b > ?`，但**不能**有效支援 `WHERE b = ?`（沒有 a 就無法定位）。`a = ? AND c = ?` 可以用 a 定位、再在範圍內過濾 c。範圍條件之後的欄位無法再用於定位：`a > ? AND b = ?` 只能用

a。設計複合索引的口訣：**等值欄位在前、範圍欄位在後、排序欄位緊接著範圍欄位**。

3.6.4 其他索引型別（PostgreSQL）

型別	適用	備註
B-tree	等值、範圍、排序、`LIKE 'abc%'`	預設；本章主角
Hash	僅等值	PostgreSQL 10 起有 WAL 保護；比 B-tree 小一點、快一點點
GiST	幾何、範圍型別、全文、最近鄰（KNN）	可擴充的平衡搜尋樹框架
SP-GiST	四元樹、基數樹、字首搜尋	非平衡、空間分割
GIN	陣列、JSONB、全文檢索（多值欄位）	倒排索引；寫入慢、查詢快
BRIN	與實體順序高度相關的大表（時間序列）	每個區塊範圍只存 min/max，極小
部分索引	`CREATE INDEX ... WHERE status='open'`	只索引子集，省空間與寫入
表達式索引	`CREATE INDEX ON t (lower(email))`	查詢必須寫出相同表達式

3.7 索引的代價

每個索引都是**寫入放大**：一筆 `INSERT` 要更新表加上每一個索引（PostgreSQL 的非 HOT `UPDATE` 亦同）；每個索引都佔緩衝池與磁碟空間，並產生 WAL。經驗法則：

- 選擇性低的欄位（如 boolean、性別）單獨建索引通常沒用——最佳化器寧可全表掃描（第 6 章解釋原因）。
- 寫多讀少的表，索引數量要克制；考慮部分索引只涵蓋熱資料。
- 用 `pg_stat_user_indexes.idx_scan` 找出從未被用過的索引並移除。
- 建索引用 `CREATE INDEX CONCURRENTLY` 避免長時間鎖表（代價是需掃描兩次且不能在交易內執行）。

3.8 B+Tree 變體速覽

- **Copy-on-Write B-Tree**（LMDB、BoltDB、*Build Your Own Database* 的範例）：修改不就地覆寫，而是複製路徑上的頁面到新位置，最後原子地切換根指標。天然提供快照與崩潰安全，不需要 WAL；代價是寫入放大與空間回收。
- **Bw-Tree**（Microsoft Hekaton）：以 delta record 鏈與 mapping table 實現 lock-free；論文漂亮但實作極難，CMU 的 Peloton 經驗顯示效能未必勝過加鎖的 B+Tree。
- **Fractal Tree / B^ϵ-tree**（TokuDB）：在內部節點放訊息緩衝區，批次下推寫入，是 B+Tree 與 LSM 之間的折衷。
- **前綴壓縮與 suffix truncation**：同頁內鍵共享前綴只存一次；分隔鍵只保留足以區分左右的最短前綴。
- **批次建立（bulk loading）**：`CREATE INDEX` 不是逐筆插入，而是先外部排序所有 (鍵, TID)，

再由下而上填滿葉頁、建內部層，速度快一個數量級且頁面緊密。

3.9 動手觀察：pageinspect 與 EXPLAIN

程式碼 3.1: 觀察 B-tree 的元頁、層級與葉頁內容

```
CREATE TABLE big AS SELECT g AS id, md5(g::text) AS payload FROM generate_series(1, 1000000) g;
CREATE INDEX big_id_idx ON big (id);

SELECT * FROM bt_metap('big_id_idx');             -- root 頁號、level（樹高−1）、fastroot
SELECT * FROM bt_page_stats('big_id_idx', 1);      -- 某頁：type (l/i/r)、live_items、free_size
SELECT itemoffset, ctid, data FROM bt_page_items('big_id_idx', 1) LIMIT 5;

EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM big WHERE id = 500000;         -- Index Scan
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM big WHERE id BETWEEN 1 AND 50000; -- Bitmap Heap Scan
EXPLAIN (ANALYZE, BUFFERS) SELECT id FROM big WHERE id < 1000;           -- Index Only Scan
```

`BUFFERS` 會顯示 `shared hit`（緩衝池命中）與 `read`（實際 I/O）的頁數——把它與你用扇出算出的樹高對照。**Bitmap Heap Scan** 是 PostgreSQL 的中間策略：先用索引收集所有符合的 TID 做成點陣圖，依頁號排序後再循序讀 heap，把隨機 I/O 變成近似循序；當符合列數佔表的數個百分比時特別划算。

MiniDB 連結：v2 的 B+Tree 索引

v2 在 v1 的 page manager 上實作 B+Tree：內部頁 `(key, page_id)`、葉頁 `(key, RID)` 加左右兄弟；先做單執行緒版本的 search / insert / split / range-scan，刪除採「不合併、只刪項」的簡化策略。第 19 章建議的測試：隨機插入十萬筆後驗證（a）中序走訪有序、（b）每個葉頁都在 fillfactor 範圍內、（c）高度符合 $\lceil \log_f N \rceil$。並行版本（crabbing）留到 v4 之後。

本章練習

1. 16 KB 的 InnoDB 頁面、`BIGINT` 主鍵、每列 200 bytes：估算叢集索引的葉頁與內部頁扇出，以及一億列的樹高。若主鍵改為 36 字元的 UUID 字串，二級索引（鍵 8 bytes）每個葉項會多出多少 bytes？
2. 以扇出 4 的 B+Tree 依序插入 1..20，畫出每次分裂後的樹；再以隨機順序插入，比較頁面利用率。
3. 解釋 Lehman–Yao 的 high key 與右鏈結如何讓「正在下降的讀者」在遇到並行分裂時仍能找到正確的葉頁。
4. 查詢 `WHERE a = 1 AND c = 5 ORDER BY b` 應建哪個複合索引？`WHERE a > 1 AND b = 5` 呢？說明理由。
5. 為何在 PostgreSQL 中 `SELECT COUNT(*) FROM t` 即使有主鍵索引也可能需要回表？什麼條件下能完全走 index-only scan？

6. （動手）對一張一百萬列的表建索引後，用 `bt_metap` 讀出 level，再用本章公式估算，解釋差異。接著 `DELETE` 90% 的列並 `VACUUM`，用 `pgstatindex` 觀察 `avg_leaf_density` 與 `leaf_fragmentation`。

延伸閱讀與資源

- CMU 15-445 Fall 2025, Lecture #06 *B+Tree Index*、#07 *Hash Tables*、#10 *Index Concurrency Control*：`https://15445.courses.cs.cmu.edu/fall2025/schedule.html`；Project #2 *Database Index*。
- PostgreSQL 文件 Ch. 11 *Indexes*（`https://www.postgresql.org/docs/current/indexes.html`）、§11.9 *Index-Only Scans and Covering Indexes*、Ch. 65.1 *B-Tree Indexes*（`https://www.postgresql.org/docs/current/btree.html`）。
- PostgreSQL 原始碼 `src/backend/access/nbtree/README`（Lehman–Yao 實作細節、頁面刪除、去重）：`https://github.com/postgres/postgres/tree/master/src/backend/access/nbtree`
- P. Lehman & S. Yao, *Efficient Locking for Concurrent Operations on B-Trees*, ACM TODS 1981。
- G. Graefe, *Modern B-Tree Techniques*, Foundations and Trends in Databases, 2011（B-tree 技術的百科全書）。
- Markus Winand, *Use The Index, Luke!*（面向開發者的索引教學，含各資料庫範例）：`https://use-the-index-luke.com/`
- Alex Petrov, *Database Internals*, Ch. 2、4、6（B-Tree 基礎、實作、變體）。

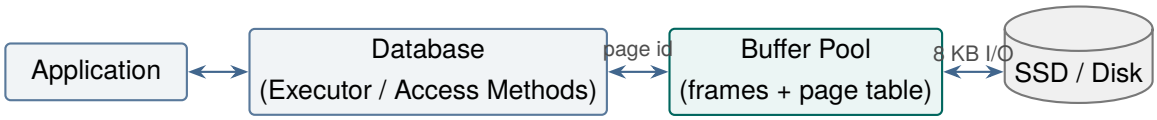
第 4 章　Buffer Pool

本章學習目標

- 說明資料庫為何要自己管理頁面快取，而不是直接依賴作業系統的 page cache 或 `mmap`。
- 畫出 buffer pool 的結構：frame、page table、dirty bit、pin count、置換用的元資料。
- 比較 LRU、Clock、LRU-K、2Q、ARC 等置換策略，並解釋「循序掃描洪水（sequential flooding）」問題與對策。
- 說明 dirty page 的寫回時機、WAL 規則（先寫日誌再寫頁面）與 steal/no-force 政策。
- 解釋 pin（釘住）與 latch 的角色，以及預讀（read-ahead）如何把隨機 I/O 變成循序 I/O。
- 完整回答：「執行 `SELECT * FROM users WHERE id=123` 時，資料庫如何從 SSD 取得資料？」

4.1　為什麼需要 Buffer Pool

資料庫不可能每次存取都讀磁碟。**Buffer pool**（又稱 buffer cache、buffer manager）是記憶體中一塊由資料庫自己管理的頁面快取，位於執行引擎與磁碟之間：



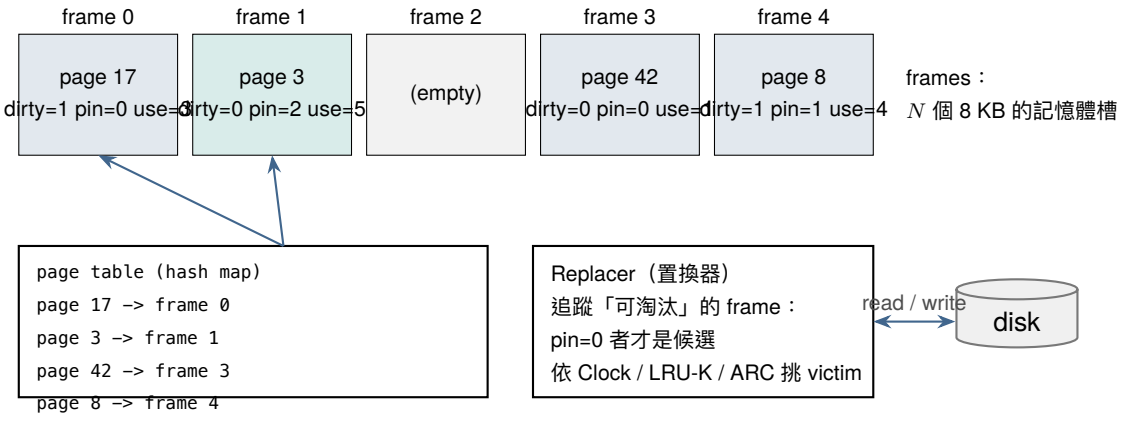
作業系統本來就有 page cache，為什麼資料庫還要自己做一份？因為資料庫比作業系統**更了解自己的存取模式**，而且有 OS 無法提供的正確性需求：

- **寫回順序必須受控**：WAL 規則要求「日誌先落盤，資料頁才能落盤」（第 8 章）。OS page cache 會在任意時間把髒頁寫出，資料庫無法阻止；而 `mmap` 的頁面被核心寫出時，資料庫甚至不知道。
- **置換策略需要語意**：資料庫知道「這是一次性的全表掃描，別把熱頁面趕走」、「這是 B+Tree 根頁，永遠留著」；OS 只看到匿名的 4 KB 區塊。
- **預讀需要知識**：資料庫知道 B+Tree 葉頁鏈結的下一頁是哪一頁、bitmap scan 接下來要讀哪些 heap 頁；OS 的 read-ahead 只能猜「下一個連續區塊」。
- **錯誤處理與可觀測性**：`mmap` 的 I/O 錯誤以 `SIGBUS` 出現，無法乾淨地回報給交易；pin 計數、命中率等統計也只有自管快取才做得到。

「Are You Sure You Want to Use MMAP in Your DBMS?」

Crotty、Leis 與 Pavlo 在 CIDR 2022 的這篇論文系統性地整理了 `mmap` 的問題：交易安全（無法控制寫回）、I/O 停頓（page fault 不可預測）、錯誤處理、效能（TLB shootdown、單執行緒的 page eviction）。MongoDB（MMAPv1）、LMDB、早期 InfluxDB 都踩過這些坑；多數成熟系統最終都回到自管的 buffer pool。論文與影片：`https://db.cs.cmu.edu/mmap-cidr2022/`

4.2 Buffer Pool 的結構



- **Frame**：固定大小的記憶體槽，一次放一頁。PostgreSQL 的 `shared_buffers` 就是一個 frame 陣列。
- **Page table**：從 page id 對應到 frame id 的雜湊表。PostgreSQL 把它切成 128 個分區（`NUM_BUFFER_PARTITIONS`），各自上鎖以降低爭用。
- **每個 frame 的元資料**（PostgreSQL 的 `BufferDesc`）：頁面標籤（哪個關聯的哪一頁）、**dirty** 位元、**pin count**（refcount）、**usage count**、I/O 進行中旗標、內容 latch。PostgreSQL 把 refcount(18 bits)、usage count(4 bits)、旗標 (12 bits) 塞進一個 64-bit 原子變數，以 CAS 操作避免加鎖。

4.3 核心操作

程式碼 4.1: Buffer pool 的四個基本操作（偽碼）

```
fetch_page(pid):
  if pid in page_table:                  # 命中 (hit)
    frame = page_table[pid]; frame.pin++; replacer.record_access(frame)
    return frame
  frame = free_list.pop() or replacer.evict()   # 未命中：找空 frame 或淘汰 victim
  if frame is None: raise "no unpinned buffers available"
  if frame.dirty:
    wal.flush_up_to(frame.page.lsn)     # WAL 規則：先刷日誌，才能寫髒頁
    disk.write(frame.page_id, frame.data)
  page_table.remove(frame.page_id)
```

```
  disk.read(pid, frame.data)            # 真正的 I/O
  page_table[pid] = frame; frame.pin = 1; frame.dirty = False
  return frame

unpin_page(pid, is_dirty):  frame.pin--; frame.dirty |= is_dirty
flush_page(pid):            wal.flush_up_to(lsn); disk.write(...); frame.dirty = False
new_page():                 pid = disk.allocate(); 取一個 frame 清零並登記
```

Pin 與 Latch 是兩回事

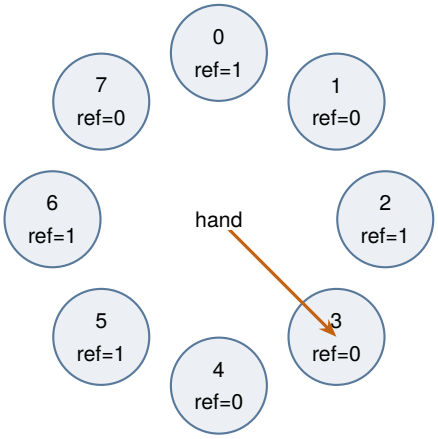
Pin（釘住）告訴置換器「這頁正在被使用，不准淘汰」；它不保護內容。**Latch**（PostgreSQL 稱 buffer content lock，讀寫鎖）保護頁面內容不被同時修改。一個典型的讀取流程：pin → 取讀 latch → 讀取 → 放讀 latch → unpin。Pin 可以持有較久（例如游標停在某頁），latch 必須極短。忘記 unpin 是 buffer pool 實作最常見的 bug——最終導致「no unpinned buffers available」。

4.4 置換策略

當沒有空 frame 時，必須選一個 **victim** 淘汰。理想策略（Bélády 的 OPT：淘汰最久以後才會用到的頁）需要預知未來，實務上以歷史近似：

策略	做法	問題 / 採用者
LRU	維護依最近使用時間排序的串列，淘汰最久未用者	每次存取都要搬動串列節點（需鎖）；對一次性掃描完全沒有抵抗力
Clock（second chance）	frame 排成環，每個有一個 reference bit；指針掃過時把 1 改 0、遇到 0 就淘汰	LRU 的廉價近似；PostgreSQL 的 clock-sweep（usage count 上限 5）
LRU-K	記錄最近 K 次存取時間，以倒數第 K 次的距離排序	只被存取一次的頁（掃描）永遠排在最後；BusTub 早期版本採 LRU-K
2Q / midpoint LRU	新頁先進「試用區」，再次被存取才升入「主區」	InnoDB：LRU 分 young/old 兩段，新頁插在 5/8 處，`innodb_old_blocks_time` 防掃描污染
ARC	同時維護「最近用過」與「頻繁用過」兩個 LRU 加上 ghost list，自適應調整兩者比例	Megiddo & Modha (FAST 2003)；CMU 15-445 Fall 2025 Project #1 採用 ARC；ZFS 使用
LFU	依存取頻率淘汰	老舊熱頁難以被淘汰（需要衰減）

4.4.1 Clock 演算法



指針指向 frame 3：ref=0 且未 pin ⇒ 淘汰。
若 ref=1：清為 0 並前進（給它第二次機會）。
PostgreSQL 的 usage_count 是 0–5 的計數器，每次命中 +1，指針掃過時 -1，到 0 才淘汰。

4.4.2 循序掃描洪水與對策

一次全表掃描會把整張表的頁面依序讀進來，每頁只用一次。在純 LRU 下，這會把所有真正熱門的頁面（索引根頁、常用的小表）全部趕出去——這就是 **sequential flooding**。對策：

- **MRU**：對已知的掃描改用「淘汰最近用的」。
- **ring buffer（環形緩衝）**：掃描只在一小塊私有的 frame 子集內循環重用。PostgreSQL 對大於 `shared_buffers/4` 的表的循序掃描使用 256 KB 的 ring（`BAS_BULKREAD`），`COPY/CREATE TABLE AS` 用 16 MB（`BAS_BULKWRITE`），VACUUM 用 2 MB（可由 `vacuum_buffer_usage_limit` 調整）。
- **優先級提示**：LRU-K、2Q、ARC 天生把「只被碰過一次」的頁排在淘汰隊伍前端。

clock-sweep 與 ring buffer 的實作

`src/backend/storage/buffer/freelist.c` 的 `StrategyGetBuffer()`：先看 strategy ring 有沒有可重用的 frame，再嘗試 free list，最後進入 clock sweep 迴圈：以原子操作讀取 `nextVictimBuffer`，若 frame 被 pin 就跳過，若 `usage_count > 0` 就減一繼續，否則以 CAS 把它 pin 住並回傳。`BM_MAX_USAGE_COUNT` 定為 5——註解寫得很清楚：值越大越接近 LRU，但最壞要掃 6 圈才找得到 victim，所以不能太大。

4.5 Dirty Page 的寫回

被修改的頁面標記為 **dirty**，不會立刻寫回磁碟（write-back 而非 write-through）——這是效能的關鍵：一頁可能被連續修改幾百次，只需寫回一次。髒頁在三種時機落盤：

1. **被選為 victim 時**（前景寫回，會拖慢查詢）。
2. **背景寫入器**（PostgreSQL 的 `bgwriter`、InnoDB 的 page cleaner）定期掃描並寫出即將被淘汰的髒頁，讓前景少做事。
3. **Checkpoint**：把某個時間點之前的所有髒頁全部寫出，以縮短當機復原時間（第 8、11 章）。PostgreSQL 的 checkpointer 會把寫入分散在 `checkpoint_completion_target`（預設 0.9）的

時間內，避免 I/O 尖峰。

WAL 規則與 steal / no-force

寫回任何髒頁之前，必須先確保「描述該頁修改的 WAL 記錄」已經落盤（頁面的 `pd_lsn` ≤ 已刷出的 WAL 位置）。在復原理論的術語中（第 11 章）：

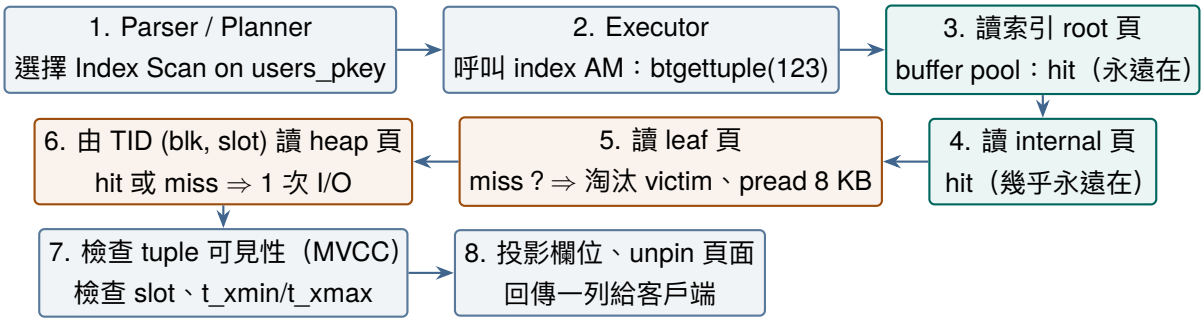
- **Steal**：允許把未提交交易修改過的髒頁寫回磁碟（因為 victim 可能屬於進行中的交易）⇒ 需要 undo 資訊。
- **No-force**：提交時不強制把資料頁寫回，只要 WAL 落盤即可 ⇒ 需要 redo 資訊。

幾乎所有高效能資料庫都是 **steal + no-force**，這正是 WAL 同時需要 redo 與 undo 能力的原因（PostgreSQL 以 MVCC 舊版本取代 undo，第 9 章）。

4.6 預讀與非同步 I/O

- **循序預讀**：循序掃描時提前發出後續頁面的讀取。PostgreSQL 18 之前主要靠 OS read-ahead；18 起引入 **read stream** 介面與真正的非同步 I/O（`io_method = sync | worker | io_uring`），由資料庫自己決定「往前看幾頁」（依最近的命中/未命中歷史動態調整），並把相鄰頁面合併成最大 `io_combine_limit` 的單一 I/O。
- **索引驅動的預讀**：bitmap heap scan 知道接下來要讀哪些 heap 頁，`effective_io_concurrency` 控制同時發出的預讀數量——在 SSD 上可以設到 100 以上。
- **AIO 的另一個收穫**：避免 **double buffering**（同一頁同時佔據 OS page cache 與 shared_buffers）。搭配 direct I/O 時資料由 DMA 直接進 buffer pool，CPU 不必複製。InnoDB 長期以 `O_DIRECT` 加大 buffer pool（常設為 RAM 的 70–80%）；PostgreSQL 傳統建議 `shared_buffers` 約 RAM 的 25%、其餘留給 OS cache，並以 `effective_cache_size` 告訴最佳化器總快取量。

4.7 完整走一遍：`SELECT * FROM users WHERE id = 123`



1. 解析器把 SQL 轉成查詢樹；最佳化器看到主鍵等值條件，估算 index scan 成本遠低於 seq scan（第 6 章），產生 `Index Scan using users_pkey`。
2. 執行器啟動計畫節點，呼叫 B-tree 存取方法搜尋鍵 123。
3. B-tree 從 metapage 取得 root 頁號，向 buffer pool 要該頁：page table 命中（root 永遠是熱頁），pin、取讀 latch、二分搜尋找到子頁號。

4. 內部頁同樣幾乎必定命中。
5. 葉頁可能未命中：buffer pool 以 clock sweep 選出 victim（若為 dirty 先刷 WAL 再寫回），然後 `pread()` 從 SSD 讀 8 KB（約 100 μs）。在葉頁中找到鍵 123 的索引項，取得 TID，例如 (block 1592, slot 3)。
6. 執行器拿 TID 向 buffer pool 要 heap 頁 1592（`ReadBuffer`）。這是**直接定址**，不需要再搜尋。
7. 從 slot 3 取得 tuple，檢查 `t_xmin`/`t_xmax` 與目前快照決定可見性（第 9 章）；若不可見且 `t_ctid` 指向新版本，就沿鏈追蹤。
8. 把需要的欄位解出（可能要去 TOAST 表），unpin 兩頁，回傳結果。

總計：**最佳情況 0 次 I/O，典型情況 1–2 次隨機讀**，加上幾次 page table 查找與 latch 操作。全表掃描則是 `relpages` 次循序讀——這個對比就是第 6 章成本模型的核心。

4.8 動手觀察：pg_buffercache

程式碼 4.2: 看看 shared_buffers 裡住著誰

```sql
SELECT c.relname, count(*) AS buffers, sum(CASE WHEN isdirty THEN 1 ELSE 0 END) AS dirty,
       round(100.0 * count(*) / (SELECT setting::int FROM pg_settings WHERE name='shared_buffers'),
    ↪  2) AS pct
FROM   pg_buffercache b JOIN pg_class c ON b.relfilenode = pg_relation_filenode(c.oid)
GROUP  BY c.relname ORDER BY buffers DESC LIMIT 10;

SELECT usagecount, count(*) FROM pg_buffercache GROUP BY 1 ORDER BY 1;   -- clock-sweep 的計數分佈

-- 緩衝池命中率（整個資料庫）
SELECT blks_hit::float / NULLIF(blks_hit + blks_read, 0) AS hit_ratio FROM pg_stat_database WHERE
    ↪ datname = current_database();
```

MiniDB 連結：v3 的 BufferPoolManager

v3 在 v1/v2 之間插入 `BufferPoolManager`：`fetch_page / unpin_page / flush_page / new_page / delete_page`，搭配可替換的 `Replacer` 介面（先做 Clock，再做 LRU-K 或 ARC）。把 v2 的 B+Tree 改成「只透過 buffer pool 存取頁面」，並用一個「隨機點查詢 + 一次全表掃描」的混合工作負載比較 LRU 與 Clock/LRU-K 的命中率——你會親眼看到 sequential flooding。

本章練習

1. 3 個 frame、存取序列 `1 2 3 1 4 1 2 5 1 2 3 4 5`：分別以 LRU、Clock、LRU-2 計算未命中次數並列出每步的 frame 內容。
2. 一張 100 萬頁的表、buffer pool 1 萬頁，熱門小表佔 500 頁。說明純 LRU 下一次全表掃描後熱門小表的命中率會發生什麼事；ring buffer 如何解決。

3. 為什麼淘汰 dirty victim 前必須先刷 WAL？若違反這條規則，當機後會出現什麼不一致？
4. 解釋 pin count 與 latch 的差別。如果某個 bug 在 fetch 之後忘了 unpin，系統會逐漸出現什麼症狀？
5. 把 `shared_buffers` 設為 RAM 的 90% 在 PostgreSQL 上通常不是好主意，原因為何？同樣設定在 InnoDB 上為何常見？
6. （動手）執行 `EXPLAIN (ANALYZE, BUFFERS)` 對同一個點查詢連跑兩次，比較 `shared read` 與 `shared hit`；再對一張大表做 `SELECT count(*)`，用 `pg_buffercache` 驗證它只佔用了約 256 KB 的 ring。

延伸閱讀與資源

- CMU 15-445 Fall 2025, Lecture #04 *Memory Management*；Project #1 *Buffer Pool Manager*（ARC replacer、disk scheduler、BPM）：`https://15445.courses.cs.cmu.edu/fall2025/project1/`
- A. Crotty, V. Leis, A. Pavlo, *Are You Sure You Want to Use MMAP in Your Database Management System?*, CIDR 2022：`https://db.cs.cmu.edu/mmap-cidr2022/`
- Hironobu Suzuki, *The Internals of PostgreSQL*, Ch. 8 *Buffer Manager*：`https://www.interdb.jp/pg/pgsql08.html`
- PostgreSQL 原始碼 `src/backend/storage/buffer/README`、`freelist.c`（clock sweep、ring）、`bufmgr.c`；`src/backend/storage/aio/README.md`（PostgreSQL 18 非同步 I/O 設計）。
- PostgreSQL 文件 §19.4 *Resource Consumption*（`shared_buffers`、`effective_io_concurrency`、`io_method`）與 F. `pg_buffercache`：`https://www.postgresql.org/docs/current/pgbuffercache.html`
- N. Megiddo & D. Modha, *ARC: A Self-Tuning, Low Overhead Replacement Cache*, FAST 2003；E. O'Neil et al., *The LRU-K Page Replacement Algorithm*, SIGMOD 1993。
- MySQL 參考手冊 *Buffer Pool*（midpoint insertion、young/old sublist）：`https://dev.mysql.com/doc/refman/8.4/en/innodb-buffer-pool.html`

第三篇

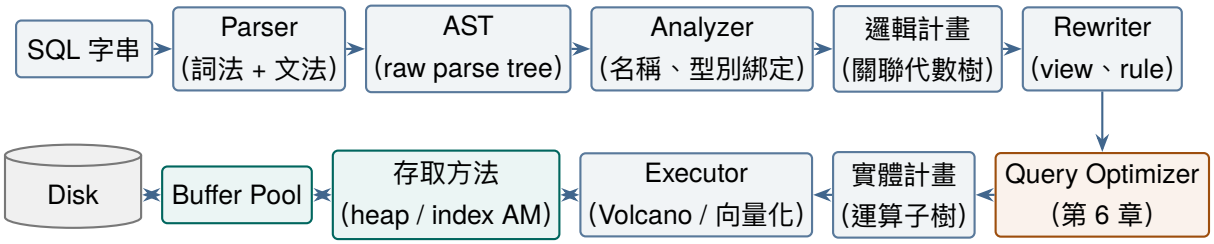
查詢處理

第 5 章　查詢執行

本章學習目標

- 描述一句 SQL 從字串到結果的完整管線：Parser → AST → 邏輯計畫 → 最佳化 → 實體計畫 → 執行器 → 儲存引擎。
- 區分邏輯運算子（Join）與實體運算子（Hash Join、Nested Loop、Merge Join）。
- 比較三種處理模型：Volcano 迭代器（tuple-at-a-time）、向量化（batch）、推送式程式碼生成。
- 以 I/O 成本公式分析 nested-loop、sort-merge、hash join，知道各自適用的情境。
- 說明外部合併排序與雜湊聚合如何在記憶體不足時溢出到磁碟。
- 看懂 `EXPLAIN ANALYZE` 的每一行，並對照到本章的運算子。

5.1　從 SQL 到結果的管線

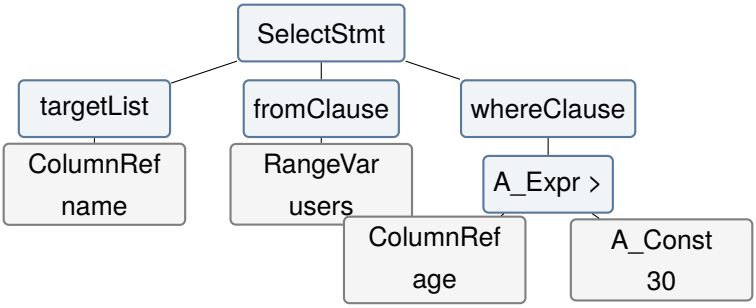


PostgreSQL 的五個階段

PostgreSQL 文件第 50 章 *Overview of PostgreSQL Internals* 把查詢處理分為：(1) **parser**（`src/backend/parser/gram.y`，bison 文法，產生 raw parse tree）；(2) **analyzer**（`analyze.c`，查系統目錄解析表名、欄位、型別、運算子，產生 `Query` 結構）；(3) **rewriter**（`src/backend/rewrite/`，展開 view 與 rule）；(4) **planner/optimizer**（`src/backend/optimizer/`，產生 `PlannedStmt`）；(5) **executor**（`src/backend/executor/`）。設定 `debug_print_parse / debug_print_rewritten / debug_print_plan = on` 可在日誌中看到每一階段的樹。

5.2　Parser 與 AST

Parser 分兩層：**詞法分析**（lexer）把字串切成 token（關鍵字、識別字、常數、運算子），**語法分析**依文法規則組成**抽象語法樹（AST）**。以 `SELECT name FROM users WHERE age > 30` 為例：



AST 只反映語法，還不知道 `users` 是哪張表、`age` 是什麼型別——那是 analyzer 的工作，它查系統目錄把每個名稱綁定到 OID，並為運算子選出正確的實作（`int4gt`）。**Prepared statement** 的價值就在這裡：解析與分析（甚至規劃）只做一次，之後以參數重複執行。

5.3　邏輯計畫與實體計畫

- **邏輯計畫**是第 1 章的關聯代數樹：$\pi_{name}(\sigma_{age>30}(\mathit{users}))$。它只說「做什麼」。
- **實體計畫**為每個邏輯運算子選定**演算法**與**存取路徑**：σ 用 Seq Scan 加 Filter 還是 Index Scan？$\bowtie$ 用 Hash Join、Nested Loop 還是 Merge Join？輸出要不要排序？要不要平行？

一個邏輯運算子可能對應多個實體運算子，反之亦然（Index Scan 同時完成 σ 與資料存取）。最佳化器（第 6 章）就是在實體計畫空間中搜尋。

5.4　處理模型

執行器如何在運算子之間傳遞資料，是效能的決定性因素之一：

5.4.1　Volcano / 迭代器模型

Graefe 的 Volcano（1994）定義每個運算子都實作 `open()`、`next()`、`close()`：根節點呼叫子節點的 `next()` 拉取一筆 tuple，子節點再向它的子節點拉，直到葉節點的掃描。**拉式（pull-based）、一次一筆**。

程式碼 5.1: Volcano 迭代器介面與一個 Filter 運算子

```
class Executor {
 public:
  virtual void Init() = 0;                       // open
  virtual bool Next(Tuple *out, RID *rid) = 0;   // 回傳 false 表示結束
  virtual ~Executor() = default;                 // close
};

class FilterExecutor : public Executor {
 public:
  void Init() override { child_->Init(); }
  bool Next(Tuple *out, RID *rid) override {
    while (child_->Next(out, rid)) {
```

```
      if (predicate_->Evaluate(*out).GetAs<bool>()) return true;   // 只讓符合者通過
    }
    return false;
  }
 private:
  std::unique_ptr<Executor> child_;
  const Expression *predicate_;
};
```

優點：簡單、組合性極佳（任何運算子都能接任何運算子）、**管線化**（不需物化中間結果）、`LIMIT` 可以提早停止。缺點：每筆 tuple 都要經過多層虛擬函數呼叫，CPU 分支預測差、快取局部性低——在資料都在記憶體的時代成為瓶頸。PostgreSQL、MySQL、SQLite（以 bytecode VM 形式）、早期多數系統都是 Volcano。

5.4.2　物化模型（Materialization）

每個運算子一次處理全部輸入、產生完整輸出再交給父節點。函數呼叫少，適合 OLTP 的小結果集與記憶體資料庫（VoltDB、早期 MonetDB）；但中間結果可能極大，不適合 OLAP。

5.4.3　向量化模型（Vectorized / Batch）

MonetDB/X100（Boncz et al. 2005）的洞見：仍用 Volcano 的 `next()`，但每次回傳**一批**（如 1024–2048 筆）同一欄位的值（向量），運算子內部以緊湊迴圈處理整批資料。函數呼叫開銷攤到上千筆、迴圈可被編譯器自動 SIMD 化、資料留在 CPU 快取。DuckDB（`STANDARD_VECTOR_SIZE = 2048`）、ClickHouse、Velox、Snowflake、Databricks Photon 都是向量化引擎（第 13 章）。

5.4.4　推送式與程式碼生成

Neumann 的 HyPer（2011）反其道而行：**推式（push-based）**——葉節點把 tuple 推給父節點；並把整條管線（pipeline）**編譯成機器碼**（LLVM），使資料盡量留在暫存器中直到遇到 pipeline breaker（排序、雜湊建表、聚合）。產生的程式碼接近手寫 C++。代價是編譯延遲（毫秒級，對短查詢不划算）與實作複雜度。PostgreSQL 11 起以 LLVM JIT 編譯*表達式求值*與 tuple 解構（`jit = on`，由成本門檻 `jit_above_cost` 控制），但運算子仍是 Volcano。

模型	資料粒度	優點	代表系統
Volcano 迭代器	一筆 tuple	簡單、可組合、管線化	PostgreSQL、MySQL、SQLite、Oracle
物化	整個中間結果	函數呼叫少	VoltDB、早期 MonetDB
向量化	一批（1k–4k）欄位值	CPU 快取友善、SIMD、容易實作	DuckDB、ClickHouse、Velox、Snowflake
推送式 + 編譯	一筆（留在暫存器）	接近手寫碼的速度	HyPer/Umbra、Spark Tungsten、SingleStore

5.5 存取方法

葉節點負責把資料從儲存引擎拉出來：

- **Seq Scan**：依序讀 heap 的每一頁，對每筆 tuple 套用 Filter。成本 ≈ 頁數 × 循序 I/O。永遠可用，對小表或低選擇性查詢反而最快。
- **Index Scan**：以索引找到 TID，逐筆回 heap 取 tuple。高選擇性時極快，但 heap 存取是隨機 I/O。
- **Index Only Scan**：全部欄位都在索引中、且 VM 顯示頁面 all-visible 時，完全不碰 heap（第 3 章）。
- **Bitmap Index Scan + Bitmap Heap Scan**：先用一或多個索引收集 TID 建成點陣圖（可對多個索引做 AND/OR），依頁號排序後循序讀 heap；中等選擇性的最佳解。點陣圖太大時退化為「每頁一個 bit」的 lossy 模式，需重新檢查條件。
- **TID Scan**：`WHERE ctid = '(0,1)'`，直接定址。

5.6 Join 演算法

設 R 有 M 頁、m 筆 tuple，S 有 N 頁、n 筆 tuple，緩衝池可用 B 頁。以下成本以 I/O 頁數計。

5.6.1 Nested Loop Join

```
for each tuple r in R:            -- 外層 (outer)
  for each tuple s in S:          -- 內層 (inner)
    if r.key == s.key: emit (r, s)
```

- **簡單 NLJ**：成本 $M + m \cdot N$。每筆外層 tuple 都把內層掃一遍，幾乎不可接受。
- **Block NLJ**：一次讀一塊外層（$B-2$ 頁），內層每塊只掃一次：$M + \lceil M/(B-2) \rceil \cdot N$。外層選小表。
- **Index NLJ**：內層有索引時，每筆外層 tuple 做一次索引查找：$M + m \cdot C$（C 為一次查找成本，通常 1–3 頁）。這是 OLTP 中最常見的 join：外層幾筆、內層有主鍵索引。

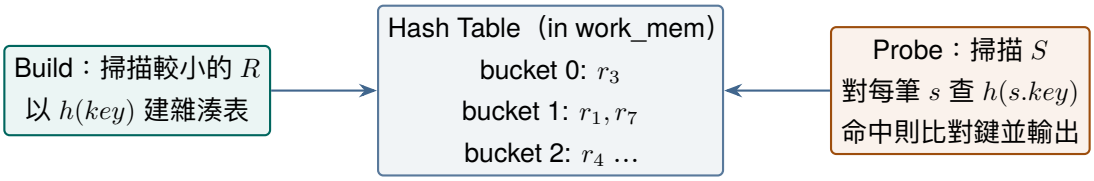
NLJ 的優點：不需要等值條件（可做任意 θ-join）、可管線化、外層很小時無人能敵。

5.6.2 Sort-Merge Join

1. 分別對 R、S 依 join 鍵排序（若已有序或有索引可省略）。
2. 兩個游標同步前進：鍵相等就輸出配對（注意重複鍵需要回溯），較小的一方前進。

成本 $=$ sort(R) + sort(S) + $(M+N)$；排序以外部合併排序計（下一節）。當輸入已排序、或輸出需要依 join 鍵排序（`ORDER BY`、後續 merge）時特別划算；也是不等值條件（`<`）之外的範圍 join 友善演算法。

5.6.3　Hash Join



成本 $\approx M + N$（兩表各讀一次），前提是雜湊表放得進記憶體

- **In-memory hash join**：成本 $M + N$，是等值 join 的首選。
- **Grace / partitioned hash join**（雜湊表放不下時）：先用 h_1 把 R、S 各自分割成 $B - 1$ 個桶寫回磁碟，再逐桶配對（桶內用 h_2 建表）。成本 $3(M + N)$：讀寫各一次分割、再讀一次配對。
- **Hybrid hash join**：記憶體夠時把第一個分區直接留在記憶體，省掉它的寫回。
- **Bloom filter**：建表時順便建 bloom filter，probe 端先用它過濾掉大多數不會配對的 tuple，減少雜湊表查找與下推至掃描端。
- 雜湊 join 只支援等值條件、且必須等 build 端全部讀完才能輸出第一筆（pipeline breaker）。

PostgreSQL 的 Hash Join

`src/backend/executor/nodeHash.c` 以 `work_mem` 為上限：若估計放不下，就把 build 端切成多個 **batch**（2 的冪個），其餘 batch 暫存到磁碟檔；probe 端也依同樣的雜湊切 batch。`EXPLAIN ANALYZE` 會顯示 `Buckets: 1024 Batches: 4 Memory Usage: 3 kB`——`Batches > 1` 代表溢出到磁碟。PostgreSQL 11 起有 **Parallel Hash Join**：多個 worker 共同建一張共享雜湊表。

5.6.4　三種 join 的選擇

	Nested Loop	Sort-Merge	Hash
條件類型	任意	等值、範圍	僅等值
最佳情境	外層極小且內層有索引	輸入已排序 / 需有序輸出 / 記憶體很小	兩表都大、無序、記憶體足夠放較小表
最差情境	兩表都大且無索引	需要排序大表	資料傾斜（某鍵極多）、記憶體不足多輪溢出
管線化	是	排序後是	build 階段阻塞
典型工作負載	OLTP 點查詢關聯	大範圍報表、已叢集資料	OLAP 大表關聯

5.7　排序與聚合

5.7.1　外部合併排序

資料放不進記憶體時：**第一趟**每次讀 B 頁排序後寫出一個 run（共 $\lceil M/B \rceil$ 個）；**之後每趟**以 $B - 1$ 路合併。總趟數 $1 + \lceil \log_{B-1} \lceil M/B \rceil \rceil$，每趟讀寫 $2M$ 頁。以 $B = 1000$ 頁（8 MB）為

