CMU 15-445/645 数据库系统（Intro to Database Systems）学习笔记

> **课程**：Carnegie Mellon University 15-445/645 *Intro to Database Systems*，Fall 2026。 **课程主页**：https://15445.courses.cs.cmu.edu/ ｜ 本学期路径前缀：`https://15445.courses.cs.cmu.edu/fall2026/` **授课教师**：Fall 2026 主页 Instructors 栏与讲义首页均署名 **Jignesh Patel**（以课程当学期主页为准）；本文引用的 Fall 2025 归档讲义则署名 Andy Pavlo。 **排课/讲义页**：https://15445.courses.cs.cmu.edu/fall2026/schedule.html ｜ **作业页**：https://15445.courses.cs.cmu.edu/fall2026/assignments.html ｜ **大纲页**：https://15445.courses.cs.cmu.edu/fall2026/syllabus.html ｜ **FAQ**：https://15445.courses.cs.cmu.edu/fall2026/faq.html **教材**：Silberschatz, Korth, Sudarshan《Database System Concepts》(7th ed.)，课程中按章节号引用（如 Chapters 12.1–12.4、13.2–13.3）。 **项目框架**：BusTub（C++ 教学数据库）https://github.com/cmu-db/bustub 。项目说明页：`fall2026/project0/` … `fall2026/project4/`。 **数据来源与边界**：本笔记逐讲下载并 `pdftotext -layout` 提取了 Fall 2026 站点上**实际存在**的讲义（slides 01–16，notes 01–14）与作业 PDF（hw2/hw3/hw4），并抓取了 assignments / syllabus / FAQ / index / project0–project4 页面。**第 16–17 讲只有 slides 而没有 notes；第 18 讲及以后（并发控制、恢复、分布式）在本学期站点上尚无任何 PDF**，这部分改以 **Fall 2025 归档**的同主题讲义（`fall2025/notes/15-optimization1.pdf` … `24-distributed2.pdf`，署名 Andy Pavlo）为据，正文中逐处标注来源归档学期。 **访问日期**：所有 URL 均于 **2026-11** 访问。 **版权声明**：本文档为个人学习笔记。讲义、作业、项目说明、BusTub 源码的版权归 CMU 与课程作者所有，本文仅用于个人学习；引用处标注文件名以便回查，不复刻原文大段文字。凡材料未明说、由笔者推断之处标注“推断”；凡课程政策/评分可能逐学期变动之处标注“以课程当学期主页为准”。

1.1 一句话定位
1.2 先修、工作量与“隐形门槛”
1.3 评分与考试（以 syllabus 页 Fall 2026 版本为准）
1.4 25 讲的讲题地图（来自 schedule 页，Fall 2026）
1.5 五个项目 + 六个作业总览
1.6 材料可得性与本文的写作边界（重要）
1.7 术语约定（读后文前先对齐）
2.1 动机
2.2 机制：三层抽象
2.3 机制：Modern SQL 的“窗口”与“递归”
2.4 关键数据结构与不变量
2.5 经典陷阱
3.1 动机：为什么 DBMS 不直接 `mmap` 一个文件
3.2 机制：页的三个层次
3.3 机制：堆文件（heap file）与页目录
3.4 机制：槽式页（slotted page）——本课最重要的一个布局
3.5 机制：元组布局与数据表示
3.6 关键数据结构与不变量（本章层面）
3.7 经典陷阱
4.1 动机
4.2 机制：buffer pool 的解剖
4.3 机制：替换策略谱系
4.4 机制：把 OS 挡在门外
4.5 关键数据结构与不变量
4.6 经典陷阱
5.1 动机
5.2 机制：LSM Tree 的写入路径
5.3 机制：读路径与代价
5.4 机制：索引组织表（index-organized storage）
5.5 关键数据结构与不变量
5.6 经典陷阱
6.1 动机：负载决定布局
6.2 机制：NSM / DSM / PAX
6.3 机制：压缩的层次与手法
6.4 关键数据结构与不变量
6.5 经典陷阱
7.1 动机
7.2 机制：静态哈希三兄弟
7.3 机制：动态哈希
7.4 关键数据结构与不变量
7.5 经典陷阱
8.1 动机
8.2 机制：结构与不变量
8.3 机制：设计选择（考试爱出“为什么”）
8.4 机制：聚簇索引与“索引扫描的页排序”
8.5 关键数据结构与不变量

8.6 经典陷阱
9.1 动机：索引回答“在哪里”，过滤器回答“在不在”
9.2 机制：Bloom filter 的数学与三条性质
9.3 机制：跳表（skip list）
9.4 机制：Trie 与压缩
9.5 机制：倒排索引与向量索引（本讲的两个“新东西”）
9.6 关键数据结构与不变量
9.7 经典陷阱
10.1 动机：为什么“加一把大锁”不行
10.2 机制：latch 的三种实现
10.3 机制：B+ 树的 latch crabbing（螃蟹法）
10.4 机制：叶子扫描的死锁与“自杀”策略
10.5 机制：写操作的回滚需求
10.6 关键数据结构与不变量
10.7 经典陷阱
11.1 动机
11.2 机制：外部归并排序（external merge sort）
11.3 机制：比较本身也可能很贵
11.4 机制：聚合的两条路
11.5 关键数据结构与不变量
11.6 经典陷阱
12.1 动机
12.2 机制：四种算法与代价
12.3 机制：Hash Join 的两个阶段与“溢出”
12.4 机制：输出物化的两种策略
12.5 关键数据结构与不变量
12.6 经典陷阱
13.1 动机
13.2 机制：三种处理模型
13.3 机制：流水线与断点
13.4 机制：访问方法（access method）
13.5 机制：修改算子与 Halloween 问题
13.6 机制：表达式求值
13.7 关键数据结构与不变量
13.8 经典陷阱
14.1 动机
14.2 机制：parallel vs distributed
14.3 机制：进程模型
14.4 机制：三种查询内并行
14.5 机制：I/O 并行
14.6 关键数据结构与不变量
14.7 经典陷阱
15.1 动机
15.2 机制：编译流水线
15.3 机制：逻辑优化（heuristic / rule-based）
15.4 机制：物理优化（cost-based）
15.5 机制：单表与多表
15.6 机制：子查询与表达式
15.7 关键数据结构与不变量
15.8 经典陷阱
16.1 动机
16.2 机制：统计信息的来源
16.3 机制：选择性与三条“不成立的假设”
16.4 机制：连接基数估计
16.5 机制：误差为什么会爆炸
16.6 关键数据结构与不变量
16.7 经典陷阱
17.1 动机：两个“送命题”
17.2 机制：ACID 逐条落地
17.3 机制：冲突与可串行化
17.4 关键数据结构与不变量
17.5 经典陷阱
18.1 动机
18.2 机制：锁 vs 闩（这张表必须能默写）
18.3 机制：2PL 的两个阶段
18.4 机制：死锁的检测与预防
18.5 机制：锁粒度与意向锁
18.6 关键数据结构与不变量
18.7 经典陷阱
19.1 动机
19.2 机制：时间戳与 OCC 三阶段
19.3 机制：幻读（phantom）与四种对策
19.4 机制：隔离级别（SQL-92 口径）
19.5 关键数据结构与不变量
19.6 经典陷阱
20.1 动机
20.2 机制：版本链与可见性
20.3 机制：五个设计维度
20.4 机制：写偏斜（write skew）
20.5 关键数据结构与不变量
20.6 经典陷阱
21.1 动机：两个策略决定一切
21.2 机制：WAL 的两条铁律
21.3 机制：三种日志粒度
21.4 机制：checkpoint 与日志膨胀
21.5 机制：group commit 与 CDC
21.6 关键数据结构与不变量
21.7 经典陷阱
22.1 动机与三条核心思想
22.2 机制：五种 LSN
22.3 机制：正常执行中的三个特殊记录
22.4 机制：checkpoint 时的两张表

22.5 机制：三阶段恢复
22.6 关键数据结构与不变量
22.7 经典陷阱
23.1 动机：三个目标
23.2 机制：四种系统架构
23.3 机制：分区（partitioning / sharding）
23.4 机制：分布式事务的协调
23.5 机制：联邦数据库（federated）
23.6 关键数据结构与不变量
23.7 经典陷阱
24.1 机制：原子提交协议谱系
24.2 机制：2PC 的消息流
24.3 机制：Paxos 与 2PC 的关系
24.4 机制：CAP 与 PACELC
24.5 机制：分布式 Join 的四种场景
24.6 关键数据结构与不变量
24.7 经典陷阱
25.1 三种页布局的取舍
25.2 元组编码的四个决策
25.3 替换策略的量化对照
25.4 行存 vs 列存：一句话判据
25.5 LSM 的三个必背代价
26.1 插入导致的分裂（叶子）
26.2 内节点分裂：push up 而非 copy up
26.3 删除：先借，后并
26.4 阶数与容量公式（一页纸要能算）
26.5 哈希索引 vs B+ 树索引
26.6 索引选择的经验规则
27.1 四种执行模型的对照
27.2 Join 算法决策树
27.3 排序的三种场景
27.4 聚合的两条路与选择
27.5 流水线断点清单（写出执行计划时要能标出来）
28.1 逻辑 vs 物理
28.2 代价模型的四项成本
28.3 基数估计的三条假设与三个修正
28.4 Join 顺序的搜索空间
28.5 三条“优化器会犯的错”（也是面试常问）
29.1 协议 × 异常
29.2 死锁策略对照
29.3 MVCC 版本存储三选一
29.4 隔离级别的“实际语义”提醒
30.1 WAL 的四条规则（可当检查表用）
30.2 ARIES 三阶段的时序图
30.3 三种 checkpoint 的成本对照
30.4 日志结构存储的恢复差异
31.1 分区策略对照
31.2 复制：两种经典形态
31.3 分布式事务：2PC 的三种失败点
31.4 NewSQL 的三件套
32.1 五个项目一览（Fall 2026 实际形态，与历史学期不同）
32.2 实现顺序（经验顺序，非官方）
32.3 评分点与“测试读法”
32.4 政策红线（必须如实遵守）
34.1 复习清单（按“能不能在白纸上画出来”自查）
34.2 自测题（先自己写，再看右列）

第 1 章 课程概览：15–445 到底在教什么

1.1 一句话定位

15–445 是一门**“把数据库管理系统从磁盘页一路造到分布式事务”**的系统课。主页的课程描述把它概括为一串“怎么存、怎么找、怎么算、怎么保证正确”的问题：

> 数据模型（关系、文档、键值）→ 存储模型（n-ary、decomposition）→ 查询语言 → 存储架构（heap、log-structured）→ 索引与过滤器（tree、hash table、vector）→ 事务处理（ACID、concurrency control）→ 恢复（logging、checkpoint）→ 查询处理（join、sorting、aggregation、optimization）→ 并行架构（multi-core、distributed）。

主页还给出一句相当“劝退”的自我定位：这门课适合“准备好把生活方式献给数据库、并愿意折腾系统编程技能”的学生（原文：*ready to commit to a database-centric lifestyle and flex their systems programming skills*）。这不是修辞：5 个项目全部在 BusTub 上写 C++，且项目**累进**（后者依赖前者）。

它与相邻课程的分工可以这样记：

课程	关注点	与本课的关系
15–213 / CSAPP	机器级程序、cache、链接、并发原语	本课先修；buffer pool、latch 的硬件直觉来自这里
15–445（本课）	**单机 DBMS 内核** + 分布式数据库入门	你在这里实现 page/buffer pool/B+tree/executor
15–721（Advanced DBMS）	现代分析型系统、编译执行、新型硬件	本课之后的进阶
15–440/640（Distributed Systems）	共识、复制、一致性模型	本课只讲数据库视角的 2PC/Paxos/CAP

1.2 先修、工作量与“隐形门槛”

- **正式先修**：CMU 15–213/513（即 CSAPP 程度的系统编程）。大纲页把课程定位为高年级课，明确“**重项目**”，并且“课程组不会在教学期内教你写/调试 C++17 程序，这是先修内容”。
- **隐形门槛 1：C++**。项目 0 就是一次 C++ 能力筛查；FAQ Q13 说项目代码库是 C++ 环境（大纲与 P0 页面写作 C++17，FAQ 处提到 C++20），并直言“我们不会在学期内教你 C++”。
- **隐形门槛 2：调试能力**。P1 起就是多线程 + 死锁 + 内存泄漏，工具链是 GTest + ASAN/LSAN + TSAN + gdb。不会用调试器的人会在 P1 就卡死。
- **工作量**：作业页给出 6 次作业、5 个项目，项目页给出的节奏是 P1 四周、P2 四周、P3 四周、P4 四周（均含 buffer）。P2 页面有一句加粗警告：*This project is more difficult than previous projects/homework! We strongly recommend that you START EARLY*。

1.3 评分与考试（以 syllabus 页 Fall 2026 版本为准）

大纲页的权重表（“approximately”）：

组成	权重	说明
Homeworks	15%	每次作业满分 100，**等权**计入；逾期**不接受**
Programming Projects	40%	见下方拆分
Midterm Exam	20%	期中，课堂内进行（日程表：Oct 07）
Final Exam	25%	期末，大学统一考试周

40% 的项目分再拆为：

子项	权重	说明
项目提交（5 个）	30%	每个项目满分 100，**等权**
Debug & Reasoning Check（DRC）	10%	**两次 30 分钟当堂笔试**（日程表：Oct 05 与 Dec 02）

DRC 的形式（FAQ Q2）值得单独记住：给你**你自己提交过的项目代码**的打印稿（注释可能被删掉，变量名保留），让你解释它怎么工作，或者**注入一个 subtle bug** 让你定位、说明现象、给出修法。也就是说：**你不只要让测试变绿，还要能在纸上讲清楚自己的代码**。此外 FAQ Q1/Q4 说明还会抽 **Code Reasoning Interview（CRI）**，每个学生**至少被叫到一次**，且可能问的是**更早的项目**。

几条容易踩的政策细节（均引自 syllabus / 项目页）：

- **P0 必须满分通过**（Sep 06 前），否则“被建议退课，且后续作业一律计 0”；P0 不允许用 late day。

- 项目晚交：每人共 **4 个日历日** 的自动延期，只适用于项目；每迟 24 小时扣 **10%**；作业晚交不受理。
- 项目**不公布**评分测试与答案；本地测试只是子集（P1/P2 页面明确写了“we will use additional test cases”，P0 与 P3 页面写“no hidden tests”）。
- 代码风格用 Google C++ Style Guide + clang-tidy 检查，**风格不过关项目记 0 分**。
- 每个项目都要交一页 design document（`docs/pN-design.md`），`make submit-pN` 会一并打包。

1.4 25 讲的讲题地图（来自 schedule 页，Fall 2026）

下表把日程页的日期/讲号/主题、站点上的讲义文件名、教材章节对应起来。“讲义文件”列写的是**实际可下载**的文件名；未发布的留空并在 1.6 节说明。

#	日期	主题（页面原文）	讲义文件（notes / slides）	教材章节
01	Aug 24	Relational Model & Algebra	01-relationalmodel	Ch. 1–2
02	Aug 26	Modern SQL	02-modernsql	Ch. 3–5
03	Aug 31	Database Storage I	03-storage1	Ch. 12.1–12.4, 13.2–13.3
04	Sep 02	Memory Management	04-bufferpool	Ch. 13.2–13.5
05	Sep 09	Database Storage II	05-storage2	Ch. 14.8.1, 24.2
06	Sep 14	Storage Models & Compression	06-storage3	Ch. 11.2, 13.6
07	Sep 16	Hash Tables	07-hashtables	Ch. 14.5, 24.5
08	Sep 21	Indexes & Filters I	08-indexes1	Ch. 14.1–14.4
09	Sep 23	Indexes & Filters II	09-indexes2	Ch. 14.1–14.4, 24.1
10	Sep 28	Index Concurrency Control	10-indexconcurrency	Ch. 18.10.2
11	Sep 30	Sorting & Aggregations Algorithms	11-sorting	Ch. 15.4–15.5
12	Oct 05	**DRC**（当堂，无讲义）	—	—
13	Oct 19	Joins Algorithms	12-joins（文件编号比讲号小 1）	Ch. 15.4–15.6
14	Oct 21	Query Execution I	13-queryexecution1	Ch. 15.1–15.3, 15.7
15	Oct 26	Query Execution II	14-queryexecution2	Ch. 22
16	Oct 28	Query Planning & Optimization I	15-optimization1（仅有 slides）	Ch. 16
17	Nov 02	Query Planning & Optimization II	16-optimization2（仅有 slides）	Ch. 16
18	Nov 04	Concurrency Control Theory	未发布	Ch. 18
19	Nov 09	Two-Phase Locking Concurrency Control	未发布	Ch. 18.1–18.3, 18.9
20	Nov 11	Timestamp Ordering Concurrency Control	未发布	Ch. 18.5–18.6
21	Nov 16	Multi-Version Concurrency Control	未发布	Ch. 18.7–18.8
22	Nov 18	Database Logging	未发布	Ch. 19.1–19.8
23	Nov 23	Database Recovery	未发布	Ch. 19.1–19.9

#	日期	主题（页面原文）	讲义文件（notes / slides）	教材章节
24	Nov 30	Distributed Databases	未发布	Ch. 20.4–20.5, 20.7, 21, 22.9, 23.1–23.4
25	Dec 02	**DRC**（当堂，无讲义）	—	—

> **文件编号陷阱**：因为第 12 讲是 DRC，notes/slides 的文件编号从第 13 讲起**比讲号小 1**（第 13 讲 Joins 的文件是 `12-joins.pdf`，第 16 讲 Optimization I 是 `15-optimization1.pdf`）。照讲号去猜文件名会 404，这是最容易浪费半小时的坑。
>
> 另外标题也有漂移：日程页第 14/15 讲写作 *Query Execution I/II*，而讲义内部标题是 *Query Processing I/II*。指同一节课。

1.5 五个项目 + 六个作业总览

作业页（assignments）的表格，加上各项目页的 due date：

类型	名称	发布	截止	主题落点
Homework	SQL	Aug 26	Sep 06	关系代数 / SQL 语义
Homework	Storage	Sep 09	Sep 20	槽式页、日志结构、压缩（hw2 PDF：30+35+35 分）
Homework	Indexes & Filters	Sep 23	Oct 04	线性/可扩展/布谷鸟哈希、B+ 树、Bloom filter（hw3 PDF：18+20+27+20+15 分）
Homework	Execution & Planning	Oct 21	Nov 01	排序、连接、执行/优化（hw4 PDF：32+44+24 分）
Homework	Transactions	Nov 04	Nov 22	ACID / 并发控制
Homework	Recovery	Nov 23	Dec 06	日志 / ARIES
Project	P0 C++ Primer	Aug 24	Sep 06	**Robin Hood 哈希集合**（Fall 2026 版，见第 32 章）
Project	P1 Buffer Pool Manager	Aug 31	Sep 27	**ARC 替换策略** + Disk Scheduler + BPM + Page Guard
Project	P2 Database Index	Sep 28	Oct 25	B+ 树（含 tombstone buffer）、迭代器、latch crabbing
Project	P3 Query Execution	Oct 19	Nov 15	扫描/增删改/索引扫描 + 聚合/连接 + 排序/窗口 + 优化规则
Project	P4 Concurrency Control	Nov 09	Dec 06	事务/锁管理（**规格页截至 2026-11 未发布**）

> **警告（必须如实标注的政策）**：Fall 2026 对 AI 的使用有硬性边界。syllabus 的 *Fall 2026 Update* 与 FAQ Q1 都写明：**P1 与 P2 必须在没有 coding agent 的前提下完成**；**P3 与 P4 允许使用 coding agent，但需要写一页 reflection**（FAQ 的表述是 P3、P4 **都**需要；syllabus 的措辞只点了 P4，两处略有差异，**以课程当学期主页/FAQ 为准**）。所有项目都要求 **code ownership**：能解释、能辩护、能修改你提交的每一行代码；无法证明则可能被下调分数或判 AIV。**作业一律禁止使用 AI 工具**。因此本笔记的项目章节只提供“目标—关键类—实现顺序—测试读法”，不提供可直接粘贴的实现，也不建议把项目代码外包给 AI。

1.6 材料可得性与本文的写作边界（重要）

翻遍 Fall 2026 站点后，材料的“形状”是这样的：

材料	可得性（2026-11）	本文如何使用
slides 01–16	全部 200，可下载	用作图示与讲师强调点的补充
notes 01–14	全部 200，`pdftotext` 质量良好	**主力依据**（每讲 3–8 页，结构化）
notes 15–24	**未发布**（站点上无文件）	主题切换为 **Fall 2025 归档 notes**，逐章标注
作业 PDF	hw2/hw3/hw4 可下载（hw1 未在文件中挂出）	用于“考点形状”，不抄题面答案
项目规格	project0–project3 完整；project4 目录**只有 `img/` 与 `faq.html`**，无规格正文	P4 只写“大纲已声明的范围 + 依赖关系”，不编造任务
视频	仅 CMU 学生可看；非 CMU 看 Fall 2025 YouTube（FAQ Q11）	本文不引用视频时间点

因此本文的策略是：**能用本学期材料的地方严格用本学期材料；不能用本学期材料的主题，用上一学期同学术主题的归档讲义补位，并在小节标题或正文首句写明“本节依据 Fall 2025 归档”**。凡是“归档讲义 + 本学期教材章节”都没覆盖的细节（例如某个具体系统的实现细节），一律标注“推断”或不写。

1.7 术语约定（读后文前先对齐）

中文	英文（首次出现即给）	本课语境下的精确含义
页	page	逻辑数据块，BusTub 中 **8192 字节（8 KB）**
帧	frame	内存里容纳一个页的固定 8 KB 槽位
缓冲池	buffer pool	帧数组 + 页表 + 替换策略 + 元数据
页表	page table	内存哈希表：page_id → frame（**不是** OS 的页表）
页目录	page directory	磁盘上：page_id → 文件+偏移；与 page table 不同
闩	latch	保护**内部数据结构**的底层原语（mutex/spinlock），不可回滚
锁	lock	保护**数据库内容**的高层原语，事务级，可回滚、可死锁检测
事务	transaction	一组读写操作的原子序列，`BEGIN` … `COMMIT` / `ABORT`
流水线断点	pipeline breaker	必须等子算子吐完所有元组才能继续的算子（join 的 build 侧、ORDER BY）
记录 ID	record id / RID	(page_id, slot_num)，BusTub 中 `RID` 为 64-bit

第 2 章 第 01–02 讲：关系模型、关系代数与现代 SQL

2.1 动机

DBMS 的第一层抽象决定了后面所有物理设计：如果数据模型是“一组无序的元组集合”，那么“顺序”就必须由**索引**或**排序算子**在物理层重新制造出来——这解释了为什么后面 B+ 树、external merge sort、clustered index 会反复出现。

2.2 机制：三层抽象

```
逻辑层          关系（relation）= 无序元组集合，schema 定义属性与域
   │
语言层          关系代数（σ π ⋈ × ∪ −）→ SQL（声明式）
   │
物理层          页 / 记录 / 索引 / 算子（本课 80% 的篇幅）
```

关系模型的三条“硬约束”，后面每一讲都会回来收账：

1. **无序**：表没有“第 n 行”的概念 → 要 `ORDER BY` 就必须排序或走有序索引（第 11 讲）。
2. **集合语义**：默认去重；`DISTINCT` / `UNION` 要真去重 → 需要排序或哈希（第 11 讲）。
3. **NULL 是三值逻辑**：`NULL = NULL` 为 UNKNOWN → 影响谓词求值、聚合、连接（P3 的聚合与 NLJ 都要专门处理）。

关系代数的六个基本算子与 SQL 的对应关系：

代数	符号	SQL 对应	物理实现落点
Selection	σ	`WHERE`	Filter 算子 / 谓词下推
Projection	π	`SELECT col…`	Projection 算子 / 列裁剪
Join	⋈	`JOIN … ON`	NLJ / sort-merge / hash join（第 12 章）
Cross product	×	`FROM a, b`	一般被改写为 join
Union	∪	`UNION`	去重或不去重
Difference	−	`EXCEPT`	通常排序/哈希实现

2.3 机制：Modern SQL 的“窗口”与“递归”

第 02 讲的重点不是 `SELECT` 语法，而是**声明式语言里那些“必须由执行引擎特殊支持”的构造**：

构造	为什么需要引擎支持	后面的落点
窗口函数 `OVER (PARTITION BY … ORDER BY … frame)`	需要“分区内有序 + 逐行滑窗”的执行模式，不是简单聚合	P3 Task #4 要写窗口算子
CTE / `WITH`	可能被物化，也可能是优化器的展开点	优化器改写
`GROUPING SETS` / `ROLLUP` / `CUBE`	一次扫描产出多组聚合	聚合算子的扩展
递归 CTE	需要迭代到不动点	一般不在项目范围

构造	为什么需要引擎支持	后面的落点
`NULLS FIRST/LAST`	排序语义的一部分，影响索引可用性	P3 external merge sort 的排序模式

> 关键洞察（贯穿全课）：SQL 是**声明式**的，所以“怎么写”与“怎么算”解耦——这正是优化器存在的理由（第 15–16 章）。而“声明式”的代价是：**同一句 SQL 可能有 4^n 种连接顺序**，优化器必须限缩搜索空间。

2.4 关键数据结构与不变量

- **Schema / Catalog**：属性名、类型、长度、约束；系统目录本身也是表（第 5 讲 system catalogs）。
- **不变量**：任何算子输出的元组必须符合 plan node 声明的 schema（P3 页面明确要求“output schema exactly as specified in the plan node, or they will fail our unit tests”）。这条不变量是 P3 最容易翻车的地方——多加一列、少加一列都会让整个 SQLLogicTest 失败。
- **表达式树**：`ColumnValueExpression`（`#0.0` = 第 0 个孩子的第 0 列）、`ConstantExpression`、`ArithmeticExpression`、比较/逻辑节点。

2.5 经典陷阱

1. **把 `#0.0` 当成“第 0 行第 0 列”**。它是“第 0 个**孩子算子**的第 0 列”。join 里常见 `#0.0 = #1.0`，表示左孩子第 0 列等于右孩子第 0 列。
2. **以为 `SELECT *` 不需要 Projection**。binder 会把 `*` 展开成实际列，planner 仍可能生成 Projection，优化器再把它消掉（P3 背景章节的例子正是如此）。
3. **忘记 NULL 的三值逻辑**：`WHERE` 只保留谓词为 TRUE 的行；`NULL`/UNKNOWN 一律丢弃。聚合里 `COUNT(*)` 与 `COUNT(col)` 的差别就来自这里。
4. **把 `EXCEPT` / `UNION` 当成“顺序敏感”**。集合运算不保证顺序，除非显式 `ORDER BY`。

第 3 章 第 03 讲：数据库存储 I —— 页、堆文件与记录布局

> 依据：`notes/03-storage1.pdf`（Fall 2026），教材 Ch. 12.1–12.4、13.2–13.3。

3.1 动机：为什么 DBMS 不直接 `mmap` 一个文件

磁盘是**块寻址**的：要读某个字节，必须先把包含它的 4 KB 块读进内存。OS 的 page cache 看起来能替我们做这件事，但讲义列了四条反对理由（并配了一句狠话 *The Operating System is not your friend*）：

问题	具体后果
Transaction safety	OS 可以在任意时刻刷脏页，DBMS 无法保证 WAL 顺序
I/O stalls	DBMS 不知道哪些页在内存里，缺页会**随机**阻塞某个线程
Error handling	任何访问都可能触发 `SIGBUS`，难以校验页有效性
Performance	OS 内部数据结构争用、TLB shootdown

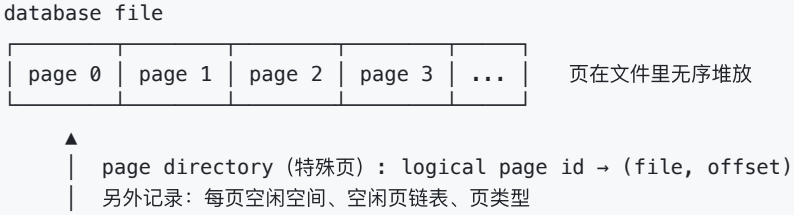
于是 DBMS 自己造一层 cache：**buffer pool**（第 4 章）。这是本课第一个“自建轮子优于复用 OS”的论据，后面还会出现（磁盘调度、日志）。

3.2 机制：页的三个层次

```
硬件页（hardware page）   通常 4 KB —— 设备保证“原子写”的粒度
OS 页   （OS page）        4 KB
数据库页（database page）  512 B ~ 32 KB —— DBMS 自己选
```

- **读写型负载**（OLTP）倾向小页（4–16 KB）：一次点查不必拖进无关数据。
- **读重型负载**（OLAP）倾向大页（讲义提到可达 ⩾ 1 MB）：摊薄每次 I/O 的固定开销。
- **原子写边界**：设备只保证硬件页大小的原子写。若**数据库页 > 硬件页**，写到一半崩溃就会留下 **torn page（撕裂页）**——这正是后面 WAL/ARIES 与 checksum 存在的根本原因之一。