例，兩趟就能排序 $B(B-1) \approx 10^6$ 頁（8 GB）。**Top-N heapsort**（`ORDER BY ... LIMIT 10`）只需維護大小 N 的堆，一趟掃描即可，PostgreSQL 會自動選用。

5.7.2 聚合

- **排序聚合**（GroupAggregate）：先依 `GROUP BY` 鍵排序，相同鍵相鄰，一趟掃過即可；輸出天然有序。
- **雜湊聚合**（HashAggregate）：以分組鍵為雜湊表鍵，維護每組的累計狀態；不需排序、通常較快，但群組數太多時雜湊表會超過 `work_mem`——PostgreSQL 13 起會分區溢出到磁碟（之前會直接超用記憶體）。
- `DISTINCT` 本質上是沒有聚合函數的分組；`COUNT(DISTINCT x)` 需要額外的去重步驟。

5.8 其他運算子與平行執行

- **Materialize**：把子節點結果暫存，供 nested loop 內層重複掃描。
- **Append / MergeAppend**：`UNION ALL`、分割表的子表掃描。
- **Limit**、**Unique**、**WindowAgg**、**SetOp**、**SubPlan**（相關子查詢，每筆外層執行一次——常是效能殺手）。
- **Gather / Gather Merge**：Volcano 論文中的 **exchange** 運算子——它把平行處理封裝成一個普通的迭代器，子樹由多個 worker 各跑一份（Parallel Seq Scan 以頁為單位分工、Parallel Hash Join 共享雜湊表），結果由 leader 收集。`max_parallel_workers_per_gather` 控制 worker 數。

5.9 讀懂 EXPLAIN ANALYZE

程式碼 5.2: 一個典型的兩表 join 計畫

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT c.name, count(*)
FROM   customers c JOIN orders o ON o.customer_id = c.id
WHERE  c.country = 'TW'
GROUP  BY c.name;
```

```
HashAggregate  (cost=4512.30..4515.30 rows=300 width=40) (actual time=38.1..38.2 rows=287 loops=1)
  Group Key: c.name
  Batches: 1  Memory Usage: 61kB
  Buffers: shared hit=1730
  ->  Hash Join  (cost=27.80..4388.60 rows=24740 width=32) (actual time=0.4..31.9 rows=24611 loops
    ↪ =1)
        Hash Cond: (o.customer_id = c.id)
        Buffers: shared hit=1730
        ->  Seq Scan on orders o  (cost=0.00..3670.00 rows=200000 width=8) (actual time=0.01..12.5
    ↪ rows=200000 loops=1)
              Buffers: shared hit=1670
```

```
        ->  Hash  (cost=24.00..24.00 rows=304 width=32) (actual time=0.3..0.3 rows=287 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 27kB
              ->  Seq Scan on customers c  (cost=0.00..24.00 rows=304 width=32) (actual time
    ↪ =0.01..0.2 rows=287 loops=1)
                    Filter: (country = 'TW'::text)
                    Rows Removed by Filter: 2213
                    Buffers: shared hit=60
Planning Time: 0.35 ms
Execution Time: 38.4 ms
```

閱讀方法：**由內而外、由下而上**。(1) 掃描 `customers` 並過濾國家，287 筆進入 Hash 建表（build 端是小表，正確）；(2) 掃描 `orders` 20 萬筆 probe；(3) 雜湊聚合 287 組，全部在記憶體完成；(4) `cost=startup..total` 是估計、`actual time` 是毫秒實測、`rows` 估計 vs 實際接近表示統計良好；`loops` 大於 1 時 actual time 與 rows 是每次的平均，要乘上 loops。`Buffers: shared hit` 全部命中，沒有磁碟 I/O。

MiniDB 連結：v1 的執行器

v1 以 Volcano 模型實作 `SeqScanExecutor`、`FilterExecutor`、`ProjectionExecutor`、`NestedLoopJoinExecutor`、`InsertExecutor`；每個都只有 `Init()/Next()`。計畫先用手寫的「規則式」轉換（有索引且等值條件就用 IndexScan），成本式最佳化留到 v6。v2 加入 `IndexScanExecutor`，v6 再加 `HashJoinExecutor`、`SortExecutor`、`HashAggregateExecutor`。

本章練習

1. R：1000 頁、10 萬筆；S：500 頁、4 萬筆；$B = 100$。計算簡單 NLJ、block NLJ（兩種外層選擇）、sort-merge（兩表都需排序）、Grace hash join 的 I/O 成本，並指出最佳者。
2. 為什麼 hash join 的 build 端應選較小的表？若最佳化器因統計過時把 build/probe 選反，會發生什麼？
3. Volcano 模型下 `SELECT ... ORDER BY x LIMIT 10` 為何仍需讀完整個輸入？哪些運算子是 pipeline breaker？
4. 解釋向量化為何能改善分支預測與快取命中；一個 `WHERE a > 10` 的向量化 Filter 在內部如何表示「哪些列通過」（提示：selection vector）？
5. 外部合併排序：$M = 10^6$ 頁、$B = 101$ 頁，需要幾趟？總 I/O 多少？若 B 加倍呢？
6. （動手）對本章的 join 查詢分別 `SET enable_hashjoin = off`、`SET enable_mergejoin = off` 強迫改用其他演算法，比較 `EXPLAIN ANALYZE` 的實際時間；再把 `work_mem` 調到 `64kB` 觀察 `Batches` 的變化。

延伸閱讀與資源

- CMU 15-445 Fall 2025, Lecture #11 *Sorting & Aggregation*、#12 *Join Algorithms*、#13–14 *Query Execution I/II*；Project #3 *Query Execution*：https://15445.courses.cs.cmu.edu/fall2025/project3/
- G. Graefe, *Volcano—An Extensible and Parallel Query Evaluation System*, IEEE TKDE 1994：https://15799.courses.cs.cmu.edu/spring2025/papers/04-volcano/graefe-icde1994.pdf
- P. Boncz, M. Zukowski, N. Nes, *MonetDB/X100: Hyper-Pipelining Query Execution*, CIDR 2005。
- T. Neumann, *Efficiently Compiling Efficient Query Plans for Modern Hardware*, VLDB 2011。
- PostgreSQL 文件 Ch. 50 *Overview of PostgreSQL Internals*（https://www.postgresql.org/docs/current/overview.html）、§14.1 *Using EXPLAIN*（https://www.postgresql.org/docs/current/using-explain.html）、§15 *Parallel Query*。
- Hironobu Suzuki, *The Internals of PostgreSQL*, Ch. 3 *Query Processing*：https://www.interdb.jp/pg/pgsql03.html
- PostgreSQL 原始碼 `src/backend/executor/README`（執行器架構、tuple table slot、JIT）。
- DuckDB 文件 *Execution Format*（向量格式）：https://duckdb.org/docs/current/internals/vector

第 6 章　查詢最佳化

本章學習目標

- 區分啟發式（規則式）重寫與成本式最佳化，知道兩者在最佳化器中各自的位置。
- 列舉常見的邏輯重寫：述詞下推、投影下推、子查詢去相關、view 合併、外部 join 簡化。
- 說明統計資訊（n_distinct、MCV、直方圖、相關性）如何轉成選擇率與基數估計，以及獨立性假設為何常常失靈。
- 用 PostgreSQL 的成本常數手算一個 Seq Scan 與 Index Scan 的成本。
- 解釋 Selinger 動態規劃的 join 排序法與「interesting order」，以及為何表數太多時要改用啟發式/遺傳演算法。
- 診斷估計錯誤的計畫，並知道對應的修正手段。

6.1　為什麼需要最佳化器

SQL 是宣告式語言：使用者說「要什麼」，系統決定「怎麼做」。同一個查詢的不同實體計畫，執行時間可以相差好幾個數量級——例如 10 表 join 的左深計畫有 $10! \approx 3.6 \times 10^6$ 種排列，再乘上每個 join 的演算法與存取路徑選擇。最佳化器的目標並非找到*最佳*計畫（那本身就是 NP-hard），而是在可接受的時間內*避開糟糕的*計畫。

現代最佳化器幾乎都源自 IBM System R（Selinger et al. 1979）的架構：

1. **邏輯重寫**：套用一定有益的代數等價轉換（啟發式）。
2. **成本式搜尋**：對候選實體計畫用統計資訊估計成本，選最便宜者。

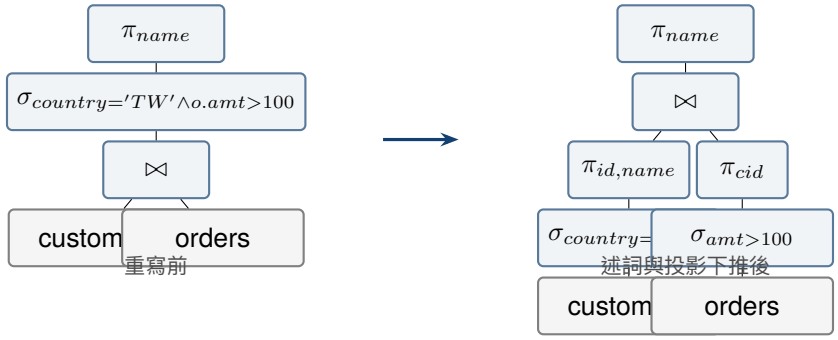
6.2　邏輯重寫：啟發式最佳化

這些轉換不需要統計，因為幾乎永遠有益（第 1 章的等價規則是其理論基礎）：

- **述詞下推（predicate pushdown）**：把 σ 盡量推向葉節點，越早過濾越好；也推進 view、子查詢與分割表（**partition pruning**）。
- **投影下推（projection pushdown）**：只往上傳需要的欄位，縮小中間結果——對行式儲存尤其重要。
- **常數摺疊與表達式簡化**：`WHERE 1 = 1`、`x + 0`、`NOT (a > 5)` → `a <= 5`；`x IN (1)` → `x = 1`。
- **子查詢去相關（decorrelation / unnesting）**：把 `WHERE EXISTS (SELECT ... WHERE o.cid`

= c.id）改寫成 semi-join，避免「每筆外層執行一次子查詢」。PostgreSQL 會把 IN/EXISTS 子查詢拉平成 semi-join，NOT EXISTS 成 anti-join；但相關的純量子查詢多半仍以 SubPlan 執行。

- **View 合併與子查詢拉平**：把 view 定義內嵌進外層查詢，讓述詞可以穿越。
- **外部 join 簡化**：LEFT JOIN ... WHERE right.col IS NOT NULL 等價於 inner join；也允許更多的重排。
- **Join 消除**：若只用到左表欄位、右表以主鍵/唯一鍵關聯，LEFT JOIN 可直接刪除。
- **OR 展開**：a = 1 OR b = 2 可改成兩個索引掃描的 bitmap OR。



6.3 統計資訊

成本式最佳化的燃料是**統計資訊**。PostgreSQL 的 ANALYZE（手動或由 autovacuum 觸發）隨機抽樣表中的列（約 300× default_statistics_target 列，預設 target 為 100，即 30,000 列），為每個欄位計算並存入 pg_statistic（可讀 view 為 pg_stats）：

統計量	意義與用途
null_frac	NULL 的比例；IS NULL 的選擇率。
n_distinct	相異值個數；負值表示佔總列數的比例（如 -1 代表每列唯一）。等值條件的預設選擇率 $1/n_{distinct}$。
most_common_vals / freqs	最常見值（MCV）清單與各自頻率（最多 target 個）；資料傾斜時的救星。
histogram_bounds	排除 MCV 後、以等高（equi-depth）方式切成 target 個 bucket 的邊界值；用於範圍條件。
correlation	欄位值順序與實體儲存順序的相關係數（$-1 \sim 1$）；接近 ± 1 表示 index scan 幾乎是循序 I/O。
avg_width	平均欄寬，用於估計中間結果大小（決定 hash join 是否放得進 work_mem）。

表層級的 reltuples（列數）與 relpages（頁數）存在 pg_class。ALTER TABLE t ALTER COLUMN c SET STATISTICS 1000 可對特定欄位提高 target；CREATE STATISTICS（**擴充統計**，PostgreSQL 10+）可對多個欄位收集 ndistinct、函數相依（dependencies）與多欄 MCV，修正下一節將看到的獨立性假設問題。

6.4　選擇率與基數估計

選擇率（selectivity） $sel(p) \in [0,1]$ 是述詞 p 讓多少比例的列通過；**基數（cardinality）**是估計的輸出列數。教科書（與 `selfuncs.c`）的基本公式：

述詞	估計	說明
$a = c$	MCV 中有 c：其頻率；否則 $(1 - \sum \text{MCV freq})/(n_{distinct} - \lvert MCV \rvert)$	無統計時預設 0.005。
$a < c$	由直方圖內插：c 落在第 k 個 bucket 的位置	加上 MCV 中 $< c$ 的頻率。
$p_1 \wedge p_2$	$sel(p_1) \cdot sel(p_2)$	**獨立性假設**；相關欄位會嚴重低估。
$p_1 \vee p_2$	$sel(p_1) + sel(p_2) - sel(p_1)\,sel(p_2)$	同樣假設獨立。
$\neg p$	$1 - sel(p)$	
$R \bowtie_{R.a=S.b} S$	$\lvert R \rvert \cdot \lvert S \rvert \,/\, \max(V(R,a), V(S,b))$	**包含假設**：值域較小者包含於較大者；外鍵 join 最準確。
`LIKE 'abc%'`	轉成範圍 `>= 'abc' AND < 'abd'`	`'%abc'` 無法用直方圖，預設低選擇率。

三個常見失靈

- **相關欄位**：`city = 'Taipei' AND zip LIKE '10%'` 兩者高度相關，相乘後低估數個數量級，導致選了 nested loop 而實際跑了幾百萬次。解法：`CREATE STATISTICS s (dependencies, mcv) ON city, zip FROM t`。
- **述詞中的函數或運算**：`WHERE lower(email) = ...`、`WHERE created_at + interval '1 day' > now()` 沒有統計可用，只能套預設值。解法：建表達式索引（`ANALYZE` 會為它收集統計）或改寫為可 sargable 的形式。
- **過時的統計**：大量載入後未 `ANALYZE`，`reltuples` 仍是舊值。解法：載入後手動 `ANALYZE`；調低 `autovacuum_analyze_scale_factor`。

Leis et al. (VLDB 2015) 用 IMDB 真實資料（Join Order Benchmark）證明：主流資料庫的基數估計在多表 join 後常偏差 $10^2 \sim 10^4$ 倍，而估計錯誤比成本模型的誤差對計畫品質影響大得多。

6.5　成本模型

PostgreSQL 的成本是**無單位的相對值**，以一次循序讀一頁為 1.0，各常數可在 `postgresql.conf` 調整：

參數	預設	意義
`seq_page_cost`	1.0	循序讀一頁
`random_page_cost`	4.0	隨機讀一頁；SSD 上常調為 1.1–1.5，否則最佳化器過度偏好 Seq Scan
`cpu_tuple_cost`	0.01	處理一筆 tuple
`cpu_index_tuple_cost`	0.005	處理一筆索引項目
`cpu_operator_cost`	0.0025	執行一個運算子或函數
`effective_cache_size`	4GB	估計 OS + shared_buffers 可快取的量；只影響 index scan 的成本估計，不配置記憶體
`work_mem`	4MB	每個排序/雜湊運算子可用的記憶體；超過即溢出

Seq Scan 成本：

$$\text{cost} = \text{relpages} \times \text{seq_page_cost} + \text{reltuples} \times (\text{cpu_tuple_cost} + \text{cpu_operator_cost} \times n_{quals})$$

以 `orders`（1670 頁、20 萬列、一個過濾條件）為例：$1670 + 200000 \times (0.01 + 0.0025) = 4170$。

Index Scan 成本更複雜：索引部分為 B+Tree 高度（隨機 I/O）加上掃過的葉頁與索引項目；heap 部分依**相關性**在「全隨機」（每筆一次 `random_page_cost`）與「全循序」之間內插，並用 Mackert–Lohman 公式考慮重複存取同一頁時的快取效果（受 `effective_cache_size` 影響）。每個計畫都有 **startup cost**（輸出第一筆前的成本）與 **total cost**；`LIMIT` 查詢偏好低 startup 的計畫（如 index scan 而非 sort）。

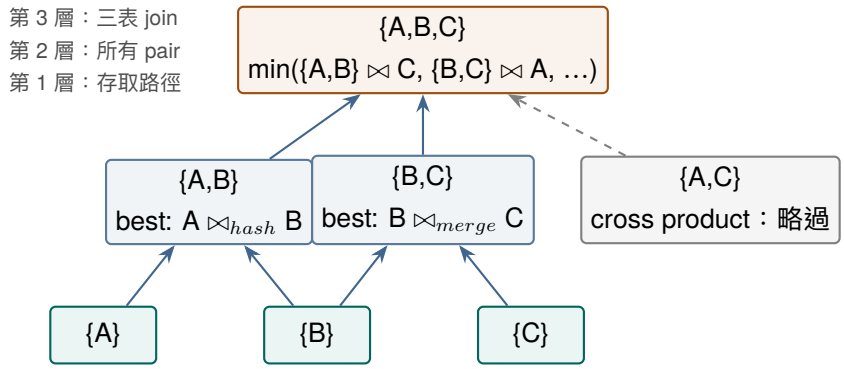
6.6 Join 排序

6.6.1 搜尋空間

n 表 join 的**左深樹**（每個 join 的右側都是基本表）有 $n!$ 種順序；允許**濃密樹（bushy）**則是 $\frac{(2(n-1))!}{(n-1)!}$ 種——10 表就超過 1.7×10^{10}。System R 只考慮左深樹（適合管線化、內層可用索引），現代系統多數也考慮濃密樹但設上限。

6.6.2 Selinger 動態規劃

1. 對每個單表，列舉所有存取路徑（seq scan、各索引），記下*每種輸出順序*的最便宜者。
2. 對每個大小為 2 的表集合，嘗試所有 join 順序與演算法，只保留每個集合（每種「interesting order」）的最佳子計畫——**最佳子結構**：全域最佳計畫的子計畫也必須是該子集合的最佳。
3. 重複至大小 n。複雜度 $O(3^n)$（濃密）或 $O(n \cdot 2^n)$（左深），遠小於 $n!$。
4. **Interesting order**：排序過的子計畫雖然較貴，但可能讓上層的 merge join 或 `ORDER BY` 免費——所以不能只留一個。
5. 預設避免 **cross product**（沒有 join 條件的組合），除非無其他選擇。



6.6.3　表數太多時

PostgreSQL 在 FROM 中的項目數 ≤ `join_collapse_limit`（預設 8）時做完整的動態規劃；超過 `geqo_threshold`（預設 12）則改用**遺傳演算法（GEQO）**隨機搜尋。其他系統採**貪婪**（每次加入成本最低的表）或 **Cascades** 框架（Graefe 1995；SQL Server、CockroachDB、Calcite 使用）——由上而下、以記憶化的轉換規則搜尋，並支援剪枝。Neumann 與 Radke（2018）展示了針對數百表 join 的自適應演算法。

6.7　實體計畫的其他決策

- **避免排序**：若某個索引的順序就是 `ORDER BY` 或 `GROUP BY` 的順序，走索引就能省掉排序節點；interesting order 的實際應用。
- **平行度**：表大於 `min_parallel_table_scan_size`（8 MB）時才考慮 Gather；worker 數依表大小以對數成長。
- **CTE**：PostgreSQL 12 起 `WITH` 子查詢預設可被內嵌（inline），除非寫 `MATERIALIZED` 或被引用多次。
- **計畫快取**：prepared statement 前 5 次用**自訂計畫**（帶入實際參數），之後若**泛型計畫**（generic plan）成本不明顯更高就改用它——參數分佈傾斜時會踩雷（parameter sniffing），可用 `plan_cache_mode` 強制。
- **分割表**：規劃期與執行期（run-time）的 partition pruning；partition-wise join/aggregate。

6.8　診斷與修正錯誤的計畫

診斷流程

1. `EXPLAIN (ANALYZE, BUFFERS)`：找 `rows=` 估計與實際差距最大的節點（以 loops 乘算）。那就是問題源頭，因為錯誤會往上層放大。
2. 看估計錯的節點是哪種述詞：相關欄位 → 擴充統計；函數 → 表達式索引；`LIKE '%x'`、JSON → 考慮 GIN/trigram；過時 → `ANALYZE`。
3. 檢查 `Batches > 1`、`Sort Method: external merge Disk`——調 `work_mem`（可只在 session 內調）。

4. 隨機 I/O 的設定是否合理：SSD 上 `random_page_cost = 1.1`；`effective_cache_size` 設為實際記憶體的 50–75%。
5. 最後手段：改寫查詢（拆成兩段、`OFFSET 0` 阻擋拉平、`MATERIALIZED`）或擴充套件 `pg_hint_plan`；`enable_*` 參數只應用於實驗，不要在生產環境永久關閉。
6. 用 `auto_explain` 自動記錄慢查詢的實際計畫、`pg_stat_statements` 找最耗資源的查詢樣板。

PostgreSQL 的 planner 原始碼

`src/backend/optimizer/README` 是必讀文件。主要流程：`planner()` → `subquery_planner()`（拉平子查詢、簡化表達式、`prep/`）→ `query_planner()` → `make_one_rel()`：先 `set_base_rel_pathlists()` 為每個基本表產生 Path（`path/allpaths.c`、`path/indxpath.c`），再 `standard_join_search()`（`path/joinrels.c`）動態規劃，`geqo/` 是遺傳演算法。成本在 `path/costsize.c`，選擇率在 `utils/adt/selfuncs.c`。最後 `plan/createplan.c` 把最便宜的 Path 轉為 Plan 樹。

MiniDB 連結：v6 成本式最佳化器

v6 把 v1 的規則式規劃器升級：(1) `ANALYZE` 指令對每欄收集 n_distinct 與 100-bucket 等高直方圖；(2) 以本章公式估計選擇率；(3) 成本模型採 PostgreSQL 的五個常數；(4) 以左深 Selinger DP 排 join 順序（表數 ≤ 6），每個子集保留最佳與每個 interesting order 的最佳；(5) 實作 `EXPLAIN` 印出估計與實際列數。這會讓你親身體會「估計錯誤如何沿樹向上放大」。

本章練習

1. 表 `t` 有 100 萬列、`status` 欄 MCV 為 {active: 0.9, closed: 0.08}、n_distinct = 5。估計 `WHERE status = 'active'`、`WHERE status = 'pending'`、`WHERE status <> 'active'` 的列數。
2. 承上，`created_at` 的直方圖有 100 個 bucket，`'2025-01-01'` 落在第 73 個 bucket 的 40% 處。估計 `WHERE created_at >= '2025-01-01'` 與加上 `AND status = 'active'` 後的列數；指出所用假設。
3. 以本章的常數計算：表 5000 頁、50 萬列、索引高度 3、條件選擇率 0.001、相關性 0（全隨機），Index Scan 的粗略成本是多少？與 Seq Scan 相比何者勝出？若 `random_page_cost = 1.1` 呢？
4. 4 表鏈狀 join $A \bowtie B \bowtie C \bowtie D$（只有相鄰表有 join 條件），列出 Selinger DP 第 2 層與第 3 層需要評估的子集合（排除 cross product）。
5. 為什麼 `WHERE date_trunc('day', ts) = '2025-01-01'` 既無法使用 `ts` 上的索引、估計又不準？寫出等價且可 sargable 的條件。
6. （動手）建立兩個高度相關的欄位並載入資料，比較 `CREATE STATISTICS` 前後 `EXPLAIN` 的估計列數與選用的 join 演算法。

延伸閱讀與資源

- P. G. Selinger et al., *Access Path Selection in a Relational Database Management System*, SIGMOD 1979（成本式最佳化的開山之作）：`https://courses.cs.duke.edu/compsci516/cps216/spring03/papers/selinger-etal-1979.pdf`
- S. Chaudhuri, *An Overview of Query Optimization in Relational Systems*, PODS 1998。
- V. Leis et al., *How Good Are Query Optimizers, Really?*, VLDB 2015（Join Order Benchmark）：`https://www.vldb.org/pvldb/vol9/p204-leis.pdf`
- G. Graefe, *The Cascades Framework for Query Optimization*, IEEE Data Eng. Bull. 1995。
- CMU 15-445 Fall 2025 Lecture #15 *Query Planning & Optimization*；CMU 15-799 *Special Topics in Database Systems: Query Optimization*（進階課）。
- PostgreSQL 文件 Ch. 14 *Performance Tips*、Ch. 72 *How the Planner Uses Statistics*（`https://www.postgresql.org/docs/current/planner-stats-details.html`）、§20.7 *Query Planning* 參數。
- Hironobu Suzuki, *The Internals of PostgreSQL*, Ch. 3（含成本計算的逐步示範）。
- PostgreSQL 原始碼 `src/backend/optimizer/README`、`path/costsize.c`、`utils/adt/selfuncs.c`。
- Markus Winand, *Use The Index, Luke!*（索引與最佳化器的實務指南）：`https://use-the-index-luke.com/`

第四篇

交易、並行控制與復原

第 7 章　交易與 ACID

本章學習目標

- 精確定義 ACID 四個性質，並知道每個性質由資料庫的哪個機制負責（以及哪個其實是應用程式的責任）。
- 以衝突可序列化與優先圖判定一個排程是否正確。
- 辨認 dirty read、non-repeatable read、phantom、lost update、write skew、read skew 等異常，並寫出重現它們的交錯時序。
- 說明 ANSI 四個隔離等級的定義、Berenson 等人對它的批評，以及 PostgreSQL / MySQL / Oracle 各自的實際行為。
- 理解 snapshot isolation 為何不是可序列化，以及如何處理 serialization failure。

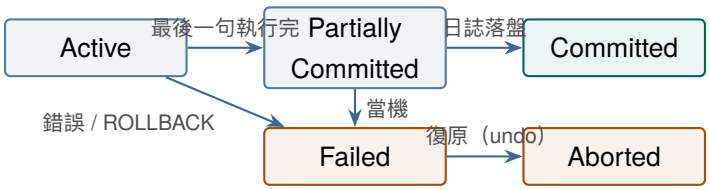
7.1　交易是什麼

交易（transaction）是一組被視為單一工作單位的資料庫操作：要嘛全部生效，要嘛全部不生效。它是資料庫對應用程式最重要的抽象——讓開發者在面對當機與並行時，仍能以「一次只有我在操作、而且操作不會做到一半」的方式思考。

程式碼 7.1: 交易控制語法（PostgreSQL）

```
BEGIN;                                  -- 或 START TRANSACTION
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
SAVEPOINT before_fee;                   -- 部分回滾點
UPDATE accounts SET balance = balance - 5 WHERE id = 1;    -- 手續費
ROLLBACK TO SAVEPOINT before_fee;       -- 取消手續費，但保留第一筆更新
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;                                 -- 或 ROLLBACK 放棄全部
```

沒有明確 `BEGIN` 時，多數系統處於 **autocommit**：每個語句自成一筆交易。



7.2　ACID

Atomicity（原子性）

交易中的操作「全有或全無」。由**日誌**（undo 資訊）或 **shadow paging** 實現；PostgreSQL 靠 MVCC——中止的交易所寫的版本只是被標為無效，不需要物理回復（第 9 章）。

Consistency（一致性）

交易把資料庫從一個合法狀態帶到另一個合法狀態。資料庫只能強制它*知道*的規則（主鍵、外鍵、CHECK、觸發器）；業務不變量（「帳戶總額守恆」）是**應用程式的責任**——C 是 ACID 中唯一不完全由資料庫保證的字母。

Isolation（隔離性）

並行執行的交易彼此看不到對方的中間狀態，結果等價於某種序列執行。由**並行控制**（鎖、MVCC、OCC）實現，是本篇的主軸。

Durability（持久性）

一旦 commit，即使當機、斷電也不會遺失。由 **WAL + fsync**（第 8 章）實現；PostgreSQL 的 `synchronous_commit = off` 允許以「最多遺失最後幾百毫秒已回報成功的交易」換取吞吐。

7.3　排程與可序列化

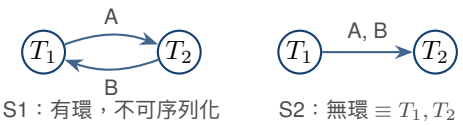
排程（schedule）是多筆交易操作的交錯順序。**序列排程**是交易一個接一個執行；一個排程若效果等價於某個序列排程，稱為**可序列化（serializable）**——這是隔離性的黃金標準。

兩個操作**衝突**的條件：來自不同交易、存取同一資料項、且至少一個是寫入。三種衝突：

- **R–W**（unrepeatable read）：T_1 讀 A 後 T_2 寫 A。
- **W–R**（dirty read）：T_1 寫 A 後 T_2 讀 A。
- **W–W**（lost update）：T_1 寫 A 後 T_2 寫 A。

衝突可序列化：能透過交換*不衝突*的相鄰操作變成序列排程。判定方法：畫**優先圖（precedence graph）**——節點是交易，若 T_i 的某操作與 T_j 的操作衝突且 T_i 在前，畫 $T_i \to T_j$。**無環 ⇔ 衝突可序列化**，拓撲排序就是等價的序列順序。

```
S1: R1(A) W1(A) R2(A) W2(A) R2(B) W2(B) R1(B) W1(B)
S2: R1(A) W1(A) R2(A) W2(A) R1(B) W1(B) R2(B) W2(B)
```



S1：有環，不可序列化　　S2：無環 $\equiv T_1, T_2$

View serializability 是更寬鬆的定義（允許 blind write 的特例），但判定是 NP-complete，實務系統都以衝突可序列化為目標。

此外還要考慮**可復原性**：若 T_2 讀了 T_1 的寫入，T_2 必須在 T_1 之後 commit（**recoverable**）；否則 T_1 中止時 T_2 已 commit 的結果建立在不存在的資料上。更強的 **strict** 排程（不讀也不覆寫未 commit 的資料）避免**連鎖中止（cascading abort）**並讓 undo 變簡單——這是多數系統實際採用的。

7.4 異常現象

以下以兩筆交易 T_1、T_2 的時間軸呈現（時間由上而下）。

異常	時序	後果
Dirty read	T_1: `UPDATE x = 10`（未 commit） T_2: 讀到 `x = 10` T_1: `ROLLBACK`	T_2 看到從未存在的值。
Non-repeatable read	T_1: 讀 x 得 5 T_2: `UPDATE x = 7; COMMIT` T_1: 再讀 x 得 7	同一交易內兩次讀同一列結果不同。
Phantom	T_1: `SELECT count(*) WHERE age > 30` 得 10 T_2: `INSERT (age = 40); COMMIT` T_1: 同一查詢得 11	集合層級的不可重複讀：多出「幽靈」列。
Lost update	T_1: 讀 `x = 10` T_2: 讀 `x = 10` T_1: `x = 10 + 1; COMMIT` T_2: `x = 10 + 1; COMMIT`	兩次加一只加了一次。
Write skew	約束：至少一位醫師值班。目前 Alice、Bob 都值班。 T_1: 查到 2 人值班，Alice 下班 T_2: 查到 2 人值班，Bob 下班 兩者 commit	各自檢查都通過，合起來違反約束；兩者寫不同列，故 W–W 偵測抓不到。
Read skew	T_1: 讀帳戶 A = 100 T_2: A → 50、B → 150 轉帳並 commit T_1: 讀 B = 150	T_1 看到總額 250，不一致的快照。

7.5 隔離等級

ANSI/ISO SQL-92 用「禁止哪些異常」定義四個等級：

等級	Dirty read	Non-repeatable	Phantom	Write skew
Read Uncommitted	可能	可能	可能	可能
Read Committed	✗	可能	可能	可能
Repeatable Read	✗	✗	可能	可能
Serializable	✗	✗	✗	✗

ANSI 定義的問題：Berenson et al. 1995

A Critique of ANSI SQL Isolation Levels 指出：ANSI 的定義是以鎖為心智模型寫的、對異常的描述有歧義、而且漏掉了 lost update、write skew 等現象。該文定義了 **Snapshot Isolation（SI）**：每筆交易看到開始時刻的一致快照，寫入時若發現同一列已被並行交易改過則中止（first-committer-wins）。SI 沒有 dirty/non-repeatable/phantom read，卻**允許 write skew**——所以它不是 serializable，但 Oracle 把 SI 稱為「SERIALIZABLE」，PostgreSQL 9.1 之前亦然。

各系統的實際行為：

系統	預設等級	Repeatable Read 的實作	Serializable 的實作
PostgreSQL	Read Committed（每個語句取新快照）	Snapshot Isolation：交易級快照；無 phantom；更新衝突報 `40001`	**SSI**（Serializable Snapshot Isolation，9.1+）：偵測危險結構並中止，真正可序列化
MySQL InnoDB	Repeatable Read	MVCC 快照 + **gap/next-key lock** 防 phantom（僅對鎖定讀）	全部讀加 S 鎖（2PL）
Oracle	Read Committed	不提供（直接跳到 Serializable）	其實是 SI，允許 write skew
SQL Server	Read Committed（鎖式；可開 RCSI）	鎖式	鎖式 2PL；另有 SNAPSHOT 等級
CockroachDB / Spanner	Serializable	—	預設即可序列化（分別為 SSI 變體與 2PL + TrueTime）

PostgreSQL 把 Read Uncommitted 當成 Read Committed 處理——MVCC 架構下根本不會讀到未 commit 的版本。

7.6 Snapshot Isolation 與 write skew

程式碼 7.2: 在 PostgreSQL REPEATABLE READ 下重現 write skew

```
-- 兩個 session 同時執行
BEGIN ISOLATION LEVEL REPEATABLE READ;
SELECT count(*) FROM doctors WHERE on_call;          -- 兩邊都得到 2
UPDATE doctors SET on_call = false WHERE name = 'Alice';   -- session 1
UPDATE doctors SET on_call = false WHERE name = 'Bob';     -- session 2
COMMIT;                                              -- 兩邊都成功：0 人值班！
```

兩筆交易寫的是不同列，first-committer-wins 不會發生衝突。防範方法：(1) 改用 `SERIALIZABLE`——SSI 會在其中一方 commit 時偵測到 rw-conflict 的危險結構並回報 `40001 serialization_failure`；(2) 把「隱含的衝突」**物化**：`SELECT ... FOR UPDATE` 鎖住所讀的值

班列、或在一張 `shift` 表上做 `UPDATE` 製造 W–W 衝突；(3) 用約束（如對「至少一人值班」建立觸發器或 exclusion constraint）。

應用程式必須處理 serialization failure

在 REPEATABLE READ 與 SERIALIZABLE 下，PostgreSQL 可能以 SQLSTATE `40001`（或 `40P01` deadlock）中止任何交易，**這不是錯誤，而是協定的一部分**：應用程式應該 重試整筆交易（含重新讀取），通常搭配指數退避並限制次數。框架常提供 `@Transactional(retry)` 類的工具。同時避免**長交易**：它們阻礙 VACUUM 回收舊版本（第 9 章），也放大衝突機率；`idle_in_transaction_session_timeout` 可以自動砍掉忘記 commit 的連線。

7.7　並行控制的三大家族

接下來三章分別深入三種實現隔離性的策略，這裡先建立地圖：

策略	機制	假設	代表
悲觀：兩階段鎖（2PL）	讀前取 S 鎖、寫前取 X 鎖，commit 時一併釋放；需偵測死結	衝突常見	SQL Server、DB2、InnoDB（寫入）
樂觀：OCC	讀/寫在私有空間進行，commit 時驗證讀集沒被改過，失敗則重做	衝突罕見、交易短	記憶體資料庫（Hekaton、Silo）、FoundationDB
多版本：MVCC	寫入產生新版本，讀者看快照永不阻塞寫者；搭配 2PL 或 SSI 處理寫衝突	讀多寫少	PostgreSQL、Oracle、InnoDB（讀取）、CockroachDB
時間戳排序（T/O）	依交易時間戳決定順序，違反者中止	理論基礎；分散式常用	Spanner（TrueTime）、部分 NewSQL

PostgreSQL 的交易識別

每筆寫入交易取得一個 32 位元遞增的 **XID**（`txid_current()` / `pg_current_xact_id()`）；唯讀交易不消耗 XID（只用虛擬 XID），這就是為何「唯讀為主」的負載較不會遇到 XID 回繞。交易狀態（in progress / committed / aborted）存在 **commit log（pg_xact，舊稱 clog）**，每個 XID 兩個 bit。Savepoint 以**子交易（subtransaction）**實作，各有自己的 XID；`pg_subtrans` 記錄父子關係。超過 64 個子交易的交易會讓快照的子交易快取溢出，造成明顯的效能退化——ORM 的「每個語句一個 savepoint」設定要特別留意。

MiniDB 連結：v4 交易管理器

v4 加入 `TransactionManager`：`Begin()` 配置遞增的 txn id、`Commit()` 寫 commit 記錄並等 WAL 落盤、`Abort()` 依 undo 資訊回復。初版隔離以表級 2PL 實現（簡單但正確），v5 換成 MVCC 快照。測試中應包含本章每一種異常的重現腳本，確認在你宣稱的隔離等級下它們不會發生。

本章練習

1. 排程 $R_1(A)\,R_2(B)\,W_1(B)\,W_2(A)\,C_1\,C_2$：畫優先圖，判定是否衝突可序列化；是否 recoverable？是否 strict？
2. 寫出一個 view serializable 但非 conflict serializable 的排程（提示：blind write）。
3. 在 Read Committed 下，`UPDATE accounts SET balance = balance - 100 WHERE id = 1` 兩個並行執行會不會 lost update？若改成應用程式先 `SELECT` 再 `UPDATE ... SET balance = :計算值` 呢？說明 PostgreSQL 在 RC 下對已被更新列的「重新檢查」行為。
4. 為 write skew 例子設計三種不同的防護方式並比較它們對並行度的影響。
5. 為什麼 PostgreSQL 的 Repeatable Read 沒有 phantom，而 ANSI 允許 RR 有 phantom？這代表 PostgreSQL 的 RR 比 ANSI 的 RR 強還是弱？
6.（動手）開兩個 `psql`，在 RC、RR、SERIALIZABLE 下分別重現表格中的每一種異常，記錄哪些等級擋住了哪些異常。

延伸閱讀與資源

- H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil, *A Critique of ANSI SQL Isolation Levels*, SIGMOD 1995：`https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-95-51.pdf`
- PostgreSQL 文件 Ch. 13 *Concurrency Control*（隔離等級、SSI 行為、明確鎖定）：`https://www.postgresql.org/docs/current/mvcc.html`
- PostgreSQL Wiki *SSI*（含值班醫師等範例）：`https://wiki.postgresql.org/wiki/SSI`
- M. Kleppmann, *Hermitage: Testing the "I" in ACID*（各資料庫隔離等級實測腳本）：`https://github.com/ept/hermitage`
- Jepsen, *Consistency Models*（隔離等級與一致性模型的統一地圖）：`https://jepsen.io/consistency`
- CMU 15-445 Fall 2025 Lecture #16 *Concurrency Control Theory*。
- P. Bernstein, V. Hadzilacos, N. Goodman, *Concurrency Control and Recovery in Database Systems*（免費 PDF）：`https://www.microsoft.com/en-us/research/people/philbe/book/`
- J. Gray, A. Reuter, *Transaction Processing: Concepts and Techniques*, 1992（經典巨著）。
- M. Kleppmann, *Designing Data-Intensive Applications*, 2nd ed., Ch. 8 *Transactions*。

第 8 章　預寫式日誌（WAL）

本章學習目標

- 說明為何「先寫日誌、再寫資料頁」能同時提供原子性與持久性，並與 steal/no-force 緩衝策略連結。
- 陳述 WAL 的兩條規則與 LSN、pageLSN、flushedLSN 之間的不變式。
- 比較實體、邏輯、生理（physiological）三種日誌記錄，知道 PostgreSQL 選了哪一種、為什麼。
- 解釋 full-page write 如何解決 torn page，以及 checkpoint 如何限制復原時間並讓日誌可回收。
- 讀懂 PostgreSQL 的 LSN 格式、`pg_wal` 目錄與 `pg_waldump` 輸出，並正確設定 `synchronous_commit` 等參數。

8.1　為什麼需要日誌

第 4 章提到緩衝池的 **steal**（未 commit 的髒頁可先寫回）與 **no-force**（commit 時不必寫回資料頁）策略能把隨機寫入延後、合併，但也帶來兩個問題：

- steal ⇒ 當機時磁碟上可能有未 *commit* 交易的修改，需要 **undo**。
- no-force ⇒ 當機時已 *commit* 交易的修改可能還沒落盤，需要 **redo**。

解法是把「發生了什麼修改」**循序**追加到一個獨立的日誌檔並 fsync——循序寫比隨機寫快一到兩個數量級，而且一次 fsync 就能讓一批交易持久。資料頁本身可以晚點、合併後再寫。這就是 **預寫式日誌**（Write-Ahead Logging, WAL）。

8.2　WAL 協定

每筆日誌記錄有一個單調遞增的 **日誌序號**（Log Sequence Number, LSN）（本質上是它在日誌串流中的位元組位移）。每個資料頁的標頭記錄 **pageLSN**：最後一次修改這頁的日誌記錄 LSN。系統維護 **flushedLSN**：已持久化到磁碟的日誌最大 LSN。

WAL 的兩條規則

1. **寫頁前**：把資料頁 P 寫回磁碟之前，必須先確保 pageLSN(P) $\le$ flushedLSN——描述這頁修改的所有日誌已落盤。（保證可 undo / 可 redo）
2. **Commit 前**：回報交易成功之前，該交易的所有日誌記錄（包含 commit 記錄）必須已

落盤。（保證持久性）



8.3　日誌記錄的內容

類型	記錄什麼	取捨
實體（physical）	頁面某位移前後的位元組	重做簡單、冪等；但記錄大，且無法描述跨頁操作
邏輯（logical）	高層操作（「對 `users` 插入這筆 tuple」）	記錄小；但重做時結果必須完全一致（不確定性、並行交錯難以重現），復原慢
生理（physiological）	以頁為單位、頁內以邏輯方式描述（「在 page 7 的 slot 3 插入 tuple」）	兩者折衷，主流選擇（ARIES、PostgreSQL、InnoDB）

典型記錄欄位：LSN、交易 ID、**prevLSN**（同一交易前一筆記錄，串成該交易的回溯鏈）、記錄類型、受影響的頁與位移、修改前映像（before image，供 undo）、修改後映像（after image，供 redo）。PostgreSQL 因為 MVCC 不需要 undo 日誌，**只記 redo 資訊**。

PostgreSQL 的 WAL 記錄

每筆記錄以 `XLogRecord` 標頭開頭（`xl_tot_len` 總長、`xl_xid`、`xl_prev`、`xl_info`、`xl_rmid`、`xl_crc`），後接零到多個 block reference（受影響的頁，可含 full-page image）與資源管理器專屬資料。**資源管理器（rmgr）**依子系統劃分：`Heap`（`INSERT/UPDATE/DELETE/HOT_UPDATE/LOCK`）、`Heap2`（`PRUNE/VACUUM/MULTI_INSERT`）、`Btree`、`Transaction`（`COMMIT/ABORT`）、`XLOG`（`CHECKPOINT_ONLINE/FPI`）、`Standby`、`Gin/Gist/Hash`……每個 rmgr 提供 `rm_redo` 函數，復原時依 `xl_rmid` 分派。

8.4　Full-Page Write 與 torn page

第 2 章提到：8 KB 頁寫到只保證 4 KB 原子性的磁碟上，斷電可能造成**半新半舊**的頁。生理日誌無法對一個損壞的頁做 redo（頁內結構已不可信）。PostgreSQL 的解法是 **full-page write（FPW）**：每個 checkpoint 之後，第一次修改某頁時把整頁映像寫進 WAL；復原時直接用該映像覆蓋，再套用後續的增量記錄。代價是 checkpoint 剛結束時 WAL 量暴增（可用 `wal_`

`compression = on` 壓縮 FPI）。InnoDB 則用 **doublewrite buffer**：先把頁寫到一塊連續區域、再寫到真正位置，兩者至少有一份完整。

8.5　Checkpoint

日誌不能無限成長，復原也不能從創世紀開始重播。**Checkpoint** 把緩衝池的髒頁寫回磁碟，並記下一個 **redo point**：復原只需從此點開始。簡單的做法是暫停所有交易、寫光髒頁——不可接受；實務上用 **fuzzy checkpoint**：記下開始時的 LSN，在正常服務的同時慢慢把當時的髒頁寫出，完成後才把 redo point 寫入控制檔（第 11 章 ARIES 會精確處理其正確性）。

PostgreSQL checkpoint 參數

- 觸發：`checkpoint_timeout`（預設 5 min）或 WAL 量達到 `max_wal_size`（預設 1 GB）；也可手動 `CHECKPOINT`。
- `checkpoint_completion_target`（預設 0.9）：checkpointer 把寫出攤平在 90% 的間隔內，避免 I/O 尖峰。
- redo point 與最新 checkpoint 位置寫入 `global/pg_control`（`pg_controldata` 可查）。
- 日誌出現 `checkpoints are occurring too frequently` 表示 `max_wal_size` 太小——每次 checkpoint 都重置 FPW，太頻繁反而放大 WAL 量。生產環境常設 `max_wal_size` 數 GB、`checkpoint_timeout` 15–30 min，以復原時間換取平穩的 I/O。

8.6　Commit 的成本：fsync 與 group commit

一次 `fsync` 在 NVMe 上約 20–100 µs、在雲端網路磁碟上可能數毫秒——若每筆交易都 fsync，吞吐上限就是 1/fsync 延遲。**Group commit**：多個同時 commit 的交易共用一次 fsync。PostgreSQL 的做法是：commit 的 backend 若發現有人正在 flush，就等待那次 flush 完成（`WALWriteLock`）；`commit_delay` 可刻意多等一點以湊更多交易。另有背景的 **walwriter** 定期（`wal_writer_delay`，200 ms）寫出 WAL 緩衝區。

`synchronous_commit`	**回報成功的時機**
`off`	WAL 進入記憶體緩衝即回報；當機最多遺失約 3× `wal_writer_delay` 的交易，但**不會損壞資料**（與 `fsync = off` 不同）。
`local`	本機 WAL fsync 完成。
`remote_write`	加上：同步備援已收到並寫入 OS（未 fsync）。
`on`（預設）	加上：同步備援已 fsync。
`remote_apply`	加上：同步備援已重播，讀備援也看得到。

fsync 的陷阱

- `fsync = off` 會讓 checkpoint 也不等資料落盤，當機後資料庫可能**損壞且無法啟動**——只能用於可隨時重建的測試環境。

- `wal_sync_method` 在 Linux 預設 `fdatasync`；某些檔案系統 / 虛擬磁碟的 fsync 行為不可靠（「fsyncgate」事件：PostgreSQL 12 之前曾在 fsync 回報錯誤後重試，但 Linux 已清除髒頁狀態，導致靜默遺失；現在遇到 fsync 失敗會直接 PANIC）。
- 消費級 SSD 可能在回報寫入完成後資料仍在揮發性快取；有電容保護（PLP）的企業級硬碟才能安全啟用寫入快取。

8.7 PostgreSQL 的 WAL 實體佈局

- WAL 位於 `$PGDATA/pg_wal/`，切成 16 MB 的**段檔（segment）**（`initdb --wal-segsize` 可改），每段內再分 8 KB 頁。
- 檔名 `000000010000000000000003` 由三個 8 位十六進位數組成：**timeline**（PITR / failover 後遞增）、**log id**、**segment no**。
- LSN 以 高 32 位/低 32 位 的十六進位表示，如 `0/16B3748`；`pg_current_wal_lsn()` 取得目前位置、`pg_walfile_name(lsn)` 對應段檔、`pg_wal_lsn_diff()` 計算位元組差（監控複寫延遲的基礎）。
- `wal_level`：`minimal`（只夠當機復原）、`replica`（預設；支援串流複寫與 PITR）、`logical`（加入邏輯解碼所需資訊）。
- **封存（archiving）**：`archive_mode = on` 加 `archive_command`（或 `archive_library`）把寫滿的段檔複製到外部儲存，搭配基礎備份即可做 **Point-in-Time Recovery**。
- 段檔會被**回收重用**（rename 為未來的檔名）而非刪除，介於 `min_wal_size` 與 `max_wal_size` 之間。

程式碼 8.1: 用 pg_waldump 觀察一筆 UPDATE 產生的 WAL