3.3 机制：堆文件（heap file）与页目录



堆文件 = **无序**的页集合。定位一页靠 page directory；定位一个元组靠 **RID = (page_id, slot_num)**。

3.4 机制：槽式页（slotted page）——本课最重要的一个布局



四条不变量（P1/P2 的 bug 大多源于违反其中之一）：

1. **槽数组与数据区相向生长**，`free_ptr` 永远指向“数据区最低地址”，槽数组末端不得越过它。
2. **RID 的 slot_num 一旦分配就不变**（即使元组被移动）；移动元组只需改槽里的偏移。
3. **删除元组 = 清空槽**（并把空间归还给空闲区/做压缩），不能让中间出现“洞”导致后续扫描读到垃圾。
4. **变长字段不内联**时要用 overflow page，元组里只留指针。

对应的三种经典操作代价（讲义明确列了 order-oriented 存储的三个痛点）：

操作	代价来源
插入	找一个有空槽的页（可能需要 page directory 帮忙）

操作	代价来源
更新（变长）	新值放不下 → 标旧值删除 + 插入新元组 → **行迁移**，索引要更新
删除	产生碎片；块设备必须整块读写 →“无用 I/O”

3.5 机制：元组布局与数据表示

```
tuple
┌──────────────────┬──────────────────────────────────┐
│ Tuple Header     │ Tuple Data                       │
│ · 可见性信息     │ · 属性按建表顺序存放             │
│   (txn id...)    │ · 必须字对齐 (word aligned)      │
│ · NULL bitmap    │ · 变长字段：长度+字节 或 指针    │
└──────────────────┴──────────────────────────────────┘
```

类型	存储方式（讲义口径）
INT/BIGINT/SMALLINT/TINYINT	原生 C/C++ 布局
FLOAT/REAL	IEEE-754
NUMERIC/DECIMAL	定点/任意精度，变长二进制（精确但慢）
VARCHAR/BLOB	长度头 + 字节，或指向 overflow page 的指针
TIME/DATE/TIMESTAMP	32/64 位整数（Unix epoch 起的微/毫秒）

两个值得记的工程技巧：

- **German Strings**：≤16 字节的字符串**完全内联**；更长的存 `（长度，前缀，指向 overflow 的指针）`，使得大量谓词比较不必去取 overflow 页。
- **NULL 的三种表示**：集中式 **NULL bitmap**（行存主流）、**特殊值**（列存主流，如 INT32_MIN）、**每属性 flag**（要 padding，最差）。

3.6 关键数据结构与不变量（本章层面）

结构	关键字段	不变量
Page header	page size, checksum, DBMS version, txn visibility, LSN	checksum 必须覆盖整页内容，否则 torn page 检测失效
Slotted page header	num_slots, free_space_ptr	`slot_array_end <= free_space_ptr`
Page directory	page_id → (file, offset), free-space 统计	必须与真实页同步，且**持久化**（重启后要能找到页）
Tuple header	txn id / NULL bitmap	变长属性的长度信息必须可自解释（self-contained）

3.7 经典陷阱

1. **混淆 page directory 与 page table**。前者在**磁盘**上（page → 文件位置），后者在**内存**里（page → 帧）。这是第 4 讲反复强调的一对概念，考试爱考。
2. **假设“页大小 = 硬件页”**。BusTub 的页是 8 KB，而设备原子写粒度通常 4 KB —— 所以“一次写 8 KB”并不原子。
3. **把 RID 当作稳定主键**。讲义明确：RID 是物理位置，应用程序**不能**依赖它（更新可能导致行迁移）。

4. **认为“删除元组就释放了空间”**。碎片需要压缩（compaction）或 freelist 管理，否则页会被“看起来满、其实空”的状态拖死。

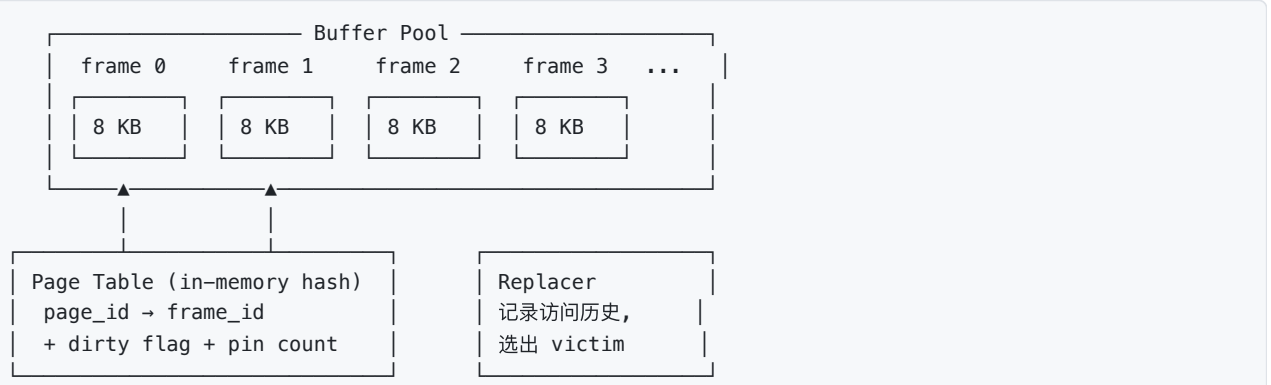
第 4 章 第 04 讲：内存与磁盘管理 —— Buffer Pool

> 依据：`notes/04-bufferpool.pdf`（Fall 2026），教材 Ch. 13.2–13.5。P1 直接考这一讲。

4.1 动机

DBMS 要假装“整个数据库都在内存里”，但内存装不下。于是需要一层缓存，把**空间控制**（页在磁盘上放哪儿）与**时间控制**（何时把页换进来/换出去）都握在自己手里。讲义把优化目标写成两条：spatial control（常一起用的页放近）+ temporal control（减少因缺页造成的停顿）。

4.2 机制：buffer pool 的解剖



三个元数据字段决定了全部正确性：

字段	含义	违反后果
pin count	有多少线程正在用这一页	归零前被驱逐 → 线程读写到别人的页（use-after-free 级别的错）
dirty flag	页被修改过、尚未落盘	驱逐时漏写 → 丢更新
page table 条目	page → frame 的唯一映射	同一页出现两份 → 两个版本互相覆盖（P1 规格里那个 “Data race ☠” 例子）

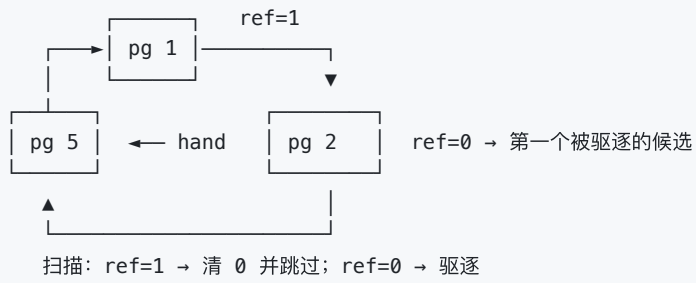
> **写回缓存（write-back）** vs 写穿（write-through）：buffer pool 是 write-back —— 脏页先留在内存，驱逐时才写盘。这带来“必须遵守 WAL 顺序”的责任（第 21 章）。

4.3 机制：替换策略谱系

策略	依据	优点	缺点
LRU	最近一次访问时间戳	直观	需维护有序结构，开销大；**顺序 flooding**

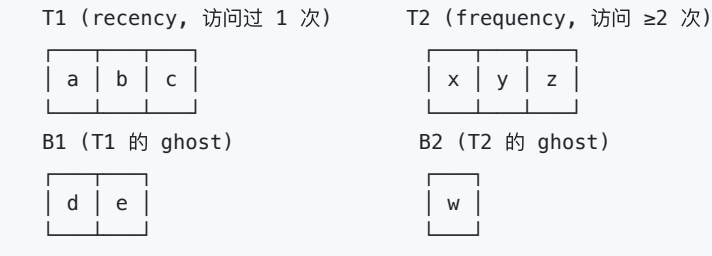
策略	依据	优点	缺点
CLOCK	每页 1 个 reference bit + 时钟指针	近似 LRU，代价低	仍是“只看新旧不看频次”
LFU	访问计数	抗“偶发扫描”	忽略时间，历史热点永不淘汰
LRU-K	最近 K 次访问的时间戳，用**间隔**预测下次访问	比 LRU 稳	存储开销大；需要 ghost cache
ARC	T1(recency) + T2(frequency) + 两个 ghost list + 自适应目标 p	同时兼顾 recency/frequency	实现复杂（**P1 就考它**）

CLOCK 的手绘图（考试常要求画）：



LRU-K 的核心指标（P1 之外的课程内容，但概念要清楚）：对页 p 维护最近 K 次访问时刻 `t1 > t2 > … > tK`，用 `t_{K-1} - t_K`（第 K 近与第 K-1 近的间隔）作为“下次还会不会被访问”的预测；K 越大越抗扫描，但冷启动越慢。工程上常用 **LRU-2 + ghost cache**（被驱逐页的 page_id 历史），命中 ghost 时给更高优先级，避免“刚驱逐又被读回”。

ARC 的四个列表（P1 Task #1 的规格原文结构）：



4.4 机制：把 OS 挡在门外

- **Direct I/O（`O_DIRECT`）**：绕过 OS page cache，避免“同一页在 OS cache 和 buffer pool 各存一份、且两套淘汰策略打架”。PostgreSQL 是反例（用 OS cache）。
- **fsync 的坑**（讲义 side note）：默认情况下 fsync 的错误会被静默吞掉，甚至在出错时把页标记为 clean。
- **后台写（background writing）**：周期性扫 page table 刷脏页，把“驱逐时的同步写”摊平成后台 I/O。

4.5 关键数据结构与不变量

结构	不变量
Page table	一个 page_id 在池中**至多一个** frame；`frame.page_id == INVALID` 表示该帧空闲
Pin count	`pin_count > 0` ⇒ 不可驱逐；`pin_count == 0` ⇔ replacer 中标记为 evictable（**必须同步维护**）
Replacer	`Size()` 只统计 **evictable** 帧；`Evict()` 在无可驱逐帧时返回空

结构	不变量
Dirty bit	脏页驱逐前必须先 `Flush`；`FlushPage` 后清 dirty

4.6 经典陷阱

1. **pin/unpin 不配对**：异常路径提前 return，pin count 泄漏 → 池子逐渐“全是钉子”，最后 OOM。
2. **在持有 page latch 时再去拿 buffer pool latch**，与“先 BP latch 再 page latch”的线程形成环 → 死锁。
3. **忘记 `SetEvictable(false)`**：pin 了但 replacer 认为可驱逐，于是页被换出（P1 里最经典的 bug）。
4. **以为 pin 住就“别人读不到”**。pin 只防驱逐，不防并发读写；并发控制要靠 page latch（读/写闩）。
5. **顺序扫描把池子冲干净（sequential flooding）**：大表扫描把热点页全冲走。解法：buffer pool bypass（扫描算子自留内存）、per-query 局部化、或 ARC/LRU-K 这类抗扫描策略。

第 5 章 第 05 讲：数据库存储 II —— 日志结构存储与索引组织表

> 依据：`notes/05-storage2.pdf`（Fall 2026），教材 Ch. 14.8.1、24.2。

5.1 动机

槽式页假设“可以原地更新（in-place update）”。但很多存储介质（HDFS、对象存储、部分云盘）**只允许追加**；而且原地更新带来随机 I/O、碎片、写放大。日志结构（log-structured）把“更新”变成“追加一条记录”。

5.2 机制：LSM Tree 的写入路径

```
write ──► MemTable (内存, 有序: skiplist / sorted map)
              │  满了
              ▼
          flush → SSTable L0 (不可变、按 key 排序、整体顺序写)
              │
              ▼  compaction (sort-merge, 只保留每个 key 最新版本)
      L1 (key 范围不重叠) ──► L2 ──► L3 ...
```

组件	作用
MemTable	吸收写，内存有序结构
SSTable	不可变、顺序写、可二分查找
SummaryTable	每张 SSTable 的 min/max key + 每层 Bloom filter，用来**跳过不可能命中的 SSTable**
Compaction	合并 SSTable、丢弃旧版本/墓碑，回收空间

两种 compaction 策略（讲义对比）：

策略	结构	适合
Universal compaction	单一“universal”层，按大小/重叠阈值触发	**写密集**、时间序查询
Level compaction	L0 可重叠，L1+ 层内 key 范围不重叠，逐层下推	**读密集**

5.3 机制：读路径与代价

读一个 key：`MemTable` → L0 各表（可重叠，可能要多张）→ L1 → L2 …，**遇到第一个命中即可停**（只关心最新版本）。

操作	代价特征
写	顺序写，快；但 compaction 造成 **write amplification**
读	可能要探多层；靠 Bloom filter + min/max 剪枝
空间	旧版本 + 墓碑会占空间，直到 compaction

5.4 机制：索引组织表（index-organized storage）

把表本身存成索引结构（B+ 树 / skiplist / trie）的 value，页内按 key 排序。对比：

存储方式	元组定位	范围扫描	典型场景
槽式页 + 二级索引	索引给 RID，再回表	需回表，随机 I/O	通用行存
日志结构	多层查找 + 版本合并	需 merge 多层	写密集 KV
索引组织	索引即数据（clustered）	**顺序 I/O，最优**	MySQL InnoDB 主键表

5.5 关键数据结构与不变量

- **SSTable 不可变**：一旦落盘不再修改，只做合并 —— 这让读路径可以无锁、让缓存友好。
- **每层 key 范围有序**：除 L0 外同层不重叠，否则“遇到第一个命中即停”就不成立。
- **墓碑（tombstone）**：删除必须写成一条 DELETE 记录，且在 compaction 之前不能被丢弃，否则旧值会“复活”。
- **系统目录（system catalog）**：DBMS 自己的元数据（表、列、索引、用户权限、统计信息）也存成表，用特殊代码 bootstrap。

5.6 经典陷阱

1. **把 LSM 当成“写一定更快”**。写放大在小 value + 多层 compaction 下非常可观。
2. **忘了墓碑的生命周期**：过早回收墓碑 → 已删除的 key 复活。
3. **读路径假设“一张 SSTable 一个 key 只出现一次”**。实际同层/跨层都可能有多版本，必须按“最新优先”判断。
4. **把 page directory 与 catalog 混为一谈**：catalog 是“有哪些表/列/索引”，page directory 是“页在文件的哪个偏移”。

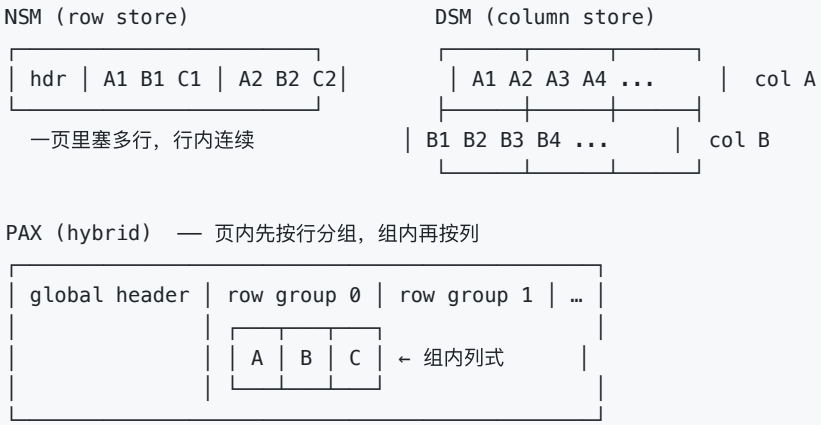
第 6 章 第 06 讲：存储模型与压缩

> 依据：`notes/06-storage3.pdf`（Fall 2026），教材 Ch. 11.2、13.6。hw2 的后两题（35+35 分）就考这一讲。

6.1 动机：负载决定布局

负载	特征	布局倾向
OLTP	短事务、写多、单实体访问	**行存（NSM）**
OLAP	长查询、读多、扫大范围少列	**列存（DSM）**
HTAP	两者混合	PAX / 双格式（fractured mirror、delta store）

6.2 机制：NSM / DSM / PAX



模型	优点	缺点
NSM	插入/更新快；整行读取一次 I/O；可用索引聚簇	扫部分列时浪费 I/O；压缩难（页内多种值域）
DSM	只读需要的列；局部性好；**压缩率高**	点查/增删改要“拼元组”；写放大
PAX	兼顾页内列式处理与行级空间局部性；Parquet/ORC/Arrow 都是它	实现复杂

> 讲义的原话很实用：**“当 DBMS 说自己是列存时，它很可能用的是 PAX。”**

DSM 如何把元组拼回来：主流是**定长偏移**（各列同 offset 属于同一元组，变长用 padding 或字典压缩）；少数用**嵌入 tuple id**（每列都存一份 id，开销大，讲义直接点评“a bad idea”）。

6.3 机制：压缩的层次与手法

压缩的四条设计原则（讲义）：**定长输出**（除变长池）、**尽量延迟解压（late materialization）**、**无损**、粒度可选（block / tuple / attribute / columnar）。

手法	一句话	适用
Naive（LZ4/Snappy/Zstd…）	通用算法压整页	通用；但每次访问都要解压，且无法延迟物化

手法	一句话	适用
RLE	连续重复值 → (value, offset, length)	排序后高度重复的列
Bit-packing	用更少 bit 表示“都小于声明宽度”的值	小整数域
Patching / mostly encoding	例外值放 lookup table	“大多数”值很小
Bitmap	每个 distinct 值一张 N-bit 位图	**低基数**列（高基数会爆炸：讲义给的例子是 10M 行 × 43k 个 ZIP → 53.75 GB）
Delta	存相邻差值（可再套 RLE）	时间序列、递增 id
Dictionary	值 → 短码 + 字典；需**保序**才能做范围查询	最常用；字符串/枚举

报错式记忆点：bitmap 编码的空间是 `N × D` bit（N 行、D 个不同值），所以“低基数才划算”；字典编码若破坏序关系，`WHERE x > 5` 就无法在压缩数据上直接算。

6.4 关键数据结构与不变量

- **PAX 的 row group header**：记录本组行数、各列偏移、min/max（可当 zone map）。
- **不变量**：同一列内所有值的编码必须同构；字典必须保序（否则范围谓词失效）；变长数据必须能靠 offset 定位。
- **mod log（naive 压缩的补丁）**：写时不减压，先记在 mod log；读时先查 mod log；空闲时把 mod log 合并回压缩页。

6.5 经典陷阱

1. **对高基数列用 bitmap**：空间爆炸，比不压缩还大。
2. **忘了“保序”要求**：用哈希做字典 → 范围查询只能全解压。
3. **以为列存一定快**：点查要拼元组，且更新代价高；“列存适合 OLAP”不是“列存总是更快”。
4. **HTAP 的两种实现搞混**：fractured mirrors 里元组**同时**存在于 NSM 与 DSM；delta store 里元组**只存在一处**（新数据在 delta，迁移后进 DSM）。

第 7 章 第 07 讲：哈希表

> 依据：`notes/07-hashtables.pdf`（Fall 2026），教材 Ch. 14.5、24.5。hw3 前三题（18+20 分）覆盖线性/布谷鸟/可扩展哈希。

7.1 动机

哈希表是 DBMS 内部最常用的数据结构：page table、join 的 build 侧哈希表、聚合哈希表、索引（等值）。它**不保序**，因此不能做范围扫描与部分键查找——这就是为什么 B+ 树仍然是默认索引（第 8 章）。

7.2 机制：静态哈希三兄弟

```
线性探查 (linear probe), capacity=8, h(k)=k%8
 idx:  0     1     2     3     4     5     6     7
      [6]   [14]  [22]  [–]   [–]   [–]   [–]   [–]
        6→6   6→7   6→0(绕回)       删除 14 → 墓碑 [T]
      查 22: 6 → 7(墓碑继续) → 0 命中
```

方案	插入	查找	删除	备注
Linear probe	线性找空位	走到空槽即失败	**墓碑**或搬移	最快、cache 友好；高负载因子下 **primary clustering**
Robin Hood	探测距离大者**抢位**，被抢者继续	同线性	同线性	让探测距离分布更均匀，压低长尾
Cuckoo	多哈希函数/多表，踢出旧元素重哈希	**O(1)**（两三个位置）	O(1)	插入可能成环 → 换种子或扩容

Robin Hood 的“抢位”规则值得手推一遍（P0 就考它）：

```
capacity=4, h(k)=k
insert 0 → [0(0), _, _, _]
insert 4 → [0(0), 4(1), _, _]        4 的 home=0, 探测距离 1
insert 1 → [0(0), 4(1), 1(1), _]     home=1 被 4 占, 4 的 dist=1 不小于 1 的 dist=0 → 1 让路
insert 8 → home=0, 走到 idx2 时 dist=2 > 1 的 dist=1 → 8 抢位, 1 被挤到 idx3
最终     [0(0), 4(1), 8(2), 1(2)]
```

7.3 机制：动态哈希

方案	扩容方式	关键点
Chained	每个槽挂链表（可加 Bloom filter 剪枝）	最常见；链表可溢出到多页
Extendible	分裂 bucket + 增加“检查的位数”	全局 bit 数 = 目录大小 log；bucket 有局部 bit 数；多个目录项可指向同一 bucket
Linear	**split pointer** 轮转分裂，不必一次重建	查找时先 h1，若落在 split pointer 之上再用 h2

Extendible hashing 的不变量（hw3 有 20 分在这里）：

1. `global_depth` = 目录项数的 log2；目录项 `i` 指向“用 `i` 的低 `global_depth` 位”确定的 bucket。
2. 每个 bucket 的 `local_depth ≤ global_depth`；多个目录项可共享同一 bucket。
3. bucket 溢出 → `local_depth++` 并分裂；若 `local_depth > global_depth`，则**目录翻倍**（`global_depth++`）。
4. 删除导致 bucket 空 → 可合并（`local_depth--`），必要时目录减半。

7.4 关键数据结构与不变量

- 槽内必须同时存 **key 与 value**（否则无法判定“这个槽是不是我要的 key”）。
- 非唯一键两种处理：链表溢出 / **冗余存重复 key**（后者更常见，线性探查照样工作）。
- 变长 key：把 key/value 放独立临时表，哈希表里存 `hash → RID`。
- 版本号技巧：清空整表时**递增表版本号**，槽内版本号不匹配即视为空，免去逐槽标记。
- 并发：静态哈希的 latch 粒度可以是 global / page / slot；也可用 **CAS 做无锁线性探查**（对 null 槽 CAS 插入）。

7.5 经典陷阱

1. **删除直接置空** → 切断了探测链，后面的 key 查不到。必须用墓碑。
2. **把 Cuckoo 的“均摊 O(1)”说成“最坏 O(1)”**。讲义特意加了“Professor's note”：实践中多个哈希函数映射到**同一张表**的不同槽，且哈希本身不保证常数时间。
3. **负载因子算错**：Robin Hood / 线性探查的容量通常取预期元素数的 2 倍。
4. **认为哈希索引能做范围查询**。不能——这也是 B+ 树仍是默认索引的原因。

第 8 章 第 08 讲：索引与过滤器 I —— B+ 树

> 依据：`notes/08-indexes1.pdf`（Fall 2026），教材 Ch. 14.1–14.4。P2 与 hw3 的 B+ 树题（27 分）都以此为准。

8.1 动机

索引是“表属性子集的副本 + 组织方式”，让你不必顺序扫描。B+ 树之所以统治磁盘索引，是因为它把**随机 I/O 转成顺序 I/O**，并且支持等值、范围、前缀、有序扫描四类访问。

8.2 机制：结构与不变量

```
                [ 5 | 9 ]                ← inner: k 个 key, k+1 个孩子
               /    |    \
        [1 3]      [6 7]      [9 13]      ← leaf: key → RID
          ↔          ↔          ↔         ← sibling pointers (Blink 树带来的)
```

形式化定义（讲义原文口径）：

- **m 路搜索树**（m = fanout = 最大孩子数）；
- **完美平衡**：所有叶子同深度；
- **除根外每个节点至少半满**：`⌈m/2⌉ − 1 ≤ #keys ≤ m − 1`；
- **k 个 key 的内节点有 k+1 个非空孩子**；
- 内节点的 key 只是**路标（guide post）**，可以不出现在叶子里。

插入/删除的“copy up” vs “push up”是必背差异：

操作	叶子分裂	内节点分裂
中间 key 的去向	**复制**到父节点（叶子保留）	**上推**到父节点（内节点不保留）
原因	叶子要存全部 key	内节点 key 只是分隔符

8.3 机制：设计选择（考试爱出“为什么”）

维度	选项	取舍
节点大小	HDD ≈ 1 MB，SSD ≈ 10 KB，内存 ≈ 512 B	大节点摊薄 I/O；小节点贴合 cache

维度	选项	取舍
合并阈值	严格半满 / 延迟合并 / 定期重建	严格合并会 **thrashing**（删了又插）
变长 key	指针 / 变长节点 / padding / **key map（slot 数组 + 偏移）**	key map 最常用
节点内查找	线性（可 SIMD）/ 二分 / **插值**	插值最快但只适合整数等有分布规律的键
重复 key	重复存储 / 存一次 + value 列表	见 deduplication

六种常见优化（讲义 4.1–4.6，P2 的 leaderboard 也围绕它们）：

1. **Pointer swizzling**：页被 pin 住时把 page_id 换成内存指针，省一次 page table 查找；驱逐前要还原。
2. **Write-optimized B+tree（Bε-tree）**：把变更缓存在内节点，批量下推 —— **P2 的 tombstone buffer 就是它的简化版**。
3. **Prefix compression**：同节点内共享前缀只存一次。
4. **Deduplication**：非唯一索引里相同 key 只写一次 + value 列表。
5. **Suffix truncation**：内节点的 key 只保留“够路由”的最短前缀。
6. **Bulk insert**：先建好叶子链表，再自底向上建索引，避免逐条插入的连续分裂。

8.4 机制：聚簇索引与“索引扫描的页排序”

- **聚簇（clustered）**：表按索引键的顺序存放 → 顺着叶子链表扫就是顺序 I/O，**几乎总优于外部排序**。
- **非聚簇**：按索引顺序回表会反复读同一页 → 优化手段是**先把 RID 收集起来、按 page_id 排序、再逐页取**（index scan page sorting）。这条优化在 P3 的 IndexScan 里有实际价值。

8.5 关键数据结构与不变量

结构	不变量
`BPlusTreePage`	`page_type_` ∈ {invalid, leaf, internal}；`size_ ≤ max_size_`
Internal page	`key_array_[0]` **无效**，查找从 `key_array_[1]` 开始；`value_array_[i]` 是孩子 page_id
Leaf page	`size_` 个 (key, RID)；`next_page_id_` 串成链表
根	`root_page_id` 存在 header page；根分裂时**必须**更新它

> P2 页面特别提醒：**“lookups should always start from the second key”** —— 内节点第 0 个 key 是占位，把它当有效 key 是新手最常犯的错。

8.6 经典陷阱

1. **内节点分裂时把中间 key 也留在节点里**（应当 push up 并删除）。
2. **叶子分裂后忘记维护 sibling pointer**，迭代器扫描到一半断链。
3. **删除时只处理“借”（redistribute）不处理“并”（merge）**，或者合并后忘记删父节点的分隔 key。
4. **根的特殊性**：根可以少于半满；根被合并到 0 个 key 时要**降低树高**并更新 `root_page_id`。
5. **把内节点 key 当成“一定存在于叶子”**。它只是路标，可能根本不存在。

第 9 章 第 09 讲：索引与过滤器 II —— Bloom、跳表、Trie、倒排与向量索引

> 依据：`notes/09-indexes2.pdf`（Fall 2026），教材 Ch. 14.1–14.4、24.1。

9.1 动机：索引回答“在哪里”，过滤器回答“在不在”

- **索引（index）**：给定 key，返回元组位置（RID）。
- **过滤器（filter）**：给定 key，回答“**可能**在集合里 / **一定不**在集合里”。

两者的组合价值在于**省掉昂贵的查找**：链式哈希的每个 bucket 指针挂一个 Bloom filter，过滤器说“不在”就直接跳过整条链（第 07 讲已埋伏笔）。

9.2 机制：Bloom filter 的数学与三条性质

```
m bit 的位图，k 个哈希函数
insert(x):  对每个 h_i(x) % m 置 1
lookup(x):  任一位置为 0 → 一定不在（无假阴性）
            全为 1       → 可能在（有假阳性）
```

性质	结论
假阴性	**永不存在**（这是它能安全用于剪枝的前提）
假阳性	存在；随插入量上升
删除	**不可能**（除非 Counting Bloom / Cuckoo filter）
参数	位图大小 m 与哈希个数 k 决定假阳性率；讲义给出在线计算器作参考