```
$ pg_waldump -p $PGDATA/pg_wal -s 0/16B3748 -n 4
rmgr: Heap        len (rec/tot):     54/   218, tx:        748, lsn: 0/016B3748, prev 0/016B3710,
      desc: HOT_UPDATE old_xmax: 748, old_off: 2, old_infobits: [], flags: 0x20, new_xmax: 0,
    ↪ new_off: 3,
      blkref #0: rel 1663/16384/16385 blk 0 FPW
rmgr: Transaction len (rec/tot):     34/    34, tx:        748, lsn: 0/016B3828, prev 0/016B3748,
      desc: COMMIT 2026-10-08 10:12:31.448521 CST
```

第一筆是 heap 的 HOT update（帶 full-page image，因為是 checkpoint 後首次修改該頁），第二筆是 commit 記錄；`prev` 串起同一交易的記錄。可加 `--stats` 看各 rmgr 的 WAL 量分佈，是診斷「為什麼 WAL 這麼多」的第一步。

8.8 其他系統的日誌設計

- **InnoDB**：redo log 是固定大小的**環狀**檔（`innodb_redo_log_capacity`），寫滿前必須完成 checkpoint；undo 資訊存在 undo tablespace（供 MVCC 與回滾，第 9 章）；`innodb_flush_log_at_trx_commit` 對應 `synchronous_commit`。
- **SQLite WAL 模式**：修改的整頁追加到 `-wal` 檔，`-shm` 檔是頁號到 WAL 位置的索引；check-

point 把頁複製回主檔。讀者讀 WAL 中的最新版本，寫者不阻塞讀者——一種以頁為粒度的 MVCC。

- **RocksDB**：每個 memtable 對應一個 WAL，memtable flush 成 SSTable 後 WAL 即可刪除（第 12 章）。

8.9　WAL 的其他用途

WAL 是資料庫中「所有變更的全序」，因此順理成章地成為：**實體複寫**的傳輸內容（第 14 章 streaming replication）、**PITR** 的素材、**邏輯解碼**（logical decoding）與 CDC（Debezium 把 WAL 轉成 Kafka 事件）的來源。理解 WAL 等於同時理解了復原、複寫與變更擷取。

程式碼 8.2: MiniDB v4 的 LogManager 介面

```
class LogManager {
 public:
  // 追加記錄到記憶體緩衝區，回傳其 LSN；不保證落盤
  lsn_t Append(const LogRecord &rec);
  // 阻塞直到 flushed_lsn_ >= lsn（commit 與寫髒頁前呼叫）
  void Flush(lsn_t lsn);
  lsn_t FlushedLsn() const { return flushed_lsn_.load(); }
 private:
  void FlusherThread();   // 背景執行緒：定期或被喚醒時 write + fsync，實作 group commit
  std::vector<char> buffer_;  std::mutex mu_;  std::condition_variable cv_;
  std::atomic<lsn_t> next_lsn_{0}, flushed_lsn_{0};
};
```

MiniDB 連結：v4 WAL

v4 的記錄類型：`BEGIN`、`COMMIT`、`ABORT`、`INSERT`、`DELETE`、`UPDATE`（含 before/after image）、`NEW_PAGE`、`CHECKPOINT`。`BufferPoolManager::FlushPage()` 先呼叫 `log_manager->Flush(page->GetLSN())` 以遵守規則 1；`TransactionManager::Commit()` 寫 COMMIT 後 `Flush` 以遵守規則 2。加入「寫到一半 kill -9」的測試：重啟後所有已回報成功的交易都必須存在，未 commit 的都必須消失。

本章練習

1. 若只採用 no-steal/force 策略，還需要日誌嗎？需要 undo 還是 redo？這種策略的效能問題在哪？
2. 說明「pageLSN ≤ flushedLSN 才能寫頁」為何同時保護 undo 與 redo 的正確性。若違反會出現什麼具體錯誤情境？
3. 一個交易修改了同一頁 1000 次：實體日誌、生理日誌、full-page write 各會產生多少資料？為何 PostgreSQL 只在 checkpoint 後首次修改時寫 FPI？
4. 解釋為什麼 `synchronous_commit = off` 不會損壞資料庫，而 `fsync = off` 會。

5. 系統每秒 20,000 筆小交易、fsync 延遲 0.5 ms。沒有 group commit 時最大吞吐是多少？有 group commit 時瓶頸會移到哪裡？
6. （動手）執行一批 `UPDATE` 前後用 `pg_current_wal_lsn()` 與 `pg_wal_lsn_diff()` 計算產生的 WAL 量；然後 `CHECKPOINT` 後再做一次，比較 `pg_waldump --stats` 中 FPI 的比例。

延伸閱讀與資源

- PostgreSQL 文件 Ch. 28 *Reliability and the Write-Ahead Log*（含 WAL 設定、非同步 commit、WAL 內部）：`https://www.postgresql.org/docs/current/wal.html`
- PostgreSQL 文件 *pg_waldump*：`https://www.postgresql.org/docs/current/pgwaldump.html`；§20.5 *Write Ahead Log* 參數。
- Hironobu Suzuki, *The Internals of PostgreSQL*, Ch. 9 *Write Ahead Logging*：`https://www.interdb.jp/pg/pgsql09.html`
- PostgreSQL 原始碼 `src/backend/access/transam/README`（交易、WAL、復原的設計筆記）、`xlog.c`、`xloginsert.c`、`src/include/access/xlogrecord.h`。
- CMU 15-445 Fall 2025 Lecture #19 *Logging Schemes*。
- SQLite, *Write-Ahead Logging*：`https://www.sqlite.org/wal.html`
- MySQL 文件 *InnoDB Redo Log* 與 *Doublewrite Buffer*。
- T. Munro, *fsync() errors* 討論與 PostgreSQL 12 的 PANIC 行為：`https://wiki.postgresql.org/wiki/Fsync_Errors`

第 9 章　多版本並行控制（MVCC）

本章學習目標

- 說明 MVCC 的三個設計維度：版本如何儲存、讀者如何決定可見性、舊版本如何回收。
- 比較 PostgreSQL 的 append-only 版本鏈與 InnoDB/Oracle 的 undo 差異儲存，以及各自的代價。
- 讀懂 PostgreSQL tuple 的 `xmin/xmax/ctid/infomask`，並手動套用可見性規則。
- 解釋快照的結構、Read Committed 與 Repeatable Read 的差別、更新衝突時的行為。
- 說明 HOT、heap pruning、VACUUM 的各階段與 autovacuum 參數，診斷膨脹(bloat)的成因。
- 理解 32 位元 XID 回繞的風險、凍結機制與相關監控。
- 描述 SSI 如何用 rw-antidependency 偵測非可序列化執行。

9.1　核心想法

鎖式並行控制（第 10 章）讓讀者與寫者互相阻塞。**多版本並行控制**（Multi-Version Concurrency Control, MVCC）改變規則：**寫入不覆蓋舊值，而是產生新版本**；每筆交易讀取一個固定時間點的**快照**。於是：

- 讀者永遠不阻塞寫者、寫者也不阻塞讀者（寫者之間仍需協調）。
- 唯讀交易可以看到一致的「時間切片」而不需要任何鎖——報表查詢與 OLTP 共存。
- 支援「時間旅行」查詢（Oracle flashback、CockroachDB `AS OF SYSTEM TIME`）。

代價是：空間（舊版本）、回收舊版本的工作（garbage collection）、以及索引必須處理多個版本。任何 MVCC 實作都要回答三個問題：**版本存哪裡？讀者如何判定可見？舊版本何時回收？**

9.2　版本儲存的三種設計

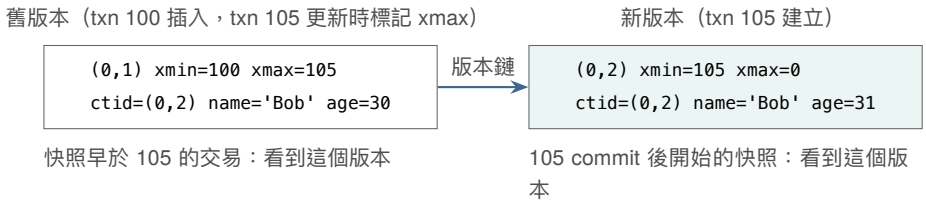
設計	做法	優點	代價
Append-only（PostgreSQL）	新版本寫進同一個 heap，舊版本原地保留，以 `ctid` 串成版本鏈（舊 → 新）	回滾免費（版本只是作廢）、讀舊版本不需重建、實作簡單	`UPDATE` = 插入 + 標記刪除，表與索引膨脹；需要 VACUUM；每個索引都要指向新版本（除非 HOT）
Delta / undo（InnoDB、Oracle、SQL Server）	主表只存最新版本，舊值的差異寫進 undo log / rollback segment / version store，讀舊版本時沿 undo 鏈重建	主表緊湊、索引只指向一個位置、讀最新版本最快	讀舊快照需重建（長交易下很慢）、回滾要套用 undo、undo 空間需回收（purge）、Oracle 的 `ORA-01555 snapshot too old`
Time-travel 表	舊版本複製到另一張「歷史表」	主表緊湊	寫入成本加倍；較少系統採用（早期 SQL Server、HANA 部分）

版本鏈的方向也有講究：PostgreSQL 由舊指向新（oldest-to-newest），讀者可能要走過多個已死版本才到最新；InnoDB 由新指向舊（newest-to-oldest），讀最新版本是 $O(1)$。Wu 等人（VLDB 2017）對記憶體資料庫的實測顯示：沒有單一最佳設計，瓶頸取決於工作負載與版本回收策略。

9.3　PostgreSQL 的 tuple 版本

回顧第 2 章的 `HeapTupleHeaderData`，MVCC 相關欄位：

欄位	意義
`t_xmin`	建立這個版本的交易 XID（INSERT 或 UPDATE 的新版本）。
`t_xmax`	刪除 / 更新這個版本的交易 XID；0 表示仍是活的。也用於**列鎖**（`FOR UPDATE` 把自己的 XID 寫進 xmax 並設 `HEAP_XMAX_LOCK_ONLY`）。
`t_cid`	同一交易內的命令序號（cmin/cmax 合用一個欄位，衝突時用 combo CID 表），讓同一交易內「先 INSERT 後 SELECT」看得到自己的寫入、但 `INSERT ... SELECT` 不會看到自己正在插入的列。
`t_ctid`	自己或**新版本**的位置 (block, offset)；被 UPDATE 後指向新版本，形成版本鏈。
`t_infomask`	**hint bits**：`HEAP_XMIN_COMMITTED`、`HEAP_XMIN_INVALID`、`HEAP_XMAX_COMMITTED`、`HEAP_XMAX_INVALID`、`HEAP_XMAX_LOCK_ONLY`、`HEAP_XMAX_IS_MULTI`、`HEAP_UPDATED`、`HEAP_HOT_UPDATED`、`HEAP_ONLY_TUPLE` ……



`DELETE` 只是設定 `xmax`；`UPDATE` 是「舊版本設 xmax + 插入新版本」。這就是為什麼 PostgreSQL 的回滾幾乎零成本——中止的交易留下的版本因為 `xmin` 對應的交易狀態為 aborted 而自動不可見。

9.4　快照

快照（snapshot）描述「在取得快照的瞬間，哪些交易已經完成」。PostgreSQL 的快照由三部分組成，`pg_current_snapshot()` 以 `xmin:xmax:xip_list` 顯示，例如 `100:106:101,104`：

- **xmin**（100）：所有 XID $<$ xmin 的交易都已結束（committed 或 aborted）。
- **xmax**（106）：第一個尚未配發的 XID；所有 XID $\geq$ xmax 的交易在快照時還不存在，一律不可見。
- **xip**（101、104）：介於兩者之間、快照時*仍在進行中*的交易；它們的寫入不可見（即使後來 commit 了）。

XID 102、103、105 已結束，可見與否取決於它們 commit 還是 abort——查 **commit log（pg_xact）**，並把結果以 hint bit 寫回 tuple 省下以後的查詢。

Read Committed：每個*語句*開始時取新快照；**Repeatable Read / Serializable**：整筆交易在第一個語句時取一次快照。這就是兩個等級全部的差別。

9.5　可見性規則

對快照 S 與 tuple t（簡化版，完整版見 `heapam_visibility.c` 的 `HeapTupleSatisfiesMVCC`）：

程式碼 9.1: MVCC 可見性判定（簡化）

```
visible(t, S):
  -- 第一步：建立者是否「對我而言已 commit」
  if t.xmin == my_xid:
      if t.cmin >= my_current_cid: return false      -- 本交易稍後的命令才插入
  elif t.xmin in S.xip or t.xmin >= S.xmax: return false   -- 快照時仍在進行 / 尚未存在
  elif status(t.xmin) != COMMITTED: return false     -- 已中止
  -- 第二步：刪除者是否「對我而言已 commit」
  if t.xmax == 0 or XMAX_INVALID or LOCK_ONLY: return true   -- 沒人刪它
  if t.xmax == my_xid: return t.cmax >= my_current_cid       -- 自己刪的：之前的命令看不到
  if t.xmax in S.xip or t.xmax >= S.xmax: return true        -- 刪除者快照時未完成：仍看得到
  return status(t.xmax) != COMMITTED                         -- 刪除者 commit 了就看不到
```

兩個關鍵洞見：(1) 可見性只靠 tuple 本身的欄位、快照與 clog 就能決定，不需要鎖；(2)「比較 XID 大小」在 32 位元回繞下必須用**模運算**（`TransactionIdPrecedes`），稍後會談。

9.6　寫入衝突

兩筆交易更新同一列時，第二個到達者發現該列的 `xmax` 屬於進行中的交易，必須**等待**對方結束（這是 MVCC 中唯一會阻塞的情境之一）：

- 對方 abort：接手，正常更新。
- 對方 commit，**Read Committed**：PostgreSQL 重新取得該列的最新版本，重新評估 `WHERE` 條件（`EvalPlanQual` 機制）；仍符合就更新新版本。這就是為什麼 RC 下 `UPDATE ... SET n = n + 1` 不會 lost update。
- 對方 commit，**Repeatable Read / Serializable**：丟出 `ERROR: could not serialize access due to concurrent update`（`40001`），因為快照中看不到新版本，繼續更新會違反快照語意——**first-updater-wins**。

`SELECT ... FOR UPDATE / FOR SHARE / FOR NO KEY UPDATE / FOR KEY SHARE` 也透過 xmax 實作列鎖；多個交易共享鎖定同一列時 xmax 指向一個 **MultiXact** ID（`pg_multixact`）。

9.7　HOT 與 heap pruning

Append-only 最大的痛點：每次 UPDATE 都要在每一個索引插入指向新版本的項目，即使更新的欄位沒被索引。PostgreSQL 8.3 引入 **Heap-Only Tuple（HOT）**：若 (a) 沒有任何索引欄位被改動、(b) 新版本放得進同一頁，則新版本不進索引；索引仍指向舊版本的 line pointer，讀者沿頁內的 `ctid` 鏈走到最新版本。舊版本死掉後，**heap pruning**（任何讀寫該頁的 backend 都可能順手做）把死版本的空間回收、line pointer 改為 **redirect**，整個過程不需要碰索引也不需要 VACUUM。

讓 HOT 發生

- 不要索引頻繁更新的欄位（如 `updated_at`、計數器）；`pg_stat_user_tables.n_tup_hot_upd / n_tup_upd` 顯示 HOT 比例。
- 對更新頻繁的表設 `fillfactor = 70`–`90`，在頁內預留空間給新版本。
- PostgreSQL 16 起，若被更新的索引欄位是 summarizing 索引（BRIN）也不會破壞 HOT。

9.8　VACUUM

HOT 處理不了跨頁的死版本與索引項目，這是 **VACUUM** 的工作。一次（lazy）VACUUM 的流程：

1. 依 Visibility Map 跳過 all-visible 的頁，掃描其餘 heap 頁：prune、收集死 tuple 的 TID（存於 `maintenance_work_mem`，PostgreSQL 17 起改用 radix tree 大幅省記憶體）、凍結夠老的

tuple、更新 FSM/VM。

2. 掃描每個索引，刪除指向這些 TID 的項目（index vacuum；大表最耗時的一步）。
3. 回到 heap，把死 tuple 的 line pointer 設為 `LP_UNUSED`。
4. 若檔案尾端全空，截斷檔案歸還 OS；更新 `pg_class.reltuples/relpages`、`relfrozenxid`。

VACUUM **不會**把空間還給作業系統（除尾端外），只是讓同表的新資料可以重用；`VACUUM FULL` 會重寫整張表（需要 `ACCESS EXCLUSIVE` 鎖與兩倍空間），線上環境改用 `pg_repack` 擴充套件。

autovacuum

autovacuum launcher 定期（`autovacuum_naptime`，1 min）檢查每張表：死 tuple 數 > `autovacuum_vacuum_threshold`（50）+ `autovacuum_vacuum_scale_factor`（0.2）× 列數就啟動 worker（最多 `autovacuum_max_workers`，3 個）。對十億列的表，20% 是兩億個死 tuple——大表應個別把 scale factor 調到 0.01 或更低，並降低 `autovacuum_vacuum_cost_delay`（預設 2 ms）讓它跑得更快。PostgreSQL 13 起純插入的表也會觸發（`autovacuum_vacuum_insert_scale_factor`），以便設定 VM 讓 index-only scan 有效。

什麼會阻止 VACUUM 回收

VACUUM 只能回收**對所有活動快照都不可見**的版本，由系統中最老的 `xmin` 水位決定。常見的「水位卡住」來源：長交易（含 `idle in transaction`）、閒置的 **replication slot**（備援或 CDC 消費者斷線）、未完成的 `PREPARE TRANSACTION`、以及 `hot_standby_feedback = on` 時備援上的長查詢。查 `pg_stat_activity.backend_xmin`、`pg_replication_slots.xmin`、`pg_prepared_xacts`。Uber 2016 年「從 PostgreSQL 搬到 MySQL」的文章正是圍繞 UPDATE 放大與 VACUUM 行為的批評，之後多個版本（HOT 改進、index deduplication、bottom-up index deletion）即為回應。

9.9　XID 回繞與凍結

XID 是 32 位元，約 42 億個，PostgreSQL 把它當成**環**：對任一 XID，比它「小」的 21 億個是過去、「大」的 21 億個是未來。若某個舊 tuple 的 `xmin` 放著不管超過 21 億個交易，它會突然變成「未來」而對所有人**不可見**——資料看似消失。

解法是**凍結（freeze）**：VACUUM 把夠老的 tuple 標為 `HEAP_XMIN_FROZEN`（舊版本是把 xmin 改成常數 2），表示「對所有人永遠可見、不再比較」。

參數	預設	意義
`vacuum_freeze_min_age`	5000 萬	tuple 的 xmin 年齡超過此值，VACUUM 時順便凍結
`vacuum_freeze_table_age`	1.5 億	表的 `relfrozenxid` 年齡超過此值，VACUUM 改為**積極模式**（不跳過 all-visible 頁，只跳過 all-frozen 頁）
`autovacuum_freeze_max_age`	2 億	強制啟動 anti-wraparound autovacuum，**即使 autovacuum 關閉**
（硬限制）	約 21 億	剩餘 4000 萬時開始警告；剩餘 300 萬時資料庫拒絕配發新 XID，必須以單使用者模式 VACUUM（PostgreSQL 14+ 的 failsafe 模式會在剩 16 億前跳過索引清理全力凍結）

程式碼 9.2: 監控回繞風險

```
SELECT datname, age(datfrozenxid) AS xid_age,
       round(age(datfrozenxid) / 2147483647.0 * 100, 2) AS pct_to_wraparound
FROM   pg_database ORDER BY 2 DESC;
SELECT relname, age(relfrozenxid) FROM pg_class WHERE relkind = 'r' ORDER BY 2 DESC LIMIT 10;
```

MultiXact ID 也是 32 位元、也需要凍結（`autovacuum_multixact_freeze_max_age`）。PostgreSQL 內部已有 64 位元的 `xid8`（含 epoch），但 heap tuple 標頭仍是 32 位元；把頁面標頭改為帶 epoch 的 64-bit XID 是社群多年進行中的工作。

9.10　Serializable Snapshot Isolation

PostgreSQL 9.1 的 SSI（Ports & Grittner 2012，基於 Cahill 2008）讓 SI 變成真正可序列化，而且**不使用阻塞鎖**。核心理論：

- **rw-antidependency**（$T_1 \xrightarrow{rw} T_2$）：T_1 讀了某資料，T_2 之後寫了它的新版本（T_1 看不到）。T_1 在序列化順序中必須排在 T_2 之前。
- 定理（Fekete et al. 2005）：SI 下所有非可序列化的執行，其序列化圖中都存在一條**危險結構**：$T_1 \xrightarrow{rw} T_2 \xrightarrow{rw} T_3$ 兩條連續的 rw-antidependency，且 T_3 最早 commit。
- SSI 追蹤 rw 邊（用 **SIREAD lock**——不阻塞任何人的「述詞鎖」，記錄誰讀過哪些 tuple / 頁 / 表），一旦偵測到危險結構就中止其中一方（`40001`）。會有少量**誤判**（結構存在但實際上可序列化），但不會漏判。



值班醫師例子：T_1 讀值班表、T_2 寫 Alice 下班；T_2 讀值班表、T_1 寫 Bob 下班
形成 $T_1 \xrightarrow{rw} T_2 \xrightarrow{rw} T_1$ 的環，SSI 中止後 commit 的那一方

實務要點：SIREAD lock 存在共享記憶體，`max_pred_locks_per_transaction` 等參數控

制上限（超過會升級為頁 / 表粒度，增加誤判）；純報表交易宣告 `BEGIN ISOLATION LEVEL SERIALIZABLE READ ONLY DEFERRABLE` 可等到一個安全快照後完全不參與偵測；序列化失敗率高時表示工作負載衝突多，應考慮重新設計而非只是重試。

9.11　其他系統速覽

- **InnoDB**：每筆列有隱藏欄 `DB_TRX_ID`、`DB_ROLL_PTR`（指向 undo 記錄）；**read view** 等同快照；**purge thread** 回收不再需要的 undo；`SHOW ENGINE INNODB STATUS` 的 history list length 是「undo 積壓」指標；長交易會讓 undo tablespace 暴漲，而非表膨脹。
- **Oracle**：undo tablespace 大小與 `undo_retention` 決定能回溯多久；超過就 `ORA-01555`。
- **SQL Server**：RCSI/Snapshot 把舊版本放進 tempdb 的 version store（2019 起可用 ADR 的持久版本存放區）。
- **CockroachDB / TiDB**：MVCC 建在 LSM 鍵值層上——鍵為 `user_key@timestamp`，舊版本由 GC TTL（預設 4 小時 / 10 分鐘）與 compaction 清除（第 12、16 章）。

MiniDB 連結：v5 MVCC

v5 把 tuple 標頭加上 `xmin`、`xmax`、`next_version`，`TransactionManager` 維護活動交易集合並產生快照；`SeqScanExecutor` 以本章的可見性函數過濾；UPDATE 走 append-only；RR 等級下寫衝突回報 serialization error。再實作一個最簡單的 `VACUUM`：找出 xmax 小於所有活動快照 xmin 且已 commit 的版本並回收 slot。挑戰題：加入 HOT 鏈與 heap pruning，量測對索引寫入量的影響。

本章練習

1. 快照 `100:106:101,104`，tuple A（xmin 103, xmax 0）、B（xmin 101, xmax 0）、C（xmin 99, xmax 105）、D（xmin 102, xmax 104）。clog：103、105 committed，102 aborted。逐一判定可見性並說明理由。
2. 為什麼 PostgreSQL 在 RC 下 `UPDATE` 要重新評估 `WHERE`，而在 RR 下直接報錯？若 RR 也「重新評估後更新」會違反什麼？
3. 一張表每秒更新 1 萬次、每次都改到被索引的欄位。估算一天產生的死 tuple 與索引項目數；討論 HOT 為何幫不上忙，以及你會怎麼改 schema。
4. 說明「long-running transaction 造成 bloat」在 PostgreSQL 與 InnoDB 分別表現在哪裡，哪個對線上查詢的影響更直接？
5. 一個 10 TB 的表 `relfrozenxid` 年齡已達 1.9 億，但目前的 VACUUM 速度每小時只能處理 200 GB。會發生什麼？你有哪些選項？
6. 畫出 write skew 例子的序列化圖，標出兩條 rw 邊；若 T_1 改用 `SELECT ... FOR UPDATE`，圖會如何改變？
7. （動手）用 `pageinspect` 的 `heap_page_items` 觀察一列經過 INSERT、UPDATE、DELETE、VACUUM 後 `t_xmin/t_xmax/t_ctid/t_infomask` 的變化，並用 `heap_`

`tuple_infomask_flags()` 解讀旗標。

延伸閱讀與資源

- Hironobu Suzuki, *The Internals of PostgreSQL*, Ch. 5 *Concurrency Control*（可見性、快照、SSI 的完整推導）：`https://www.interdb.jp/pg/pgsql05.html`；Ch. 6 *Vacuum Processing*：`https://www.interdb.jp/pg/pgsql06.html`
- PostgreSQL 文件 Ch. 13 *Concurrency Control*；§24.1 *Routine Vacuuming*（含 *Preventing Transaction ID Wraparound Failures*）：`https://www.postgresql.org/docs/current/routine-vacuuming.html`
- D. R. K. Ports, K. Grittner, *Serializable Snapshot Isolation in PostgreSQL*, VLDB 2012：`https://drkp.net/papers/ssi-vldb12.pdf`
- M. Cahill, U. Röhm, A. Fekete, *Serializable Isolation for Snapshot Databases*, SIGMOD 2008。
- Y. Wu et al., *An Empirical Evaluation of In-Memory Multi-Version Concurrency Control*, VLDB 2017：`https://www.vldb.org/pvldb/vol10/p781-wu.pdf`
- CMU 15-445 Fall 2025 Lecture #18 *Multi-Version Concurrency Control*。
- PostgreSQL 原始碼：`src/backend/access/heap/README.HOT`、`README.tuplock`、`heapam_visibility.c`、`src/backend/storage/lmgr/README-SSI`、`src/backend/access/heap/vacuumlazy.c`。
- E. Klitzke, *Why Uber Engineering Switched from Postgres to MySQL*, 2016：`https://www.uber.com/blog/postgres-to-mysql-migration/`（搭配 PostgreSQL 社群的回應閱讀）。
- MySQL 文件 *InnoDB Multi-Versioning*：`https://dev.mysql.com/doc/refman/8.4/en/innodb-multi-versioning.html`

第 10 章　鎖與並行控制

本章學習目標

- 區分 lock（交易層級、邏輯物件）與 latch（執行緒層級、記憶體結構），以及 PostgreSQL 的三層鎖：heavyweight lock、LWLock、spinlock。
- 寫出 S/X/IS/IX/SIX 的相容矩陣，解釋多粒度鎖與意向鎖的目的。
- 證明兩階段鎖（2PL）保證衝突可序列化，並說明為何實務使用 strict 2PL。
- 以等待圖偵測死結、以 wait-die/wound-wait 預防死結，並知道 PostgreSQL 的 `deadlock_timeout` 機制。
- 熟記 PostgreSQL 的 8 種表級鎖與 4 種列級鎖的衝突關係，避免 DDL 鎖住整個系統的事故。
- 用 `pg_locks`、`pg_blocking_pids()` 診斷鎖等待。

10.1　Lock 與 Latch

	Lock（交易鎖）	Latch（閂鎖）
保護對象	邏輯物件：表、列、索引鍵範圍、述詞	記憶體結構：buffer frame、雜湊表桶、B+Tree 頁
持有者	交易	執行緒 / 程序，一小段程式碼
持有時間	到交易結束（2PL）	關鍵區段（微秒）
模式	多種（S、X、意向鎖……）	通常只有 read/write
死結	偵測或預防	以編碼紀律避免（固定順序、不等待）
PostgreSQL 名稱	heavyweight lock（`pg_locks` 可見）	LWLock、spinlock、buffer pin

第 3、4 章的 latch crabbing、buffer header 鎖都屬於 latch；本章的主角是交易鎖。

10.2　鎖模式與多粒度鎖

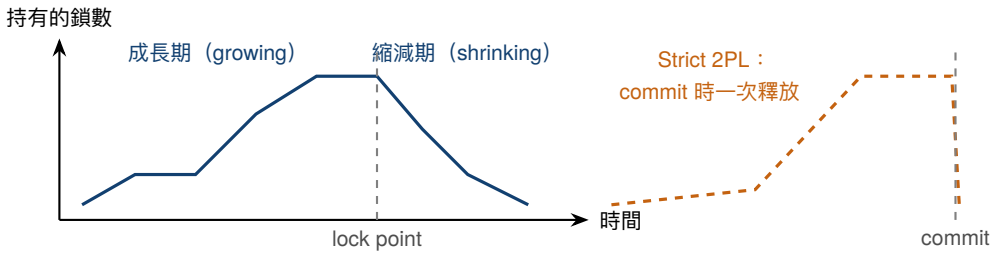
最基本的兩種模式：**共享鎖 S**（讀）與**排他鎖 X**（寫）。若鎖的粒度是整張表，並行度太低；若只有列鎖，一個全表掃描要取百萬個鎖，鎖表爆炸。Gray 等人（1976）提出**多粒度鎖**與**意向鎖（intention lock）**：在上層節點（資料庫 → 表 → 頁 → 列）宣告「我打算在下層取某種鎖」。

已持有 \ 請求	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✗	✗	✗
S	✓	✗	✓	✗	✗
SIX	✓	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

- **IS / IX**：打算在下層取 S / X。要鎖某列 X 之前，先在表上取 IX、在頁上取 IX。
- **SIX** = S + IX：讀整張表，同時更新其中少數列。
- 全表掃描在表上取 S，就與任何 IX（有人在改某列）衝突——正確地阻止了讀到半更新狀態；而兩個 IX 相容，改不同列的交易互不干擾（列層級再判斷）。
- **鎖升級（escalation）**：列鎖太多時換成表鎖以省資源（SQL Server、DB2 會；PostgreSQL 不會，因為列鎖存在 tuple 裡不佔鎖表）。

10.3　兩階段鎖

兩階段鎖（2PL）：每筆交易分為**成長期**（只取鎖、不放鎖）與**縮減期**（只放鎖、不取鎖）。一旦釋放任何鎖，就不能再取得新鎖。



定理：2PL 產生的排程都是衝突可序列化的。證明概要：令每筆交易的 **lock point** 為它取得最後一個鎖的時刻；若 $T_i \to T_j$ 有衝突邊，則 T_i 必須先釋放該資料的鎖、T_j 才能取得，故 $\text{lockpoint}(T_i) < \text{lockpoint}(T_j)$。若優先圖有環，會推出 lock point 的嚴格遞增環，矛盾。

但基本 2PL 允許在縮減期釋放 X 鎖後被別人讀到尚未 *commit* 的值 ⇒ 連鎖中止。**Strict 2PL**：X 鎖持有到 commit/abort；**Rigorous（strong strict）2PL**：所有鎖持有到結束。實務系統都用 strict/rigorous 2PL，代價是並行度降低與死結。

10.4　死結

10.4.1　偵測

維護**等待圖（wait-for graph）**：$T_i \to T_j$ 表示 T_i 在等 T_j 持有的鎖。有環即死結，選一個**受害者**中止（依據：已完成的工作量、持有的鎖數、重啟次數避免餓死）。

PostgreSQL 的死結偵測

PostgreSQL 不常態性地檢查等待圖：一個 backend 等鎖超過 `deadlock_timeout`（預設 1 s）後，才由它自己跑一次等待圖檢查（`src/backend/storage/lmgr/deadlock.c`）；若發現環，**發現者自己**成為受害者，收到 `ERROR: deadlock detected`（SQLSTATE `40P01`），訊息會列出涉及的程序與各自等待的鎖。設 `log_lock_waits = on` 可把等待超過 `deadlock_timeout` 的事件（即使不是死結）寫進日誌。

10.4.2　預防

以交易時間戳（越老優先權越高）決定誰等、誰死，保證等待圖無環：

- **Wait-Die**（非搶占）：老交易等年輕交易；年輕交易要等老交易時直接中止（die）。
- **Wound-Wait**（搶占）：老交易要鎖時直接把年輕持有者中止（wound）；年輕交易等老交易。

中止的交易重啟時**保留原時間戳**，否則會餓死。分散式系統（如早期 Spanner、CockroachDB 的某些路徑）偏好預防，因為跨節點建等待圖代價高。

應用層避免死結

- **固定的存取順序**：多筆更新依主鍵排序後再執行（`UPDATE ... WHERE id IN (...) ORDER BY id` 不保證順序——用 `SELECT ... ORDER BY id FOR UPDATE` 先鎖）。
- 交易越短越好；把使用者互動、外部 API 呼叫移出交易。
- 設 `lock_timeout` 讓等待快速失敗並重試，而不是堆積。
- 外鍵：更新父表主鍵或刪父列會對子表取鎖；子表的外鍵欄位沒索引時會全表掃描並長時間持鎖。

10.5　Phantom 與述詞鎖

Strict 2PL 鎖住的是*已存在*的列。T_1 執行 `SELECT count(*) WHERE age > 30` 鎖了符合的 10 列，T_2 插入一列 `age = 40` 不會與任何鎖衝突——phantom 出現。解法：

- **述詞鎖（predicate lock）**：鎖住條件 `age > 30` 本身，與任何可能符合的插入衝突；判定述詞相交代價高，少有系統直接實作。
- **索引範圍鎖 / gap lock**：鎖住索引上 `(30, +∞)` 的區間。InnoDB 的 **next-key lock**（記錄鎖 + 前方的 gap 鎖）就是這個，這也是 InnoDB 的 RR 沒有 phantom（對鎖定讀而言）的原因，但也帶來不少「為什麼插入會被鎖住」的困惑。
- **SSI 的 SIREAD lock**：PostgreSQL 的做法——不阻塞，只記錄讀過的範圍（tuple / 頁 / 表），事後偵測衝突（第 9 章）。

10.6　PostgreSQL 的表級鎖

PostgreSQL 有 8 種表級鎖模式，名稱源自歷史，不要望文生義（`ROW EXCLUSIVE` 是表級鎖，表示「我要改列」）。每個 SQL 命令自動取得對應模式，衝突時排隊等待：

已持有 \ 請求	AS	RS	RX	SUX	S	SRX	X	AX
ACCESS SHARE (AS)								✗
ROW SHARE (RS)							✗	✗
ROW EXCLUSIVE (RX)					✗	✗	✗	✗
SHARE UPDATE EXCLUSIVE (SUX)				✗	✗	✗	✗	✗
SHARE (S)			✗	✗		✗	✗	✗
SHARE ROW EXCLUSIVE (SRX)			✗	✗	✗	✗	✗	✗
EXCLUSIVE (X)		✗	✗	✗	✗	✗	✗	✗
ACCESS EXCLUSIVE (AX)	✗	✗	✗	✗	✗	✗	✗	✗

模式	誰取得它
ACCESS SHARE	`SELECT`
ROW SHARE	`SELECT ... FOR UPDATE / FOR SHARE`
ROW EXCLUSIVE	`INSERT`、`UPDATE`、`DELETE`、`MERGE`
SHARE UPDATE EXCLUSIVE	`VACUUM`（非 FULL）、`ANALYZE`、`CREATE INDEX CONCURRENTLY`、`ALTER TABLE ... VALIDATE CONSTRAINT`、多數 `ALTER TABLE SET (...)`
SHARE	`CREATE INDEX`（非 CONCURRENTLY）——阻擋寫入但允許讀
SHARE ROW EXCLUSIVE	`CREATE TRIGGER`、部分 `ALTER TABLE`（如 `ADD FOREIGN KEY`）
EXCLUSIVE	`REFRESH MATERIALIZED VIEW CONCURRENTLY`
ACCESS EXCLUSIVE	`DROP TABLE`、`TRUNCATE`、`VACUUM FULL`、`CLUSTER`、`REINDEX`、多數 `ALTER TABLE`（加欄、改型別、加 NOT NULL……）、`LOCK TABLE` 預設

DDL 鎖住整個系統的經典事故

一個報表查詢在表 `orders` 上持有 ACCESS SHARE 已跑了 10 分鐘。工程師執行 `ALTER TABLE orders ADD COLUMN ...`，它需要 ACCESS EXCLUSIVE，於是**排隊等待**。問題是：鎖的等待是**先進先出**的——之後所有新的 `SELECT`（只需要 ACCESS SHARE）都排在 ALTER 後面，因為它們與排隊中的 ACCESS EXCLUSIVE 衝突。10 秒內連線池耗盡，整個服務停擺。對策：遷移腳本一律先 `SET lock_timeout = '3s'`，拿不到鎖就失敗重試；用 `CREATE INDEX CONCURRENTLY`、`ADD CONSTRAINT ... NOT VALID` 再 `VALIDATE`、PostgreSQL 11+ 的 `ADD COLUMN ... DEFAULT`（常數預設不重寫表）等低鎖手段。

10.7　PostgreSQL 的列級鎖

已持有 \ 請求	KEY SHARE	SHARE	NO KEY UPDATE	UPDATE
FOR KEY SHARE				✗
FOR SHARE			✗	✗
FOR NO KEY UPDATE		✗	✗	✗
FOR UPDATE	✗	✗	✗	✗

- `UPDATE` 不改主鍵 / 唯一鍵欄位時取 **FOR NO KEY UPDATE**；改鍵或 `DELETE` 取 **FOR UPDATE**。
- 外鍵檢查對被參照的父列取 **FOR KEY SHARE**——所以更新父列的非鍵欄位（NO KEY UPDATE）不會與子表的插入互相阻塞。這是 9.3 引入這兩種弱鎖的原因。
- 列鎖記錄在 tuple 的 `xmax`（多個持有者時用 MultiXact），不佔鎖表，所以鎖百萬列也不會耗盡 `max_locks_per_transaction`，但會產生 WAL 與髒頁。
- `NOWAIT`：拿不到立即失敗；`SKIP LOCKED`：跳過被鎖的列——實作**工作佇列**的標準寫法。

程式碼 10.1: 以 SKIP LOCKED 實作多消費者佇列