变体：**Counting Bloom filter**（用计数器代替 bit，支持增删）、**Cuckoo filter**（存指纹，支持增删）、**SuRF**（succinct range filter，支持范围过滤的压缩 trie）。

9.3 机制：跳表（skip list）

```
level 2:  ──────────────────── 9 ──────────────►
level 1:  ──────── 4 ──────────── 9 ─── 13 ────►
level 0:  ── 1 ─ 4 ─ 6 ─ 7 ── 9 ─ 11 ─ 13 ──►
每个 key 有 1/2 概率升到上一层；期望每层是下层的一半
```

操作	做法
Find	从顶层向右走，直到下一个值大于目标，再下降一层
Insert	抛硬币决定层数；**自底向上**插入（否则读者会看到“有节点但无下层指针”的中间态）
Delete	先打删除标记（防读者访问已释放对象），由后台线程真正摘除；**先摘上层再摘下层**

优点：无需再平衡、内存比 B+ 树省（不留反向指针）；缺点：**不 cache/disk 友好**，反向扫描难。所以跳表常见于内存结构（MemTable），不是磁盘索引。

9.4 机制：Trie 与压缩

Trie 把 key 拆成 digit（字符串按字节、其它类型按 bit），路径即前缀，**形状只依赖 key 集合、不依赖插入顺序**，所有操作 O(k)（k = key 长度）。

压缩	做法	代价
水平压缩	已知 span 时用数组代替 map	空间随 fanout 固定
垂直压缩	单孩子链合并（**Radix tree**）	丢失完整 key → 可能假阳性，需回表校验

9.5 机制：倒排索引与向量索引（本讲的两个“新东西”）

倒排索引（inverted index）：term → posting list（包含该词的记录列表）。两种工业实现：

实现	词典结构	特点
Lucene	**有限状态转换器（FST）**，边上带权重，累加即偏移	不可变；增量靠“新 FST + 后台合并”
PostgreSQL GIN	B+ 树做词典	posting list 小则存有序 RID 列表，大则再建 B+ 树；另有 pending list 攒批量更新

向量索引（vector index）：LLM 产生 embedding，语义相近则几何相近，于是“语义搜索”变成最近邻（ANN）问题。两类：

类别	结构	例子
倒排式	先 k-means 聚类，再建 centroid → 记录的倒排表；查询时先定位簇再查簇内（可看邻近簇提高召回）	IVFFlat
图式	每个向量连 n 个近邻，查询时贪心走向更近的邻居；可用多层图（类似跳表）加速	FAISS、HNSW

> 讲义的提醒很重要：向量检索**没有唯一正确答案**（近似），而且常需要“先/后过滤”（pre-/post-filter）来结合结构化谓词。

9.6 关键数据结构与不变量

结构	不变量
Bloom filter	位图只增不减；`lookup=false` 是**确定**结论
Skip list	每层有序；上层节点必在下层出现；删除标记先于物理摘除
Trie	节点路径唯一确定前缀；压缩后必须回表验证
倒排表	posting list 有序（便于求交/求并）；词典保序（便于范围查 term）

9.7 经典陷阱

1. **拿 Bloom filter 当“精确集合”**：只允许用它做“否定”判断。
2. **跳表插入自顶向下**：并发读者会踩空指针。
3. **以为 Trie 一定省空间**：字符集大、key 稀疏时指针开销爆炸，必须压缩。
4. **在 B+ 树上做“非前缀”谓词**：`INDEX(a,b,c)` 上只有 `b = 1` 时无法收窄范围（部分系统有 skip scan，但不是通用能力）。

第 10 章 第 10 讲：索引并发控制

> 依据：`notes/10-indexconcurrency.pdf`（Fall 2026），教材 Ch. 18.10.2。P2 Task #4 就是这一讲。

10.1 动机：为什么“加一把大锁”不行

单线程数据结构 + 一把全局读写锁当然正确，但： – 并发度归零（写者互斥所有人）； – 树高 3–4 层，若每层都拿写闩，根成为**串行化瓶颈**。

课程只要求 **physical correctness**（结构不被破坏、指针不悬空），logical correctness 留给事务层（第 17 章起）。

10.2 机制：latch 的三种实现

实现	原语	优点	缺点
Test-and-Set spin latch	CAS / `std::atomic`	单指令、DBMS 自己控制	高争用下忙等、cache line 抖动
Blocking OS mutex	`std::mutex`（futex）	简单	系统调用开销（讲义给的量级是 ~25 ns/次），不可扩展
Reader-Writer latch	`std::shared_mutex`	允许并发读	元数据大；需处理读写队列饥饿（读者优先/写者优先/公平）

10.3 机制：B+ 树的 latch crabbing（螃蟹法）

```
[root]  ← 先闩父
  |
[inner] ← 再闩子；若“子安全”则放掉父
  |
[leaf]  ← 拿到叶子就可以一路放掉祖先
```

“安全”的定义（必须背）：

操作	安全条件
插入	节点**未满**（不会分裂）
删除	节点**超过半满**（不会合并/重分配）
查找	恒安全（不改结构）

两种协议：

协议	下行方式	代价
基础版	一路拿 **写闩**	正确但根节点串行化，并发差
乐观版（optimistic）	先按查找拿**读闩**到叶子，再在叶子上拿写闩；若叶子不安全 → 全部释放、退回基础版重来	分裂/合并稀少时最快

```
// 乐观 crabbing 的伪代码（P2 Task #4 的骨架）
auto ctx = Context{};
ctx.header_page_ = FetchPageWrite(header_page_id_);   // 保护 root_page_id_
auto node = FetchPageRead(root_page_id_);
while (!node->IsLeafPage()) {
  auto child = FetchPageRead(node->ChildAt(IndexOf(key)));
  ctx.read_set_.push_back(std::move(node));           // 保留读闩? 可提前 drop
  node = std::move(child);
}
auto leaf = FetchPageWrite(leaf_page_id);
if (!IsSafe(leaf)) { /* 释放全部闩，restart with pessimistic path */ }
```

10.4 机制：叶子扫描的死锁与“自杀”策略

纵向 crabbing 是**单向**的（只从父到子），所以不可能成环 → 无死锁。但**叶子水平扫描**引入了“两个方向同时抢闩”：

```
T1（扫描）: leaf A ──要──► leaf B
T2（删除）: leaf B ──要──► leaf A     ← 环
```

latch 本身**没有死锁检测**（没有 lock manager 记录谁持有谁），所以只能靠**编码纪律**：水平移动时，“拿到下一个叶子的闩之后才释放当前叶子的闩”，并且**必须支持 no-wait 模式** —— 拿不到就立刻释放自己所有闩、重试整个操作。这也是讲义反复说的“latches 的死锁靠 avoidance，不靠 detection”。

10.5 机制：写操作的回滚需求

既然可能中途放弃，就必须能撤销已做的修改 → 每个 worker 维护私有 **write-set**（节点 id + 已插入/删除的 key）；持写闩直到操作完成，从而**永不向其他线程暴露半成品状态**。

10.6 关键数据结构与不变量

结构	不变量
Latch 顺序	只允许“父 → 子”与“左 → 右（扫描）”两个方向；跨方向必须 no-wait
write-set	记录足够信息以撤销；只有持写闩的节点才能进 write-set
header page 闩	只在“可能改 root_page_id”时持有（插入/删除）；查找不需要
读闩重入	**同一线程不得对同一页拿两次读闩**（会自死锁，P2 页面明确写了）

10.7 经典陷阱

1. **判断“安全”时用了错误的阈值**（例如用 `size_ < max_size_` 判插入安全，但 `max_size_` 是 key 数上限，边界差一）。
2. **在乐观路径里拿着读闩去升级为写闩**：多数 latch 实现不支持升级，直接死锁。
3. **忘了释放祖先的读闩**（read_set 没清理）→ 后续线程全部卡住，表现为“跑得越来越慢”。
4. **叶子扫描拿到后立刻释放当前叶子**：窗口期内另一个线程删了当前叶子，指针悬空。
5. **把事务的 lock 和索引的 latch 混用**：latches 不可回滚、不参与死锁检测；用锁的思路写 latch 代码必然出错。

第 11 章 第 11 讲：排序与聚合算法

> 依据：`notes/11-sorting.pdf`（Fall 2026），教材 Ch. 15.4–15.5。hw4 第 1 题（32 分）就是外部归并排序的算术。

11.1 动机

关系模型不保证顺序，但 `ORDER BY`、`GROUP BY`、`DISTINCT`、sort-merge join 都需要顺序。数据装不下内存时，就要用**外部**算法：把 I/O 次数当作代价。

11.2 机制：外部归并排序（external merge sort）

```
Phase 1 排序：每次读 B 页 → 内存快排 → 写回，得到 ⌈N/B⌉ 个 run
  [==== B 页 ====] → run0
  [==== B 页 ====] → run1
  ...
Phase 2 归并：每次合并 B-1 个 run，用 1 页做输出缓冲
  run0 run1 ... run_{B-2}  ⟶  run0'
```

代价公式（必背）：

量	公式
排序阶段 run 数	`⌈N / B⌉`
归并趟数	`1 + ⌈log_{B-1} ⌈N/B⌉⌉`
总 I/O	`2N ×（趟数）`（每趟每页读一次写一次）
两路归并（B=3）	趟数 = `1 + ⌈log2 N⌉`

两个优化：

- **双缓冲（double buffering）**：预取下一个 run 到第二组缓冲页，用并行 I/O 掩盖延迟；代价是**可用缓冲减半**。
- **Top-N 堆排序**：`ORDER BY … LIMIT n` 只需一次扫描 + 大小为 n 的优先队列；`WITH TIES` 时按需扩容。

11.3 机制：比较本身也可能很贵

技巧	说明
Code specialization	把比较器硬编码/模板特化，避免函数指针
Suffix truncation	先比长字符串的二进制前缀，相同再比全串
Key normalization	把变长 key 编码成保序的定长串
Collation	字符串排序规则不是全局统一的（德语 Ä 与 ASCII 的位置不同）；多级比较：基字母 → 重音 → 大小写

11.4 机制：聚合的两条路

```
排序聚合：  scan → sort(GROUP BY key) → 顺序扫一遍，边扫边累计
哈希聚合：  scan → 建 hash(GROUP BY key → running value) → 输出
            （装不下 → 分区溢出 → 每个分区再哈希）
```

维度	排序聚合	哈希聚合
输出顺序	**天然有序**（省一次 ORDER BY）	无序
计算成本	排序更贵	通常更便宜
溢出处理	外部归并排序	分区（`h1` 分 B−1 个分区，再对每分区用 `h2` 建表）
适用	数据已排序 / 需要有序输出	大多数场景

哈希聚合的 running value 是“增量可合并”的：`AVG` 要存 `(SUM, COUNT)`，`COUNT(*)` 存计数，`MIN/MAX` 存当前极值。

11.5 关键数据结构与不变量

结构	不变量
Run	每个 run 内部有序；run 之间无序（直到最后一趟）
归并堆	每个 run 一个游标；输出页满即刷
聚合哈希表	`GroupByKey → RunningValue`；同 key 只出现一次
NULL 语义	聚合忽略 NULL 输入；空表的 `COUNT(*)` = 0，其它聚合为 NULL；**GROUP BY 的 NULL 要当作一个分组**（P3 的坑）

11.6 经典陷阱

1. **趟数公式用错底数**：是 `B−1` 路归并，不是 `B` 路（要用 1 页做输出）。
2. **忘记 `2N`**：每趟都“读 + 写”，不是只读。
3. **聚合时把 NULL 当值参与 `COUNT`**：`COUNT(col)` 不计 NULL，`COUNT(*)` 计。
4. **在流水线里把聚合的 build 放在 `Next()` 里**：聚合是 pipeline breaker，build 应放在 `Init()`（P3 明确提示）。
5. **Top-N 用了全排序**：`LIMIT` 场景下这是数量级的浪费。

第 12 章 第 13 讲：连接算法

> 依据：`notes/12-joins.pdf`（Fall 2026，文件编号比讲号小 1），教材 Ch. 15.4–15.6。hw4 第 2 题（44 分）是连接代价计算。

12.1 动机

规范化把信息拆开，join 负责拼回去。连接是 OLAP 的主要成本来源，也是优化器最想选对的一步（第 15–16 章）。

12.2 机制：四种算法与代价

记号：外表 R（M 页、m 元组），内表 S（N 页、n 元组），B 个缓冲页。

算法	代价	说明
Naive NLJ	`M + (m × N)`	每个外表元组扫一遍内表
Block NLJ	`M + (M × N)`；有 B 页时 `M + (⌈M/(B−2)⌉ × N)`	按**页**而非按元组
Index NLJ	`M + (m × C)`	C = 每次索引探测的代价；内表必须有索引
Sort-Merge	`sort(R) + sort(S) + (M + N)`；最坏合并 `M × N`	数据已排序时极优
Hash（Grace）	`3 × (M + N)`	分区 `2(M+N)` + 探测 `M+N`

讲义给的对照实验（M=1000, m=100000, N=500, n=40000, B=100, 0.1 ms/IO）：

算法	I/O 量级	讲义给出的耗时量级
Naive NLJ	~10^8 次	约 1.4 小时
Block NLJ	~10^4 次	约 0.65 秒
Sort-Merge	排序 6000 + 合并 1500	约 0.75 秒
Hash Join	3 × 1500 = 4500	约 0.45 秒

> 这张表的意义不是背数字，而是建立“**算法选择能带来 4 个数量级差异**”的直觉。

12.3 机制：Hash Join 的两个阶段与“溢出”

```
Build:  扫描外表 R，建 hash table（key = join key）
        ┌────┬────┬────┬────┐
        │ h1 │ h1 │ h1 │ h1 │  ← 桶
        └────┴────┴────┴────┘
Probe:  扫描内表 S，对每行算 h1 定位桶，再比对原值（解决哈希冲突）
```

- **只能做等值连接**（完整 join key 上的等值）。
- **build 侧必须装得下内存**；装不下 → **Grace Hash Join**：两侧都按 `h1` 分区落盘，逐分区再做 hash join；单分区还太大 → 递归用 `h2` 再分。

- **Bloom filter**（sideways information passing）：build 时顺手建，probe 时先过滤，省掉注定不命中的 I/O。
- **Hybrid hash join**：热点分区留在内存直接算，其余落盘 —— 讲义点评“难以正确实现，实践中很少做”。

12.4 机制：输出物化的两种策略

策略	内容	适合
Early materialization	直接拼出完整输出元组	行存（反正整行都要读）
Late materialization	只带 join key + RID	列存（避免搬运用不到的列）

12.5 关键数据结构与不变量

结构	不变量
Hash table	key = join key 的**拼接**；桶内必须能区分不同 key（比对原值）
分区文件	同一 key 的两个表元组必须落到**同一分区号**（否则永远匹配不上）
排序归并游标	两侧按同一 key 序推进；相等时可能需要“回溯”处理重复 key（最坏 M×N）
外表选择	NLJ 系列应以**页数少**的表为外表（不只是元组少）

12.6 经典陷阱

1. **NLJ 内外表选反**：代价从 `M + M/(B-2)·N` 变成 `N + N/(B-2)·M`，可能差一个数量级。
2. **Hash join 用错 build 侧**：应 build **较小**的表（内存约束）。
3. **忘了处理哈希冲突**：只比 hash 不比原值 → 错误结果（P3 明确要求处理 collision）。
4. **Grace hash 分区后忘了“同 key 同分区”**：用不同哈希函数分两侧 → 结果错。
5. **LEFT JOIN 在 build/probe 反向后语义错**：外表必须是“保留全部行”的那一侧。

第 13 章 第 14 讲：查询执行 I —— 处理模型与访问方法

> 依据：`notes/13-queryexecution1.pdf`（Fall 2026，讲号 #14），教材 Ch. 15.1–15.3、15.7。P3 的执行器全部建立在这一讲的模型上。

13.1 动机

同一个逻辑计划有多种物理执行方式；**处理模型（processing model）** 决定了算子之间怎么传数据、控制流怎么走，从而决定 CPU 效率与内存占用。

13.2 机制：三种处理模型

模型	接口	数据单位	适合	缺点
Iterator / Volcano	`Next()` 返回一个元组或 EOF	1 元组	行存 OLTP；易组合；`LIMIT` 好实现	每元组一次虚函数调用 + 分支
Materialization	`Output()` 一次返回全部	整个结果	OLTP（结果小）	OLAP 中间结果可能溢出到磁盘
Vectorized / Batch	`Next()` 返回一批	`BUSTUB_BATCH_SIZE` 个	OLAP；易用 SIMD；CPU 流水友好	批量大小的调参

```
Volcano:      root.Next() → child.Next() → ... → leaf 吐 1 行，逐层 return
Vectorized:   root.Next() → child.Next() → ... → leaf 吐 1024 行，逐层 return
Push 模型:    leaf 主动 push 到 parent（对 cache/寄存器更友好，但中间结果大小难控）
```

BusTub 的选择（P3 页面原文）：row-based **vectorization**，每个执行器实现 `Next()` 返回最多 `BUSTUB_BATCH_SIZE` 个元组，**同时还返回一批 RID**。

13.3 机制：流水线与断点

- **流水线（pipeline）**：元组可以连续流过一串算子而不落盘。
- **断点（pipeline breaker）**：必须等孩子吐完才能继续 —— join 的 build 侧、`ORDER BY`、聚合、子查询。

> 这条区分直接对应 P3 的工程决策：`Init()` 里做 build（一次性），`Next()` 里做 probe（流式）。

13.4 机制：访问方法（access method）

方法	做法	优化点
Sequential scan	逐页逐槽，谓词过滤	压缩、预取、buffer pool bypass、并行、堆聚簇、zone map、JIT
Index scan	用索引定位	选对索引；聚簇索引最优
Multi-index scan	多个索引各算 RID 集合，再交/并	位图、哈希表、Bloom filter 求交

Zone map（无损跳过）：每页预存 min/max 等聚合，若 `WHERE x > 600` 而该页 max=400，则整页跳过。云数据库里尤其值钱（省的是网络传输）。

索引选择的考虑因素（讲义列出）：索引含哪些属性、查询引用哪些属性、值域、谓词组合、唯一/非唯一。

13.5 机制：修改算子与 Halloween 问题

- `INSERT` 两种实现：算子内部物化 / 直接插入孩子传来的元组。
- `UPDATE` / `DELETE`：孩子传 RID，算子必须**记录已处理的 RID**，否则——
- **Halloween Problem**（1976 年 System R 在万圣节发现）：更新改变了元组的物理位置，扫描算子**再次访问到**同一元组，导致重复更新。解法：跟踪本次查询已修改的 RID。

13.6 机制：表达式求值

`WHERE` 被表示成表达式树（比较、AND/OR、算术、常量、列引用）。求值时要带一个**执行上下文**（当前元组、参数、schema）。三条优化：常量折叠（constant folding）、公共子表达式消除、向量化/JIT。

13.7 关键数据结构与不变量

结构	不变量
Executor 接口	`Init()` 只做一次；`Next()` 可被反复调用直到返回空批次
输出 schema	必须与 plan node 声明**逐列一致**（P3 硬性要求）
批次	每批 ≤ `BUSTUB_BATCH_SIZE`；RID 批次与元组批次一一对应
扫描游标	记录 (page_id, slot) 位置；**前自增 vs 后自增**的语义差异是 P3 的经典坑

13.8 经典陷阱

1. **`Next()` 里重复做 build**：每次调用都重建哈希表 → 正确但慢到超时。
2. **输出被删除的元组**：SeqScan 必须检查 `TupleMeta::is_deleted_`。
3. **前/后自增混用**：`iter++` 返回旧值、`++iter` 返回新值；`TableIterator` 上写错会漏掉第一行或重复最后一行。
4. **以为 Volcano 一定比向量化慢**：点查场景下 Volcano 的常数更小；向量化赢在扫描型负载。
5. **忘了 `LIMIT` 可以提前终止**：Volcano 模型的天然优势，别在 `Next()` 里把子算子读干。

第 14 章 第 15 讲：查询执行 II —— 并行与进程模型

> 依据：`notes/14-queryexecution2.pdf`（Fall 2026，讲号 #15），教材 Ch. 22。

14.1 动机

单线程执行遇到两类瓶颈：CPU（算不动）与 I/O（读不快）。并行执行把查询摊到多核/多机；但“并行 DBMS”与“分布式 DBMS”的假设完全不同。

14.2 机制：parallel vs distributed

维度	Parallel DBMS	Distributed DBMS
节点距离	同一机箱/机架	可跨数据中心
互联	高速 LAN	公网，慢且不可靠
通信代价	可忽略	必须计入代价模型
故障	基本假设不发生	**必须处理**（丢包、节点崩溃）

14.3 机制：进程模型

模型	代表系统	优点	缺点
Process per worker	Postgres、Oracle、DB2（早期）	进程崩溃不拖垮全库	依赖 OS 调度；共享状态要靠共享内存/消息
Thread per worker	SQL Server、MySQL、近 20 年多数系统	上下文切换便宜；共享地址空间	一个线程崩溃可能带走整个进程
Embedded	DuckDB、SQLite、RocksDB	无 IPC，应用自己调度	与宿主应用同生共死

14.4 机制：三种查询内并行

```
Intra-operator（水平）：同一算子切成多份，各处理不相交数据
       ┌── Exchange(Gather) ──┐
    scan A1                 scan A2
Inter-operator（垂直）：算子串成流水线，不同阶段同时跑
    scan ──► filter ──► project    （不同 worker 各站一段）
Bushy（混合）：不同子树同时执行
       join ─┬─── join(A,B)
             └─── join(C,D)
```

Exchange 算子三种：`Gather`（多流合一，最常见）、`Distribute`（一分为多）、`Repartition`（重分区）。

14.5 机制：I/O 并行

方式	说明
Multi-disk parallelism	RAID / 存储设备层面，DBMS **无感知**
Database partitioning	DBMS 自己把表切成不相交子集放到不同磁盘/节点（逻辑分区 vs 物理分区）

14.6 关键数据结构与不变量

结构	不变量
Exchange	下游算子必须等 exchange 收齐所有孩子的数据（它本身是 pipeline breaker）
分区映射	同一 key 的所有元组必须在同一分区（否则 join/聚合出错）
任务划分	每个 fragment 的数据子集**两两不相交且并集为全集**

14.7 经典陷阱

1. **把“多线程”当成“查询内并行”**：thread-per-worker 只保证多请求并发（inter-query），不等于单查询加速。
2. **在 exchange 之上做假设“数据已就绪”**：它是断点。
3. **分区键选错**：数据倾斜 → 某个 worker 成为长尾。
4. **忽略 NUMA**：跨 socket 访问内存的代价可能抵消并行收益。

第 15 章 第 16 讲：查询优化 I —— 规则改写与计划枚举

> 依据：`fall2026/slides/15-optimization1.pdf`（本学期仅有 slides）＋ **Fall 2025 归档** `fall2025/notes/15-optimization1.pdf`，教材 Ch. 16。

15.1 动机

SQL 是声明式的：它说“要什么”，不说“怎么算”。优化器的任务是在**等价**的计划里挑最便宜的一个。讲义把优化称为“构建 DBMS 中最难的部分”，并提到历史上第一个优化器是 1970 年代的 IBM System R —— “在此之前，人们不相信机器能比人写出更好的计划”。

15.2 机制：编译流水线

```
SQL 文本
  │ parser
  ▼
语法树（AST）
  │ binder（查 catalog，把名字换成内部 id）
  ▼
逻辑计划（关系代数树）
  │ rewriter / 优化器
  ▼
物理计划（带访问路径、算法选择、代价标注）
  │ executor factory
  ▼
执行器树
```

BusTub 的 `EXPLAIN` 正好把这三段打出来（P3 页面示例）：

```
=== BINDER ===   BoundSelect { table=..., columns=[...], ... }
=== PLANNER ===  Projection { exprs=["#0.0","#0.1"] } | (schema)
                 MockScan { table=__mock_table_1 } | (schema)
=== OPTIMIZER === MockScan { ... }        ← Projection 被消掉了
```

15.3 机制：逻辑优化（heuristic / rule-based）

规则	作用
谓词下推（predicate pushdown）	尽早过滤，减少上游数据量
谓词重排	选择性高的先算
拆分合取谓词（split conjunctive predicates）	把 `a AND b` 拆开分别下推
笛卡尔积 → join	用 join 谓词消掉 cross product
投影下推（projection pushdown）	尽早裁列，缩小元组
表达式改写	常量折叠、合并冗余谓词、识别恒假谓词

15.4 机制：物理优化（cost-based）

- 枚举候选计划 → 估代价 → 选最小。
- **搜索空间爆炸**：n 路 join 有 `4^n` 种顺序（结合律 + 交换律）。
- **终止条件**：墙钟超时、代价阈值、枚举穷尽、变换次数上限。

两种搜索范式：

范式	方向	代表
Generative / bottom-up	从空开始，逐步搭出计划；动态规划 + **左深树**	System R、DB2、MySQL、Postgres、多数开源
Transformation / top-down	从目标出发，用变换规则 + 实现规则向下搜索；memoization 去重；**物理属性是一等公民**	MSSQL、Greenplum、CockroachDB、Volcano

15.5 机制：单表与多表

- **单表**：核心是选访问方法（seq scan / index scan / 二分…）。OLTP 查询多为 **sargable**（存在最优索引），所以工业界常用简单启发式而非复杂代价模型。
- **多表**：限制搜索空间是关键；常见做法是只考虑**左深树**（left-deep），把 bushy 树留给启发式。

15.6 机制：子查询与表达式

手段	说明
去相关 / 展平（de-correlate / flatten）	把嵌套子查询改写成 join
分解（decompose）	把子查询结果存临时表，分块优化
表达式改写	恒假谓词直接短路；`x BETWEEN 1 AND 150` 合并冗余条件

15.7 关键数据结构与不变量

结构	不变量
逻辑计划	只表达“算什么”，不含算法与访问路径
物理计划	每个逻辑算子映射到一个具体实现 + 访问路径
等价变换	变换前后**结果集必须相同**（顺序除非语义要求，否则不算）
Memo（top-down）	同一 group 的多个等价表达式共享最优代价

15.8 经典陷阱

1. **认为“逻辑等价”就等于“代价相同”**：`A ⋈ B` 与 `B ⋈ A` 结果同，代价可能差几个数量级。
2. **下推谓词越过外连接**：`LEFT JOIN` 的 `WHERE` 条件下推会改变语义（`ON` 与 `WHERE` 不等价）。
3. **投影下推裁掉了排序/分组需要的列**。

4. **以为优化器一定选对**：基数估计误差会级联放大（第 16 章），这也是“优化器仍是研究热点”的原因。

第 16 章 第 17 讲：查询优化 II —— 统计、基数估计与代价模型

> 依据：`fall2026/slides/16-optimization2.pdf` + **Fall 2025 归档** `fall2025/notes/16-optimization2.pdf`，教材 Ch. 16。

16.1 动机

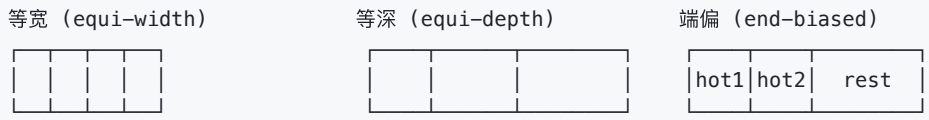
代价模型要回答一个问题：**这个算子会输出多少行？** 这个数字错了，后面所有决策都会错。而“准确估计”与“低成本维护”天然矛盾。

16.2 机制：统计信息的来源

DBMS 在 catalog 里为每个关系 R 维护：

符号	含义
`N_R`	R 的元组数
`V(A, R)`	属性 A 的**不同值个数**（distinct count）
`SC(A, R)`	选择基数 = `N_R / V(A,R)`（平均每个值有多少行）

直方图三兄弟：



每桶宽度相同，简单，但倾斜数据下失真

每桶行数近似相同，代价是桶边界不规则

前 N−1 桶存精确热点计数，最后一桶存其余的平均

此外还有 **采样**（sampling：维护只读样本，变更超过阈值如 10% 时刷新）与 **sketch**（Count-Min Sketch、HyperLogLog）。

16.3 机制：选择性与三条“不成立的假设”

`sel(P)` = 满足谓词 P 的元组比例。等值谓词的经典估算：`sel = 1 / V(A,R)`。合取谓词假设独立则相乘。

三条假设（讲义明确说“现实中经常不成立”）：

假设	内容	被什么打破
均匀性（uniformity）	除热点外取值分布均匀	倾斜数据（Zipf）
独立性（independence）	不同属性上的谓词互不影响	相关属性（城市 ↔ 邮编）
包含性（containment）	内表 join key 的每个值都出现在外表	外键之外的 join、稀疏 key