```
WITH next_job AS (
  SELECT id FROM jobs
  WHERE  status = 'pending'
  ORDER  BY created_at
  FOR UPDATE SKIP LOCKED
  LIMIT  1
)
UPDATE jobs SET status = 'running', worker = :me
FROM   next_job WHERE jobs.id = next_job.id
RETURNING jobs.*;
```

Advisory lock（`pg_advisory_lock(key)`、`pg_try_advisory_xact_lock(key)`）讓應用程式借用資料庫的鎖管理器協調任意資源（如「同一時間只跑一個排程器」）。

10.8　診斷鎖等待

程式碼 10.2: 誰在等誰

```
SELECT a.pid, a.state, a.wait_event_type, a.wait_event,
       pg_blocking_pids(a.pid) AS blocked_by,
       now() - a.xact_start AS xact_age,
       left(a.query, 60) AS query
FROM   pg_stat_activity a
WHERE  cardinality(pg_blocking_pids(a.pid)) > 0 OR a.pid IN (SELECT unnest(pg_blocking_pids(pid))
    ↪ FROM pg_stat_activity)
ORDER  BY a.xact_start;
```

`pg_locks` 列出所有鎖（`granted = false` 為等待中）；`pg_stat_activity.wait_event_type = 'Lock'` 表示在等交易鎖，`'LWLock'` 則是內部閂鎖爭用（如 `WALWrite`、`BufferMapping`、`LockManager`）——那是另一類問題（I/O 或 CPU 瓶頸），不是應用程式的鎖。找到源頭後可 `pg_cancel_backend(pid)`（取消查詢）或 `pg_terminate_backend(pid)`（斷線）。

10.9　樂觀並行控制

Kung 與 Robinson（1981）的 **OCC**：交易在私有工作區讀寫（**讀取階段**），commit 時**驗證**讀集沒有被已 commit 的並行交易改過（backward validation：與所有在我開始後 commit 的交易比對），通過才把寫集**寫入**（寫入階段，通常在一個短暫的全域臨界區內）。衝突罕見時沒有任何鎖開銷；衝突多時重做成本高，且長交易容易一直被年輕交易「驗證失敗」。記憶體資料庫（Hekaton、Silo、TicToc）與 FoundationDB 採用 OCC 變體。對照：MVCC + first-updater-wins 其實也是一種「對寫入樂觀」的策略。

MiniDB 連結：v4 LockManager

v4 實作 `LockManager`：`LockTable(txn, table_id, mode)`、`LockRow(txn, rid, mode)`、`Unlock*`，支援 IS/IX/S/SIX/X 與本章的相容矩陣，strict 2PL（鎖到 commit/abort 才釋放），背景執行緒每 50 ms 建等待圖找環並中止最年輕的交易。測試：兩個執行緒交叉更新兩列必須有一方收到 deadlock 錯誤且另一方成功；壓測中不得有鎖洩漏。這正是 CMU 15-445 Project #4 的核心。

本章練習

1. 交易 T_1 在表 R 上取 SIX，T_2 想在 R 的某列取 X：T_2 需要先在 R 上取什麼鎖？能否取得？若 T_2 只想讀某列呢？
2. 給出一個遵守 2PL 但發生連鎖中止的排程，再說明 strict 2PL 如何避免。
3. 三筆交易 $T_1(ts{=}10)$、$T_2(ts{=}20)$、$T_3(ts{=}30)$，T_1 等 T_3、T_3 等 T_2、T_2 想要 T_1 持有的鎖。在 wait-die 與 wound-wait 下分別會發生什麼？
4. 為什麼 `CREATE INDEX CONCURRENTLY` 要掃描表兩次並等待所有既有交易結束？它取的 SHARE UPDATE EXCLUSIVE 與哪些操作衝突？
5. 設計一個遷移腳本把 `users.email` 加上 NOT NULL 與唯一約束，要求全程不阻塞讀寫超過 1 秒。列出每一步取得的鎖。
6. 用 `SKIP LOCKED` 的佇列在消費者當機時工作會不會遺失？與「先 UPDATE 標記再處理」相比，兩者的失敗模式有何不同？
7. （動手）開三個 `psql` 重現「DDL 排隊導致 SELECT 全部阻塞」，用本章的診斷查詢觀察，然後加入 `lock_timeout` 再試一次。

延伸閱讀與資源

- PostgreSQL 文件 §13.3 *Explicit Locking*（表級 / 列級 / 頁級鎖、死結、advisory lock）：https://www.postgresql.org/docs/current/explicit-locking.html
- PostgreSQL 原始碼 `src/backend/storage/lmgr/README`（鎖管理器設計）、`README.tuplock`（列鎖與 MultiXact）、`deadlock.c`。
- J. Gray, R. Lorie, G. Putzolu, I. Traiger, *Granularity of Locks and Degrees of Consistency in a Shared Data Base*, 1976。
- H. T. Kung, J. T. Robinson, *On Optimistic Methods for Concurrency Control*, ACM TODS 1981。
- CMU 15-445 Fall 2025 Lecture #17 *Two-Phase Locking Concurrency Control*；Project #4 *Concurrency Control*。
- M. Slot (Citus), *PostgreSQL rocks, except when it blocks: Understanding locks*：https://www.citusdata.com/blog/2018/02/15/when-postgresql-blocks/；*Seven tips for dealing with Postgres locks*：https://www.citusdata.com/blog/2018/02/22/seven-tips-for-dealing-with-postgres-locks/
- H. Ringhus, *pglocks.org*（每個 SQL 命令取得哪些鎖的互動查詢表）：https://pglocks.org/
- MySQL 文件 *InnoDB Locking*（record / gap / next-key / insert intention lock）：https://dev.mysql.com/doc/refman/8.4/en/innodb-locking.html

第 11 章　當機復原與 ARIES

本章學習目標

- 以 steal/no-steal × force/no-force 的矩陣定位各種復原策略，包括 shadow paging。
- 陳述 ARIES 的三個原則：WAL、redo 時重複歷史、undo 時寫 CLR。
- 描述 ARIES 的資料結構（ATT、DPT、pageLSN、prevLSN、undoNextLSN）與 fuzzy checkpoint 的內容。
- 一步步執行 Analysis、Redo、Undo 三個階段，並說明為何復原過程中再次當機仍然正確。
- 解釋 PostgreSQL 為何沒有 undo 階段、啟動時如何用 `pg_control` 與 WAL 重播到一致狀態，以及 PITR 與 timeline。

11.1　失效的類型與復原的目標

- **交易失效**：邏輯錯誤、約束違反、死結受害者——只需回滾單筆交易（undo）。
- **系統當機**：程序崩潰、OS panic、斷電——揮發性記憶體（buffer pool、鎖表）全部遺失，磁碟完好。本章主題。
- **媒體失效**：磁碟損毀——需要備份 + 日誌封存（PITR）或複本（第 14 章）。

復原演算法的目標是在系統當機後恢復 **原子性**（未 commit 的交易效果消失）與**持久性**（已 commit 的交易效果存在）。

11.2　設計空間：steal/no-steal 與 force/no-force

	No-Steal（未 commit 的髒頁不可寫回）	**Steal**（可寫回）
Force（commit 時髒頁必須寫回）	不需 undo、不需 redo。最簡單，但 commit 要等隨機寫入；buffer pool 必須容納交易全部的髒頁。	需 undo、不需 redo。
No-Force（commit 不寫回）	不需 undo、需 redo。	**需 undo 與 redo。效能最佳（ARIES、多數商用系統）。**

Shadow paging（System R 早期、LMDB、SQLite 的 rollback-journal 模式精神相近）是 no-steal/force 的代表：修改時複製頁面（copy-on-write）到新位置，commit 時原子地切換頁

表根指標。優點是不需要日誌與復原程序；缺點是每次 commit 都要寫出所有頁並 fsync、資料碎片化、頁表切換難以支援高並行。現代系統幾乎都選擇 steal/no-force + WAL。

11.3　ARIES 的三個原則

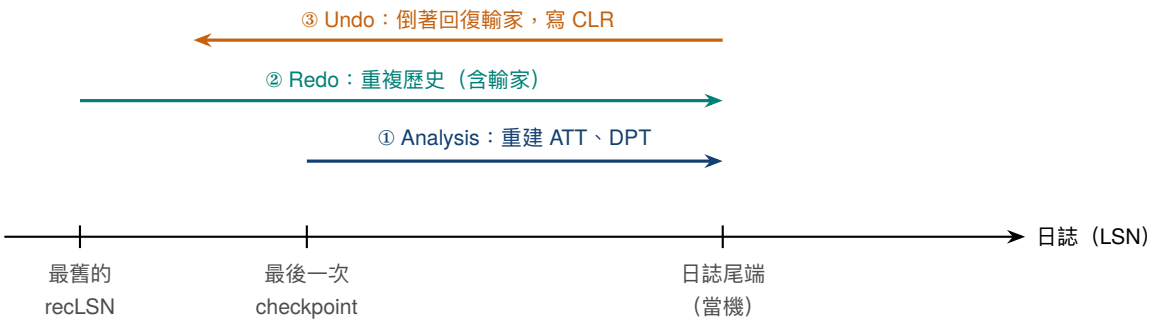
Mohan 等人（1992）的 **ARIES**（Algorithms for Recovery and Isolation Exploiting Semantics）是幾乎所有現代資料庫復原機制的藍本：

1. **Write-Ahead Logging**：第 8 章的兩條規則。
2. **Repeating history during redo**：當機後，先把資料庫完整重建到當機那一刻的狀態（包含未 commit 交易的修改），然後才 undo 輸家。這讓 redo 邏輯極其簡單，也讓細粒度鎖與複雜的頁面結構（B+Tree 分裂）得以正確復原。
3. **Logging changes during undo**：undo 的每一步也寫日誌（**Compensation Log Record, CLR**），所以 undo 進行到一半再當機，不會重複 undo。

11.4　資料結構

結構	內容
日誌記錄	`LSN`、`type`（UPDATE / COMMIT / ABORT / END / CLR / CHECKPOINT）、`txnId`、`prevLSN`（同交易前一筆）、`pageId`、位移、**before image**、**after image**；CLR 另有 `undoNextLSN`：下一個要 undo 的記錄。
`pageLSN`	每個資料頁標頭：最後修改它的 LSN。
Transaction Table（ATT）	每筆活動交易：`txnId`、狀態（running / committing / aborting）、`lastLSN`（最新記錄）。
Dirty Page Table（DPT）	每個髒頁：`pageId`、**recLSN**——第一次把它弄髒的 LSN；該頁磁碟版本保證含 recLSN 之前的所有修改。
Checkpoint	`BEGIN_CHECKPOINT` + `END_CHECKPOINT`（含當時的 ATT 與 DPT 快照）；**master record** 記下最新 BEGIN_CHECKPOINT 的 LSN。Fuzzy：checkpoint 期間交易照常進行、不寫出髒頁。

11.5　三個階段



11.5.1 Analysis

從 master record 指向的最後一次 checkpoint 開始**向前**掃到日誌尾端，重建當機瞬間的 ATT 與 DPT：

- 以 checkpoint 中的 ATT、DPT 為起點。
- 遇到某交易的記錄：加入 / 更新 ATT 的 `lastLSN`；遇到 `END`：從 ATT 移除；遇到 `COMMIT`：狀態改為 committing。
- 遇到修改頁 P 的記錄且 P 不在 DPT：加入，`recLSN` = 該 LSN。

結束時，ATT 中的交易就是**輸家（loser）**（除了 committing 者，它們只需補寫 END）；DPT 中最小的 `recLSN` 是 redo 的起點。

11.5.2 Redo

從 min(recLSN) **向前**掃描，對每一筆 UPDATE 或 CLR 記錄 r（修改頁 P）：若以下任一成立則**跳過**，否則重做並把 pageLSN 設為 r.LSN：

1. P 不在 DPT（當機前已寫回）；
2. P 在 DPT 但 r.LSN $<$ recLSN（這個修改已在磁碟上）；
3. 讀入 P 後發現 pageLSN $\geq r$.LSN（此修改已存在；這一步需要實際讀頁）。

注意 redo **不區分贏家輸家**——這就是「重複歷史」。Redo 結束後，資料庫等同當機瞬間的記憶體狀態。對 committing 的交易寫 END 並移出 ATT。

11.5.3 Undo

把所有輸家的 `lastLSN` 放進 **ToUndo** 集合，反覆取出**最大**的 LSN（確保所有交易的 undo 以 LSN 遞減的順序交錯進行，只需一趟向後掃描）：

- 若是 UPDATE：套用 before image，寫一筆 CLR（`undoNextLSN` = 該記錄的 `prevLSN`），把 `prevLSN` 加入 ToUndo。
- 若是 CLR：不 undo 它（它描述的 undo 已經生效），把它的 `undoNextLSN` 加入 ToUndo。
- 若 `prevLSN` / `undoNextLSN` 為 null：該交易完全回復，寫 END。

11.5.4　完整範例

LSN	記錄	prevLSN	說明
10	BEGIN_CHECKPOINT		
20	END_CHECKPOINT		ATT = {}，DPT = {}
30	T_1 UPDATE P1	–	DPT: P1 recLSN 30
40	T_2 UPDATE P2	–	DPT: P2 recLSN 40
50	T_1 UPDATE P3	30	DPT: P3 recLSN 50
60	T_2 COMMIT	40	T_2 進入 committing
70	T_1 UPDATE P1	50	P1 已在 DPT，recLSN 不變
80	T_2 END	60	T_2 移出 ATT
	（當機；假設 *P2* 已被寫回磁碟，*P1*、*P3* 未寫回）		

Analysis（從 LSN 10）：ATT = {T_1: lastLSN 70}；DPT = {P1: 30, P2: 40, P3: 50}。輸家 = T_1。
Redo（從 LSN 30）：30 重做 P1；40 讀 P2 發現 pageLSN = 40 跳過；50 重做 P3；70 重做 P1。
Undo（ToUndo = {70}）：undo 70 寫 CLR 90（undoNext 50）；undo 50 寫 CLR 100（undoNext 30）；undo 30 寫 CLR 110（undoNext null）；寫 T_1 END 120。

復原中再當機

若在寫完 CLR 100 之後再度當機：下一次 Analysis 會看到 T_1 仍在 ATT（lastLSN 100）；Redo 會重做 CLR 90、100（它們也是修改頁的記錄）；Undo 取出 100，它是 CLR，直接跳到 `undoNextLSN` = 30，只剩 LSN 30 需要 undo。每一步 undo 最多做一次——**冪等**。這就是 CLR 的意義：ARIES 的復原工作量有上界，不會因反覆當機而無限增長。

11.6　ARIES 的進階主題

- **邏輯 undo 與 nested top action**：B+Tree 分裂屬於「結構修改」，即使發起的交易最後中止，分裂本身也不該回復（其他交易可能已經用到新頁）。ARIES 以 nested top action（寫一筆 dummy CLR 跳過整段）讓結構修改在邏輯上立即 commit；索引項目的刪除則以**邏輯 undo**（重新搜尋鍵再刪除）進行，因為鍵可能已搬到別頁。
- **部分回滾**：savepoint 的實作就是 undo 到某個 LSN 為止。
- **並行復原**：redo 可依頁或依 LSN 範圍切成多個執行緒；PostgreSQL 的 redo 到 18 為止仍是單一 startup process（但可用 `recovery_prefetch` 預讀頁面）。
- **媒體復原**：還原最近一次完整備份後，從備份開始的 LSN 重播封存的日誌——本質上就是 redo 階段的延伸，這就是 PITR。

11.7　PostgreSQL 的復原

沒有 undo 階段的 ARIES

PostgreSQL 採 steal/no-force 與 WAL，Analysis 與 Redo 的精神與 ARIES 相同，但**沒有 Undo 階段**：MVCC 下，中止或未完成交易所寫的 tuple 版本只是不可見（其 XID 在 clog 中不是 committed），不需要物理回復，之後由 VACUUM 清理。因此 WAL 只含 redo 資訊，也不需要 CLR。代價是中止交易留下的垃圾要靠 VACUUM 處理，以及長交易回滾雖然瞬間完成、但不會釋放空間。

啟動流程（`src/backend/access/transam/xlogrecovery.c`）：

1. 讀 `global/pg_control`：若狀態為 `DB_SHUTDOWNED` 則是乾淨關機，不需復原；否則進入 crash recovery（日誌出現 `database system was interrupted`、`database system was not properly shut down; automatic recovery in progress`）。
2. 取最後一次 checkpoint 記錄的 **redo point**，從該處開始**順序重播** WAL（`StartupXLOG()`）：每筆記錄依 `xl_rmid` 呼叫對應資源管理器的 `rm_redo`；有 full-page image 的 block 直接覆蓋；否則比較 pageLSN，只套用更新的記錄（與 ARIES 的條件 3 相同）。
3. 遇到不完整或 CRC 錯誤的記錄視為日誌尾端（`redo done at ...`）。此時資料庫狀態等同當機瞬間，未 commit 的交易「自動」消失。
4. 建立一個新 checkpoint（end-of-recovery checkpoint，PostgreSQL 15 起較小的 end-of-recovery record），開放連線。

Archive recovery 與 PITR：若存在 `recovery.signal` / `standby.signal`，startup process 透過 `restore_command` 從封存取回段檔持續重播，直到 `recovery_target_time` / `_lsn` / `_xid` / `_name` 指定的位置，然後進入新的 **timeline**（timeline ID 加一，`.history` 檔記錄分支點），確保新舊歷史的 WAL 不會混淆。Hot standby（第 14 章）就是永不結束的 archive recovery，並允許唯讀查詢。

永遠不要碰的工具

`pg_resetwal` 會偽造一個乾淨的 checkpoint 讓損壞的叢集啟動——但它**丟棄所有未重播的 WAL**，資料庫幾乎必然處於不一致狀態（索引與 heap 不符、clog 遺失）。只在已經有備份、且目的是搶救部分資料並立即 `pg_dump` 重建時使用。同理，`fsync = off` 之後的當機沒有復原可言。

11.8　其他系統

- **InnoDB**：接近教科書 ARIES——先從最後 checkpoint 重播 redo log，再以 undo log 回滾未完成交易（回滾在背景進行，啟動後即可服務）。
- **SQLite**：rollback-journal 模式是「先把舊頁備份到 journal、改主檔、commit 時刪 journal」（undo-only）；WAL 模式是 redo-only（第 8 章）。
- **RocksDB**：重播 WAL 重建 memtable，SSTable 本身不可變，不需 undo（第 12 章）。

MiniDB 連結：v4 Recovery

v4 可二選一：(a) 完整 ARIES——實作 ATT、DPT、CLR 與三階段，是最扎實的練習（也是 CMU 15-445 舊版 Project 的內容）；(b) PostgreSQL 式 redo-only——配合 v5 的 MVCC，只需 Analysis + Redo。無論哪一種，都要寫一個「隨機時刻 `kill -9` 後重啟並驗證不變量」的壓力測試（例如轉帳總額守恆），並重複數百次。

本章練習

1. 為什麼 ARIES 要「重複歷史」而不是只 redo 贏家？舉一個只 redo 贏家會出錯的例子（提示：兩筆交易修改同一頁的不同 slot，且頁面曾被寫回）。
2. 沿用本章範例，但假設當機前 P1 已被寫回磁碟（pageLSN = 70）。重新執行三個階段並列出每一步。
3. Fuzzy checkpoint 的 DPT 快照可能包含「其實已經寫回」的頁，或漏掉「checkpoint 期間新弄髒」的頁。這兩種情況為何都不影響正確性？
4. 若 CLR 不記錄 `undoNextLSN`，而是把 undo 當作一般 UPDATE 記錄，復原中再當機會發生什麼？
5. PostgreSQL 沒有 undo 階段，代價轉嫁到哪裡？對比 InnoDB，在「10 億列的交易在 99% 時中止」的情境下，兩者的行為差異為何？
6. （動手）啟動 PostgreSQL 執行一批交易，`kill -9` postmaster 後重啟，閱讀日誌中的復原訊息；再用 `pg_controldata` 比較前後的 checkpoint 位置與狀態欄位。

延伸閱讀與資源

- C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, P. Schwarz, *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*, ACM TODS 1992：`https://cs.stanford.edu/people/chrismre/cs345/rl/aries.pdf`
- CMU 15-445 Fall 2025 Lecture #20 *Database Recovery*（ARIES 的教學版推導）。
- R. Ramakrishnan, J. Gehrke, *Database Management Systems*, 3rd ed., Ch. 18 *Crash Recovery*（最清楚的 ARIES 教科書章節）。
- PostgreSQL 文件 Ch. 25 *Backup and Restore*（含 *Continuous Archiving and Point-in-Time Recovery*）：`https://www.postgresql.org/docs/current/continuous-archiving.html`；`pg_controldata`、`pg_resetwal` 說明頁。
- Hironobu Suzuki, *The Internals of PostgreSQL*, Ch. 9（§9.8 復原流程）、Ch. 10 *Base Backup & PITR*：`https://www.interdb.jp/pg/pgsql10.html`
- PostgreSQL 原始碼 `src/backend/access/transam/xlogrecovery.c`、`xlog.c`（`StartupXLOG`）、`src/backend/access/transam/README`。
- J. Hellerstein, M. Stonebraker, J. Hamilton, *Architecture of a Database System*, 2007, §6 *Transactions: Concurrency Control and Recovery*。

- A. Petrov, *Database Internals*, Ch. 5 *Transaction Processing and Recovery*。

第五篇

替代儲存架構：LSM-Tree 與行式儲存

第 12 章　LSM-Tree

本章學習目標

- 解釋 B+Tree 原地更新的寫入放大問題，以及 LSM-Tree 如何以「記憶體緩衝 + 循序寫出 + 背景合併」換取寫入吞吐。
- 描述 MemTable、SSTable、Manifest、WAL 的角色與 SSTable 的檔案格式。
- 走一遍點查詢與範圍掃描的讀取路徑，說明 Bloom filter 與 block cache 如何壓低讀取放大。
- 比較 leveled 與 tiered（universal）compaction 在寫入 / 讀取 / 空間放大上的取捨（RUM 三角）。
- 理解 tombstone、write stall、key-value 分離等實務議題，並能解讀 RocksDB 的主要參數。
- 判斷何時該選 LSM、何時該選 B+Tree。

12.1　為什麼需要另一種結構

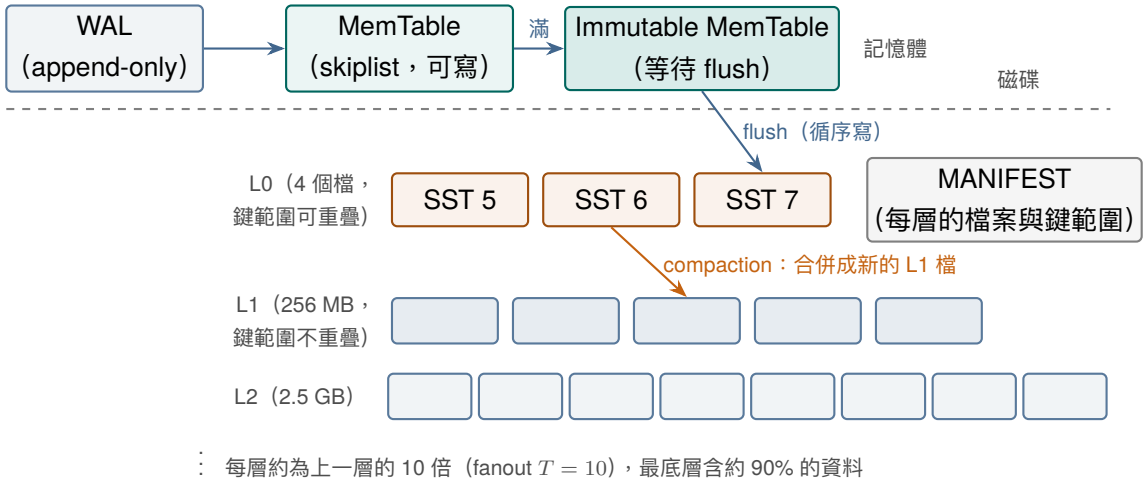
B+Tree 的每次更新都是**原地修改**一個頁面：即使只改 100 bytes，最終也要把整個 8–16 KB 的頁寫回磁碟（**寫入放大** 80–160 倍），而且這些頁散佈各處，是**隨機寫入**。在 HDD 時代這是尋道瓶頸，在 SSD 上則加速磨損並放大 FTL 內部的垃圾回收。隨機寫入的另一個代價：每頁寫入前都要取 latch、處理分裂、寫 WAL。

O'Neil 等人（1996）的 **Log-Structured Merge-Tree**（LSM-Tree）改變策略：

1. 寫入先進入**記憶體中的排序結構**（並 append 到 WAL 保證持久）。
2. 滿了就**整批循序寫出**成一個不可變的排序檔案。
3. 背景把多個檔案**合併（compaction）**成更大的檔案，清掉被覆蓋的舊值。

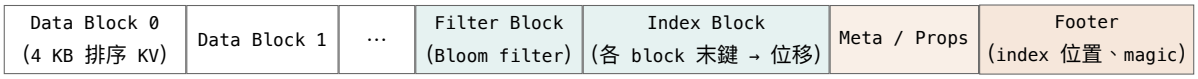
所有磁碟寫入都是大塊循序寫，讀取則要查多個檔案——LSM 把成本從寫入端搬到讀取端與背景工作。Google Bigtable（2006）、LevelDB（2011）、RocksDB（2012）讓它成為現代寫入密集系統的標準配備。

12.2 結構



- **MemTable**：記憶體中的排序結構，通常是 **skiplist**（並行插入友善；RocksDB 預設），也可用紅黑樹或 B-tree。寫滿（`write_buffer_size`，64 MB）後凍結成 immutable 並換新的。
- **WAL**：每筆寫入先 append 到 WAL；一個 memtable flush 完成後對應的 WAL 即可刪除。復原只需重播 WAL 重建 memtable（第 11 章）。
- **SSTable（Sorted String Table）**：不可變、鍵排序的檔案。寫出後永不修改，只會在 compaction 中被讀取與刪除——因此不需要 latch、也沒有 torn page 問題。
- **Levels**：L0 由 flush 直接產生，各檔鍵範圍可重疊；L1 以下每層內的檔案鍵範圍互不重疊，整層等同一個巨大的排序陣列，查找只需二分搜尋 MANIFEST 中的鍵範圍。
- **MANIFEST**：描述目前版本（每層的檔案清單）的日誌；每次 flush/compaction 追加一筆版本編輯。

12.2.1 SSTable 檔案格式



Data block 內的鍵以**前綴壓縮**儲存（每 16 個鍵一個 restart point 供二分搜尋），整個 block 可用 LZ4/Zstd 壓縮並附 CRC。讀一個鍵：footer → index block（通常常駐快取）→ 二分搜尋找到 data block → 讀取、解壓、在 block 內搜尋。

12.3 讀取路徑

點查詢 `get(k)` 由新到舊依序找，第一個命中即為答案（新值蓋舊值）：

1. MemTable → immutable memtable(s)。
2. L0 的每一個檔案（鍵範圍重疊，所以都得查；這是 L0 檔數要控制的原因）。
3. L1、L2……每層最多一個檔案（二分搜尋 MANIFEST 的鍵範圍）。

最壞情況要讀「L0 檔數 + 層數」個 SSTable——這是 **讀取放大**。兩個關鍵武器：

- **Bloom filter**：每個 SSTable（或每個 data block）一個；10 bits/key 時誤判率約 1%，讓不存在於該檔的查詢幾乎免費跳過。對不存在鍵的查詢（例如「insert if not exists」）尤其關鍵。Bloom filter 對範圍查詢無效（除非用前綴 bloom）。
- **Block cache**：快取解壓後的 data block（RocksDB 預設 LRU，8 MB——務必調大），index/-filter block 可固定在快取中。

範圍掃描無法用 bloom filter，必須在每一層開一個游標，用**合併迭代器（merging iterator）**（最小堆）同時推進並去重；層數越多、tombstone 越多，掃描越慢。這就是 LSM 的範圍查詢通常比 B+Tree 慢的原因。

12.4 Compaction

Compaction 是 LSM 的心臟：它**回收空間**（丟棄被覆蓋的舊版本與已刪除的鍵）、**限制讀取放大**（減少要查的檔案數）、**維持層級結構**。代價是消耗 I/O 與 CPU，並造成**寫入放大**：同一筆資料在生命週期中會被重寫多次。

12.4.1 Leveled compaction

RocksDB / LevelDB 預設。每層有大小上限（$L_1 = 256$ MB、每層 $\times 10$）；某層超標時，挑一個檔案與下一層鍵範圍重疊的檔案合併、寫成新檔到下一層。L0 累積 4 個檔時觸發 L0→L1。因為每層內鍵不重疊，每層只佔約 10% 的「冗餘」空間，**空間放大約 1.1**；但一筆資料每往下一層就要和下一層約 $T = 10$ 倍大的資料一起重寫，**寫入放大約** $T\times$ **層數** $\approx 10 \times L$（實務 10–30）。

12.4.2 Tiered / Universal compaction

每層容許 T 個鍵範圍重疊的 run；累積到 T 個就把它們合併成一個 run 放到下一層。一筆資料每層只被重寫一次，**寫入放大約等於層數**（低），但每層要查 T 個 run、且被覆蓋的舊值可能同時存在多個 run，**讀取與空間放大高**（空間最壞 $2\times$ 以上）。Cassandra 的 size-tiered、RocksDB 的 universal compaction、HBase 都屬此類；寫入極重、讀取可容忍較慢時使用。

策略	寫入放大	讀取放大（點查）	空間放大	適用
Leveled	高：$O(T \cdot L)$	低：$O(L)$（有 bloom 時接近 1）	低：≈ 1.1	讀多、空間敏感（預設）
Tiered / Universal	低：$O(L)$	高：$O(T \cdot L)$	高：≈ 2 或更多	寫入極重、時序資料
FIFO	無（只刪舊檔）	高	低	快取、僅保留最近資料
B+Tree（對照）	每次更新重寫整頁	$O(\log_B N)$，穩定	頁填充率 $\approx 70\%$	讀多寫少、範圍查詢

Athanassoulis 等人（2016）的 **RUM 猜想**：Read、Update、Memory（空間）三種開銷不可能同時最小化——任何存取方法都是三角形上的一點，LSM 的各種 compaction 策略就是在

這個三角形上移動。

12.5 刪除、tombstone 與其他實務問題

- **刪除是寫入**：`delete(k)` 寫入一個 **tombstone**。它必須存活到抵達最底層並確認沒有更舊的版本後才能丟棄；大量刪除後讀取反而變慢（要跳過成堆的 tombstone），Cassandra 的「tombstone 警告」即源於此。**範圍 tombstone** 以一筆記錄刪除整個鍵區間。
- **Write stall**：寫入速度超過 compaction 時，L0 檔案與待合併資料堆積。RocksDB 在 L0 達 20 個檔減速、36 個檔停止寫入（`level0_slowdown/stop_writes_trigger`），以及 `soft/hard_pending_compaction_bytes_limit`。延遲尖峰常來自這裡；對策是限制寫入速率（rate limiter）、增加 compaction 執行緒、調大 L0 觸發值。
- **壓縮分層**：上層用快速的 LZ4（頻繁重寫）、最底層用高壓縮比的 Zstd（90% 的資料、很少重寫）。
- **Key-value 分離（WiscKey）**：大 value 存到獨立的 value log，LSM 只存鍵與指標，compaction 不再搬動 value（RocksDB BlobDB、TiKV Titan、Badger）。代價是範圍掃描要多一次隨機讀，value log 自己需要 GC。
- **MVCC 內建**：RocksDB 的每筆寫入帶一個遞增的 **sequence number**，snapshot 就是一個 sequence number；CockroachDB、TiKV 進一步把交易時間戳編進鍵（`key@ts`），LSM 天然保留多版本，舊版本由 compaction 依 GC 門檻清除（第 9、16 章）。

RocksDB 的關鍵參數

`write_buffer_size`（64 MB）與 `max_write_buffer_number`（2）決定記憶體緩衝；`level0_file_num_compaction_trigger`（4）、`max_bytes_for_level_base`（256 MB）、`max_bytes_for_level_multiplier`（10）、`target_file_size_base`（64 MB）決定層級形狀；`BlockBasedTableOptions`：`block_size`（4 KB）、`block_cache`、`filter_policy = NewBloomFilterPolicy(10)`、`cache_index_and_filter_blocks`；`max_background_jobs` 控制 flush/compaction 執行緒數。官方 *Tuning Guide* 的第一句話仍然適用：「先量測，再調整」——`rocksdb.stats` 與 `compaction stats` 會列出每層的大小、檔數、讀寫量與放大倍數。

12.6 採用 LSM 的系統

- **嵌入式 KV**：LevelDB、RocksDB（Meta；MySQL 的 MyRocks、Kafka Streams、Flink 狀態後端）、Pebble（CockroachDB，Go 重寫）、Badger、Sled。
- **分散式資料庫**：Cassandra / ScyllaDB、HBase、TiKV（TiDB）、CockroachDB、YugabyteDB（DocDB）、FoundationDB（可選 Redwood B+Tree）。
- **時序 / 搜尋**：InfluxDB（TSM）、Prometheus TSDB（類 LSM 的 block）、Lucene/Elasticsearch 的 segment merge 也是同一思想。
- 對照：MongoDB WiredTiger 預設仍是 B+Tree（可選 LSM），PostgreSQL / MySQL InnoDB / SQL Server / Oracle 皆為 B+Tree。

12.7 LSM 還是 B+Tree？

	B+Tree	LSM-Tree
寫入吞吐	隨機 I/O、每頁 latch；中等	循序 I/O、批次；高
點查詢	$O(\log N)$ 穩定、可預期	需查多層；bloom filter 後接近 B+Tree
範圍掃描	葉頁鏈結，極佳	多路合併、受 tombstone 影響；較慢
空間	頁填充率 $\sim 70\%$；原地更新無額外版本	leveled $\sim 1.1\times$；壓縮效果好（大塊排序資料）
延遲尾端	分裂 / 合併造成小幅抖動	compaction 與 write stall 造成明顯尖峰
並行與交易	需要 latch crabbing 等複雜協定	檔案不可變，簡單；交易多由上層實作
典型場景	OLTP 關聯式資料庫	寫入密集、KV、時序、分散式儲存底層

MiniDB 連結：可選的 LSM 儲存後端

在 v2 之後可以把 `HeapFile + BPlusTree` 替換成一個 LSM 引擎作為實驗分支：skiplist memtable、簡單 SSTable（data blocks + index + bloom）、MANIFEST、leveled compaction、merging iterator，並量測寫入放大（寫入的位元組 ÷ 使用者資料位元組）。CMU 的 *mini-lsm* 教學（Rust）是極佳的逐步指南；完成後再回頭比較兩種後端在 YCSB 式負載下的差異。

本章練習

1. Leveled compaction、$T = 10$、資料總量 1 TB、L1 = 256 MB。需要幾層？估算一筆資料在生命週期中被重寫的次數（寫入放大）。若 $T = 4$ 呢？
2. 每個 SSTable 含 100 萬個鍵，bloom filter 10 bits/key。(a) 這個 filter 多大？(b) 誤判率約多少？(c) 一次對不存在鍵的查詢，在 6 層、L0 有 4 個檔的情況下預期讀幾個 data block？
3. 為什麼 L0 的檔案鍵範圍可以重疊、L1 以下卻不行？若讓 L1 也允許重疊，讀寫放大會怎麼變？
4. 解釋為何 tombstone 不能在到達底層之前丟棄；給一個過早丟棄導致「已刪除的資料復活」的例子。
5. 一個工作負載 95% 是對最近 1 小時資料的 `INSERT`、5% 是對任意時間的點查詢。你會選哪種 compaction？若改為大量的範圍掃描呢？
6. 比較 RocksDB 的 sequence number 快照與 PostgreSQL 的 XID 快照：舊版本分別存在哪裡、由誰回收？
7. （動手）用 RocksDB 的 `db_bench` 跑 `fillrandom` 後 `readrandom`，觀察 `rocksdb.stats` 中

每層的大小與 W-Amp 欄位；再把 compaction 改成 universal 重跑比較。

延伸閱讀與資源

- P. O'Neil, E. Cheng, D. Gawlick, E. O'Neil, *The Log-Structured Merge-Tree (LSM-Tree)*, Acta Informatica 1996：`https://www.cs.umb.edu/~poneil/lsmtree.pdf`
- F. Chang et al., *Bigtable: A Distributed Storage System for Structured Data*, OSDI 2006。
- RocksDB Wiki：*Leveled Compaction*、*Universal Compaction*、*RocksDB Tuning Guide*、*Bloom Filter*、*Block Cache*：`https://github.com/facebook/rocksdb/wiki`
- LevelDB 實作說明 `doc/impl.md` 與 `doc/table_format.md`：`https://github.com/google/leveldb/tree/main/doc`
- S. Dong et al., *RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications*, ACM TOS 2021。
- N. Dayan, M. Athanassoulis, S. Idreos, *Monkey: Optimal Navigable Key-Value Store*, SIGMOD 2017；N. Dayan, S. Idreos, *Dostoevsky*, SIGMOD 2018（lazy leveling）。
- L. Lu et al., *WiscKey: Separating Keys from Values in SSD-Conscious Storage*, FAST 2016。
- C. Luo, M. Carey, *LSM-based Storage Techniques: A Survey*, VLDB Journal 2020：`https://arxiv.org/abs/1812.07527`
- M. Athanassoulis et al., *Designing Access Methods: The RUM Conjecture*, EDBT 2016。
- A. Chi (skyzh), *LSM in a Week*（mini-lsm，Rust 動手教學）：`https://skyzh.github.io/mini-lsm/`
- CMU 15-445 Fall 2025 Lecture #4 *Database Storage: Log-Structured Storage*；CMU 15-721 *Advanced Database Systems* 的儲存模型講次。

第 13 章　行式儲存與 OLAP

本章學習目標

- 從 I/O、壓縮、CPU 三個角度解釋為何分析型查詢在行式（column-oriented）儲存上快一到兩個數量級。
- 熟悉主要的輕量級壓縮編碼（字典、RLE、bit-packing、delta、bitmap）並知道何時適用、如何「直接在壓縮資料上運算」。
- 說明向量化執行、SIMD、延遲物化（late materialization）、zone map 跳塊如何協同運作。
- 讀懂 Parquet 檔案結構，理解 PAX ／ row group 的設計理由，以及 Arrow 的角色。
- 了解行式系統如何處理寫入（delta store + merge）與 HTAP 的常見架構，並知道 PostgreSQL 在分析負載上的選項與限制。

13.1　OLTP 與 OLAP 的分歧

第 1 章區分過兩種負載：OLTP 以主鍵存取少數列的全部欄位並頻繁更新；OLAP 掃描上億列但只碰少數欄位，做聚合，幾乎只讀。用列式（row-oriented，NSM）儲存跑 OLAP 有三個根本問題：

1. **I/O 浪費**：`SELECT avg(price) FROM sales` 只需要一欄，卻得讀入整筆 tuple（可能 50 欄）。
2. **壓縮不佳**：一頁內混雜各種型別與值域，通用壓縮器找不到規律。
3. **CPU 低效**：tuple-at-a-time 的 Volcano 每筆都要解讀欄位位移、虛擬函數呼叫，分支多、快取差（第 5 章）。

13.2　列式與行式

sales