16.4 机制：连接基数估计

对 R ⋈ S（共享属性 A，非主键）：

```
|R ⋈ S| ≈ (N_R × N_S) / max(V(A,R), V(A,S))
```

若 A 是 S 的主键（R 是外键侧），则 `|R ⋈ S| = N_R`（每行最多匹配一行）。

16.5 机制：误差为什么会爆炸

```
真实 1000 行 ──估成 100──► 下一层再乘一个偏差 ──► 最终差 100 倍
                              → 选错 join 顺序 / 选错算法
```

讲义的原话是“一处估计误差会级联并在后续算子中被放大”。工程上的对策：更好的统计（多列直方图、采样）、自适应执行（adaptive query processing）、以及“猜错就换计划”的重优化。

16.6 关键数据结构与不变量

结构	不变量
Catalog 统计	与实际数据可能**不一致**（异步更新）；优化器必须容忍
直方图	桶的边界有序；桶内计数之和 = N_R（近似）
代价模型	至少包含 CPU 与 I/O 两项；分布式再加 network

16.7 经典陷阱

1. **用均匀假设估倾斜列**：热点值被严重低估（这是“优化器选错索引”的最常见根因）。
2. **把 `V(A,R)` 当成“最大值 − 最小值”**：它是不同值个数。
3. **忘记包含性原则**：内外表搞反时 `max(V(A,R), V(A,S))` 的分母会变，估计值差很多。
4. **以为“统计信息越新越好”**：频繁全表扫描统计会拖慢系统；工业界用采样 + 阈值触发。

第 17 章 第 18 讲：并发控制理论 —— ACID 与可串行化

> 依据：**Fall 2025 归档** `fall2025/notes/17-concurrencycontrol.pdf`（本学期对应讲号 #18，讲义尚未发布），教材 Ch. 18。

17.1 动机：两个“送命题”

- **丢失更新（lost update）**：两个事务同时改同一对象，谁最后写谁赢，另一次修改蒸发。
- **持久性（durability）**：提交后断电，数据还在吗？

事务是“变更的基本单位”，**不允许部分完成**。最笨的正确做法是串行执行 + 每次拷贝整个数据库（shadow paging 的雏形），性能不可接受；于是需要“交错执行但结果等价于某个串行顺序”。

17.2 机制：ACID 逐条落地

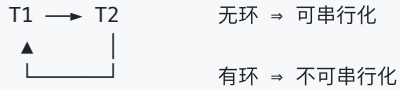
字母	保证	实现手段（本课后半部分）
Atomicity	全做或全不做	Logging（undo）/ Shadow paging
Consistency	约束在事务前后都成立	SQL 约束 + 应用逻辑；DBMS 只保证“若事务各自一致，则串行结果一致”
Isolation	互不干扰，等价于某个串行顺序	2PL / TSO / OCC / MVCC（第 18–20 章）
Durability	提交后不可丢	WAL + fsync（第 21–22 章）

17.3 机制：冲突与可串行化

冲突的三要素：不同事务、同一对象、至少一个是写。

冲突类型	别名	现象
Read–Write	Unrepeatable Read	同一对象两次读结果不同
Write–Read	Dirty Read	读到别人未提交的写
Write–Write	Lost Update	覆盖别人未提交的写

调度 S 是 **conflict serializable** ⟺ 其**优先图（precedence graph）无环**：节点是事务，`Ti → Tj` 表示 Ti 的某操作与 Tj 的某操作冲突且 Ti 在前。



调度空间的包含关系（必背，考试常画成同心圆）：

```
Serial ⊂ ConflictSerializable ⊂ ViewSerializable ⊂ All
                ∪
        StrongStrict2PL（也在 ConflictSerializable 内）
```

View serializability 更宽松（允许盲写 blind write），但要判断“应用怎么解释值”，无法高效实施，**实践中不用**。

17.4 关键数据结构与不变量

结构	不变量
事务状态	`RUNNING → COMMITTING → COMMITTED` / `→ ABORTED`；不可逆
优先图	加边只增不减（一次判定内）；有环即拒绝该调度
对象集合	讨论可串行化时假设对象集合**固定**；有插入/删除就出现幻读（第 19 章）

17.5 经典陷阱

1. **把“临时不一致”当成错误**：转账中间态可见是允许的，**永久不一致**才是错误。
2. **以为冲突可串行化是“唯一正确”**：它比视图可串行化更严格，会拒绝一些其实正确的调度（这是性能代价）。
3. **忽略“DBMS 管不了外部副作用”**：发邮件、调外部 API 无法回滚。
4. **把 `COMMIT` 当成“一定成功”**：DBMS 可以覆盖并 abort（例如检测到冲突）。

第 18 章 第 19 讲：两阶段锁（2PL）与死锁处理

> 依据：**Fall 2025 归档** `fall2025/notes/18-twophaselocking.pdf`（本学期对应讲号 #19），教材 Ch. 18.1–18.3、18.9。

18.1 动机

前面的可串行化是“事后判定”，而 DBMS 需要在**运行时**边执行边保证正确。锁是最直接的动态手段：**先拿锁再访问**。

18.2 机制：锁 vs 闩（这张表必须能默写）

维度	Lock	Latch
保护对象	数据库内容（元组/表/库）	内存数据结构（哈希表、页）
持有时间	整个事务	临界区（很短）
模式	S / X / Update / Intention	Read / Write
死锁处理	检测与恢复（waits-for、超时、abort）	**靠编码纪律避免**
管理者	Lock manager（集中）	被保护的结构自身
可回滚	需要	不需要

18.3 机制：2PL 的两个阶段



- 单独的 2PL 已能保证**冲突可串行化**（优先图无环）。
- 但它允许**脏读**与**级联回滚（cascading aborts）**，也可能死锁。
- **Strong Strict 2PL（Rigorous 2PL）**：所有锁**提交时才释放** → 无级联回滚，回滚只需恢复原值；代价是并发度更低。
- **Strict 2PL**：只在 shrinking 阶段释放**共享锁**，保证略弱（仍可能脏读）。

18.4 机制：死锁的检测与预防

检测：维护 waits-for 图（`Ti → Tj` 表示 Ti 等 Tj），周期性找环，选**牺牲者（victim）**回滚。牺牲者选择的启发式：最年轻/最老、进度最少/最多、已持锁数、需要连带回滚的事务数、重启次数（防饿死）。回滚可以是**全事务**或**部分（savepoint）**。

预防：给事务优先级（通常用时间戳，越老越高），只允许**单向等待**：

方案	请求者优先级高	请求者优先级低	口诀
Wait-Die	等待	自杀（abort）	“老的等，小的死”
Wound-Wait	杀死持有者（wound）	等待	“老的杀，小的等”

> 记忆法：协议名 `X-Y` 中，请求者优先级**高**就做 `X`，**低**就做 `Y`。重启时复用原时间戳，避免饿死。

18.5 机制：锁粒度与意向锁

级别	常见度
Database	罕见
Table	很常见
Page	常见
Tuple	很常见
Attribute	罕见

层次锁的规则：在祖先上加**意向锁**，才能在后代上加显式锁。

模式	含义
IS	后代将加 S 锁
IX	后代将加 X 或 S 锁
SIX	本节点 S + 后代 X（“读全表 + 改几行”）

18.6 关键数据结构与不变量

结构	不变量
Lock table	**不需要持久化**（崩溃时活跃事务一律 abort）
锁兼容矩阵	S–S 兼容；S–X、X–X 不兼容；意向锁之间 IS/IX 兼容、与 S/X 冲突
waits-for 图	允许“漏检”（下一轮会发现），所以构造时无需加闩

18.7 经典陷阱

1. **以为 2PL 就没有脏读**：普通 2PL 有；要 Strict/Rigorous 才消除。
2. **在事务中途随意放锁**：进入 shrinking 后再拿锁 → 不再是 2PL。

3. **把 Wait-Die 与 Wound-Wait 记反**：写反了在考试里必扣分，在代码里会造成活锁。
4. **忘记 `SELECT … FOR UPDATE`**：读-改-写场景先用 S 锁再用 X 锁，容易升级失败而回滚；提前拿 X 锁更省。
5. **层次锁忘了意向锁**：直接在某行加 X 锁而父节点没有 IX，另一个事务可以在表级加 S 锁 → 冲突漏检。

第 19 章 第 20 讲：时间戳排序与乐观并发控制（OCC）

> 依据：**Fall 2025 归档** `fall2025/notes/19-timestampordering.pdf`（本学期对应讲号 #20），教材 Ch. 18.5—18.6。

19.1 动机

2PL 是**悲观**的：先锁再干。若冲突其实很少（只读事务多、数据不相交），锁的开销与等待就是纯浪费。**乐观**路线是：先干，提交时再验证。

19.2 机制：时间戳与 OCC 三阶段

每个事务拿一个单调递增的时间戳 `TS(Ti)`；`TS(Ti) < TS(Tj)` 则调度必须等价于 Ti 在 Tj 之前。

```
Read Phase      → 读写都发生在私有 workspace，记录 read set / write set
Validation Phase→ 提交时检查与其它事务的 RW / WW 冲突
Write Phase     → 验证通过则原子安装到全局数据库；否则 abort & restart
```

验证的两条方向：

方向	比较对象	说明
Forward	与**尚未提交**的运行中事务比（未进验证阶段的 `TS = ∞`）	保证“我”不破坏别人
Backward	与**自我启动以来已提交**的事务比	更常用

前向验证的三个条件（`TS(Ti) < TS(Tj)` 时必须满足其一）：

1. Ti 的写阶段在 Tj 的读阶段开始前完成（完全串行）；
2. Ti 的写阶段在 Tj 的写阶段前完成，且 `WriteSet(Ti) ∩ ReadSet(Tj) = ∅`；
3. Ti 的读阶段在 Tj 的读阶段前完成，且 `WriteSet(Ti) ∩ ReadSet(Tj) = ∅` 且 `WriteSet(Ti) ∩ WriteSet(Tj) = ∅`。

19.3 机制：幻读（phantom）与四种对策

事务两次扫描同一范围，中间别人插入/删除了落在这个范围内的行 → 两次结果不同。根因是**只锁已存在的记录，锁不住“还不存在的记录”**。

对策	做法	评价
锁全表/全页	简单粗暴	并发差
提交时重扫（re-execute scans）	记录所有 WHERE，提交时重跑校验	代价高

对策	做法	评价
谓词锁（predicate locking）	按谓词加锁	System R 提出，实现复杂（HyPer 的 precision locking 近似）
索引锁（index locking）	key-value lock / **gap lock** / key-range lock / 层次化	**最常用**

19.4 机制：隔离级别（SQL-92 口径）

级别	脏读	不可重复读	幻读	典型实现
SERIALIZABLE	✗	✗	✗	Strict 2PL + 索引锁
REPEATABLE READ	✗	✗	**可能**	Strict 2PL
READ COMMITTED	✗	**可能**	可能	X 锁到提交、S 锁读完即放
READ UNCOMMITTED	**可能**	可能	可能	只对写加锁，读不加锁

> 注意：SQL-92 的级别只覆盖 **2PL 体系下**的三种异常；快照隔离（第 20 章）下的**写偏斜**不在表里 —— 这正是“RR 在 MVCC 系统里其实不是可串行化”的根源。

19.5 关键数据结构与不变量

结构	不变量
时间戳分配	单调递增；可用物理时钟（UTC，避免夏令时）或逻辑计数器
workspace	写只落私有区；读是否复制取决于隔离级别（repeatable read 必须复制）
验证集	read set / write set 必须完整（否则验证漏判）

19.6 经典陷阱

1. **OCC 在冲突多时性能崩坏**：abort 发生在“已经干完活”之后，浪费最大。
2. **时间戳分配成为瓶颈**：所有事务抢同一个原子计数器（分布式下更糟）。
3. **以为“快照读”就没有幻读**：快照隔离下幻读表现为“看不见新插入的行”，但写偏斜仍可能破坏一致性。
4. **忘记 workspace 的复制成本**：大事务把整页数据拷进私有区，内存与 CPU 都要付账。

第 20 章 第 21 讲：多版本并发控制（MVCC）

> 依据：**Fall 2025 归档** `fall2025/notes/20-multiversioning.pdf`（本学期对应讲号 #21），教材 Ch. 18.7—18.8。

20.1 动机

2PL 的根本矛盾：**读会阻塞写，写会阻塞读**。MVCC 的解法是“不覆盖，而是新增版本”：读者读旧版本，写者写新版本，互不阻塞。讲义称它是“过去十年几乎所有新 DBMS 都在用的方案”。

20.2 机制：版本链与可见性

```
index ──► [ v0 | txn=10 | begin=5  | end=8  ]
            │ next
            ▼
          [ v1 | txn=12 | begin=8  | end=∞  ]   ← head（最新）
```

读规则：事务用**启动时的快照**（复制事务状态表）判断可见性：版本由“已提交且 `begin ≤ snapshot` 且 `end > snapshot`”的事务产生，才可见。

- **写不阻塞读、读不阻塞写**；**写仍阻塞写**（同一对象的并发写要抢锁或靠 first-writer-wins）。
- 天然支持 **Snapshot Isolation（SI）**；不做 GC 时还能支持 **time-travel 查询**。

20.3 机制：五个设计维度

维度	选项	取舍
并发控制协议	2PL / TSO / OCC	与 MVCC 正交，可组合（MV-2PL、MV-TO、MV-OCC）
版本存储	Append-only（O2N / N2O）、Time-travel table、**Delta storage**	追加式简单；time-travel 读快写慢；delta 写快读慢
垃圾回收	Tuple-level（后台 vacuum / 协作清理）、Transaction-level	后台扫描可与任意存储配合；协作清理只适用 O2N
索引管理	逻辑指针（多一层间接）/ 物理指针	逻辑指针更新成本低；物理指针要更新所有索引
删除	deleted flag / tombstone tuple	墓碑可省空间但要特定位模式

新旧顺序（O2N vs N2O）很关键：N2O（最新在链头）让“读最新”只需一次寻址，但每次更新都要改索引指针；O2N 相反。

20.4 机制：写偏斜（write skew）

```
约束：x + y >= 0     初始 x=1, y=1
T1：读 y=1 → 写 x = -1          （认为 y 还在，安全）
T2：读 x=1 → 写 y = -1          （认为 x 还在，安全）
提交后：x=-1, y=-1  →  违反约束
```

快照隔离下两个事务修改**不同**对象，无法用“同对象写冲突”发现 → 结果是**不可串行化**但 SI 允许。这是 SI ≠ SERIALIZABLE 的经典证明（讲义用“黑白弹珠互换”的比喻）。