```
id date country amount
1 2026-01-03 TW 120
2 2026-01-03 TW 80
3 2026-01-04 JP 300
4 2026-01-04 TW 50
```

SELECT sum(amount) WHERE country = 'TW'

DSM 只讀 country 與 amount 兩個區段，且可在字典碼上比較。

```
NSM（列式，一頁連續存放整筆 tuple）：
1|2026-01-03|TW|120
2|2026-01-03|TW|80
```

```
DSM（行式，每欄一個區段）：
id: 1 2 3 4
date: 2026-01-03 ×2, 2026-01-04 ×2 （RLE）
country: TW TW JP TW → 字典 {TW:0,JP:1} → 0 0 1 0
amount: 120 80 300 50 → bit-packing (9 bits)
```

DSM（Decomposition Storage Model）把每個欄位獨立存放；同一列的各欄位值靠**相同**

的位置（position / row id）關聯，或者（較少見）每個值帶一個明確的 tuple id。好處對應上面三個問題：只讀需要的欄、同型別連續資料壓縮率高、整欄緊湊的陣列正適合向量化與 SIMD。代價是：**tuple 重建**（查詢需要多欄時要把各欄依位置拼回）與**單列更新昂貴**（要改 n 個區段）。

Abadi 等人（SIGMOD 2008）的實驗證明：把行式系統「假裝」成列式（垂直切表、每欄一個索引）仍慢一個數量級——關鍵不只是儲存格式，而是整個執行引擎為欄位設計：**壓縮**、**延遲物化**、**block iteration**（向量化）與 **invisible join**。

13.3　壓縮編碼

行式系統偏好**輕量級**（lightweight）編碼：解碼極快、甚至不必解碼就能運算，再視情況疊一層 LZ4/Zstd。

編碼	做法	適用	直接運算
字典（dictionary）	相異值編號，欄位只存整數碼；可**保序**（order-preserving）以支援範圍比較	低基數字串（國家、狀態）	等值 / 範圍比較在碼上進行；`GROUP BY` 直接以碼為鍵
RLE	連續相同值存成 (值, 起點, 長度)	排序欄、低基數欄	聚合直接乘長度；過濾以 run 為單位
Bit-packing / FOR	以實際值域所需位元數存（frame of reference：存最小值 + 偏移）	整數、小範圍數值	SIMD 一次解多個
Delta	存相鄰差值（再 bit-packing）	時間戳、遞增 id	需前綴和重建；常配區塊內的 base
Bitmap	每個相異值一個位元向量（Roaring 等壓縮位元圖）	極低基數（性別、布林）	AND/OR 直接合併位元圖；計數 = popcount
Null suppression	省略 null / 前導零	稀疏欄	—

典型壓縮比：事實表 5–10 倍（Parquet + Zstd 常見），且因為解碼便宜、讀取 I/O 大減，壓縮同時提升速度而非拖慢。選擇編碼通常由系統自動依取樣統計決定（DuckDB、Parquet writer 都會為每個 column chunk 選擇）。

13.4　排序、zone map 與跳塊

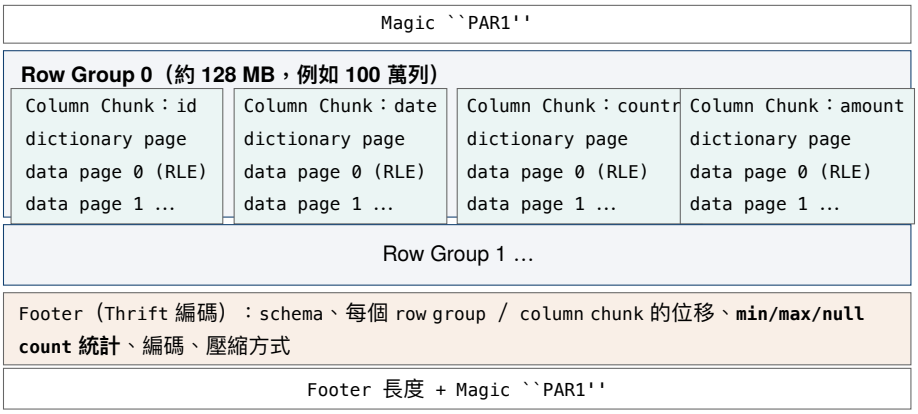
把資料依常用過濾欄位**排序**（ClickHouse 的 `ORDER BY`、Redshift 的 sort key、C-Store 的 projection）有雙重效果：排序欄的 RLE 效果極佳，且每個區塊的**最小 / 最大值（zone map / min-max 統計）**可以在不讀資料的情況下判斷整塊是否可能符合條件——**block skipping**。範圍查詢 `WHERE date BETWEEN ...` 在排序欄上只需讀少數區塊。PostgreSQL 的 BRIN 索引（第 3 章）就是 heap 上的 zone map。更進一步可以為一個表維護多種排序的副本（C-Store/Vertica 的 projections），以空間換查詢。

13.5　執行引擎

- **向量化**（第 5 章）：運算子一次處理一個欄位的 1k–4k 個值；過濾結果以 **selection vector**（通過者的位置陣列）或位元圖傳遞，不複製資料。
- **SIMD**：AVX2/AVX-512 一條指令比較 8–16 個 32 位元整數；字典碼、bit-packed 整數都能直接餵給 SIMD；編譯器對簡單緊湊迴圈能自動向量化，關鍵路徑常用 intrinsics 手寫。
- **延遲物化（late materialization）**：盡量只傳遞位置與必要欄位，到最後輸出（或 join 真正需要時）才取其他欄位——過濾選擇率低時省下絕大多數的重建工作。對照早期物化（一開始就拼成 tuple，然後跑行式引擎）。
- **Morsel-driven 平行**（Leis et al. 2014）：把輸入切成小塊（morsel，約 10 萬筆）由工作執行緒動態取用，NUMA-aware，負載均衡；DuckDB、HyPer/Umbra 採用。
- **聚合與 join**：雜湊聚合在欄位陣列上以分區（radix partitioning）提高快取局部性；星型 join 用 invisible join / bloom filter 先在事實表過濾，再查維度表。

13.6　Parquet、PAX 與 Arrow

純 DSM 把每欄放成一個檔案，讀一列要開 n 個檔；Ailamaki 等人（2001）的 **PAX**（Partition Attributes Across）折衷：**頁內**（或區塊內）按欄位分組存放，一個區塊仍包含同一批列的所有欄位。Apache Parquet 正是這個思路：



讀取器先讀 footer，依查詢需要的欄位與統計**只取相關的 column chunk 與 page**（projection + predicate pushdown），這是雲端物件儲存（S3）上以 HTTP range request 讀資料湖的基礎。ORC 結構類似（stripe / stream）。**Apache Arrow** 則是記憶體中的標準欄位格式（含 validity bitmap、offset buffers），讓 Pandas、DuckDB、Spark、Polars 之間零複製交換資料；Parquet 是磁碟格式、Arrow 是記憶體格式，兩者常搭配。Iceberg、Delta Lake、Hudi 等「表格式」在 Parquet 檔之上加入 manifest、快照與 ACID 提交協定，構成 lakehouse。

13.7　寫入與 HTAP

行式儲存對單列插入 / 更新不友善（要改每個欄位區段、且區塊通常不可變）。通用解法是 **delta store + merge**：

- 新寫入先進一個小的列式或記憶體結構（Vertica 的 WOS、SAP HANA 的 delta、ClickHouse 的新 part），查詢同時掃描主儲存與 delta 並合併。
- 背景定期把 delta 排序、壓縮、合併進主儲存（本質上就是 LSM 的 compaction——ClickHouse MergeTree 的名稱即由此而來）。
- 刪除與更新以**刪除位元圖（deletion vector）**或 tombstone 標記，merge 時實際移除（Iceberg/Delta 的 deletion vectors 亦同）。

HTAP（混合負載）系統通常維持**雙格式**：列式主副本服務 OLTP，透過複寫產生行式副本服務 OLAP（TiDB 的 TiFlash、Oracle Database In-Memory、SingleStore、AlloyDB 的 columnar engine）。

PostgreSQL 與分析負載

PostgreSQL 是純列式 heap 儲存，但有一組工具可以把中等規模（數百 GB 至數 TB）的分析做得不錯：**BRIN** 索引與依時間**分割表**提供跳塊效果、**平行查詢**（Parallel Seq Scan / Hash Join / Aggregate）、**物化視圖**、JIT 表達式編譯、`work_mem` 調大避免溢出。超過這個範圍，常見的選擇是：Citus columnar（`USING columnar` 存取方法，壓縮與欄位投影，但不支援更新與索引）、把資料輸出成 Parquet 由 DuckDB / `pg_duckdb` / `pg_analytics` 查詢，或以邏輯複寫 / CDC 同步到 ClickHouse、Snowflake、BigQuery 等專用系統。

13.8 系統速覽

系統	特色
MonetDB / X100	學術先驅：BAT（binary association table）、bulk 處理、X100 向量化（後成為 VectorWise/Actian Vector）
C-Store / Vertica	projections（多種排序副本）、WOS/ROS 讀寫分離、壓縮上直接運算
DuckDB	嵌入式（「分析界的 SQLite」）、向量 2048、morsel 平行、直接查 Parquet/CSV/Arrow、PostgreSQL 相容語法
ClickHouse	MergeTree：按主鍵排序的 parts、稀疏主索引（每 8192 列一個標記 granule）、極端的壓縮與向量化、適合即時分析與日誌
Snowflake	計算與儲存分離、micro-partition（50–500 MB、含 min/max）、自動叢集、多叢集虛擬倉儲
BigQuery / Dremel	巢狀資料的行式表示（repetition/definition level，Parquet 繼承）、無伺服器、數千節點平行
Redshift	基於 ParAccel（PostgreSQL 分支）的 MPP 行式倉儲，sort key / dist key
Druid / Pinot	即時攝取 + 倒排索引 + 行式 segment，面向低延遲的事件分析

MiniDB 連結：v6 的行式實驗

在 v6 的執行器加入一個 `ColumnChunk` 格式（每欄一個陣列 + null bitmap，區塊 2048 列）與兩個向量化運算子：`VectorFilter`（輸出 selection vector）與 `VectorSum`。對同一張 1000 萬列的表比較 `SELECT sum(amount) WHERE country = 'TW'` 在列式 Volcano 與行式向量化下的執行時間，再加上字典編碼與 zone map 跳塊，觀察每一步的收益。這個小實驗會比任何文字更能說明本章的論點。

本章練習

1. 一張 100 欄、每欄平均 8 bytes、10 億列的表。查詢只涉及 3 欄。估算列式與行式儲存各需讀取多少資料（未壓縮）；若行式平均壓縮 6 倍呢？
2. 欄位 `status` 有 4 個相異值、1 億列、未排序。比較字典 + bit-packing 與 bitmap 編碼的空間；若資料依 status 排序後再用 RLE 呢？
3. 解釋為什麼保序字典能支援 `WHERE name < 'M'`，但新值插入時可能需要重編碼；系統通常如何緩解？
4. 畫出 `SELECT c.name, sum(s.amount) FROM sales s JOIN customers c ON ... WHERE s.date >= '2026-01-01' GROUP BY c.name` 在延遲物化下各運算子傳遞的資料（位置？欄位？）。
5. Parquet 的 row group 取 128 MB 而非 8 KB 的理由是什麼？太大或太小各有什麼問題？
6. 為什麼 ClickHouse 的 MergeTree 與 LSM-Tree 有相同的名字成分？列出兩者的相似與相異處（至少各三點）。
7. （動手）用 DuckDB 把一個 CSV 轉成 Parquet（`COPY ... TO 'x.parquet'`），以 `parquet_metadata()` 觀察每個 column chunk 的編碼、壓縮大小與統計；再比較 `EXPLAIN ANALYZE` 中帶與不帶過濾條件時實際讀取的 row group 數。

延伸閱讀與資源

- D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, S. Madden, *The Design and Implementation of Modern Column-Oriented Database Systems*, Foundations and Trends in Databases 2013（本章最完整的延伸讀物）：`https://stratos.seas.harvard.edu/files/stratos/files/columnstoresfntdbs.pdf`
- D. Abadi, S. Madden, N. Hachem, *Column-Stores vs. Row-Stores: How Different Are They Really?*, SIGMOD 2008：`https://www.cs.umd.edu/~abadi/papers/abadi-sigmod08.pdf`
- M. Stonebraker et al., *C-Store: A Column-oriented DBMS*, VLDB 2005。
- A. Ailamaki, D. DeWitt, M. Hill, M. Skounakis, *Weaving Relations for Cache Performance*（PAX）, VLDB 2001。
- V. Leis, P. Boncz, A. Kemper, T. Neumann, *Morsel-Driven Parallelism*, SIGMOD 2014。
- M. Raasveldt, H. Mühleisen, *DuckDB: an Embeddable Analytical Database*, SIG-

MOD 2019；DuckDB 文件 *Internals*：https://duckdb.org/docs/current/internals/overview

- Apache Parquet 格式規範：https://parquet.apache.org/docs/file-format/；Apache Arrow 欄位格式：https://arrow.apache.org/docs/format/Columnar.html
- S. Melnik et al., *Dremel: Interactive Analysis of Web-Scale Datasets*, VLDB 2010。
- B. Dageville et al., *The Snowflake Elastic Data Warehouse*, SIGMOD 2016。
- ClickHouse 文件 *MergeTree*：https://clickhouse.com/docs/en/engines/table-engines/mergetree-family/mergetree
- CMU 15-445 Fall 2025 Lecture #5 *Storage Models & Data Layout*、#6 *Database Compression*；CMU 15-721 *Advanced Database Systems*（向量化、編譯、行式執行的研究所課程）。

第六篇

分散式資料庫

第 14 章　分散式資料庫基礎

本章學習目標

- 說明分散的三個動機（規模、可用性、延遲）與它帶來的根本困難（部分失效、不可靠網路、沒有全域時鐘）。
- 比較單領導者、多領導者、無領導者三種複寫架構，以及同步 / 非同步複寫的取捨與複寫延遲造成的異常。
- 設定並理解 PostgreSQL 的串流複寫與邏輯複寫（slot、hot standby、synchronous_standby_names、publication/subscription）。
- 解釋範圍分割、雜湊分割與一致性雜湊，以及次級索引、再平衡與熱點問題。
- 精確陳述 CAP 定理（Gilbert–Lynch 版本）、它常被誤解之處，以及 PACELC 的補充。
- 在 Jepsen 的一致性模型地圖上定位 linearizability、sequential、causal、eventual consistency 與 serializability。

14.1　為什麼要分散，以及為什麼困難

單機資料庫在三個方向上會碰到天花板：**規模**（資料量或吞吐超過一台機器）、**可用性**（一台機器當機就全停）、**延遲**（使用者分佈全球，光速往返 150 ms）。分散到多台機器解決這些問題，代價是必須面對分散式系統的三個根本事實：

1. **部分失效**：某些節點掛了、某些還活著，而你往往無法分辨「掛了」和「很慢」。
2. **不可靠的網路**：訊息可能延遲、遺失、重複、亂序；分割（partition）會發生。
3. **沒有全域時鐘**：各節點的時鐘有偏差（NTP 下常見數毫秒到數百毫秒），「先後」需要另外定義。

本章處理資料的擺放（複寫與分割）與一致性語意；第 15 章處理共識、第 16 章處理跨節點交易、第 17 章處理網路與儲存層。

14.2　複寫

複寫（replication）把同一份資料放在多個節點，目的為可用性、讀取擴展與地理延遲。難的是**寫入如何在複本間同步**。

14.2.1 單領導者（primary / standby）

所有寫入送到一個 **leader**，它把變更（WAL 或邏輯變更）串流給 **followers**；讀取可以分散到 followers。

- **同步複寫**：leader 等至少一個 follower 確認後才回報 commit——不會遺失已確認的資料，但 follower 慢或斷線會卡住寫入。實務上用「一個同步 + 其餘非同步」（semi-synchronous）或 quorum 式的同步（PostgreSQL 的 `ANY 2 (s1, s2, s3)`）。
- **非同步複寫**：leader 不等。吞吐與延遲最好，但 leader 當機時最後幾筆已確認的交易可能遺失（**RPO** > 0）。
- **Failover**：偵測 leader 失效（timeout）→ 選出新 leader（最新的 follower）→ 重新導向客戶端。危險：**腦裂（split brain）**——舊 leader 其實沒死，兩個 leader 同時接受寫入。需要 **fencing**（STONITH、fencing token、租約）與第 15 章的共識來安全地選主。

複寫延遲的異常（讀 follower 時）：

- 寫完馬上讀卻看不到（需 **read-your-writes**：寫後一段時間內讀 leader，或帶 LSN 等待 follower 追上）。
- 連續兩次讀結果「倒退」（需 **monotonic reads**：同一使用者固定讀同一個 follower）。
- 看到回答卻看不到問題（需 **consistent prefix reads**：因果相關的寫入保持順序）。

14.2.2 多領導者

多個資料中心各有 leader，彼此非同步互相複寫；或離線裝置（手機行事曆）各自是 leader。寫入延遲低、容忍資料中心斷線，但**寫入衝突**無可避免：同一列在兩地被同時修改。解法：避免衝突（依使用者固定路由到某個 leader）、**Last-Writer-Wins**（以時間戳決勝，會靜默丟資料）、保留所有版本交給應用程式合併、或使用 **CRDT**（可交換合併的資料型別）。PostgreSQL 16 的邏輯複寫加入 `origin = none` 選項，讓雙向複寫不會把收到的變更再傳回去（但衝突處理仍是使用者的責任；BDR/pgEdge 等產品補上這一塊）。

14.2.3 無領導者（Dynamo 風格）

Amazon Dynamo（2007）、Cassandra、Riak、DynamoDB：沒有 leader，客戶端（或協調節點）把寫入同時送到 N 個複本，W 個確認即成功；讀取送到 N 個，R 個回應即可。若 $W + R > N$，讀寫集合必定有交集，讀到的至少有一個最新版本（以版本號或向量時鐘判斷）。典型設定 $N = 3, W = 2, R = 2$。配套機制：**sloppy quorum 與 hinted handoff**（節點暫時不可達時先寫到別的節點，之後補回）、**read repair**（讀到舊版本時順手更新）、**anti-entropy**（背景以 Merkle tree 比對修復）。代價是一致性弱（即使 $W + R > N$ 也不是 linearizable，見 Kleppmann DDIA 第 5 章的反例）、衝突需要解決。

	單領導者	多領導者	無領導者
寫入衝突	無	有，需解決	有，需解決
寫入延遲	到 leader 的往返	本地 leader，低	到 W 個複本
一致性	可達 linearizable（同步 + 讀 leader）	最終一致	可調（R、W），通常最終一致
失效處理	failover 複雜、腦裂風險	容忍資料中心斷線	天然容忍節點失效
代表	PostgreSQL、MySQL、MongoDB、Kafka	多區 PostgreSQL（BDR）、CouchDB、行事曆同步	Dynamo、Cassandra、Riak、ScyllaDB

PostgreSQL 的複寫

串流複寫（physical）：standby 以 `pg_basebackup` 建立，透過 **walreceiver** 從 primary 的 **walsender** 持續接收 WAL 並重播（永不結束的復原，第 11 章）。

- **Replication slot**（`pg_create_physical_replication_slot`）：primary 保留 standby 尚未取走的 WAL——standby 斷線太久會塞爆 `pg_wal`（`max_slot_wal_keep_size` 設上限）。
- **Hot standby**：standby 可服務唯讀查詢；長查詢與 WAL 重播（如 VACUUM 清掉查詢仍需要的版本）衝突時，依 `max_standby_streaming_delay` 取消查詢，或以 `hot_standby_feedback` 讓 primary 延後清理。
- **同步**：`synchronous_standby_names = 'FIRST 1 (s1, s2)'` 或 `'ANY 2 (s1, s2, s3)'`，搭配 `synchronous_commit` 的等級（第 8 章）。
- **Failover 工具**：Patroni（以 etcd/Consul/ZooKeeper 做領導者選舉與 fencing）、repmgr、pg_auto_failover；`pg_rewind` 讓舊 primary 不必重建就能回頭當 standby；PostgreSQL 17 的 failover slot 讓邏輯複寫在切換後延續。
- **監　控**：`pg_stat_replication`（`sent_lsn / write_lsn / flush_lsn / replay_lsn`、`replay_lag`）。

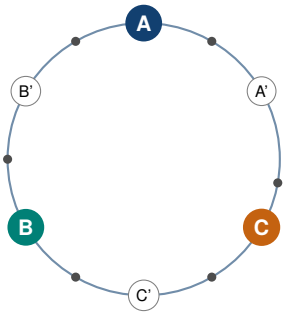
邏輯複寫（PostgreSQL 10+）：`CREATE PUBLICATION / CREATE SUBSCRIPTION`，透過邏輯解碼（`pgoutput`）傳送列層級的變更，可跨大版本、選擇性複寫部分表或列（row filter、column list），用於升級、資料整合、CDC（Debezium 使用 `pgoutput` 或 `wal2json`）。限制：不複寫 DDL、序列、大物件；初始同步大表耗時；衝突（如唯一鍵）會讓 subscription 停下來。

14.3　分割（sharding / partitioning）

當資料或寫入量超過單一節點，就必須把資料**切分**到多個節點（每個分片通常再複寫）。

- **範圍分割**：依鍵的區間（`[a, f)`, `[f, m)` …）。範圍查詢高效；但循序鍵（時間戳、自增 id）會把所有寫入打到同一分片——**熱點**。HBase、Bigtable、CockroachDB、TiDB、Spanner 採用（自動分裂過大的 range）。

- **雜湊分割**：對鍵取雜湊再分配。負載均勻；但範圍查詢要散播到所有分片。Cassandra、DynamoDB、Citus 預設採用。可以用**複合鍵**（雜湊第一部分、第二部分範圍）兼得：`(user_id, timestamp)`。
- **一致性雜湊**（Karger et al. 1997）：把雜湊空間視為環，節點與鍵都雜湊到環上，鍵歸屬順時針第一個節點；增減節點只需搬動 $1/N$ 的資料。搭配**虛擬節點**（每個實體節點在環上有數百個點）讓負載均勻。Dynamo、Cassandra、分散式快取採用。
- **固定分區數**：預先切成遠多於節點數的分區（如 Citus 預設 32 個 shard、Elasticsearch 的 shard），再平衡時整個分區搬移，不需重新雜湊。



實心黑點：鍵 $h(k)$ 落在環上，順時針找到第一個節點
A、B、C：實體節點；A'、B'、C'：虛擬節點，讓每個節點負責多個不連續區段
移除 B 時只有 B 負責的區段改由順時針的下一個節點接手

分割後的難題：

- **次級索引**：*本地索引*（每個分片索引自己的資料；查詢要 scatter-gather 所有分片）vs *全域索引*（依索引鍵分割；寫入要跨分片更新，需要分散式交易）。
- **跨分片查詢與 join**：盡量**共置（co-locate）**——以 tenant id 分割所有表，讓同租戶的 join 落在同一分片（Citus 的 distribution column）。
- **再平衡**：線上搬移分區而不中斷服務；避免「加一台機器就要重雜湊全部資料」。
- **路由**：客戶端、代理層或節點自己知道「鍵在哪」——元資料存在 etcd/ZooKeeper 之類的共識儲存。

14.4　CAP 定理

Brewer 2000 年的猜想、Gilbert 與 Lynch 2002 年的證明：在一個**非同步網路**中，分散式系統不可能同時保證

- **C**onsistency：**linearizability**（每個操作看起來像在某個瞬間原子地發生在單一副本上），
- **A**vailability：每個送到*未失效*節點的請求最終都得到（非錯誤的）回應，
- **P**artition tolerance：網路可以任意丟訊息。

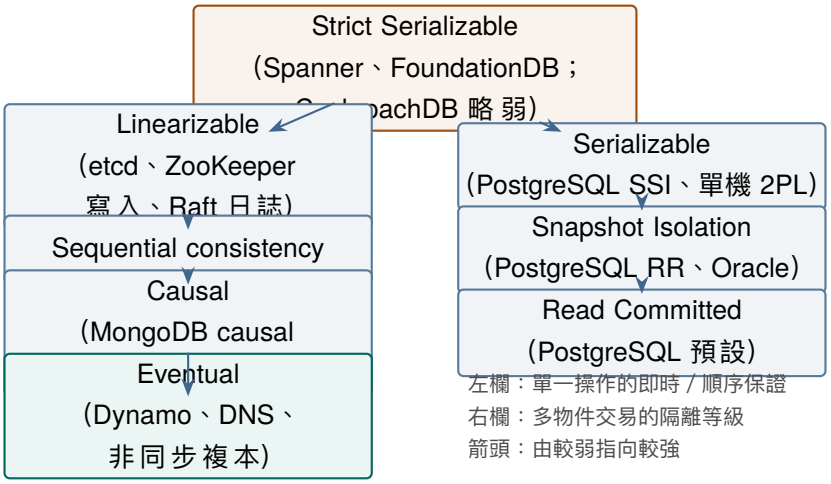
證明極簡單：分割成兩半後，一半收到寫入，另一半收到讀取；要「可用」就得回答，要「一致」就得回答新值，但訊息過不去——二選一。

CAP 常被誤解的地方

- **P 不是選項**：網路分割一定會發生，真正的選擇只在分割發生時的 C 或 A。「CA 系統」在分散式語境下沒有意義。
- CAP 的 C 是 linearizability（很強），A 是「每個非失效節點都要回應」（很特殊）。ACID 的 C、一般說的「高可用（99.99%）」都不是 CAP 的定義。
- 它只談一個暫存器的一種一致性與一種失效。Kleppmann（2015）的 *Please stop calling databases CP or AP* 指出：多數真實系統既不是 CAP 意義的 CP 也不是 AP。
- **PACELC**（Abadi 2012）更實用：**P**artition 時選 **A** 或 **C**；**E**lse（正常運作時）選 **L**atency 或 **C**onsistency。Dynamo/Cassandra 是 PA/EL；Spanner、CockroachDB 是 PC/EC；PostgreSQL 非同步複寫是 PA/EL、同步複寫偏 PC/EC。

14.5　一致性模型

「一致性」一詞在不同脈絡下有不同意思。Jepsen 的地圖把**單物件 / 多物件的順序保證**（分散式系統傳統）與**交易隔離等級**（資料庫傳統）放在同一張圖：



- **Linearizability**：所有操作有一個與真實時間一致的全序；一旦某個讀看到新值，之後的讀都看到。單物件、單操作的最強保證。實作需要共識或同步複寫；CAP 的 C。
- **Sequential**：有全序，且符合每個客戶端自己的程式順序，但不必符合真實時間。
- **Causal**：只對有因果關係的操作保序；可在分割下保持可用（是「可用系統能提供的最強模型」之一）。
- **Eventual**：停止寫入後，所有複本終將收斂。沒有任何關於「何時」與「中間看到什麼」的保證。
- **Serializability** 談的是多物件交易的等價序列順序，不談真實時間——PostgreSQL 的 SERIALIZABLE 允許「先 commit 的交易在序列化順序中排後面」。**Strict serializability** = serializability + linearizability，Spanner 的「external consistency」即此（CockroachDB 預設略弱，為 serializable 加上單鍵 linearizable）。

14.6　時間與順序

沒有全域時鐘，分散式系統以邏輯手段定義「先後」：

- **Lamport 時鐘**（1978）：每個節點一個計數器，送訊息時附上、收到時取 max 加一；若 $a \to b$（因果先於）則 $L(a) < L(b)$，反之不成立。
- **向量時鐘**：每個節點一個分量；能判定兩事件是因果相關還是**並行**（Dynamo 用來偵測衝突版本）。
- **混合邏輯時鐘（HLC）**：實體時間 + 邏輯計數器，接近牆上時鐘又保持因果；CockroachDB、YugabyteDB 以 HLC 做 MVCC 時間戳。
- **TrueTime**：Spanner 用 GPS + 原子鐘把時鐘誤差界定在 ϵ（約 7 ms）內，commit 時等待 ϵ（commit wait）換取 external consistency（第 16 章）。

14.7　現代分散式資料庫的架構分類

架構	做法	代表
Shared-nothing 分片 + 每分片共識	資料切成 range / hash 分片，每個分片一個 Raft/Paxos 群組，分散式交易跨分片	Spanner、CockroachDB、TiDB、YugabyteDB（NewSQL）
協調者 + worker 分片	在既有單機資料庫上加一層路由與分散式查詢規劃	Citus（PostgreSQL）、Vitess（MySQL）、ShardingSphere
Shared-storage / 計算儲存分離	單一寫入節點，WAL 送到分散式儲存層；儲存層負責複寫與頁面物化（「the log is the database」）	Aurora、Neon、AlloyDB、PolarDB
Dynamo 風格	無領導者、一致性雜湊、可調 quorum	Cassandra、ScyllaDB、DynamoDB、Riak
主從複寫 + 外部協調	傳統資料庫加 failover 管理器	PostgreSQL + Patroni、MySQL Group Replication / Orchestrator

MiniDB 連結：v7 複寫

v7 第一步是**單領導者串流複寫**：leader 的 `LogManager` 多一個 `WalSender` 把 WAL 記錄透過 TCP 送給 follower，follower 的 `WalReceiver` 寫入本地 WAL 並由 redo 程序套用（重用 v4 的復原程式碼），並開放唯讀查詢。提供 `sync_commit = on/off` 兩種模式，量測吞吐差異；再寫一個客戶端測試「寫入後立刻讀 follower」觀察 read-your-writes 問題，並用「附帶 LSN、等待 follower 追上」修好它。第 15 章會把選主交給 Raft。

本章練習

1. $N = 3$。列出 (W, R) 的所有組合，指出哪些滿足 $W + R > N$、各自的寫入 / 讀取可用性（可容忍幾個節點失效）與延遲特性。
2. 給一個 $W + R > N$ 但讀到舊值的具體交錯（提示：寫入只在部分節點成功後失敗、或

讀寫並行）。

3. 一個以自增 id 為主鍵的訂單表要分成 16 個分片。比較範圍分割、雜湊分割、`(customer_id hash, order_id range)` 三種設計在「寫入熱點」「查某客戶最近訂單」「查某時間區間所有訂單」上的表現。
4. 用 CAP 的精確定義分析：PostgreSQL 單 primary + 一個同步 standby，在 primary 與 standby 之間網路分割時，系統是 C 還是 A？若改為非同步呢？若 standby 自動升級為 primary 呢？
5.「我們的系統是最終一致的，所以可能會讀到舊資料」與「我們的系統是 causal consistent」對一個聊天應用的差別是什麼？舉出使用者可觀察到的現象。
6. 解釋為何 PostgreSQL 的 SERIALIZABLE 不是 strict serializable，並構造一個例子：交易 T_2 在真實時間上於 T_1 commit 之後才開始，卻在序列化順序中排在 T_1 之前。
7.（動手）用 Docker 建一個 primary + hot standby，在 standby 上執行長查詢同時於 primary 大量更新並 VACUUM，觀察 `max_standby_streaming_delay` 的取消行為；開啟 `hot_standby_feedback` 後再觀察 primary 的膨脹。

延伸閱讀與資源

- M. Kleppmann, *Designing Data-Intensive Applications*, 2nd ed. (2026), Ch. 5 *Replication*、Ch. 6 *Partitioning*、Ch. 9–10 *Consistency and Consensus*——本章與後三章的首選教材。
- S. Gilbert, N. Lynch, *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services*, SIGACT News 2002：`https://users.ece.cmu.edu/~adrian/731-sp04/readings/GL-cap.pdf`
- E. Brewer, *CAP Twelve Years Later: How the "Rules" Have Changed*, IEEE Computer 2012；D. Abadi, *Consistency Tradeoffs in Modern Distributed Database System Design*（PACELC）, IEEE Computer 2012。
- M. Kleppmann, *Please stop calling databases CP or AP*, 2015：`https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html`
- Jepsen, *Consistency Models*：`https://jepsen.io/consistency`；Jepsen 對各資料庫的分析報告：`https://jepsen.io/analyses`
- G. DeCandia et al., *Dynamo: Amazon's Highly Available Key-value Store*, SOSP 2007：`https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf`
- D. Karger et al., *Consistent Hashing and Random Trees*, STOC 1997。
- L. Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System*, CACM 1978。
- PostgreSQL 文件 Ch. 26 *High Availability, Load Balancing, and Replication*：`https://www.postgresql.org/docs/current/high-availability.html`；Ch. 29 *Logical Replication*：`https://www.postgresql.org/docs/current/logical-replication.html`；Patroni 文件：`https://patroni.readthedocs.io/`
- A. Verbitski et al., *Amazon Aurora: Design Considerations for High Throughput Cloud-*

Native Relational Databases, SIGMOD 2017。

- R. Taft et al., *CockroachDB: The Resilient Geo-Distributed SQL Database*, SIGMOD 2020。
- MIT 6.5840 *Distributed Systems*（講義、論文清單與 Raft/KV 實作 lab）：`https://pdos.csail.mit.edu/6.824/`

第 15 章　共識：Raft 與 Paxos

本章學習目標

- 定義共識問題的安全性與活性性質，說明 FLP 不可能結果的含意與實務上的繞法。
- 解釋複寫狀態機（replicated state machine）為何是共識最重要的應用形式。
- 走一遍單值 Paxos 的兩個階段並說明多數決交集為何保證安全。
- 完整描述 Raft：任期、領導者選舉、日誌複寫、commit 規則、選舉限制，以及 Figure 8 的微妙之處。
- 理解成員變更、快照、唯讀查詢的正確處理與 PreVote/CheckQuorum 等工程修正。
- 說明失效偵測、租約與 fencing token，以及它們在 PostgreSQL 高可用（Patroni + etcd）中的角色。

15.1　共識問題

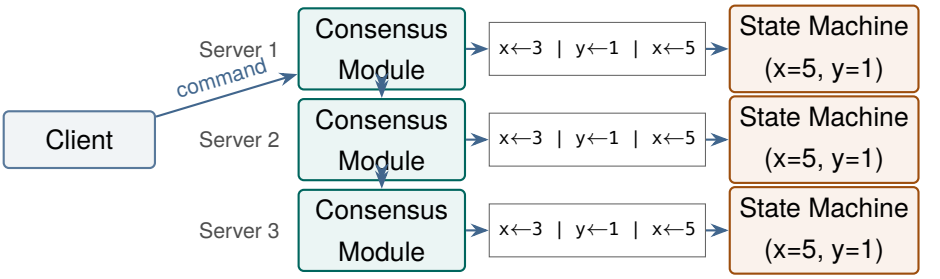
一群節點要對**一個值**（或一串值）達成一致，即使部分節點當機、網路延遲或分割。正確的共識演算法滿足：

- **Safety**：*Agreement*（所有決定的節點決定同一個值）、*Validity*（決定的值是某個節點提議過的）、*Integrity*（每個節點最多決定一次）。
- **Liveness**：*Termination*（所有未失效的節點最終都會決定）。

FLP 不可能結果（Fischer, Lynch, Paterson 1985）：在完全非同步的系統中，只要可能有一個節點當機，就沒有確定性演算法能保證終止。實務的繞法是假設**部分同步**（timeout 最終會可靠）並使用隨機化（Raft 的隨機選舉逾時）——演算法永遠保證安全性，活性則在網路「夠好」時成立。

共識、原子廣播（所有節點以相同順序收到相同訊息）、linearizable 的 compare-and-set 暫存器、以及領導者選舉在計算能力上互相等價。資料庫用它來：選主（避免腦裂）、複寫日誌（讓複本一致）、儲存叢集設定與分片對照表、實作分散式鎖。

15.2　複寫狀態機



共識模組確保每台伺服器的日誌內容與順序完全相同；確定性的狀態機套用相同日誌後必然處於相同狀態。

複寫狀態機（RSM）是共識在資料庫中的標準形態：客戶端命令先進入共識模組，共識保證所有伺服器的**日誌**一致，每台伺服器把已 commit 的日誌依序餵給**確定性**的狀態機（可以是 KV 儲存、資料庫的 WAL 套用器、設定儲存）。只要多數節點存活，系統就能持續服務，且任何存活節點的狀態都可以重建出來。

15.3　Paxos

Lamport 的 Paxos（1989 年寫成、1998 年發表、2001 年 *Paxos Made Simple*）是第一個被證明正確的實用共識演算法。角色：**Proposer**（提議值）、**Acceptor**（投票；通常 $2f+1$ 個，容忍 f 個失效）、**Learner**（得知結果）。單值（single-decree）Paxos 分兩階段，提案編號 n 全域唯一且遞增：

1. **Phase 1a Prepare**(n)：proposer 向 acceptors 送出編號 n。
   Phase 1b Promise：acceptor 若 n 大於它見過的任何編號，承諾不再接受編號 $< n$ 的提案，並回覆它已接受的最高編號提案 (n', v')（若有）。
2. **Phase 2a Accept**(n, v)：收到多數 promise 後，proposer 選值：若任何回覆帶有已接受的值，**必須採用其中編號最高的** v'；否則可用自己的值。
   Phase 2b Accepted：acceptor 若未承諾過更高編號，接受 (n, v) 並通知 learners。多數 acceptor 接受同一 (n, v) 即**選定（chosen）**。

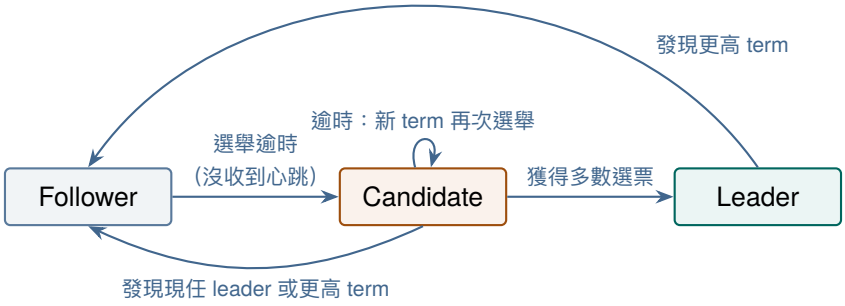
安全性的核心：任兩個多數集合必有交集。若值 v 已被選定（多數接受），之後任何 prepare(n') 的多數回覆中必有至少一個 acceptor 回報 v（或更新但同樣源自 v 的提案），proposer 被迫沿用 v，所以永遠不會選定第二個值。**活性問題**：兩個 proposer 輪流以更高編號互相搶先（dueling proposers）可能永不終止——解法是選出一個 **distinguished proposer**（leader），這就把問題帶回領導者選舉，但選舉本身不需要安全（錯選只影響活性）。

Multi-Paxos：對日誌的每個位置跑一次 Paxos 太貴；穩定的 leader 只需跑一次 Phase 1 涵蓋所有未來位置，之後每筆命令只需 Phase 2 的一輪往返。但論文沒有規定 leader 選舉、日誌補洞、成員變更、快照的細節，每個實作（Chubby、Spanner、Neon 的 safekeeper）都自行發明——Google 的 *Paxos Made Live* 詳述從論文到生產系統的鴻溝。這正是 Raft 誕生的動機。

15.4　Raft

Ongaro 與 Ousterhout（2014）以**可理解性**為設計目標：強領導者、把問題分解為選舉、日誌複寫、安全性三部分，並以 term 與日誌索引保持狀態簡單。

15.4.1　狀態與任期

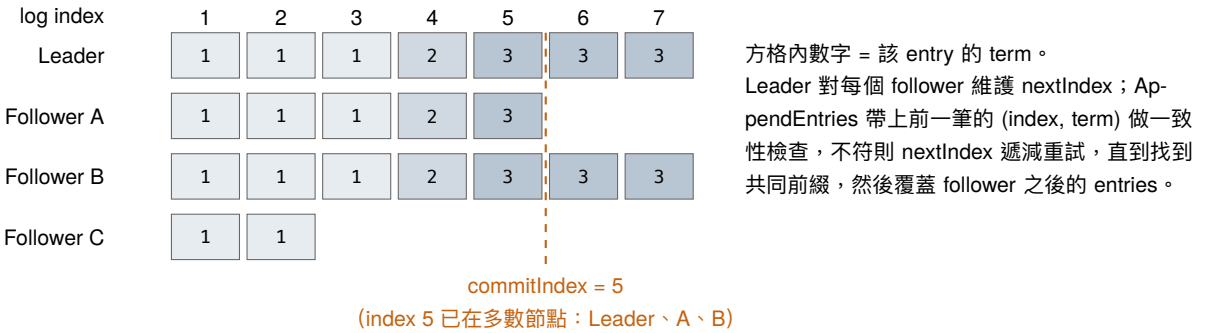


時間切成遞增的 **term**（任期），每個 term 以一次選舉開始，最多一個 leader。每個 RPC 都帶 term：收到更高 term 的節點立刻更新並退回 follower；收到較低 term 的請求則拒絕。Term 是 Raft 的邏輯時鐘，用來識別過期的 leader 與訊息。

15.4.2　領導者選舉

Follower 在**隨機化**的選舉逾時（如 150–300 ms）內沒收到 leader 的心跳（空的 `AppendEntries`），就自增 term、轉為 candidate、投自己一票並向所有節點送 `RequestVote`。每個節點在一個 term 內**只投一票**（先到先得，且須持久化），獲得多數票者成為 leader 並立刻發心跳。隨機逾時讓分票（split vote）很快自行化解。**Election Safety**：一個 term 最多一個 leader——因為多數票不可能同時給兩人。

15.4.3　日誌複寫



1. 客戶端命令到 leader，leader 追加到自己的日誌（持久化），平行送 `AppendEntries(prevLogIndex, prevLogTerm, entries, leaderCommit)` 給所有 followers。

2. Follower 檢查 `prevLogIndex` 處的 term 是否等於 `prevLogTerm`（**一致性檢查**）；是則追加（刪除衝突的後續 entries），回覆成功。
3. Entry 被**多數**節點存下後，leader 推進 `commitIndex` 並套用到狀態機、回覆客戶端；followers 從下一次 RPC 的 `leaderCommit` 得知並套用。

Log Matching 性質：若兩份日誌在某 index 的 entry 有相同 term，則它們在該 index 之前完全相同——由「leader 在一個 term 內對一個 index 最多建立一筆」加上一致性檢查歸納得到。

15.4.4 安全性：選舉限制與 commit 規則

- **選舉限制**：`RequestVote` 帶上 candidate 最後一筆 entry 的 (term, index)；投票者只在 candidate 的日誌**至少與自己一樣新**（最後 term 較大，或 term 相同而日誌較長）時才投票。因為已 commit 的 entry 存在於多數節點，而 candidate 需要多數票，所以新 leader 必定擁有所有已 commit 的 entries——**Leader Completeness**。Raft 的日誌只從 leader 流向 follower，leader 永不覆寫自己的日誌。
- **Commit 規則（Figure 8）**：leader **不能**僅因為「舊 term 的 entry 已複製到多數」就 commit 它——那個多數可能在之後被擁有更高 term 的 candidate 覆蓋。Leader 只透過計數 commit *當前 term* 的 entry；一旦當前 term 的某筆 entry commit，Log Matching 使其之前的所有 entry 也間接 commit。實務上 leader 上任時立刻追加一筆當前 term 的 no-op entry 以盡快 commit 前任遺留的 entries。
- **State Machine Safety**：若某節點在 index i 套用了某 entry，其他節點在 i 不會套用不同的 entry。

15.4.5 工程上必須處理的事

- **持久化**：`currentTerm`、`votedFor`、日誌必須在回覆 RPC 前 fsync——這是 Raft 的 WAL。
- **日誌壓縮**：狀態機定期產生**快照**（含 last included index/term），截斷之前的日誌；落後太多的 follower 以 `InstallSnapshot` 追上。
- **成員變更**：直接從舊設定切到新設定可能出現兩個不相交的多數。Raft 提供 **joint consensus**（過渡期需新舊兩組多數）或較簡單的**單節點變更**（一次只加減一台）。
- **客戶端語意**：命令可能已 commit 但回覆遺失，重送會重複執行；客戶端為每個命令帶唯一 id，狀態機去重，才有 linearizable 語意。
- **唯讀查詢**：leader 直接讀自己的狀態機可能是**過期讀**（它可能已被罷免而不自知）。正確做法：讀之前先與多數交換一輪心跳確認仍是 leader（ReadIndex），或使用**領導者租約**（假設時鐘漂移有界，在租約期內免心跳）；follower read 可透過 ReadIndex 向 leader 取得 commitIndex 後等待套用。
- **PreVote 與 CheckQuorum**（etcd）：被分割後重新加入的節點 term 已經很高，會把現任 leader 打下台（disruptive server）；PreVote 讓它先詢問「若我參選能否勝出」，CheckQuorum 讓失去多數聯繫的 leader 主動下台。
- **效能**：批次（一個 RPC 多筆 entries）、pipelining（不等回覆連續送）、平行 fsync 與網路傳送、leader 只需等多數即可 commit（慢的 follower 不拖累）。

15.5　Raft 與 Paxos 的比較

	(Multi-)Paxos	Raft
領導者	可選（提升效能），任何節點可提議	必須；所有寫入經由 leader
日誌	允許空洞，各 slot 獨立決定；需要補洞	嚴格連續，無空洞
選舉	論文未規定；實作各異	明確定義，附帶選舉限制
誰能當 leader	任何節點，之後補齊缺少的 entries	只有日誌最新的節點
理解與驗證	困難；多個生產系統有過 bug	設計目標即可理解性；有 TLA+ 規格與數十個實作
變體	Flexible Paxos（quorum 只需 $\|Q_1\|+\|Q_2\|>N$）、EPaxos（無領導者）、Mencius	etcd/raft、TiKV raft-rs、HashiCorp raft、Kafka KRaft

Howard 與 Mortier（2020）指出：撇開表述方式，兩者的核心機制幾乎同構，差別主要在 leader 選舉的策略與日誌是否允許空洞。其他家族成員：**Viewstamped Replication**（1988，比 Paxos 發表更早、與 Raft 極為相似）、**ZAB**（ZooKeeper 的原子廣播）。

15.6　失效偵測、租約與 fencing

- **心跳與逾時**：唯一的偵測手段，但無法區分「死了」「很慢」「網路斷了」。逾時太短 → 誤判與頻繁選舉；太長 → 失效後長時間不可用。
- **ϕ accrual 偵測器**（Hayashibara 2004）：以歷史心跳間隔的分佈輸出「懷疑程度」而非布林值，應用程式自選門檻；Cassandra、Akka 採用。
- **SWIM**（Das 2002）：隨機 ping 加間接 ping、以 gossip 傳播成員狀態，訊息量不隨節點數線性成長；Consul/Serf、memberlist 採用。
- **租約（lease）**：leader 在一段時間內被授權獨占；依賴「時鐘漂移有界」的假設。
- **Fencing token**：鎖服務每次授權附上遞增編號，儲存端拒絕較舊編號的寫入——即使持鎖者因 GC 暫停、網路延遲而「以為自己還持有鎖」也不會造成損害（Kleppmann 對 Redlock 的批評即在此：沒有 fencing 的分散式鎖無法保證安全）。

PostgreSQL 與共識

PostgreSQL 核心**沒有**內建共識：同步複寫保證資料不遺失，但不會自動選主，也不防止腦裂。高可用由外部工具補足——**Patroni** 把「誰是 primary」存在 etcd/Consul/ZooKeeper（統稱 DCS）的一把帶 TTL 的鎖中：primary 必須在 TTL 內持續更新租約，否則自我降級（`pg_ctl stop` 或 watchdog 重啟機器）；其他節點透過 DCS 競爭成為新 primary，並以 `pg_rewind` 讓舊 primary 回來當 standby。這是「以共識儲存做領導者選舉 + 租約 + fencing」的教科書應用。Neon 則更進一步，把 WAL 寫入一組以 Paxos 複寫的 **safekeepers**，計算節點無狀態、可任意重啟。

15.7　採用共識的系統

- **協調服務**：etcd（Raft；Kubernetes、Patroni 的後端）、ZooKeeper（ZAB；Kafka 舊版、HBase、Hadoop）、Consul（Raft）。
- **分散式資料庫（Multi-Raft）**：每個分片（range / region）一個 Raft 群組——TiKV（數萬個 region，心跳合併）、CockroachDB（leaseholder 處理讀取、follower reads 以 closed timestamp）、YugabyteDB。
- **Paxos 系**：Google Chubby、Spanner（每個 Paxos 群組一個 tablet）、Megastore、Azure Cosmos DB、Neon safekeepers。
- **訊息系統**：Kafka KRaft（2022 起取代 ZooKeeper 管理 metadata）、Pulsar/BookKeeper（quorum 寫入）。
- **MongoDB**：複本集選舉與複寫協定是 Raft 的變體（pull-based 複寫）。

MiniDB 連結：v7 以 Raft 複寫 WAL

把 v7 的「leader 串流 WAL」升級為 Raft：每筆交易的 WAL 記錄（或整個 commit group）作為一個 Raft entry，commit 後才回覆客戶端並套用到本地儲存引擎。實作順序建議依 MIT 6.5840 Lab 3：(A) 選舉與心跳、(B) 日誌複寫與 commit、(C) 持久化、(D) 快照。測試是重點：模擬網路分割、訊息延遲與重排、節點重啟，檢查「已回覆成功的寫入永不遺失」「任一時刻最多一個 leader 能 commit」兩個不變量。raft.github.io 的互動視覺化與 TLA+ 規格是除錯時的好朋友。

本章練習

1. 5 個 acceptor 的 Paxos：proposer P1 以 $n = 1$ 讓 A1、A2、A3 接受值 X；P2 以 $n = 2$ 向 A3、A4、A5 做 prepare。P2 會得到什麼回覆？它可以提議自己的值 Y 嗎？為什麼？
2. 畫出 Raft 論文 Figure 8 的情境（5 節點、S1 在 term 2 的 entry 被複製到多數後 S1 崩潰……），說明若 leader 以「舊 term entry 到達多數」為 commit 條件會如何違反安全性。
3. 3 節點 Raft 叢集，follower C 與其他節點分割 10 分鐘後重新連上。沒有 PreVote 時會發生什麼？PreVote 如何避免？
4. 為什麼 Raft 要求投票給「日誌至少一樣新」的 candidate，而 Multi-Paxos 允許任何節點當 leader？各自的代價是什麼？
5. Leader 收到唯讀請求 `GET x`，直接回傳本地狀態機的值。構造一個違反 linearizability 的情境；再說明 ReadIndex 與租約兩種修法的假設差異。
6. 一個 Raft 群組的寫入延遲由什麼決定？跨三個資料中心（往返 30 ms、80 ms）部署 5 節點時，leader 應放在哪裡？follower 數量增加會讓寫入更快還是更慢？
7. 設計 Patroni 式的 failover：primary 的租約 TTL 為 30 s、更新間隔 10 s。最壞情況下服務中斷多久？若 primary 的程序被 GC/磁碟 I/O 卡住 40 秒後恢復，會發生什麼？fencing 在哪一步生效？

延伸閱讀與資源

- D. Ongaro, J. Ousterhout, *In Search of an Understandable Consensus Algorithm (Extended Version)*, USENIX ATC 2014：`https://raft.github.io/raft.pdf`；Raft 官網（視覺化、實作清單、TLA+）：`https://raft.github.io/`；D. Ongaro 博士論文 *Consensus: Bridging Theory and Practice*, Stanford 2014（成員變更、客戶端互動、效能的完整處理）。
- L. Lamport, *Paxos Made Simple*, 2001：`https://lamport.azurewebsites.net/pubs/paxos-simple.pdf`；*The Part-Time Parliament*, ACM TOCS 1998。
- T. Chandra, R. Griesemer, J. Redstone, *Paxos Made Live: An Engineering Perspective*, PODC 2007。
- M. Fischer, N. Lynch, M. Paterson, *Impossibility of Distributed Consensus with One Faulty Process*, JACM 1985。
- H. Howard, R. Mortier, *Paxos vs Raft: Have We Reached Consensus on Distributed Consensus?*, PaPoC 2020：`https://arxiv.org/abs/2004.05074`；H. Howard et al., *Flexible Paxos: Quorum Intersection Revisited*, 2016。
- B. Oki, B. Liskov, *Viewstamped Replication*, PODC 1988；B. Liskov, J. Cowling, *Viewstamped Replication Revisited*, 2012。
- N. Hayashibara et al., *The ϕ Accrual Failure Detector*, SRDS 2004；A. Das, I. Gupta, A. Motivala, *SWIM*, DSN 2002。
- M. Kleppmann, *How to do distributed locking*, 2016（fencing token）：`https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html`
- *The Secret Lives of Data: Raft* 互動式教學：`https://thesecretlivesofdata.com/raft/`
- MIT 6.5840 Lab 3 *Raft*（Go 實作與測試框架）：`https://pdos.csail.mit.edu/6.824/labs/lab-raft.html`
- etcd raft 函式庫：`https://github.com/etcd-io/raft`；TiKV `raft-rs`：`https://github.com/tikv/raft-rs`；Patroni 文件 *Replication modes* 與 *Watchdog*。

第 16 章　分散式交易

本章學習目標

- 區分「原子提交」與「共識」兩個問題，說明為何 2PC 需要全體同意而 Paxos 只需多數。
- 完整描述兩階段提交（2PC）的訊息、日誌寫入點與每一種失效情境下的行為，理解它為何會阻塞。
- 說明 3PC 的改進與它在網路分割下失效的原因。
- 解釋現代系統如何用共識複寫協調者狀態（Paxos Commit、Spanner）、以 Percolator / parallel commit 減少往返，以及全域時間戳（TSO、HLC、TrueTime）如何提供跨分片的一致快照。
- 以 Saga 模式分解長交易，知道它放棄了什麼（隔離）以及補償、冪等、outbox 的配套做法。
- 正確使用 PostgreSQL 的 `PREPARE TRANSACTION`，並知道孤兒 prepared 交易的危害。

16.1　問題：跨節點的原子性與隔離

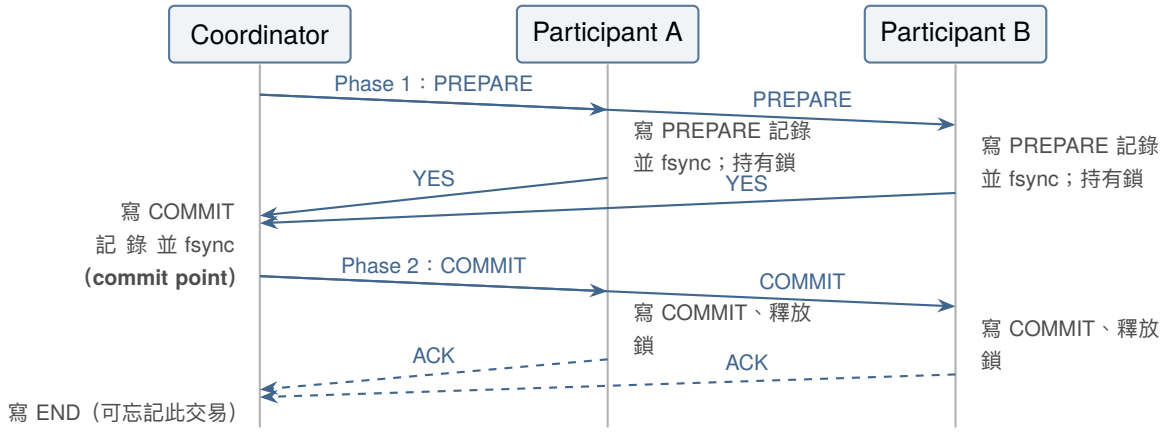
資料被分割到多個節點後，一筆交易可能同時修改分片 A 與分片 B（轉帳：A 扣款、B 入帳）。我們需要：

- **原子提交（atomic commit）**：所有參與節點要嘛全部 commit、要嘛全部 abort——即使有節點在中途當機。
- **跨分片隔離**：其他交易不能看到「A 已扣、B 未入」的中間狀態，也就是需要跨節點的一致快照或鎖。

原子提交 ≠ 共識

共識（第 15 章）：節點各自提議值，**多數**同意任一提議值即可，少數節點掛掉無妨。原子提交：每個參與者投「commit」或「abort」，**任何一票 abort 就必須 abort**，所以需要全部的票；一個參與者失聯就無法決定 commit。這個「需要全體」的本質讓 2PC 無法像 Paxos 一樣容錯——除非把每個參與者本身也用共識複寫（Spanner 的做法）。

16.2　兩階段提交（2PC）



1. **Phase 1（投票）**：協調者送 `PREPARE`。每個參與者把交易做到「只差 commit」——寫下所有修改與一筆 **PREPARE 日誌並 fsync**，之後即使當機也能 commit；然後回 `YES`（並承諾不再單方面 *abort*）或 `NO`。
2. **Phase 2（決定）**：全部 YES 則協調者寫 **COMMIT 日誌並 fsync**——這是整筆交易的 **commit point**；之後送 `COMMIT` 給所有參與者。任一 NO 或逾時則寫 ABORT 並送 `ABORT`。參與者執行、寫日誌、釋放鎖、回 `ACK`；協調者收齊後寫 END。

16.2.1　失效分析

情境	行為
參與者在投票前當機	協調者逾時 → ABORT。參與者重啟後沒有 PREPARE 記錄，自行 abort。
參與者投 YES 後當機	重啟後看到 PREPARE 記錄但沒有結果：**必須詢問協調者**（或其他參與者）才能決定；在此之前持續**持有鎖**。
協調者在寫 COMMIT 前當機	參與者處於 prepared 狀態**無限期等待**（**阻塞**）：它們不能 abort（已承諾）也不能 commit（不知結果）。鎖被持有，其他交易被擋住。這是 2PC 最大的弱點。
協調者在寫 COMMIT 後、送出前當機	重啟後讀日誌，重送 COMMIT。結果已確定，只是延遲。
網路分割	等同上述失聯情境；協調者與參與者都只能等待。

代價：正常路徑 2 輪往返、每個節點至少 2 次 fsync（參與者 PREPARE + COMMIT；協調者 COMMIT），鎖的持有時間延長至跨網路的兩個階段。**Presumed abort** 最佳化：協調者不記錄 abort 決定，查無記錄即視為 abort，省下 abort 路徑的日誌與 ACK。**唯讀參與者**可在 Phase 1 直接回 READ-ONLY 並退出。

PostgreSQL 的 PREPARE TRANSACTION

PostgreSQL 提供 2PC 的參與者端（協調者由外部交易管理器如 JTA/XA、Citus、應用程式負責）：

```
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
PREPARE TRANSACTION 'xfer-20261008-0001';   -- 交易狀態持久化到 pg_twophase/，釋放連線但保留鎖
-- 協調者收齊所有 YES 後：
COMMIT PREPARED 'xfer-20261008-0001';       -- 或 ROLLBACK PREPARED
```

預設 `max_prepared_transactions = 0`（功能關閉），因為**孤兒 prepared 交易**非常危險：它持有鎖、佔住 xmin 水位讓 VACUUM 無法清理、阻止 XID 凍結——一筆忘了 commit 的 prepared 交易可以在幾週後讓整個叢集因回繞保護而停機。使用時必須監控 `pg_prepared_xacts` 的 prepared 時間並設定告警。`postgres_fdw` 對遠端表的修改**不是** 2PC（各遠端各自 commit，可能部分成功）；Citus 的跨 shard 寫入則以 2PC 執行，並由 `pg_dist_transaction` 表在協調者重啟後復原懸而未決的交易。

16.3 三階段提交（3PC）

Skeen（1981）在 PREPARE 與 COMMIT 之間加入 **PRE-COMMIT** 階段：協調者收齊 YES 後先廣播「準備 commit」，收齊確認後才送 COMMIT。好處：任何存活的參與者只要看到 PRE-COMMIT 就知道所有人都投了 YES，可以在協調者當機時**自行推進**（不阻塞）；若沒人收到 PRE-COMMIT 則可安全 abort。代價：多一輪往返，而且它假設**失效只有當機、沒有網路分割**——分割時兩邊可能各自做出相反決定。因此 3PC 幾乎不被實際系統採用；真正解決阻塞的方法是把協調者的決定放進共識。

16.4 以共識消除阻塞

16.4.1 Paxos Commit 與 Spanner

Gray 與 Lamport（2006）指出：2PC 的阻塞源於協調者是**單點**。把「每個參與者的投票」與「協調者的決定」都以 Paxos 複寫到 $2f+1$ 個 acceptor，協調者當機後任何人都能從 acceptors 讀出已決定的結果——**Paxos Commit**。$f=0$ 時退化為 2PC。

Google **Spanner** 把這個想法產品化：每個分片（tablet）由一個 Paxos 群組複寫，分散式交易在 *Paxos 群組之間*跑 2PC，協調者是其中一個群組的 leader；參與者的 PREPARE 與協調者的 COMMIT 記錄都寫進各自的 Paxos 日誌，所以任何單一節點失效都不會阻塞。鎖以 2PL 管理（寫鎖），唯讀交易則用 **TrueTime** 時間戳走無鎖的快照讀。

16.4.2 Percolator：以資料本身當協調者日誌

Google Percolator（2010；TiDB/TiKV 採用）建在 Bigtable 的單列原子操作上，**不需要獨立的協調者日誌**：

1. 交易從 **TSO（timestamp oracle）**取得 `start_ts`，所有讀取以此快照進行，寫入先緩衝在客戶端。

2. **Prewrite**：選一個寫入鍵為 **primary**，對每個鍵寫入資料（版本 = start_ts）與一把**鎖**（secondary 的鎖指向 primary）；任一鍵已有衝突的鎖或更新的寫入則 abort。
3. 取得 `commit_ts`，**對 primary 寫入 commit 記錄並刪除其鎖**——這個單列原子操作就是 commit point。
4. 非同步地 commit 所有 secondary（寫 commit 記錄、刪鎖）。

讀者遇到 secondary 的鎖時，查看 primary：primary 已 commit 則幫忙把 secondary 補完（roll forward），primary 的鎖已逾時則清理（roll back）。協調者（客戶端）當機不會阻塞——狀態都在資料裡。TiDB 另有「async commit」與「1PC」最佳化縮短關鍵路徑。

16.4.3 CockroachDB 的 parallel commits

CockroachDB 每個 range 一個 Raft 群組；交易寫入先以 **write intent**（帶交易 id 的暫定版本）落到各 range，交易狀態存在一筆 **transaction record**。傳統做法要等所有 intent 寫完再把 record 改為 COMMITTED（兩輪共識延遲）。**Parallel commits**（2019）讓 record 進入 **STAGING** 狀態並列出所有 intent 的位置，與 intent 寫入*平行*進行；只要全部成功，交易就隱含地 commit 了——客戶端只需等一輪往返。之後非同步把 record 標為 COMMITTED、解析 intents。遇到 STAGING 記錄的讀者會自行驗證所有 intent 是否存在，以確定交易結果（status recovery）。

	2PC（XA）	Spanner	Percolator / CockroachDB
協調者狀態	單點日誌	Paxos 群組	寫在資料裡（primary / txn record），以 Raft 複寫
協調者失效	阻塞	不阻塞（群組選新 leader）	不阻塞（任何讀者可解析）
commit 延遲	2 輪往返 + 2 次 fsync	2PC over Paxos（每輪跨群組）+ commit wait	Percolator 2 輪；parallel commits 1 輪
隔離	依各參與者（通常 2PL）	Strict serializable（2PL + TrueTime）	SI（TiDB） / serializable（CockroachDB，HLC + 讀刷新）

16.5 跨分片的一致快照：全域時間戳

單機 MVCC 的快照靠一個遞增的 XID 計數器（第 9 章）；跨節點沒有這種東西，三種解法：

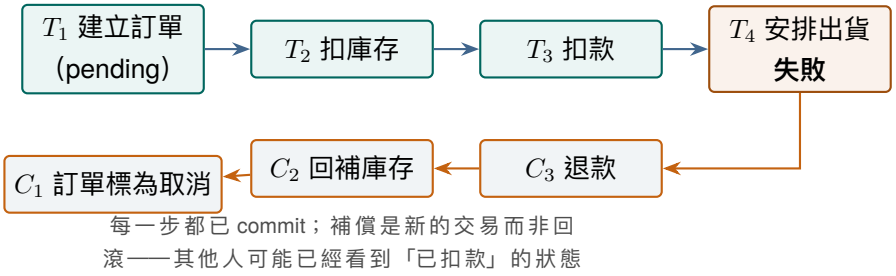
- **集中式 TSO**：一個（以 Raft 複寫、批次配發的）時間戳服務——TiDB 的 PD、Percolator、OceanBase。簡單、嚴格單調；跨地域部署時每筆交易多一次到 TSO 的往返。
- **HLC + 不確定區間**：CockroachDB / YugabyteDB 以混合邏輯時鐘配發時間戳，並假設時鐘偏差 ≤ `max_offset`（500 ms）。讀到時間戳落在「我的 ts 與 ts + max_offset 之間」的值時無法確定先後，必須**讀刷新（read refresh）**或重啟。不需特殊硬體，但只保證單鍵 linearizable、整體 serializable。
- **TrueTime**：Spanner 用 GPS/原子鐘把不確定區間縮到幾毫秒，commit 時**等待**區間過去

（commit wait），確保時間戳順序與真實時間一致——external consistency（strict serializability）。

Calvin（2012；FaunaDB 的基礎）走另一條路：**確定性**資料庫——先用共識把所有交易的輸入排成全序（sequencing layer），再讓每個分片以相同順序確定性地執行，執行期不需要 2PC（結果由順序決定，節點不會「改變主意」）。代價是交易的讀寫集合必須事先知道（或先做偵察讀），互動式交易不適用。

16.6　Saga：放棄隔離的長交易

Garcia-Molina 與 Salem（1987）為「跑數小時的長交易」提出 **Saga**：把交易拆成一串本地交易 $T_1, T_2, \ldots, T_n$，每個都立即 commit；為每個 T_i 定義**補償交易** C_i（語意上的反操作，如「取消訂位」而非「回滾」）。若 T_j 失敗，依序執行 $C_{j-1}, \ldots, C_1$（**向後復原**）；或者對可重試的步驟一直重試直到成功（**向前復原**）。微服務架構把它當作跨服務「交易」的標準模式。



Saga 保留原子性（最終全部完成或全部補償），但**沒有隔離**：步驟之間的中間狀態對外可見，可能出現 lost update（兩個 saga 交錯更新）、dirty read（讀到稍後被補償的狀態）。常見對策：**語意鎖**（訂單標為 pending，其他流程看到 pending 就等待或拒絕）、**可交換的更新**（加減而非設值）、**悲觀視角**（把風險高的步驟放最後）、**重讀驗證**。工程配套：每一步與補償都必須**冪等**（會被重送）、補償本身不能失敗（或需人工介入佇列）、以 **outbox pattern** 在同一個本地交易中寫入業務資料與待發事件，再由 CDC（PostgreSQL 邏輯解碼）或輪詢可靠地發出，避免「資料寫了、事件沒發」。協調方式分**編排（orchestration）**（一個 saga 協調器狀態機，如 Temporal、AWS Step Functions）與**編舞（choreography）**（各服務訂閱事件接力）。**TCC**（Try-Confirm-Cancel）是帶預留步驟的 Saga 變體，更接近 2PC 的語意但由應用程式實作。

16.7　選擇與實務建議

	2PC / XA	共識式提交	Saga
原子性	是	是	最終（透過補償）
隔離	是（鎖持有到 Phase 2）	是	**否**
協調者失效	阻塞	不阻塞	協調器狀態需持久化；可恢復
延遲	2 輪 + fsync	1–2 輪跨群組共識	多個獨立交易，總時間長但不持鎖
耦合	所有參與者需支援 XA、同時在線	需同一個資料庫系統	鬆耦合、跨異質服務
適用	同一組織內少數資料庫、短交易	NewSQL 內部	微服務、跨組織流程、長時間流程

最好的分散式交易是不需要的那一個

Pat Helland (2007) 的建議至今有效：以**實體 (entity)** 為單位設計資料，讓一筆業務操作盡量落在單一分片 / 單一資料庫的交易內（Citus 的 co-location、DynamoDB 的 single-table design 都是這個精神）；跨實體的操作以訊息與冪等處理達成最終一致。真的需要跨分片原子性時，優先選擇把它內建的系統（Spanner、CockroachDB、TiDB、YugabyteDB、Citus），而不是在應用層自己拼 XA。

MiniDB 連結：v7 的 2PC 實驗

在兩個 MiniDB 節點上實作 2PC：加入 `PREPARE TRANSACTION` 狀態（持久化 PREPARE 記錄並保留鎖）、一個簡單的協調者（含日誌）與 `COMMIT/ROLLBACK PREPARED`。測試：在協調者寫 COMMIT 記錄之前與之後分別 kill 它，觀察參與者的阻塞與復原；然後把協調者日誌改放到 v7 的 Raft 群組，證明阻塞消失。最後實作一個 Saga 範例（訂單 / 庫存 / 付款三個表）並寫測試重現它的隔離異常。

本章練習

1. 參與者 A 投了 YES 之後與所有人失聯 30 分鐘。它能否自行決定？如果它聯絡上另一個參與者 B 且 B 說「我投了 NO」呢？若 B 說「我投了 YES」呢？
2. 為什麼協調者「寫 COMMIT 日誌並 fsync」是 commit point，而不是「收到最後一個 YES」或「送出第一個 COMMIT」？
3. 計算一筆跨 3 個分片的交易在 2PC（協調者為其中一個分片）、Spanner 式 2PC over Paxos（每個群組 3 副本、跨機房往返 2 ms）、CockroachDB parallel commits 下的大致 commit 延遲（以往返次數與 fsync 計）。
4. Percolator 中，若客戶端在 primary commit 之後、secondary commit 之前當機，另一個讀者讀到 secondary 的鎖時如何處理？若在 primary commit 之前當機呢？鎖的「逾時」判斷為何需要 TSO？

5. 說明 Calvin 為何不需要 2PC；它如何處理「交易要先讀某值才知道要寫哪些鍵」的情況？
6. 把「使用者下單」設計成 Saga：列出步驟、補償、每一步的冪等鍵；指出哪兩個 saga 並行時會出現 lost update，並用語意鎖修正。
7. （動手）啟用 `max_prepared_transactions`，執行 `PREPARE TRANSACTION` 後中斷連線，觀察 `pg_prepared_xacts`、`pg_locks` 與其他 session 的阻塞；放著幾小時後觀察 `VACUUM VERBOSE` 的「dead but not yet removable」。

延伸閱讀與資源

- J. Gray, L. Lamport, *Consensus on Transaction Commit*, ACM TODS 2006：`https://lamport.azurewebsites.net/video/consensus-on-transaction-commit.pdf`
- J. Corbett et al., *Spanner: Google's Globally-Distributed Database*, OSDI 2012：`https://research.google/pubs/spanner-googles-globally-distributed-database-2/`
- D. Peng, F. Dabek, *Large-scale Incremental Processing Using Distributed Transactions and Notifications*（Percolator）, OSDI 2010；TiKV Deep Dive *Percolator*：`https://tikv.org/deep-dive/distributed-transaction/percolator/`
- N. VanBenschoten, *Parallel Commits: An Atomic Commit Protocol for Globally Distributed Transactions*, Cockroach Labs 2019：`https://www.cockroachlabs.com/blog/parallel-commits/`
- A. Thomson et al., *Calvin: Fast Distributed Transactions for Partitioned Database Systems*, SIGMOD 2012：`https://cs.yale.edu/homes/thomson/publications/calvin-sigmod12.pdf`
- H. Garcia-Molina, K. Salem, *Sagas*, SIGMOD 1987：`https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf`；C. Richardson, *Pattern: Saga*：`https://microservices.io/patterns/data/saga.html`
- P. Helland, *Life beyond Distributed Transactions: an Apostate's Opinion*, CIDR 2007。
- D. Skeen, *Nonblocking Commit Protocols*, SIGMOD 1981（3PC）。
- PostgreSQL 文件 *PREPARE TRANSACTION*：`https://www.postgresql.org/docs/current/sql-prepare-transaction.html`；Citus 文件 *Distributed Transactions*。
- P. Bernstein, E. Newcomer, *Principles of Transaction Processing*, 2nd ed., Ch. 8 *Two-Phase Commit*。
- CMU 15-445 Fall 2025 Lecture #22 *Distributed OLTP Database Systems*；M. Kleppmann, DDIA 2nd ed., Ch. 9（atomic commit 與 2PC）。

第 17 章　資料庫網路與分散式儲存

本章學習目標

- 比較 process-per-connection、thread-per-connection 與事件驅動的連線模型，說明 PostgreSQL 為何需要連線池。
- 讀懂 PostgreSQL 前後端協定（v3）的訊息格式、簡單查詢與擴充查詢流程、pipelining 與取消機制。
- 正確選擇 PgBouncer 的 pool mode、估算 pool 大小，並知道 transaction mode 下哪些功能會失效。
- 以往返次數思考延遲，設計 timeout、重試、退避與冪等鍵，處理「結果不明」的失敗。
- 說明 shared-nothing、shared-disk 與計算儲存分離三種架構，以及 Aurora / Neon 式「日誌即資料庫」、物件儲存為底層的設計。

17.1　連線模型

模型	做法	優缺點	代表
Process per connection	postmaster 為每個連線 `fork()` 一個 backend 程序，透過共享記憶體協作	隔離佳（一個崩潰不拖垮全部）、程式簡單；每連線數 MB 記憶體、建立成本高、數千連線後排程與快照計算開銷大	PostgreSQL、Oracle（dedicated server）
Thread per connection	每個連線一條執行緒	較輕；仍難以支撐數萬閒置連線	MySQL、SQL Server（可配合 worker pool）
Thread pool / 事件驅動	少量執行緒以非阻塞 I/O 服務大量連線	連線數與執行緒解耦；實作複雜	MySQL Enterprise thread pool、Redis、ScyllaDB（shard-per-core）

為什麼 PostgreSQL 的連線很貴

每個 backend 是獨立程序：私有記憶體（`work_mem` 是每個運算子、每個連線各自配置）、catalog 快取、prepared statement 快取都不共享；每次取得快照要掃描 `ProcArray`（所有活動連線），連線越多越慢（PostgreSQL 14 大幅改善，但仍非免費）；`max_connections` 也

決定鎖表等共享結構的大小。經驗法則：**活躍連線數**應接近 CPU 核心數的 2–4 倍，數千個應用程式連線必須經過連線池。

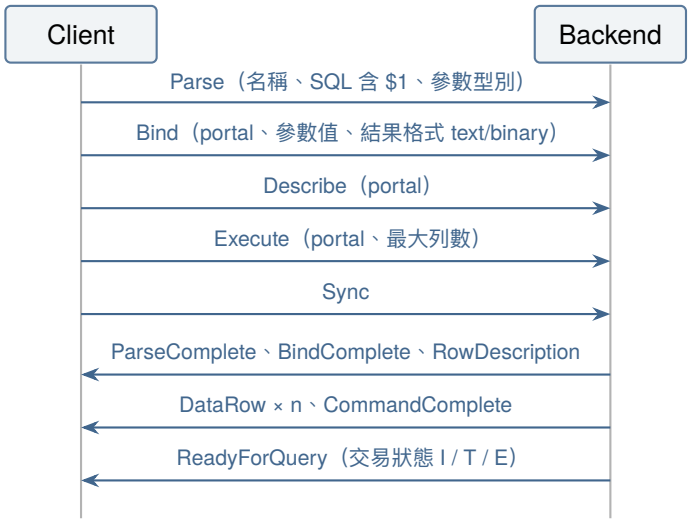
17.2　PostgreSQL 前後端協定

PostgreSQL 的 wire protocol 自 7.4 起為 **v3.0**，PostgreSQL 18 新增 **v3.2**（更長的取消金鑰、協商機制；libpq 預設仍以 3.0 連線）。所有訊息（除第一個 startup message）格式一致：**1 byte 訊息型別 + int32 長度（含自身）+ 內容**，整數為大端序，字串以 NUL 結尾。

17.2.1　連線建立

客戶端送 `StartupMessage`（協定版本、`user`、`database`、可選參數如 `application_name`）；可先送 `SSLRequest` 升級 TLS 或 `GSSENCRequest`。伺服器依 `pg_hba.conf` 回覆認證要求（`AuthenticationSASL` 走 SCRAM-SHA-256 的多輪交換、`AuthenticationMD5Password`、`AuthenticationOk`……，第 18 章），接著送多個 `ParameterStatus`（`server_version`、`client_encoding`、`DateStyle`……）、`BackendKeyData`（程序 id 與取消金鑰）與 `ReadyForQuery`。

17.2.2　簡單查詢與擴充查詢



簡單查詢：一個 `Query`（'Q'）訊息帶整個 SQL 字串（可含多個語句），回 RowDescription + DataRow + CommandComplete + ReadyForQuery。

擴充查詢：Parse 一次、Bind/Execute 多次；參數與 SQL 分離（天然防 SQL injection）；可要求二進位格式；`Sync` 之前的錯誤讓後續訊息被跳過直到下一個 Sync。

Pipelining（libpq 14+）：不等回覆連續送多組 Parse/Bind/Execute，以一個 Sync 結束，往返次數從 n 降到 1。

其他子協定：**COPY**（`CopyInResponse` 後客戶端送 `CopyData` 串流，批次載入最快的路徑）、**取消**（另開一條連線送 `CancelRequest` + 金鑰，因為原連線正忙）、**非同步通知**（`LISTEN/NOTIFY` 的 `NotificationResponse`）、**複寫協定**（`START_REPLICATION` 後以 `CopyBothResponse` 雙向串流 WAL，walreceiver 與邏輯複寫客戶端如 Debezium 都走這條）。協定是公開且穩定的，所以 PgBouncer、CockroachDB、YugabyteDB、QuestDB、Materialize、DuckDB 的 pg 伺服器模式等都「說 PostgreSQL 協定」以重用整個驅動生態。

17.3 連線池

應用程式端（HikariCP、pgx pool、SQLAlchemy pool）與**伺服器端**（PgBouncer、Odyssey、PgCat、Supavisor、RDS Proxy）兩層都常見。PgBouncer 的三種模式：

pool_mode	何時歸還伺服器連線	限制
`session`	客戶端斷線時	只省連線建立成本，不減少後端連線數
`transaction`	每筆交易結束時	跨交易的 session 狀態不可靠：`SET`（非 `SET LOCAL`）、session advisory lock、`LISTEN`、temp table、游標（`WITH HOLD` 除外）、`PREPARE`（PgBouncer 1.21+ 支援協定層的 prepared statement）
`statement`	每個語句結束時	不允許多語句交易；只適合 autocommit 的簡單負載

Pool 大小：連線並非越多越好。PostgreSQL wiki 與 HikarCP 的經驗公式 connections $\approx$ cores $\times 2 +$ effective spindles，再依實測調整——超過核心數太多只會讓所有查詢因為上下文切換與鎖競爭一起變慢。應用程式端 pool 總和應 $\leq$ 後端 pool 的 `default_pool_size`；等待 pool 的排隊要有 timeout，否則一次慢查詢會讓整個服務卡死。

17.4 延遲、RPC 與序列化

- **往返才是單位**：同機房往返約 0.2–0.5 ms、跨區域 30–150 ms；一個 ORM 的 N+1 查詢在 0.5 ms 往返下 1000 列就是 0.5 秒。對策：JOIN / `IN (...)` 合併查詢、批次寫入（多列 `INSERT ... VALUES`、`COPY`）、pipelining、把邏輯移到資料庫（CTE、函數）、prepared statement 省 parse。
- **格式**：PostgreSQL 協定可選文字或二進位（`timestamp`、`numeric`、`bytea` 二進位省大量轉換）；分散式資料庫內部節點間多用 gRPC/Protobuf（TiKV、CockroachDB）或自訂二進位 RPC。
- **TCP 細節**：`TCP_NODELAY`（關 Nagle，避免小訊息延遲）；`tcp_keepalives_idle/interval/count` 偵測死連線（特別是經過 NAT / 負載平衡器時）；TLS 握手 1–2 次往返（連線池攤平）；Unix domain socket 比 localhost TCP 快且可用 peer 認證。
- **尾端延遲**（Dean & Barroso 2013）：一個請求扇出到 100 個節點時，p99 的慢節點幾乎必然出現；分散式資料庫以 hedged request、超時重送到另一個複本、限制扇出來對抗。

17.5 Timeout、重試與冪等

伺服器端的保險絲

`statement_timeout`（單一語句上限）、`lock_timeout`（等鎖上限，遷移腳本必設）、`idle_in_transaction_session_timeout`（忘記 commit 的連線）、`idle_session_timeout`（PostgreSQL 14+）、`transaction_timeout`（PostgreSQL 17+，整筆交易上限）；客戶端另有 `connect_timeout`。沒有 timeout 的呼叫等於把整個系統的可用性交給最慢的那個依賴。

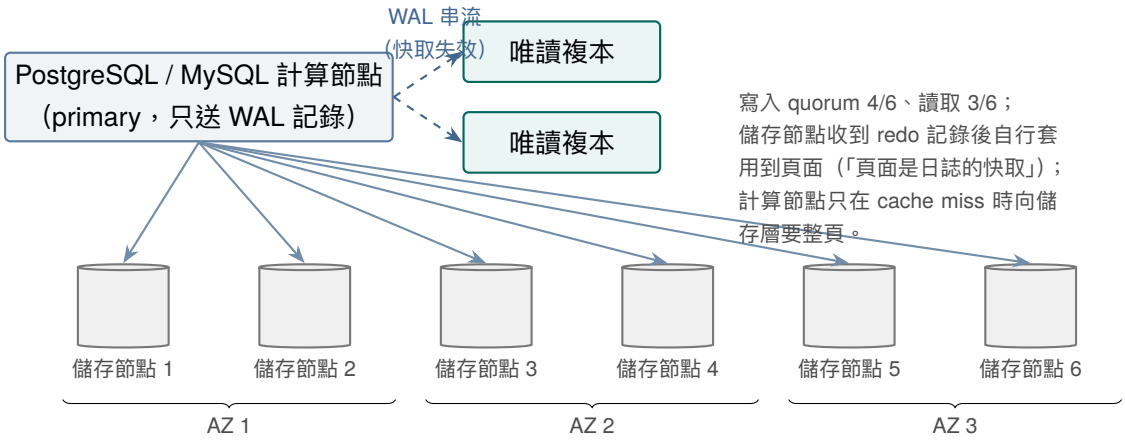
結果不明（ambiguous failure）：客戶端送出 `COMMIT` 後連線中斷——交易可能已 commit、也可能沒有。盲目重試會重複執行（重複扣款），不重試可能遺失。正確做法：

- 讓操作**冪等**：以客戶端產生的 **idempotency key** 為唯一鍵（`INSERT ... ON CONFLICT DO NOTHING`），重試時可安全重做；或把「這筆交易是否完成」寫成可查詢的狀態。
- 重試要**有上限、指數退避、加抖動（jitter）**，並區分可重試錯誤（`40001`、`40P01`、連線失敗）與不可重試錯誤（約束違反、語法錯誤）。
- 層層重試會放大流量（retry storm）：只在一層重試，搭配斷路器（circuit breaker）與負載卸除。
- 「exactly-once」在分散式系統中不存在於傳輸層，它永遠是「at-least-once + 去重」。

17.6 分散式儲存架構

架構	特徵	代表
Shared-nothing	每個節點擁有自己的磁碟與資料分片；靠網路交換；線性擴展、分割複雜	Citus、CockroachDB、TiDB、Cassandra、Teradata、Greenplum
Shared-disk	多個計算節點掛同一個（SAN / 叢集檔案系統）儲存；需要分散式鎖與快取一致性	Oracle RAC、IBM Db2 pureScale
計算儲存分離（disaggregated）	單一（或少數）寫入節點；儲存層是獨立的分散式服務，負責複寫、持久化、頁面物化；計算節點無狀態可快速擴縮	Aurora、Neon、AlloyDB、PolarDB、Snowflake（物件儲存）

17.6.1　日誌即資料庫：Aurora 與 Neon



Amazon **Aurora**（2017）的核心觀察：傳統主從複寫中網路成為瓶頸——primary 要送 WAL、整頁、double-write、備份等多份資料。Aurora 讓資料庫**只送 redo 日誌**到跨 3 個可用區、6 份複製的儲存服務（寫 4/6 quorum、讀 3/6，可容忍整個 AZ + 一個節點失效），儲存節點在背景把日誌套用成頁面、做 checkpoint、備份到 S3；網路 I/O 減少 7.7 倍，當機復原不需要重播（儲存層持續在做）。2018 年的後續論文說明它如何避免在 I/O 路徑上跑共識（以 epoch 與 quorum 讀取代替）。

Neon 以開源 PostgreSQL 做了類似拆分：**safekeepers**（Paxos 複寫的 WAL 服務，3 節點 quorum）接收 WAL 保證持久；**pageservers** 消化 WAL 成分層的不可變檔案（類 LSM），服務 `GetPage@LSN` 請求並把冷資料下沉到 S3；計算節點是修改過的無狀態 PostgreSQL。副產品是 **copy-on-write 分支與時間旅行**（任意 LSN 的資料庫分支幾乎零成本）與 scale-to-zero。

17.6.2　以物件儲存為底層

S3 類物件儲存提供近乎無限容量、11 個 9 的持久性、極低單價，但單次請求延遲 10–100 ms、只支援整物件寫入（無原地修改）。這些特性恰好與**不可變、大塊、循序**的資料結構相容：Snowflake 把表切成 micro-partition 存於 S3，計算節點用本地 SSD 當快取；LSM 的 SSTable 天然可以放在 S3（RocksDB-Cloud、SlateDB、Neon 的 layer file）；lakehouse 的 Parquet + Iceberg/Delta 更是以物件儲存為唯一真相。趨勢是「**S3 為持久層、本地 NVMe 為快取、記憶體為工作集**」的三層設計，以及 S3 Express / 條件寫入（compare-and-swap）帶來的新原語。

17.6.3　分散式檔案系統與複寫 vs 糾刪碼

GFS（2003） / HDFS 以 64–128 MB chunk、3 副本、append 為主的設計支撐了第一代大數據系統；Google 現今的 Colossus 以糾刪碼為主。**三副本**儲存成本 3×、修復只需複製；**糾刪碼（Reed–Solomon，如 6+3）**成本 1.5×、可容忍 3 個失效，但修復要讀多個分片並計算，小檔與隨機讀不友善。資料庫層通常不直接碰這些（交給雲端 EBS/S3），但理解它們有助於解讀「持久性 11 個 9」與「I/O 延遲抖動」的來源。

MiniDB 連結：v7 讓 psql 連上 MiniDB

實作 PostgreSQL 協定 v3 的子集合：startup、`AuthenticationOk`、`ParameterStatus`、`ReadyForQuery`、簡單查詢（`Q`）的 `RowDescription`、`DataRow`、`CommandComplete`、`ErrorResponse`，全部文字格式。完成後 `psql -h localhost -p 5433` 就能直接對 MiniDB 下 SQL，任何 PostgreSQL 驅動也能連——這是最有成就感的一個里程碑。連線模型用 thread-per-connection 即可；進階：加入擴充查詢（Parse/Bind/Execute）與 `statement_timeout`。

本章練習

1. 一個服務有 200 個 pod、每個應用程式 pool 10 條連線，後端 PostgreSQL 有 16 核。沒有 PgBouncer 時會發生什麼？設計一個兩層 pool 的配置並說明每個數字。
2. 列出三個在 PgBouncer transaction mode 下會出錯的 SQL 用法，並給出替代寫法。
3. 以 Wireshark / `tcpdump` 擷取一次 `psql` 查詢（關閉 SSL），辨認 StartupMessage、Authentication、Query、RowDescription、DataRow、ReadyForQuery 訊息與其長度欄位。
4. 跨區域往返 80 ms，應用程式要插入 10,000 列。比較逐筆 `INSERT`、多列 `INSERT`（每 500 列）、pipelining、`COPY` 的大致總時間。
5. 設計「建立付款」API 的冪等機制：鍵從哪裡來、存在哪裡、重試時如何回傳與第一次相同的結果、鍵要保留多久。
6. 比較 Aurora 的 4/6 quorum 與 Raft 3 節點多數決：各自容忍什麼失效？為什麼 Aurora 說它「避免了共識」？
7. （動手）寫一個 200 行以內的程式，接受 psql 連線並對任何查詢回傳固定的一列（Python 或 C++ 皆可），觀察 psql 需要哪些最少的訊息才能正常運作。

延伸閱讀與資源

- PostgreSQL 文件 Ch. 53 *Frontend/Backend Protocol*（訊息流程與格式的權威說明）：`https://www.postgresql.org/docs/current/protocol.html`；§32.5 *Pipeline Mode*（libpq）。
- PgBouncer 文件（*Features*、*Configuration*）：`https://www.pgbouncer.org/`；PostgreSQL Wiki *Number Of Database Connections*：`https://wiki.postgresql.org/wiki/Number_Of_Database_Connections`；HikariCP *About Pool Sizing*：`https://github.com/brettwooldridge/HikariCP/wiki/About-Pool-Sizing`
- A. Verbitski et al., *Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases*, SIGMOD 2017；*Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes*, SIGMOD 2018。
- Neon 文件 *Architecture overview*：`https://neon.com/docs/introduction/architecture-overview`；Neon 原始碼 `docs/` 目錄（safekeeper、pageserver 設

計）。

- S. Ghemawat, H. Gobioff, S.-T. Leung, *The Google File System*, SOSP 2003。
- B. Dageville et al., *The Snowflake Elastic Data Warehouse*, SIGMOD 2016；J. Zhou et al., *FoundationDB: A Distributed Unbundled Transactional Key Value Store*, SIGMOD 2021。
- J. Dean, L. Barroso, *The Tail at Scale*, CACM 2013。
- M. Brooker, *Timeouts, retries, and backoff with jitter*, Amazon Builders' Library：`https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/`
- M. Kleppmann, DDIA 2nd ed., Ch. 4 *Encoding and Evolution*、Ch. 8 *The Trouble with Distributed Systems*。

第七篇

安全與動手實作

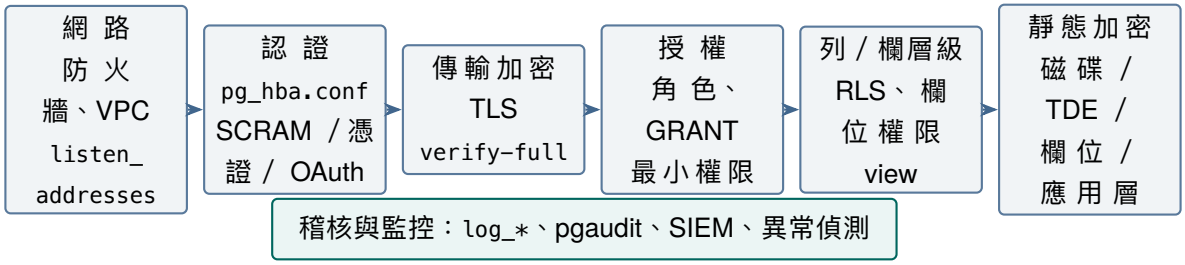
第 18 章　資料庫安全

本章學習目標

- 以縱深防禦的層次（網路 → 認證 → 授權 → 列 / 欄層級 → 加密 → 稽核）組織資料庫安全。
- 讀寫 `pg_hba.conf`，選擇正確的認證方法（SCRAM-SHA-256、憑證、Kerberos、OAuth）與 TLS 模式。
- 以角色、權限、預設權限、`search_path` 與 `SECURITY DEFINER` 的注意事項設計最小權限模型。
- 用 Row-Level Security 實作多租戶隔離，並知道其效能與繞過條件。
- 解釋 SQL injection 的機制與各種形式，並以參數化查詢、`format('%I'/'%L')`、最小權限徹底防範。
- 區分磁碟加密、TDE、欄位加密與應用層加密各自防禦的威脅，以及金鑰管理的角色。
- 設定稽核日誌（`log_*`、pgaudit），並列出一份可執行的強化檢查清單。

18.1　威脅模型與縱深防禦

資料庫是攻擊者的終極目標：它存著一切。常見攻擊面包括：暴露在網際網路的連接埠、弱或外洩的密碼、應用程式的 SQL injection、權限過大的帳號（應用程式用 superuser）、未加密的備份與日誌、內部人員、以及未修補的已知漏洞。沒有單一機制能擋住全部，**縱深防禦**讓每一層的失守都被下一層限制損害範圍：



兩個貫穿全章的原則：**最小權限**（每個主體只擁有完成工作所需的最少權限）與**預設拒絕**（沒有明確允許的都禁止）。

18.2 認證：pg_hba.conf

PostgreSQL 的**主機式認證**檔 `pg_hba.conf` 決定「誰、從哪裡、連哪個資料庫、用什麼方法」，**由上而下第一個符合的規則生效**，順序至關重要：

程式碼 18.1: ）]pg_hba.conf 範例（欄位：TYPE DATABASE USER ADDRESS METHOD [OPTIONS]）

```
# 本機維運：以 OS 使用者對應資料庫角色，不需密碼
local   all             postgres                                peer
# 應用程式：只允許來自應用子網、強制 TLS、SCRAM 密碼
hostssl appdb           app_rw          10.0.1.0/24             scram-sha-256
# 報表唯讀帳號：以客戶端憑證認證（CN 必須等於角色名）
hostssl appdb           reporting       10.0.2.0/24             cert clientcert=verify-full
# 複寫連線
hostssl replication     replicator      10.0.0.0/16             scram-sha-256
# 其他一律拒絕（明確寫出比依賴「沒有符合規則」更清楚）
host    all             all             0.0.0.0/0               reject
```

方法	說明
`trust`	不驗證。只能用在完全隔離的開發環境；許多入侵事件源於生產環境留下的 `trust`。
`peer` / `ident`	以 OS 使用者身分對應角色（`peer` 用於 Unix socket）。
`scram-sha-256`	PostgreSQL 10+，14 起為 `password_encryption` 預設。鹽化、迭代雜湊、挑戰回應，密碼不經網路傳送，伺服器只存驗證子（verifier）。
`md5`	舊方法，已不建議（可被重放、雜湊可離線破解）；僅為相容保留。
`cert`	客戶端 TLS 憑證；`clientcert=verify-full` 要求 CN 符合角色名（或以 `map` 對應）。適合服務對服務。
`gss` / `sspi`	Kerberos / Active Directory 單一登入，企業環境常見。
`ldap` / `radius` / `pam`	把密碼轉交外部系統驗證；注意 `ldap` simple bind 會把明文密碼送到 LDAP 伺服器，必須走 TLS。
`oauth`	**PostgreSQL 18 新增**：OAuth 2.0 device authorization flow，由 `oauth_validator_libraries` 指定的驗證模組檢查 token，可對接 Okta、Entra ID 等 IdP。

密碼管理：PostgreSQL 沒有內建複雜度規則（`passwordcheck` 模組可加一點），到期以 `VALID UNTIL` 設定；`CREATE ROLE ... PASSWORD` 的明文會出現在伺服器日誌與 `psql` 歷史——改用 `\password`（客戶端先雜湊）或直接提供 SCRAM verifier。雲端服務另提供 **IAM 認證**（短效 token 取代密碼，如 RDS IAM）。

18.3 傳輸加密：TLS

伺服器端 `ssl = on` 加上 `ssl_cert_file`、`ssl_key_file`（可選 `ssl_ca_file` 以驗證客戶端憑證）；`pg_hba.conf` 用 `hostssl` 強制加密、`hostnossl` 拒絕未加密連線。客戶端的 `sslmode` 決

定實際安全程度：

sslmode	竊聽防護	中間人防護	說明
disable / allow / prefer	否 / 視情況	否	prefer 是 libpq 預設：伺服器不支援就退回明文，攻擊者可降級
require	是	**否**	加密但不驗證伺服器身分
verify-ca	是	部分	驗證憑證由信任的 CA 簽發，不比對主機名
verify-full	是	**是**	驗證 CA 與主機名；**生產環境應使用**

PostgreSQL 17 的 `sslnegotiation=direct` 省去明文的 `SSLRequest` 往返；`ssl_min_protocol_version` 建議 `TLSv1.2` 以上。GSSAPI 也可提供加密（`hostgssenc`）。別忘了**複寫連線與 PgBouncer 到後端的連線**也要加密。

18.4 授權：角色與權限

PostgreSQL 統一以**角色（role）**表示使用者與群組（`LOGIN` 屬性區分）。角色可成為其他角色的成員並**繼承**權限（`INHERIT`，PostgreSQL 16 起可逐成員設定），或以 `SET ROLE` 切換。

程式碼 18.2: 最小權限的角色設計

```
-- 群組角色（NOLOGIN）承載權限；登入角色只是成員
CREATE ROLE app_readonly NOLOGIN;
CREATE ROLE app_readwrite NOLOGIN;
GRANT USAGE ON SCHEMA app TO app_readonly, app_readwrite;
GRANT SELECT ON ALL TABLES IN SCHEMA app TO app_readonly;
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA app TO app_readwrite;
GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA app TO app_readwrite;
-- 未來建立的物件也套用（由執行遷移的角色 migrator 建立）
ALTER DEFAULT PRIVILEGES FOR ROLE migrator IN SCHEMA app
  GRANT SELECT ON TABLES TO app_readonly;
ALTER DEFAULT PRIVILEGES FOR ROLE migrator IN SCHEMA app
  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO app_readwrite;
-- 登入角色：應用程式執行期、遷移、報表各自分開
CREATE ROLE app_rw LOGIN PASSWORD '...' IN ROLE app_readwrite CONNECTION LIMIT 200;
CREATE ROLE reporting LOGIN IN ROLE app_readonly;
-- 欄位層級：薪資欄只給 HR
GRANT SELECT (id, name, dept) ON app.employees TO app_readonly;
GRANT SELECT (salary) ON app.employees TO hr;
```

權限設計的常見陷阱

- **應用程式用 superuser 或物件擁有者連線**：一旦 injection 成功，攻擊者就能 `COPY ... TO PROGRAM`、讀取檔案、改任何表。執行期帳號應只擁有 DML；DDL 由獨立的遷移角色執行。

- **public schema**：PostgreSQL 15 之前所有人都能在 public 建物件，攻擊者可建同名函數劫持（CVE-2018-1058）。15 起預設撤銷 CREATE；舊叢集手動 REVOKE CREATE ON SCHEMA public FROM PUBLIC。
- **search_path 與 SECURITY DEFINER**：以定義者權限執行的函數若未固定 search_path，呼叫者可用自己 schema 的同名物件替換——務必 SET search_path = pg_catalog, pg_temp，並限制 EXECUTE 權限（函數預設 PUBLIC 可執行！）。
- **預定義角色**：pg_read_all_data、pg_write_all_data、pg_monitor、pg_signal_backend、pg_read_server_files、pg_execute_server_program 等——後兩者等同 OS 層級的存取，授予要極謹慎。
- **View 不是完美的安全邊界**：最佳化器可能把使用者的函數推到過濾條件之前執行而洩漏資料；需要 security_barrier view 與 LEAKPROOF 函數的概念（RLS 內建處理）。

18.5 Row-Level Security

RLS 讓資料庫強制「哪個角色能看到 / 修改哪些列」，常見於多租戶 SaaS：所有租戶共用表，但每個查詢自動加上租戶條件。

程式碼 18.3: 多租戶 RLS

```
ALTER TABLE orders ENABLE ROW LEVEL SECURITY;
ALTER TABLE orders FORCE ROW LEVEL SECURITY;      -- 連表的擁有者也受限制

CREATE POLICY tenant_isolation ON orders
  USING      (tenant_id = current_setting('app.tenant_id')::int)        -- 讀取 / 既有列的可見性
  WITH CHECK (tenant_id = current_setting('app.tenant_id')::int);       -- 寫入的新列必須符合

-- 應用程式在每筆交易開始時設定（transaction-scoped，連線池安全）
BEGIN;
SET LOCAL app.tenant_id = '42';
SELECT * FROM orders;      -- 自動只回傳 tenant 42 的列
COMMIT;
```

- Policy 可指定命令（FOR SELECT / INSERT / UPDATE / DELETE / ALL）與角色（TO role）；多個 **permissive** policy 以 OR 結合，**restrictive** policy（AS RESTRICTIVE）以 AND 疊加。
- 沒有任何 policy 符合時**預設拒絕**（看到空表）。Superuser 與具 BYPASSRLS 的角色不受限制；表擁有者除非 FORCE 也不受限。
- USING 條件會被加進每個查詢計畫，確保 tenant_id 有索引（通常是複合索引的前導欄）；policy 中的子查詢要留意效能。
- 比在應用程式每個查詢手寫 WHERE tenant_id = ? 可靠得多——漏寫一次就是跨租戶資料外洩。Supabase 等平台的 API 授權即建立在 RLS 上。

18.6 SQL Injection

機制：程式把使用者輸入拼接進 SQL 字串，使資料被當成程式碼執行。

程式碼 18.4: 脆弱的程式與攻擊輸入

```
# Python（錯誤示範）
query = f"SELECT * FROM users WHERE name = '{name}' AND password = '{pw}'"
# 輸入 name = "admin' --"  →  SELECT * FROM users WHERE name = 'admin' --' AND password = ''
# 輸入 pw = "' OR '1'='1"  →  條件恆真
# 輸入 name = "' UNION SELECT username, passwd FROM pg_shadow --"  →  讀出密碼雜湊（若權限足夠）
# Blind / time-based： name = "' AND (SELECT pg_sleep(5) WHERE substr(current_user,1,1)='p') --"
```

防禦（依重要性）：

1. **參數化查詢（prepared statement / bind parameters）**：SQL 結構與資料走協定的不同通道（第 17 章的 Parse/Bind），資料永遠不會被解析成 SQL。所有主流驅動與 ORM 都支援；**識別字（表名、欄名）與 `ORDER BY` 方向不能參數化**，必須白名單。
2. **動態 SQL 必須正確引用**：PL/pgSQL 用 `format('%I', ident)`（識別字）、`format('%L', literal)`（字面值）與 `EXECUTE ... USING $1`；應用層用 `quote_ident` / `quote_literal` 或驅動的 identifier 類別（如 psycopg 的 `sql.Identifier`）。
3. **最小權限**：即使注入成功，一個只有 DML 權限、受 RLS 限制、不能讀檔案的帳號能造成的損害有限。
4. **輸入驗證與輸出編碼**是輔助（防止邏輯錯誤與 XSS），不能取代前三項；WAF 只能攔截已知樣式。
5. 不要把資料庫錯誤訊息原樣回給使用者（洩漏 schema），並對 `pg_sleep` 等延時函數的呼叫保持警覺。

程式碼 18.5: 安全的動態 SQL（PL/pgSQL）

```
CREATE FUNCTION count_rows(tbl text, col text, val text) RETURNS bigint
LANGUAGE plpgsql SECURITY INVOKER SET search_path = pg_catalog, pg_temp AS $$
DECLARE n bigint;
BEGIN
  EXECUTE format('SELECT count(*) FROM %I WHERE %I = $1', tbl, col) INTO n USING val;
  RETURN n;
END $$;
```

18.7 靜態加密

層級	做法	防禦的威脅	不防禦的威脅
磁碟 / 卷加密	LUKS、雲端磁碟 KMS 加密、自加密硬碟	磁碟被偷、報廢硬碟、快照外流	任何能登入 OS 或資料庫的人；對執行中的系統透明
TDE（透明資料加密）	資料庫在寫頁面 / WAL 時加密；PostgreSQL 核心**尚無** TDE（Percona `pg_tde`、EDB、Cybertec 提供）	直接讀取資料檔與備份、不經資料庫的存取	透過 SQL 的存取、記憶體傾印；合規需求（PCI-DSS）常明文要求
欄位加密（`pgcrypto`）	`pgp_sym_encrypt(data, key)`；金鑰由應用程式傳入	有資料庫讀取權但沒有金鑰的人	金鑰若存在資料庫或日誌中則無效；加密欄無法索引 / 比較（確定性加密可等值查詢但洩漏重複）
應用層 / 信封加密	應用程式以資料金鑰加密、資料金鑰再由 KMS 主金鑰加密（envelope）	DBA、資料庫被完整拿走	應用程式伺服器被攻破；查詢能力大幅受限

- **密碼永遠不儲存明文，也不儲存可逆加密**：以 bcrypt / scrypt / Argon2 等慢雜湊加鹽（應用層，或 `pgcrypto` 的 `crypt(pw, gen_salt('bf', 12))`）。
- **金鑰管理**是加密的核心難題：金鑰與資料分離、定期輪替、以 KMS/HSM 保管、稽核金鑰使用。**Crypto-shredding**（刪除金鑰即等同刪除資料）是實作 GDPR「被遺忘權」於備份中的常用技巧。
- **備份、WAL 封存、日誌、`pg_dump` 輸出**與正式資料同等敏感，必須同樣加密並控管存取。
- 開發 / 測試環境應使用**去識別化**的資料（PostgreSQL Anonymizer 擴充套件可做靜態或動態遮罩）。

18.8 稽核與監控

- 內建日誌：`log_connections`、`log_disconnections`、`log_statement = ddl`（或 `mod`；`all` 的量與敏感資料風險很高）、`log_min_duration_statement`、`log_line_prefix` 含使用者、資料庫、來源位址、應用程式名稱。
- **pgaudit**：符合稽核需求的結構化記錄——session audit（依類別 READ/WRITE/DDL/ROLE/FUNCTION/MISC）與 object audit（針對特定表，透過授權給稽核角色來標記）；記錄的是實際執行的語句（含 prepared statement 的參數，若開啟）。
- 日誌要**送出資料庫主機**（SIEM / 集中式日誌），本機日誌可被同一個攻擊者刪除；設定保留期限與完整性保護。
- 監控異常：非上班時間的 superuser 登入、大量 `SELECT` 的匯出行為、失敗登入暴增、來自新 IP 的連線、`pg_stat_statements` 中突然出現的新查詢樣板。

PostgreSQL 強化檢查清單

1. `listen_addresses` 只綁必要介面；防火牆 / 安全群組只開給應用子網；絕不直接暴露 5432 到網際網路。
2. `pg_hba.conf`：沒有 `trust`、沒有 `0.0.0.0/0` 的允許規則、全面 `scram-sha-256` 或憑證、`hostssl` 強制 TLS；客戶端 `sslmode=verify-full`。
3. 應用程式帳號非 superuser、非擁有者、只有 DML；遷移、報表、監控、備份各用獨立角色；`CONNECTION LIMIT`。
4. `REVOKE CREATE ON SCHEMA public FROM PUBLIC`；函數明確 `REVOKE EXECUTE ... FROM PUBLIC`；`SECURITY DEFINER` 函數固定 `search_path`。
5. 全面參數化查詢；程式碼審查與靜態分析禁止字串拼接 SQL。
6. 多租戶啟用 RLS；敏感欄位以欄位權限或 view 限制。
7. 磁碟加密 + 必要時 TDE / 欄位加密；金鑰在 KMS；備份加密並定期演練還原。
8. 開啟連線日誌與 pgaudit，集中保存；設定告警。
9. 追蹤 PostgreSQL 安全公告，**定期套用 minor 版本更新**（含 CVE 修補）；擴充套件白名單。
10. `statement_timeout`、`idle_in_transaction_session_timeout` 等資源限制，降低 DoS 影響。

MiniDB 連結：v7 的最小安全功能

為 MiniDB 的 wire protocol 加入：(1) 使用者表與鹽化慢雜湊的密碼驗證（或實作 SCRAM-SHA-256 的子集合，直接相容 psql）；(2) `GRANT SELECT/INSERT ON table TO user` 與執行器中的權限檢查；(3) 擴充查詢協定的參數綁定，證明參數值永遠不經過 parser。寫一個測試：以 `' OR 1=1 --` 作為參數值查詢，確認它被當成純字串。

本章練習

1. 下列 `pg_hba.conf` 有什麼問題？`host all all 0.0.0.0/0 md5` 放在第一行、之後才是 `hostssl appdb app 10.0.1.0/24 scram-sha-256`。
2. 為什麼 `sslmode=require` 不能防止中間人攻擊？攻擊者具體怎麼做？
3. 一個 `SECURITY DEFINER` 函數內呼叫了未加 schema 的 `lower(text)`。說明攻擊者如何利用它取得定義者權限，以及兩種修法。
4. 設計一個 RLS 方案：客服人員可以看到自己負責區域的所有客戶、客戶只能看到自己、主管可以看到全部但不能修改。寫出 policy。
5. 以下 PL/pgSQL 是否安全？`EXECUTE 'SELECT * FROM ' || quote_ident(tbl) || ' WHERE id = ' || id_text;` 若 `id_text` 來自使用者呢？
6. 比較「整個資料庫用 TDE」與「只對信用卡號做應用層加密」在合規、效能、查詢能力、金鑰外洩影響上的差異。
7. （動手）安裝 pgaudit，設定 `pgaudit.log = 'write, ddl'`，執行一組操作後閱讀日誌

格式；再試 object audit 只稽核一張表的 SELECT。

延伸閱讀與資源

- PostgreSQL 文件 Ch. 20 *Client Authentication*（`pg_hba.conf` 與所有認證方法）：`https://www.postgresql.org/docs/current/client-authentication.html`；Ch. 21 *Database Roles*；§5.9 *Row Security Policies*：`https://www.postgresql.org/docs/current/ddl-rowsecurity.html`；§18.9 *Secure TCP/IP Connections with SSL*：`https://www.postgresql.org/docs/current/ssl-tcp.html`；§33.1.2 libpq *SSL Support*（sslmode 表）。
- PostgreSQL 安全公告與 CVE 清單：`https://www.postgresql.org/support/security/`
- pgaudit：`https://www.pgaudit.org/`；pgcrypto 文件：`https://www.postgresql.org/docs/current/pgcrypto.html`；Percona `pg_tde`：`https://docs.percona.com/pg-tde/`；PostgreSQL Anonymizer：`https://postgresql-anonymizer.readthedocs.io/`
- OWASP *SQL Injection Prevention Cheat Sheet*：`https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html`；OWASP Top 10。
- CIS *PostgreSQL Benchmark*（逐項的強化設定與檢查方法）：`https://www.cisecurity.org/benchmark/postgresql`
- PostgreSQL Wiki *A Guide to CVE-2018-1058: Protect Your Search Path*：`https://wiki.postgresql.org/wiki/A_Guide_to_CVE-2018-1058:_Protect_Your_Search_Path`
- R. Anderson, *Security Engineering*, 3rd ed.（免費線上版），威脅模型與系統性安全思維的基礎讀物。

第 19 章　動手打造 MiniDB

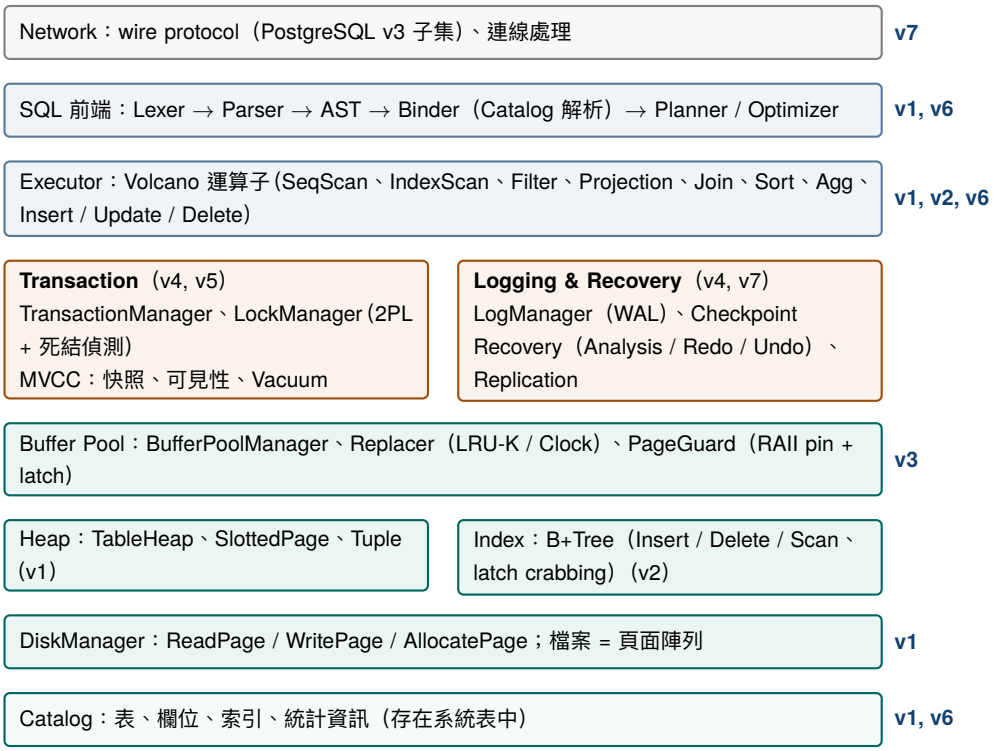
本章學習目標

- 以 C++20 從零建立一個單機關聯式資料庫，並依 v1 → v7 的里程碑逐步加入索引、緩衝池、交易、WAL、MVCC、最佳化器與複寫。
- 為每個版本定義明確的介面、驗收測試與「完成」的標準。
- 建立專業的工程工具鏈：CMake、GoogleTest、sanitizers、sqllogictest、當機注入測試、基準測試。
- 知道何時參考 BusTub、SQLite、PostgreSQL 的實作，以及各個教學專案的取捨。

19.1　為什麼要自己寫一個

前 18 章的每個概念——slotted page、latch crabbing、clock sweep、WAL 的兩條規則、可見性判定、Selinger DP——在實作它們之前都只是知識；實作之後才會變成直覺。MiniDB 的目標不是取代任何資料庫，而是一個約 10–15k 行、**你完全理解每一行**的系統。三個原則：**正確優先**（每個版本都有完整測試）、**小步前進**（每個版本都是可用的資料庫）、**抄最好的設計**（命名與架構盡量跟隨 PostgreSQL 與 BusTub，閱讀它們的原始碼時會輕鬆很多）。

19.2 架構總覽



19.3 專案佈局與工具鏈

程式碼 19.1: 目錄結構

```
minidb/
  CMakeLists.txt            # C++20, -Wall -Wextra -Werror, 可選 -fsanitize=address,undefined / thread
  src/
    common/                 # 型別、錯誤、config（PAGE_SIZE = 8192）、RID、Value
    storage/disk/           # disk_manager.{h,cc}
    storage/page/           # page.h、slotted_page.{h,cc}、b_plus_tree_page.{h,cc}
    storage/table/          # tuple.{h,cc}、table_heap.{h,cc}、table_iterator
    storage/index/          # b_plus_tree.{h,cc}、index_iterator
    buffer/                 # buffer_pool_manager.{h,cc}、lru_k_replacer / clock_replacer、page_guard.h
    catalog/                # catalog.{h,cc}、schema.h、column.h、statistics.h
    parser/                 # lexer.{h,cc}、parser.{h,cc}、ast.h
    binder/                 # 名稱解析、型別檢查 → bound statement
    planner/                # logical plan、optimizer（v6）、physical plan
    execution/              # executor_context、executors/*.cc、expressions/*.cc
    concurrency/            # transaction.h、transaction_manager.{h,cc}、lock_manager.{h,cc}
    recovery/               # log_record.h、log_manager.{h,cc}、checkpoint、recovery_manager.{h,cc}
    network/                # pg_protocol.{h,cc}、server.{h,cc}（v7）
    replication/            # wal_sender、wal_receiver、raft/（v7）
  test/                     # GoogleTest：每個模組一個 *_test.cc
  test/sql/                 # sqllogictest 格式的 .slt 檔
```

```
  tools/                  # shell (REPL) 、crash_test harness、bench
  third_party/            # googletest、fmt、 (可選) antlr 或 lemon
```

工具鏈建議：**CMake** + Ninja；**GoogleTest**；**clang-format** / clang-tidy；**AddressSanitizer + UBSan** 常開、**ThreadSanitizer** 跑並行測試；**libFuzzer** 對 parser 與 tuple 反序列化做模糊測試；`perf` / Instruments 看熱點。所有測試在 CI（GitHub Actions）上跑三種建置：Debug+ASan、Debug+TSan、Release。

19.4 v1：SQL 子集合、Heap File、Page Manager

目標：能 `CREATE TABLE`、`INSERT`、`SELECT ... FROM ... WHERE ...`（含兩表 join）、`UPDATE`、`DELETE`，資料持久化在單一檔案，重啟後仍在。

- **SQL 子集合與 parser**：手寫 lexer；陳述式用遞迴下降、表達式用 Pratt parser（運算子優先序）。支援型別 `INTEGER`、`BIGINT`、`DOUBLE`、`BOOLEAN`、`VARCHAR(n)`；運算子 `= <> < <= > >= AND OR NOT + - * /`；`NULL` 的三值邏輯從第一天就做對。先不要用 parser generator——手寫 parser 約 800 行，而且錯誤訊息更好。
- **Catalog**：`pg_class` / `pg_attribute` 的迷你版：兩張系統表存在固定頁面，啟動時載入成記憶體結構；表 id、欄位名稱 / 型別 / 位置、first page id。
- **Tuple 格式**：標頭（長度、null bitmap）+ 固定長度欄位 + 變長欄位的（offset, length）+ 變長資料區。`Tuple::GetValue(schema, col_idx)` 不需要反序列化整筆。
- **Slotted page**：第 2 章的設計，`PAGE_SIZE = 8192`；標頭含 LSN（為 v4 預留）、slot 數、free space 指標；`InsertTuple/GetTuple/MarkDelete/UpdateTuple`。
- **TableHeap**：頁面以雙向鏈結（prev/next page id）串起；插入時從最後一頁開始找空間（v6 可加 FSM）；`TableIterator` 依序走訪。
- **DiskManager**：以 `pread/pwrite` 讀寫第 n 頁；`AllocatePage` 從檔尾或 free list。v1 可以直接「每次讀寫都碰磁碟」或放一個最簡單的 map 快取，v3 再換成真正的 buffer pool。
- **Executor**：第 5 章的 Volcano 介面，`SeqScan`、`Filter`、`Projection`、`NestedLoopJoin`、`Insert`、`Update`、`Delete`、`Values`。Planner 先是「規則式」：AST → 運算子樹的直接對應。
- **REPL**：`minidb shell db.file`，支援 `\d`、`\dt`。

程式碼 19.2: v1 的核心介面

```cpp
struct RID { page_id_t page_id; uint32_t slot; };

class DiskManager {
 public:
  explicit DiskManager(const std::string &path);
  void ReadPage(page_id_t id, char *out);        // 讀 PAGE_SIZE bytes
  void WritePage(page_id_t id, const char *data); // 不保證 fsync；Sync() 另外呼叫
  page_id_t AllocatePage();
```

```
  void Sync();
};

class TableHeap {
 public:
  std::optional<RID> InsertTuple(const Tuple &t, Transaction *txn);
  std::optional<Tuple> GetTuple(RID rid, Transaction *txn) const;
  bool MarkDelete(RID rid, Transaction *txn);
  bool UpdateTuple(const Tuple &t, RID rid, Transaction *txn);  // 放不下則 delete + insert
  TableIterator Begin(Transaction *txn);  TableIterator End();
};

class AbstractExecutor {
 public:
  virtual void Init() = 0;
  virtual bool Next(Tuple *tuple, RID *rid) = 0;
  virtual const Schema &GetOutputSchema() const = 0;
  virtual ~AbstractExecutor() = default;
};
```

驗收：100 個 sqllogictest 案例通過（含 NULL 語意、字串比較、join 正確性）；插入 10 萬列後重啟能全部讀回；ASan 無報告。

19.5 v2：B+Tree 索引

目標：`CREATE INDEX idx ON t(col)`；等值與範圍查詢走索引；索引與表同步維護。

- 頁面式 B+Tree（第 3 章）：`BPlusTreeHeaderPage`（root id）、`InternalPage`（n 個 key、$n+1$ 個 child）、`LeafPage`（key、RID、next leaf id）。階數由 `PAGE_SIZE` 與鍵長算出；先支援固定長度鍵（整數、定長字串），再加變長鍵。
- API：`Insert(key, rid)`、`Delete(key)`、`GetValue(key)`、`Begin(key)` / `Begin()` 回傳 `IndexIterator` 做範圍掃描。
- 分裂與合併 / 重分配完整實作；根節點的升降；空樹與單一葉節點的邊界。
- `IndexScanExecutor` 與 planner 規則：`WHERE col = ?` 或範圍條件且 `col` 有索引時使用；`Insert/Update/Delete` 執行器同步更新所有索引。
- 並行（可延後到 v3 之後）：latch crabbing（讀：一路 read latch；寫：樂觀先用 read latch，發現需分裂再重來用 write latch）。

驗收：以 `std::map` 為 oracle 的隨機插入 / 刪除 / 查詢 100 萬次對照測試；重啟後索引可用；（並行版）TSan 乾淨，多執行緒混合讀寫結果與單執行緒一致。這與 CMU 15-445 Project #2 幾乎等價，可參考其測試設計。

19.6 v3：Buffer Pool

目標：所有頁面存取經過 `BufferPoolManager`；可以用 64 個 frame 跑 1 GB 的資料庫；多執行緒安全。

- `FetchPage / NewPage / UnpinPage(is_dirty) / FlushPage / DeletePage`；page table（`unordered_map` + mutex，進階：分區）；free list。
- **Replacer**：先做 Clock（第 4 章），再做 LRU-K（BusTub 的選擇）；可淘汰的條件是 pin count = 0。
- **PageGuard**：RAII 物件在解構時 unpin（並釋放 latch）——這會消滅一整類「忘記 unpin」的 bug。`ReadPageGuard` / `WritePageGuard` 同時取得頁面 latch。
- 把 v1、v2 所有直接呼叫 `DiskManager` 的地方換掉；加入命中率統計；`EXPLAIN (BUFFERS)` 風格的輸出。
- 進階：背景寫出執行緒；disk scheduler（I/O 佇列由專用執行緒處理，BusTub Project #1）；掃描用 ring buffer 避免沖掉熱資料。

驗收：pool size 遠小於資料量時所有 v1/v2 測試仍通過；pin 計數在每個測試結束時為 0；並行壓測無死鎖、TSan 乾淨。

19.7 v4：交易、WAL 與復原

目標：`BEGIN / COMMIT / ROLLBACK`；`kill -9` 後重啟不遺失已 commit 的交易、未 commit 的交易消失；多個交易並行正確。

- `Transaction`（id、狀態、持有的鎖、undo 用的寫入集合、prevLSN）；`TransactionManager::Begin/Commit/Abort`。
- `LockManager`：表鎖與列鎖，模式 S/X（進階 IS/IX/SIX），strict 2PL，等待圖死結偵測執行緒（第 10 章）。先用表級鎖讓一切正確，再細化。
- `LogManager`（第 8 章）：記憶體緩衝、背景 flush、group commit；`LogRecord` 型別：BEGIN/COMMIT/ABORT/INSERT/DELETE/UPDATE/NEW_PAGE/CLR/CHECKPOINT。頁面標頭的 LSN 開始發揮作用：寫出任何頁面前，先確保該頁 LSN 之前的日誌已落盤（在 `FlushPage` 內呼叫 `LogManager::Flush`）。
- `RecoveryManager`：ARIES 三階段（第 11 章），或在 v5 之後改為 redo-only。Checkpoint 以背景執行緒定期進行。
- 隔離等級：此版本以 2PL 提供 SERIALIZABLE（讀也要鎖），v5 再換成 MVCC。

驗收：**當機注入測試**——測試程式 `fork()` 出資料庫程序執行隨機轉帳負載，在隨機時刻 `SIGKILL`，重啟後驗證「總額守恆」與「所有已回覆成功的交易存在」，重複 500 次；死結測試中恰好一方被中止；`ROLLBACK` 後資料與索引一致。

19.8 v5：MVCC

目標：讀不阻塞寫、寫不阻塞讀；提供 READ COMMITTED 與 REPEATABLE READ（snapshot isolation）；垃圾回收。

- Tuple 標頭加 `xmin`、`xmax`、`next_version`（RID）；UPDATE 走 append-only + 版本鏈（第 9 章）。
- `Snapshot{xmin, xmax, active_xids}` 由 `TransactionManager` 產生；RC 每語句一個、RR 每交易一個；`IsVisible(tuple, snapshot, txn)` 實作第 9 章的規則。
- 寫寫衝突：等待對方結束；RR 下對方 commit 則回報 serialization failure；RC 下重新讀最新版本並重新評估條件。
- 索引：指向所有版本，掃描時過濾可見性（PostgreSQL 做法）；或實作 HOT 的簡化版。
- `VACUUM`：找出對所有活動快照都不可見的死版本回收 slot；維護最老活動快照的 xmin 水位。
- 進階：SSI 的簡化版（追蹤 rw-antidependency、偵測危險結構）。

驗收：以 Hermitage 的測試腳本（dirty read、non-repeatable read、phantom、lost update、write skew、read skew）逐一驗證每個等級的行為與你宣稱的一致；長時間負載下資料庫檔案大小在 VACUUM 後穩定。

19.9 v6：成本式最佳化器與更多運算子

目標：`ANALYZE`、`EXPLAIN [ANALYZE]`；多表查詢自動選擇 join 順序與演算法；TPC-H 子集（scale factor 0.01）可在合理時間完成。

- 統計：每欄 `n_distinct`、null 比例、100-bucket 等高直方圖、MCV；表的列數與頁數（存進 catalog）。
- 邏輯重寫：述詞與投影下推、常數摺疊、簡單的子查詢去相關（`IN` → semi-join）。
- 成本模型與選擇率（第 6 章公式與 PostgreSQL 常數）；存取路徑選擇（seq scan vs index scan）。
- Join 排序：左深 Selinger DP（≤ 8 表），保留 interesting order；演算法選擇 NLJ / index NLJ / hash join / sort-merge。
- 新運算子：`HashJoin`（含溢出到磁碟的 Grace 版本）、`Sort`（外部合併排序，`work_mem` 限制）、`HashAggregate` / `SortAggregate`、`Limit`、`TopN`。
- `EXPLAIN ANALYZE` 印出估計與實際列數——這是驗證統計與選擇率的工具。
- 可選的行式實驗（第 13 章）：`ColumnChunk` + 向量化 Filter/Sum。

驗收：TPC-H Q1、Q3、Q5、Q6、Q10 結果正確；對照「強制 NLJ」的計畫，最佳化器選出的計畫至少快 10 倍；估計列數在 2 倍誤差內的節點超過 80%。

19.10 v7：網路與複寫

目標：`psql` 可以直接連線；一個 primary 把 WAL 串流給一個 standby，standby 可服務唯讀查詢；（進階）以 Raft 做自動選主。

- PostgreSQL 協定 v3 子集（第 17 章）：startup、認證（先 `trust`，再 SCRAM）、簡單查詢、`ErrorResponse` 帶 SQLSTATE、`ReadyForQuery` 的交易狀態；thread-per-connection + 每連線一個 session 狀態。
- 擴充查詢（Parse/Bind/Execute）讓參數化查詢可用，也讓 ORM 與驅動能正常工作。
- 串流複寫：`WalSender` 把 LSN 之後的 WAL 傳給 `WalReceiver`；standby 持續 redo（重用 v4）並以快照服務唯讀查詢；`synchronous_commit` 開關；`pg_stat_replication` 式的延遲監控。
- 進階 A：Raft（第 15 章）複寫 WAL，自動 failover；進階 B：2PC 跨兩個 MiniDB（第 16 章）；進階 C：簡單的 hash sharding 路由層。
- 安全（第 18 章）：密碼雜湊、`GRANT`、參數化查詢的注入測試。

驗收：`psql`、Python psycopg、JDBC 都能連線並執行 sqllogictest；primary 當機後 standby 擁有所有已同步 commit 的交易；Jepsen 風格的客戶端歷史檢查（可用 `elle` 或自寫的 linearizability checker）在 Raft 版本上通過。

19.11 測試策略

層級	做法
單元測試	GoogleTest；每個資料結構獨立測（slotted page 填滿 / 碎片化、B+Tree 分裂與合併邊界、replacer 的 pin 行為、log record 序列化往返）。
sqllogictest	文字檔：SQL + 預期結果（或結果的雜湊）。SQLite、CockroachDB、DuckDB、BusTub 都用這個格式，可以借用它們的測試檔（去掉不支援的語法）。
Oracle 對照	隨機產生操作序列，同時套用到 MiniDB 與 `std::map` / SQLite，比對結果（B+Tree、MVCC 可見性最有效）。
並行測試	多執行緒隨機負載 + 不變量檢查（總額守恆、唯一鍵唯一）；TSan；注入隨機 `sleep` 放大交錯。
當機注入	`fork` + `SIGKILL` 迴圈；也可在 `DiskManager::WritePage` 中隨機「只寫一半」模擬 torn write 測試 full-page write。
模糊測試	libFuzzer 餵 parser 隨機字串、餵 tuple 反序列化隨機位元組；任何 crash 都是 bug。
基準測試	固定的 YCSB 式 KV 負載與 TPC-H 子集；每個版本記錄數字，觀察 buffer pool、索引、最佳化器各帶來多少改善。

19.12 時程建議

版本	主題	對應章節與關鍵難點	建議時間
v1	Parser、heap、executor	第 1、2、5 章；tuple 格式與 NULL 語意	3–4 週
v2	B+Tree	第 3 章；刪除時的合併 / 重分配	2–3 週
v3	Buffer pool	第 4 章；並行與 RAII guard	1–2 週
v4	交易、鎖、WAL、復原	第 7、8、10、11 章；當機注入測試	4–5 週
v5	MVCC	第 9 章；可見性與 VACUUM	2–3 週
v6	最佳化器	第 6、13 章；統計與 DP	3–4 週
v7	協定、複寫、Raft	第 14–18 章；Raft 的測試	4–6 週

每週 10–15 小時的話，整個計畫約 6–8 個月；全職約 3 個月。不必一次做完——v1 到 v4 已經涵蓋資料庫核心的 80%。

19.13 參考實作

專案	語言	值得學什麼
BusTub（CMU 15-445）	C++	與 MiniDB 架構最接近；Project 1 buffer pool（LRU-K、disk scheduler）、Project 2 B+Tree、Project 3 執行器、Project 4 並行控制（近年為 MVCC）。測試與程式骨架公開，解答不公開。
cstack *db_tutorial*	C	「Let's Build a Simple Database」：從 REPL 到 B-Tree 的 SQLite 複製品，十餘篇循序漸進，最適合 v1 之前熱身。
build-your-own.org *Database*	Go	KV 儲存、B-tree、free list、交易、到 SQL 層，每章附完整程式碼與推理。
SQLite	C	*Architecture of SQLite* 文件、bytecode VM（`EXPLAIN`）、單檔案格式規格、極致的測試文化。
PostgreSQL	C	附錄 A 的原始碼地圖；每個子系統的 README 是最好的設計文件。
MIT 6.5830 GoDB	Go	課程 lab：heap file、buffer pool、運算子、交易，與 6.5840 的 Raft lab 銜接。
toydb	Rust	Raft + MVCC + SQL 的分散式教學資料庫，結構清楚、文件完整。
mini-lsm	Rust	一週建 LSM 引擎（第 12 章）。
SimpleDB（Sciore）	Java	《Database Design and Implementation》教科書的配套系統，每章一個子系統，解說最細。

卡住時

- 先寫測試再寫程式；B+Tree 的 bug 幾乎都能用「隨機操作 + oracle 對照 + 最小化重現」抓到。
- 用 `EXPLAIN`、頁面傾印工具（自己寫一個 `minidb inspect` 印出頁面內容，等同 `pageinspect`）把狀態可視化。

- 讀 PostgreSQL 對應子系統的 README——幾乎每個你會遇到的設計問題，它都已經討論過。
- 到 CMU 15-445 的 Discord / Piazza 或 PostgreSQL 的 `pgsql-hackers` 郵件列表看別人怎麼解決類似問題。

延伸閱讀與資源

- CMU 15-445/645 Fall 2025 專案頁（BusTub）：`https://15445.courses.cs.cmu.edu/fall2025/assignments.html`；BusTub 原始碼：`https://github.com/cmu-db/bustub`
- cstack, *Let's Build a Simple Database*：`https://cstack.github.io/db_tutorial/`
- J. Smith, *Build Your Own Database From Scratch in Go*：`https://build-your-own.org/database/`
- SQLite, *Architecture of SQLite*：`https://www.sqlite.org/arch.html`；*Database File Format*：`https://www.sqlite.org/fileformat.html`；*How SQLite Is Tested*：`https://www.sqlite.org/testing.html`
- MIT 6.5830 *Database Systems*（GoDB labs）：`https://dsg.csail.mit.edu/6.5830/`
- E. Sciore, *Database Design and Implementation*, 2nd ed., Springer 2020（SimpleDB）。
- toydb：`https://github.com/erikgrinaker/toydb`；mini-lsm：`https://skyzh.github.io/mini-lsm/`
- sqllogictest 格式說明（SQLite）：`https://www.sqlite.org/sqllogictest/doc/trunk/about.wiki`；Jepsen *elle* 檢查器：`https://github.com/jepsen-io/elle`
- PostgreSQL 原始碼各子系統 README（見附錄 A）。

第八篇

附錄

附錄 A　PostgreSQL 原始碼導覽地圖

本附錄把前面各章的概念對應到 PostgreSQL 18 原始碼樹中的位置。閱讀原始碼最有效的順序是：**先讀該目錄的 `README`**（PostgreSQL 的 README 是一流的設計文件），再讀標頭檔中的結構定義（`src/include/`），最後才讀 `.c`。

A.1　取得與建置

程式碼 A.1: 取得原始碼並以除錯模式建置

```
git clone https://git.postgresql.org/git/postgresql.git && cd postgresql
git checkout REL_18_STABLE
# Meson（推薦）或傳統 configure；--enable-cassert 開啟內部斷言，-O0 方便 gdb
meson setup build -Dcassert=true -Ddebug=true -Doptimization=0 --prefix=$HOME/pgdev
ninja -C build && ninja -C build install
$HOME/pgdev/bin/initdb -D $HOME/pgdata && $HOME/pgdev/bin/pg_ctl -D $HOME/pgdata -l logfile start
# 跑迴歸測試
meson test -C build --suite setup --suite regress
# 對某個 backend 除錯：先取得 pid，再 gdb attach
psql -c 'SELECT pg_backend_pid()'
gdb -p <pid>   # break heap_insert ... continue，然後在 psql 執行 INSERT
```

常用的導覽工具：`git grep -n 'HeapTupleSatisfiesMVCC'`、`ctags -R src`、以及線上的 `https://doxygen.postgresql.org/`（可點擊的交叉參照）。`src/tools/pgindent` 說明程式碼風格；`src/backend/utils/misc/postgresql.conf.sample` 是所有 GUC 參數的清單。

A.2 目錄總覽

路徑	內容
src/backend/	伺服器端全部程式碼（下表細分）
src/include/	所有標頭檔；結構定義與巨集在這裡（如 access/htup_details.h、storage/bufpage.h）
src/bin/	客戶端工具：psql、pg_dump、pg_basebackup、pg_waldump、pg_controldata、pg_rewind、pgbench
src/interfaces/libpq/	客戶端函式庫；fe-protocol3.c 是協定的客戶端實作
src/common/、src/port/	前後端共用程式碼與可攜性層
src/test/	regress/（SQL 迴歸測試）、isolation/（並行交錯測試，第 7–10 章的異常都有對應的 spec）、recovery/（複寫與復原的 TAP 測試）
contrib/	隨附擴充套件：pageinspect、pg_buffercache、pg_stat_statements、pgcrypto、postgres_fw、pg_visibility、amcheck、auto_explain
doc/src/sgml/	官方文件原始碼

A.3 概念對應表

章	概念	路徑	入口與必讀
1	系統目錄	src/backend/catalog/、src/include/catalog/pg_class.h、pg_attribute.h	README；catalog/pg_*.dat 是初始內容
1	關聯代數 → 計畫節點	src/include/nodes/plannodes.h、primnodes.h、parsenodes.h	每種運算子一個 struct
2	頁面佈局、slotted page	src/include/storage/bufpage.h、src/backend/storage/page/bufpage.c	PageHeaderData、PageAddItem()
2	Heap tuple 與 heap AM	src/include/access/htup_details.h、src/backend/access/heap/heapam.c、heapam_handler.c	HeapTupleHeaderData；heap_insert()、heap_update()；README.HOT、README.tuplock
2	檔案與 fork、FSM、VM	src/backend/storage/smgr/、src/backend/storage/freespace/README、src/backend/access/heap/visibilitymap.c	md.c（1 GB 分段）、fsm_internals.h
2	TOAST	src/backend/access/common/toast_internals.c、heaptoast.c、src/include/access/toast_*.h	TOAST_TUPLE_THRESHOLD

章	概念	路徑	入口與必讀
3	B-Tree 索引	`src/backend/access/nbtree/`	**`README`**（Lehman–Yao、刪除、去重）；`nbtsearch.c`、`nbtinsert.c`、`nbtpage.c`
3	索引存取方法介面	`src/include/access/amapi.h`、`src/backend/access/index/indexam.c`	`IndexAmRoutine`；GIN/GiST/BRIN/Hash/SP-GiST 各在 `access/` 下有 README
4	Buffer pool	`src/backend/storage/buffer/`	**`README`**；`bufmgr.c`（`ReadBuffer`、`StrategyGetBuffer` clock sweep）、`freelist.c`（ring buffer）、`buf_table.c`、`buf_internals.h`
4	非同步 I/O（PG 18）	`src/backend/storage/aio/`	`README.md`；`method_io_uring.c`、`method_worker.c`、`read_stream.c`
5	Parser / Analyzer / Rewriter	`src/backend/parser/gram.y`、`scan.l`、`analyze.c`；`src/backend/rewrite/`	`raw_parser()`、`transformStmt()`；`rewriteHandler.c`
5	Executor	`src/backend/executor/`	**`README`**；`execMain.c`、`execProcnode.c`、`nodeSeqscan.c`、`nodeIndexscan.c`、`nodeHashjoin.c`、`nodeHash.c`、`nodeMergejoin.c`、`nodeNestloop.c`、`nodeAgg.c`、`nodeSort.c`、`nodeGather.c`
5	排序與 tuplestore	`src/backend/utils/sort/tuplesort.c`、`tuplestore.c`	外部合併排序、top-N heapsort
5	表達式求值與 JIT	`src/backend/executor/execExpr.c`、`execExprInterp.c`、`src/backend/jit/llvm/`	`README` in `jit/`
6	Planner / Optimizer	`src/backend/optimizer/`	**`README`**；`plan/planner.c`、`path/allpaths.c`、`path/indxpath.c`、`path/joinrels.c`、`path/joinpath.c`、`geqo/`

章	概念	路徑	入口與必讀
6	成本與選擇率	`src/backend/optimizer/path/costsize.c`、`src/backend/utils/adt/selfuncs.c`	`cost_seqscan()`、`cost_index()`、`eqsel()`、`scalarineqsel()`
6	統計資訊	`src/backend/commands/analyze.c`、`src/backend/statistics/`（擴充統計）、`src/include/catalog/pg_statistic.h`	`README.*` in `statistics/`
7	交易狀態機、子交易	`src/backend/access/transam/xact.c`	`CommitTransaction()`、`AbortTransaction()`；`README` in `transam/`
7, 9	XID、clog、MultiXact	`src/backend/access/transam/varsup.c`、`clog.c`、`multixact.c`、`subtrans.c`、`slru.c`	`GetNewTransactionId()`、`TransactionIdDidCommit()`
8	WAL 寫入與格式	`src/backend/access/transam/xlog.c`、`xloginsert.c`、`xlogreader.c`、`src/include/access/xlogrecord.h`	`XLogInsert()`、`XLogFlush()`；`src/backend/access/rmgrdesc/` 是 `pg_waldump` 的描述函數
8	Checkpoint、walwriter	`src/backend/postmaster/checkpointer.c`、`walwriter.c`、`bgwriter.c`	`CreateCheckPoint()` in `xlog.c`
9	快照與可見性	`src/backend/access/heap/heapam_visibility.c`、`src/backend/storage/ipc/procarray.c`、`src/backend/utils/time/snapmgr.c`	`HeapTupleSatisfiesMVCC()`、`GetSnapshotData()`
9	HOT 與 pruning	`src/backend/access/heap/pruneheap.c`、`README.HOT`	`heap_page_prune_opt()`
9	VACUUM、凍結、autovacuum	`src/backend/access/heap/vacuumlazy.c`、`src/backend/commands/vacuum.c`、`src/backend/postmaster/autovacuum.c`	`lazy_scan_heap()`；`heap_prepare_freeze_tuple()`
9	SSI	`src/backend/storage/lmgr/predicate.c`	**`README-SSI`**（含理論推導）
10	鎖管理器、死結	`src/backend/storage/lmgr/lock.c`、`deadlock.c`、`lmgr.c`、`src/include/storage/lock.h`	**`README`**；`LockAcquire()`、`DeadLockCheck()`；衝突表 `LockConflicts[]`
10	LWLock、spinlock、latch	`src/backend/storage/lmgr/lwlock.c`、`s_lock.c`、`src/backend/storage/ipc/latch.c`	`README` in `lmgr/`；wait event 名稱表 `utils/activity/wait_event_names.txt`

章	概念	路徑	入口與必讀
11	當機復原、PITR、timeline	src/backend/access/transam/xlogrecovery.c、timeline.c、src/backend/access/transam/xlog.c（StartupXLOG）	PerformWalRecovery()；src/bin/pg_controldata、pg_resetwal
11	兩階段提交（prepared）	src/backend/access/transam/twophase.c	pg_twophase/ 目錄格式
14	串流複寫	src/backend/replication/walsender.c、walreceiver.c、syncrep.c、slot.c	README in replication/；SyncRepWaitForLSN()
14	邏輯解碼與邏輯複寫	src/backend/replication/logical/、pgoutput/	README in logical/；decode.c、reorderbuffer.c、worker.c、apply
17	前後端協定	src/backend/tcop/postgres.c（主迴圈 PostgresMain）、src/backend/libpq/pqcomm.c、auth.c、hba.c、src/include/libpq/protocol.h	exec_simple_query()、exec_parse_message()、exec_bind_message()
17	Postmaster 與程序模型	src/backend/postmaster/postmaster.c、launch_backend.c	ServerLoop()、BackendStartup()
18	認證、SCRAM、OAuth、TLS	src/backend/libpq/auth.c、auth-scram.c、auth-oauth.c、be-secure-openssl.c、hba.c	ClientAuthentication()
18	權限與 RLS	src/backend/catalog/aclchk.c、src/backend/rewrite/rowsecurity.c、src/backend/commands/policy.c	pg_class_aclcheck()；RLS 在 rewriter 階段加入條件

A.4 建議的閱讀路徑

1. **一週入門**：src/backend/storage/buffer/README → bufpage.h → htup_details.h → heapam.c 的 heap_insert()，搭配 pageinspect 觀察效果。
2. **索引**：nbtree/README 全文（約一小時），再跟著 _bt_search() 與 _bt_doinsert() 走一次插入。
3. **交易**：transam/README → xact.c 的 CommitTransaction() → XLogFlush() → heapam_visibility.c。
4. **查詢處理**：executor/README 與 optimizer/README，然後用 debug_print_plan 比對實際輸出。
5. **並行與復原**：lmgr/README、README-SSI、xlogrecovery.c。
6. **動手改**：加一個 GUC、在某個執行節點加計數器、寫一個小擴充套件（src/test/modules/

有範本）——這是理解建置系統與程式碼風格最快的方法。

`src/test/isolation/specs/` 目錄值得特別一提：每個檔案都是一個並行交錯情境（如 `read-only-anomaly.spec`、`fk-deadlock.spec`、`eval-plan-qual.spec`），是本教材第 7–10 章所有異常的可執行版本。

附錄 B　完整學習資源清單

本附錄彙整全書引用與推薦的資源，依類型排列；每章末的「延伸閱讀」則依主題排列。標示 ★ 者為本教材認為該類別中最值得優先投入的資源。

B.1　課程

- ★ **CMU 15-445/645 Intro to Database Systems**（Fall 2025，Andy Pavlo）——講義、投影片、作業、BusTub 專案、YouTube 全程錄影。本教材第 2–11 章的結構深受其影響。https://15445.courses.cs.cmu.edu/fall2025/
- **CMU 15-721 Advanced Database Systems**——記憶體資料庫、向量化、編譯、行式儲存、現代 OLAP 的研究所課程。https://15721.courses.cs.cmu.edu/
- **CMU 15-799 Special Topics: Query Optimization**——最佳化器專題。https://15799.courses.cs.cmu.edu/
- **UC Berkeley CS186 Introduction to Database Systems**——投影片與影片公開，專案以 Java 實作 B+Tree、join、並行、復原。https://cs186berkeley.net/
- **MIT 6.5830/6.5831 Database Systems**——論文導讀 + GoDB lab。https://dsg.csail.mit.edu/6.5830/
- ★ **MIT 6.5840 Distributed Systems**（舊稱 6.824）——第 14–16 章的最佳搭配：講義、論文、Raft/KV/分片交易 lab。https://pdos.csail.mit.edu/6.824/
- **Stanford CS245 Principles of Data-Intensive Systems**——儲存、查詢、交易、分散式的綜合課。
- **CMU Database Group YouTube 頻道**——除課程外還有 Vaccination Database Tech Talks、Quarantine Tech Talks 等業界系統（CockroachDB、Snowflake、DuckDB、TiDB……）的架構演講。

B.2　書籍

- ★ A. Silberschatz, H. Korth, S. Sudarshan, *Database System Concepts*, 7th ed., McGraw-Hill 2019——最完整的教科書；官方網站有投影片與部分章節。https://www.db-book.com/
- R. Ramakrishnan, J. Gehrke, *Database Management Systems*, 3rd ed., 2002——第 18 章的 ARIES 與第 12–14 章的執行、最佳化寫得最清楚。
- ★ A. Petrov, *Database Internals: A Deep Dive into How Distributed Data Systems Work*, O'Reilly 2019——上半部儲存引擎（B-Tree、LSM）、下半部分散式，與本教材的章節幾乎一一對應。

- ★ M. Kleppmann, *Designing Data-Intensive Applications*, 2nd ed., O'Reilly 2025——複寫、分割、交易、一致性與共識的最佳入門與參考。
- ★ H. Suzuki, *The Internals of PostgreSQL*（免費線上）——PostgreSQL 內部機制的系統性說明，本教材的 PostgreSQL 內容大量參考。`https://www.interdb.jp/pg/`
- E. Rogov, *PostgreSQL 14 Internals*（Postgres Professional，免費 PDF）——MVCC、鎖、WAL、查詢執行的深入專著，含大量實驗。`https://postgrespro.com/community/books/internals`
- J. Gray, A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann 1992——交易處理的聖經。
- P. Bernstein, V. Hadzilacos, N. Goodman, *Concurrency Control and Recovery in Database Systems*, 1987（免費 PDF）。`https://www.microsoft.com/en-us/research/people/philbe/book/`
- P. Bailis, J. Hellerstein, M. Stonebraker (eds.), *Readings in Database Systems*, 5th ed.（Red Book，免費線上）——經典論文選集與評論。`http://www.redbook.io/`
- J. Hellerstein, M. Stonebraker, J. Hamilton, *Architecture of a Database System*, Foundations and Trends in Databases 2007（免費）——一篇論文長度的「資料庫系統架構總覽」。
- G. Weikum, G. Vossen, *Transactional Information Systems*, 2001——並行控制與復原的理論基礎。
- E. Sciore, *Database Design and Implementation*, 2nd ed., Springer 2020——以 SimpleDB 逐章實作。
- M. Winand, *SQL Performance Explained* / *Use The Index, Luke!*（免費線上）——索引與執行計畫的實務。`https://use-the-index-luke.com/`
- D. Abadi et al., *The Design and Implementation of Modern Column-Oriented Database Systems*, 2013（免費）——行式系統專著。

B.3 論文（依章節）

第 1 章

Codd, *A Relational Model of Data for Large Shared Data Banks*, CACM 1970。

第 2 章

Ailamaki et al., *Weaving Relations for Cache Performance*（PAX）, VLDB 2001；Crotty et al., *Are You Sure You Want to Use MMAP in Your DBMS?*, CIDR 2022。

第 3 章

Bayer & McCreight, *Organization and Maintenance of Large Ordered Indices*, 1972；Lehman & Yao, *Efficient Locking for Concurrent Operations on B-Trees*, TODS 1981；Graefe, *Modern B-Tree Techniques*, 2011。

第 4 章

O'Neil et al., *The LRU-K Page Replacement Algorithm*, SIGMOD 1993；Megiddo & Modha,

ARC, FAST 2003。

第 5 章

Graefe, *Volcano*, TKDE 1994；Boncz et al., *MonetDB/X100*, CIDR 2005；Neumann, *Efficiently Compiling Efficient Query Plans*, VLDB 2011；Shapiro, *Join Processing in Database Systems with Large Main Memories*, TODS 1986。

第 6 章

Selinger et al., *Access Path Selection*, SIGMOD 1979；Chaudhuri, *An Overview of Query Optimization*, PODS 1998；Leis et al., *How Good Are Query Optimizers, Really?*, VLDB 2015；Graefe, *Cascades*, 1995。

第 7 章

Berenson et al., *A Critique of ANSI SQL Isolation Levels*, SIGMOD 1995；Adya, Liskov, O'Neil, *Generalized Isolation Level Definitions*, ICDE 2000；Bailis et al., *Highly Available Transactions*, VLDB 2014。

第 8、11 章

Mohan et al., *ARIES*, TODS 1992；Gray et al., *The Recovery Manager of the System R Database Manager*, 1981。

第 9 章

Ports & Grittner, *Serializable Snapshot Isolation in PostgreSQL*, VLDB 2012；Cahill et al., *Serializable Isolation for Snapshot Databases*, SIGMOD 2008；Wu et al., *An Empirical Evaluation of In-Memory MVCC*, VLDB 2017。

第 10 章

Gray et al., *Granularity of Locks and Degrees of Consistency*, 1976；Kung & Robinson, *On Optimistic Methods for Concurrency Control*, TODS 1981。

第 12 章

O'Neil et al., *The LSM-Tree*, 1996；Chang et al., *Bigtable*, OSDI 2006；Dayan et al., *Monkey*, SIGMOD 2017；Lu et al., *WiscKey*, FAST 2016；Luo & Carey, *LSM-based Storage Techniques: A Survey*, VLDBJ 2020。

第 13 章

Stonebraker et al., *C-Store*, VLDB 2005；Abadi et al., *Column-Stores vs. Row-Stores*, SIGMOD 2008；Melnik et al., *Dremel*, VLDB 2010；Leis et al., *Morsel-Driven Parallelism*, SIGMOD 2014；Dageville et al., *Snowflake*, SIGMOD 2016；Raasveldt & Mühleisen, *DuckDB*, SIGMOD 2019。

第 14 章

Gilbert & Lynch, *Brewer's Conjecture*, 2002；Brewer, *CAP Twelve Years Later*, 2012；Abadi, *PACELC*, 2012；DeCandia et al., *Dynamo*, SOSP 2007；Karger et al., *Consistent Hashing*, STOC 1997；Lamport, *Time, Clocks*, CACM 1978；Verbitski et al., *Aurora*, SIGMOD 2017；Taft et al., *CockroachDB*, SIGMOD 2020。

第 15 章

Ongaro & Ousterhout, *In Search of an Understandable Consensus Algorithm*, ATC 2014；Lamport, *Paxos Made Simple*, 2001；Chandra et al., *Paxos Made Live*, PODC 2007；FLP, 1985；Howard & Mortier, *Paxos vs Raft*, 2020；Oki & Liskov, *Viewstamped Replication*, 1988。

第 16 章

Gray & Lamport, *Consensus on Transaction Commit*, TODS 2006；Corbett et al., *Spanner*, OSDI 2012；Peng & Dabek, *Percolator*, OSDI 2010；Thomson et al., *Calvin*, SIGMOD 2012；Garcia-Molina & Salem, *Sagas*, SIGMOD 1987；Helland, *Life beyond Distributed Transactions*, CIDR 2007。

第 17 章

Ghemawat et al., *The Google File System*, SOSP 2003；Dean & Barroso, *The Tail at Scale*, CACM 2013；Zhou et al., *FoundationDB*, SIGMOD 2021。

B.4 PostgreSQL 官方與社群資源

- ★ 官方文件（每個版本）：`https://www.postgresql.org/docs/current/`；特別是 Part VII *Internals*（第 50 章以後）。
- 原始碼：`https://git.postgresql.org/`（鏡像 `https://github.com/postgres/postgres`）；Doxygen：`https://doxygen.postgresql.org/`
- PostgreSQL Wiki：`https://wiki.postgresql.org/`（*Developer FAQ*、*Tuning Your PostgreSQL Server*、*Don't Do This*、*Lock Monitoring*、*SSI*）。
- 郵件列表 `pgsql-hackers`（設計討論與 patch 審查，許多功能的「為什麼」只在這裡找得到）與 Commitfest：`https://commitfest.postgresql.org/`
- 每個 release 的 *Release Notes*（新功能摘要）與 *PostgreSQL 18 Release Notes*：`https://www.postgresql.org/docs/18/release-18.html`
- Planet PostgreSQL（社群部落格聚合）：`https://planet.postgresql.org/`；優質技術部落格：Crunchy Data、Cybertec、pganalyze、EDB、Citus、Neon、Dalibo、Percona。
- 會議錄影：PGConf.dev、PGCon(歷史)、PGConf.EU、PGConf.ASIA、FOSDEM PGDay；台灣 PostgreSQL 使用者社群的聚會與年會。
- 安全公告：`https://www.postgresql.org/support/security/`

B.5 工具

- **觀察內部狀態**：`pageinspect`、`pg_buffercache`、`pg_visibility`、`pg_freespacemap`、`pg_waldump`、`pg_controldata`、`amcheck`、`pg_stat_statements`、`auto_explain`、`pg_stat_activity` / `pg_locks` / `pg_stat_replication`。
- **執行計畫視覺化**：`https://explain.depesz.com/`、`https://explain.dalibo.com/`、pgMustard。
- **負載與基準**：`pgbench`（TPC-B 風格，可自訂腳本）、HammerDB（TPC-C/H）、sysbench、

YCSB、TPC-H 工具（`tpch-dbgen`）。

- **監控**：pg_stat_* 視圖、pgwatch、pgBadger（日誌分析）、Prometheus postgres_exporter、pganalyze。
- **高可用與備份**：Patroni、pgBackRest、barman、pg_auto_failover、PgBouncer、PgCat、Odyssey。
- **正確性測試**：Jepsen（`https://jepsen.io/`）、Hermitage（`https://github.com/ept/hermitage`）、sqllogictest、SQLancer（自動找出邏輯 bug）。
- **互動式學習**：RelaX 關聯代數計算機（`https://dbis-uibk.github.io/relax/`）、*The Secret Lives of Data*（Raft 視覺化）、Raft 官網視覺化、Use The Index Luke 的互動範例。

B.6 動手專案

- BusTub（CMU 15-445）：`https://github.com/cmu-db/bustub`
- cstack *Let's Build a Simple Database*：`https://cstack.github.io/db_tutorial/`
- *Build Your Own Database From Scratch*：`https://build-your-own.org/database/`
- mini-lsm：`https://skyzh.github.io/mini-lsm/`；toydb：`https://github.com/erikgrinaker/toydb`
- MIT 6.5840 Raft lab：`https://pdos.csail.mit.edu/6.824/labs/lab-raft.html`
- SQLite 文件（*Architecture*、*File Format*、*WAL*、*Testing*）：`https://www.sqlite.org/docs.html`

B.7 保持更新

資料庫領域變化快，但核心概念穩定。建議每年固定閱讀：PostgreSQL 新版本的 release notes、CMU 15-445 當年度的講義更新、VLDB / SIGMOD / CIDR 的 industry track 論文（各大系統的架構論文多發表於此）、Jepsen 的新分析報告、以及 DDIA 作者 Martin Kleppmann 與 Murat Demirbas（`https://muratbuffalo.blogspot.com/`）等人的部落格。

附錄 C　中英術語對照表

本表依英文字母排序，收錄全書出現的核心術語、本教材採用的中文譯名與一句話定義，並標示主要討論的章節。中文資料庫文獻的譯名並不統一（例如 transaction 有「交易」與「事務」兩種常見譯法），本教材在全書中採用台灣常用的譯法，並在首次出現時附上英文。

English	中文	定義	章
ACID	交易四特性	Atomicity、Consistency、Isolation、Durability：交易的原子性、一致性、隔離性、持久性	7
Access method (AM)	存取方法	表或索引的儲存與存取實作（heap、B-Tree、GIN……）	2, 3
Anti-entropy	反熵	副本間以背景比對（如 Merkle tree）修復差異的機制	14
ARIES	—	Algorithms for Recovery and Isolation Exploiting Semantics：WAL、steal/no-force、repeating history 的復原演算法	11
Atomic commit	原子提交	多參與者的交易「全部提交或全部中止」；2PC 為代表協定	16
B+Tree	—	所有鍵值存放於葉節點、葉節點串連、內部節點只做導航的平衡搜尋樹	3
Bloom filter	布隆過濾器	可能有偽陽性、不會偽陰性的機率性成員測試結構	12
Buffer pool	緩衝池	以固定大小 frame 快取磁碟頁面的記憶體區域	4
Bufferpool manager	緩衝池管理器	負責 pin/unpin、替換、髒頁寫回的元件	4
CAP theorem	CAP 定理	網路分割發生時，系統必須在一致性與可用性間取捨	14
Cardinality estimation	基數估計	估計運算子輸出列數；最佳化器的核心難題	6
Catalog	系統目錄	描述資料庫中物件（表、欄位、索引、型別……）的中繼資料表	1
Checkpoint	檢查點	將髒頁寫回並記錄復原起點的動作，用來縮短復原時間	8, 11
Clock sweep	時鐘掃描	以參考位元近似 LRU 的頁面替換演算法	4
Columnar storage	行式儲存	同一欄位的值連續存放（DSM / PAX / Parquet）	13

English	中文	定義	章
Commit point	提交點	交易的 commit 記錄落盤的瞬間；之前中止、之後必定持久	8, 16
Compaction	壓實 / 合併	LSM 中合併多個 SSTable、刪除過期版本的背景程序	12
Consensus	共識	多節點對一個值（或一串值）達成一致的協定（Paxos、Raft）	15
Consistent hashing	一致性雜湊	節點增減時只需搬移少量鍵的分割方法	14
Cost model	成本模型	以 I/O、CPU 等參數估計計畫執行代價的函數	6
Deadlock	死結	等待圖出現環，交易互相等待且無法前進	10
Dirty page	髒頁	緩衝池中已修改、尚未寫回磁碟的頁面	4
Dirty read	髒讀	讀到其他交易尚未提交的資料	7
Durability	持久性	已提交的交易在當機後不會遺失	7, 8
Executor	執行器	依計畫樹逐運算子計算結果的元件	5
Extent	區段	由多個連續頁面組成的配置單位	2
Fencing token	圍籬令牌	單調遞增的編號，用來拒絕過期 leader / 鎖持有者的寫入	15
Fork (relation fork)	關聯分支	PostgreSQL 中同一個表的主檔、FSM、VM、init 等分開的檔案	2
Free space map (FSM)	可用空間地圖	記錄每頁剩餘空間，加速插入時找頁	2
fsync	—	要求作業系統把檔案的髒資料與中繼資料寫到持久儲存	8
Group commit	群組提交	多筆交易共用一次 WAL fsync	8
Hash join	雜湊連接	以較小的輸入建雜湊表、另一側探測的連接演算法	5
Heap file	堆積檔	不保證順序的表儲存結構	2
HOT (Heap-Only Tuple)	堆積內更新	更新未改動索引欄位時，新版本留在同頁且不新增索引項目	9
Hybrid logical clock (HLC)	混合邏輯時鐘	結合實體時間與邏輯計數器的時間戳	14, 16
Index-only scan	僅索引掃描	所需欄位皆在索引中，不必回表（需 visibility map 協助）	3, 9
Isolation level	隔離等級	交易間可見性的保證強度（RU、RC、RR、SI、Serializable）	7
Latch	閂鎖	保護記憶體資料結構的短期互斥鎖（相對於保護邏輯內容的 lock）	3, 10
Latch crabbing / coupling	閂鎖螃蟹法	B+Tree 由上而下取閂、確認子節點安全後釋放父節點	3
Leader election	領導者選舉	共識協定中選出唯一協調者的過程	15
Linearizability	線性一致性	每個操作看起來在呼叫與回傳之間的某個瞬間原子發生	14
Lock escalation	鎖升級	太多細粒度鎖時改持較粗粒度的鎖	10

English	中文	定義	章
Log sequence number (LSN)	日誌序號	WAL 中記錄的位元組位移，單調遞增	8
Logical replication	邏輯複寫	以列變更（而非實體頁面）為單位的複寫	14
LSM-Tree	日誌結構合併樹	以記憶體寫入 + 多層排序檔 + 合併為核心的寫入最佳化結構	12
Materialization	實體化	把中間結果完整存起來（相對於串流 pipelining）	5
MemTable	記憶體表	LSM 中接收寫入的記憶體排序結構	12
Multi-version concurrency control (MVCC)	多版本並行控制	保留多個版本讓讀寫互不阻塞	9
Nested-loop join	巢狀迴圈連接	外層每列都掃描內層的連接演算法	5
Non-repeatable read	不可重複讀	同一交易內兩次讀同一列結果不同	7
Normalization	正規化	依函數相依拆表以消除異常（1NF–BCNF）	1
Page	頁面	磁碟與記憶體間搬移的固定大小區塊（PostgreSQL 預設 8 KB）	2
Partitioning / Sharding	分割 / 分片	把資料依鍵分散到多個節點	14
Phantom read	幻讀	同一交易內兩次範圍查詢結果集不同	7, 10
Pipelining	管線化	運算子逐列把輸出交給上層而不實體化	5
Plan (execution plan)	執行計畫	最佳化器選出的運算子樹	5, 6
Predicate lock	述詞鎖	鎖住「滿足某條件的所有列（含不存在者）」以防幻讀	10
Prepared statement	預備敘述	先 parse/plan、之後多次綁定參數執行的敘述	17, 18
Quorum	法定人數	操作需多少副本回應才算成功（$R + W > N$）	14
Raft	—	以 leader 為中心、強調可理解性的共識演算法	15
Read committed	讀已提交	每個敘述看到開始時已提交的資料	7
Redo	重做	復原時重新套用已記錄的修改	11
Repeatable read	可重複讀	交易內多次讀取結果一致；PostgreSQL 以快照隔離實作	7, 9
Replication lag	複寫延遲	副本落後主節點的時間或 LSN 距離	14
Row-level security (RLS)	列層級安全	以政策限制每個角色可見 / 可改的列	18
Saga	—	以一連串本地交易與補償動作組成的長交易模式	16
Selectivity	選擇率	滿足條件的列所佔比例	6
Serializability	可序列化	並行執行的效果等同某個序列執行	7
Serializable snapshot isolation (SSI)	可序列化快照隔離	在 SI 上偵測危險結構並中止以達到可序列化	9
Slotted page	槽式頁面	頁首 + 槽目錄 + 由頁尾向前成長的 tuple 區	2

English	中文	定義	章
Snapshot isolation (SI)	快照隔離	交易讀取開始時的一致快照；可能出現 write skew	7, 9
Sort-merge join	排序合併連接	兩側依連接鍵排序後同步掃描	5
Split-brain	腦裂	兩個節點同時自認為 leader	15
SSTable	排序字串表	LSM 中不可變的排序鍵值檔	12
Steal / No-force	偷取 / 不強制	緩衝管理策略：未提交的髒頁可寫出、提交時不必寫出資料頁	11
Streaming replication	串流複寫	以 WAL 串流為單位的實體複寫	14
Strict two-phase locking	嚴格兩階段鎖	所有鎖持有到交易結束才釋放的 2PL	10
TOAST	—	PostgreSQL 把超大欄位切片 / 壓縮另存的機制	2
Transaction	交易	一組被視為單一工作單元的操作	7
Transaction ID (XID)	交易識別碼	PostgreSQL 的 32 位元交易編號，需凍結以防回捲	9
Two-phase commit (2PC)	兩階段提交	PREPARE 再 COMMIT 的原子提交協定	16
Undo	還原	復原時撤銷未提交交易的修改	11
Vacuum	清理	PostgreSQL 回收死版本、更新 VM、凍結 XID 的程序	9
Vectorized execution	向量化執行	運算子一次處理一批（如 1024 列）而非一列	5, 13
Visibility map (VM)	可見性地圖	每頁一位元記錄「全部可見」，供 vacuum 與 index-only scan 使用	9
Volcano model	火山模型	每個運算子實作 `Next()` 的迭代器執行模型	5
Write-ahead logging (WAL)	預寫式日誌	修改先寫日誌再寫資料頁，確保持久性與原子性	8
Write amplification	寫入放大	實際寫入位元組數與邏輯寫入量的比值	12
Write skew	寫入偏斜	兩個交易各讀對方要寫的資料後都寫入，違反不變式	7, 9