20.5 关键数据结构与不变量

结构	不变量
版本链	按时间戳有序；索引指向**链头**（head）；遍历直到找到可见版本
事务状态表	快照 = 启动时该表的副本；提交/回滚要原子更新
回收条件	版本对**所有活跃事务**都不可见 且 创建者非“已 abort” → 可回收
删除标记	逻辑删除后不能再产生更新的版本（所以删除不存在 WW 冲突）

20.6 经典陷阱

1. **以为 MVCC 就等于可串行化**：SI 允许写偏斜。
2. **忘了回收旧版本**：长事务/长快照会把版本链拖成“永不回收”，存储爆炸。
3. **索引指针与链头不一致**：更新后忘记更新索引，读到旧版本。
4. **把 `deleted flag` 放在错误位置**：放在版本头 vs 放在“链尾之后”，语义不同。
5. **MVCC + 非唯一索引**：同一 key 的多版本会让索引出现重复 key（MVCC duplicate key problem），取回后还要按可见性过滤。

第 21 章 第 22 讲：数据库日志（WAL）

> 依据：**Fall 2025 归档** `fall2025/notes/21-logging.pdf`（本学期对应讲号 #22），教材 Ch. 19.1–19.8。

21.1 动机：两个策略决定一切

策略	含义	选择
STEAL / NO-STEAL	是否允许把**未提交**事务的脏页写盘	STEAL 性能好
FORCE / NO-FORCE	提交时是否**强制**把所有修改写盘	NO-FORCE 性能好

四种组合，工业界只用两种：

组合	运行时	恢复	备注
NO-STEAL + FORCE	差（写集必须装进内存）	**最简单**（不需 undo/redo）	教学价值 > 实用价值；shadow paging 属于此类
STEAL + NO-FORCE	**最好**	复杂（undo + redo 都要）	**WAL 就在这里**

21.2 机制：WAL 的两条铁律

1. **先写日志，后写数据页**：页面对应的所有日志记录必须先落盘（`pageLSN ≤ flushedLSN`）。
2. **提交前必须把该事务的日志（含 `<COMMIT>`）刷盘**，之后才能向应用确认。

```
txn ──► WAL buffer（内存）──flush──► log file（稳定存储，顺序写）
                │
                └──► buffer pool 里的数据页 ──晚些/驱逐时──► 数据文件（随机写）
```

日志记录字段：`txn id`、`object id`、`before value`（undo 用）、`after value`（redo 用）、以及时间戳/checksum 等。MVCC 下 `before value` 可能可以省掉（旧版本已在版本链里）——讲义称之为“并发控制与恢复的良性协同”。

21.3 机制：三种日志粒度

粒度	记什么	优点	缺点
Physical	字节级 diff（像 `git diff`）	精确、恢复简单	记录大
Logical	高层操作（`UPDATE` / `INSERT`）	记录小，一条可影响多页	并发下**不确定**，recovery 难
Physiological	“某页的某 slot”，不管页内组织	折中；页可在日志后重组	**业界主流**

21.4 机制：checkpoint 与日志膨胀

日志会无限增长 → 必须**截断**。三种 checkpoint：

类型	是否暂停事务	记录内容
Blocking（一致性）	停新事务 + 等活跃事务结束 + 刷脏页 + 写 `<CHECKPOINT>`	简单，恢复容易
Slightly better blocking	停新事务，不等活跃事务结束	记录 ATT + DPT 快照
Fuzzy（ARIES 用）	**不暂停**	`<CHECKPOINT-BEGIN>` 与 `<CHECKPOINT-END>` 各一条，END 里带 BEGIN 时刻的 ATT/DPT

`<CHECKPOINT-BEGIN>` 的 LSN 记入 **MasterRecord**（磁盘上的“最近一次成功 checkpoint”指针）。

21.5 机制：group commit 与 CDC

- **group commit**：把多个事务的 flush 合并成一次，摊薄 fsync 成本（日志缓冲满或超时触发）。
- **Change Data Capture（CDC）**：同一份 WAL 还可以喂给副本、数仓、微服务 —— “为恢复而写的日志，顺便成了复制源”。

21.6 关键数据结构与不变量

结构	不变量
WAL buffer	追加写；LSN 单调
`flushedLSN`	已落盘的最大 LSN；**只增不减**
数据页	`pageLSN` = 最近一次修改该页的日志 LSN
checkpoint 记录	必须包含足够信息（ATT/DPT）才能从此处开始恢复

21.7 经典陷阱

1. **先刷数据页再刷日志**：崩溃后无法 undo（steal 已发生），数据库进入不一致状态。
2. **提交后不 fsync 就返回**：违反 durability；`fsync` 失败还可能被静默吞掉（第 04 讲的 side note）。
3. **混淆逻辑日志与物理日志的恢复能力**：逻辑日志要求“重放时状态一致”，并发下很难保证。
4. **checkpoint 太频繁**：运行期性能下降；太少则恢复时间不可控。

第 22 章 第 23 讲：数据库恢复（ARIES）

> 依据：**Fall 2025 归档** `fall2025/notes/22-recovery.pdf`（本学期对应讲号 #23），教材 Ch. 19.1—19.9。

22.1 动机与三条核心思想

ARIES（Algorithms for Recovery and Isolation Exploiting Semantics）是 IBM 在 1990 年代初为 DB2 设计的恢复算法。三条支柱：

1. **Write-Ahead Logging**（STEAL + NO-FORCE）；
2. **Repeating History During Redo**：重启时**先重放一切**，把数据库恢复到崩溃瞬间的精确状态（包括将要回滚的事务）；
3. **Logging Changes During Undo**：回滚时把 undo 动作也写进日志（CLR），这样“恢复中再崩溃”也不会重复 undo。

22.2 机制：五种 LSN

名称	位置	定义
`flushedLSN`	内存	已落盘的最大 LSN
`pageLSN`	每个数据页	最近一次修改该页的日志 LSN
`recLSN`	DPT 每条目	该页**变脏之前**最早的、尚未落盘的日志 LSN
`lastLSN`	ATT 每条目	该事务最新一条日志的 LSN
`MasterRecord`	磁盘	最近一次成功 checkpoint 的 LSN

写页的前置条件：`pageLSN ≤ flushedLSN`。

22.3 机制：正常执行中的三个特殊记录

记录	时机	作用
`<BEGIN>`	事务开始	标记起点
`<COMMIT>`	提交	**刷盘后**才能向应用确认
`<TXN-END>`	提交/回滚彻底完成	之后不再有该事务的日志；可延后刷盘
`<ABORT>`	主动回滚	随后逐条 undo 并写 **CLR**

记录	时机	作用
CLR（compensation log record）	undo 每条修改时	含 `undoNextLSN`；CLR 永不被 undo

```
log:  ... U1(T1,p5)  U2(T1,p7)  ABORT(T1)  CLR(U2→undoNext=U1)  CLR(U1→undoNext=null)  TXN-END
                  ▲                          │
                  └──── prevLSN 串成每事务的链表 ─┘
```

22.4 机制：checkpoint 时的两张表

- **ATT（Active Transaction Table）**：`txnId`、`status`（Running / Committing / Undo Candidate）、`lastLSN`。ATT 包含**所有还没有 TXN-END 的事务**（含已提交/已回滚的）。
- **DPT（Dirty Page Table）**：每个脏页一条，记录 `recLSN`。注意 DPT 关心的是“页脏”，与“谁弄脏的、提交没提交”无关。

22.5 机制：三阶段恢复

```
① Analysis：从 MasterRecord 指向的 <CHECKPOINT-BEGIN> 向前扫
   · 见 TXN-END → 从 ATT 删除
   · 其它记录 → 加入 ATT（状态 UNDO），见到 commit 改为 COMMIT
   · 见 UPDATE 且页 P 不在 DPT → 加入 DPT，recLSN = 该记录 LSN
② Redo：从 DPT 中最小的 recLSN 开始向前重放（**包括将回滚的事务**）
   · 跳过条件：页不在 DPT；或 记录 LSN < 页的 recLSN；或 pageLSN ≥ 记录 LSN
   · 重放后：pageLSN = 该记录 LSN
   · 结束时为所有 COMMIT 状态事务写 TXN-END 并移出 ATT
③ Undo：反向回滚崩溃时仍活跃的事务
   · 每步取所有 ATT 事务中最大的 lastLSN
   · 每次 undo 写一条 CLR（带 undoNextLSN）
```

22.6 关键数据结构与不变量

结构	不变量
DPT	`recLSN` 是该页“最早未落盘的修改”，用于决定 redo 起点
ATT	状态机单向：Running → Committing → Committed；或 → Undo Candidate
CLR	只前进不回退；`undoNextLSN` 指向下一条待 undo 的记录
恢复结束	ATT 只剩已提交事务（写 TXN-END 后清空）；日志 flush 后才接受新事务

22.7 经典陷阱

1. **redo 时“跳过已回滚事务”**：错，ARIES 要求 **repeat history**，先全部重放，再统一 undo。
2. **undo 时不写 CLR**：恢复中再次崩溃会重复 undo（幂等被破坏）。
3. **用 `lastLSN` 而不是“最大 lastLSN”**：undo 必须全局按 LSN 逆序推进，否则事务间的依赖会错。
4. **redo 起点用了 checkpoint 的 LSN 而不是最小 recLSN**：会多做一些无用 redo（慢），更糟的是若 checkpoint 时未刷该页，可能漏 redo。
5. **忘记 `pageLSN` 比较**：导致“已经落盘的页被重复 redo”，在非幂等操作上出错。

第 23 章 第 24 讲：分布式数据库 I —— 架构与分区

> 依据：**Fall 2025 归档** `fall2025/notes/23-distributed1.pdf`（本学期对应讲号 #24），教材 Ch. 20.4–20.5、20.7、21。

23.1 动机：三个目标

分布式 DBMS 把单一逻辑数据库分散到多个物理资源上，应用**通常无感**。三个目标：**容错**（单节点故障不拖垮全局）、**可扩展**（数据超单机容量）、**性能**（并行）。

23.2 机制：四种系统架构

架构	共享什么	特点	代表
Shared Everything	全部	单机；实现最简单	单机 DBMS
Shared Nothing	只有网络	各节点独立 CPU/内存/磁盘；扩展要搬数据；一致性协调难；性能潜力最大	传统分布式 DBMS
Shared Disk	同一逻辑磁盘	计算与存储解耦，各自独立扩缩；节点间要传播状态（内存是私有的）	云原生 DBMS、serverless
Shared Memory	同一地址空间	需要极快互联，成本高	少见

23.3 机制：分区（partitioning / sharding）

方式	做法	评价
Naive（按表）	一张表放一个节点	简单，但 join 与倾斜都很难看
垂直分区	按列切	每片要额外存重建元组的信息
水平分区	按行的 hash / range / predicate 切	最常用；分区键决定一切

- **逻辑分区**（节点负责一组 key 但不实际存储）常见于 shared disk；**物理分区**常见于 shared nothing。
- 目标：**最大化单节点事务**（single-node txn 不需要跨节点协调）。

扩容时的数据迁移：

方案	迁移量	说明
普通 hash（`h(k) % n`）	几乎全部	节点数一变，映射全乱
一致性哈希（consistent hashing）	约 `1/n`	节点在环上，key 顺时针归属最近节点；复制因子 k = 顺时针 k 个节点
Rendezvous（highest-random-weight）	仅受影响 key	对每个 `h(k ‖ p)` 打分取最高；无需环；一致性哈希可视为其特例

23.4 机制：分布式事务的协调

方式	做法
集中式协调者	全局“交通警察”，客户端先向它申请锁/权限，再访问各分区，最后向它请求提交决定；可作为 middleware
去中心化	客户端直连某节点，该节点当 leader，负责与其他分区通信并决定提交

23.5 机制：联邦数据库（federated）

把多个异构 DBMS 连成一个逻辑系统：middleware + connector，把查询翻译成各后端方言。难点：数据模型/查询语言不一致、无法统一优化、大量数据拷贝。

23.6 关键数据结构与不变量

结构	不变量
分区映射	同一 key 的元组必须落在同一分区（join/聚合正确性的前提）
环（一致性哈希）	每个 key 有唯一“顺时针最近节点”；副本是后续 k 个节点
透明性	应用看到的 schema 与单机一致（data transparency）

23.7 经典陷阱

1. **用 `h(k) % n` 做分区**：扩容时全量重分布。
2. **分区键选错导致跨节点 join 常态化**：网络成为瓶颈。
3. **把 shared disk 当成 shared nothing 设计**：前者内存私有，缓存一致性要自己传播。
4. **以为“分布式 = 更快”**：跨节点协调的延迟可能让简单查询变慢。

第 24 章 第 25 讲：分布式数据库 II —— 原子提交、CAP 与分布式 Join

> 依据：**Fall 2025 归档** `fall2025/notes/24-distributed2.pdf`（本学期对应讲号 #25），教材 Ch. 22.9、23.1–23.4。

24.1 机制：原子提交协议谱系

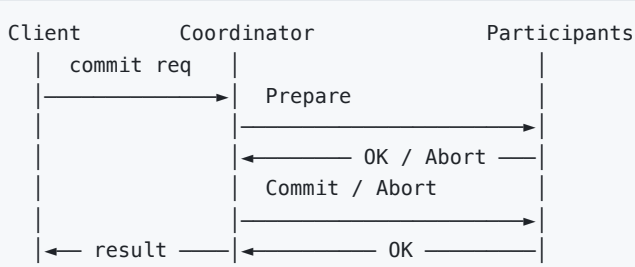
多节点事务结束时，必须让所有参与者对“提交/回滚”达成一致。协议族：2PC（1970s）、3PC（1983）、Viewstamped Replication（1988）、Paxos（1989）、ZAB、Raft（2013）。

所有协议都要求三条性质：

性质	含义
Stability	一旦决定，不可更改

性质	含义
Consistency	所有 RM（resource manager）最终同状态
Liveness	只要足够多节点存活并能通信，就能推进

24.2 机制：2PC 的消息流



- **阻塞性**：协调者在发出 Prepare 后崩溃 → 参与者只能等（或保守地 abort）。
- 优化：**Early Prepare Voting**（最后一条查询顺便带回投票）、**Early Acknowledgement after Prepare**（全票通过就先告诉客户端成功）。

24.3 机制：Paxos 与 2PC 的关系

维度	2PC	Paxos
协调者数量	1（F=0）	`2F+1`，多数派存活即可推进
阻塞	协调者挂则阻塞	多数派存活则不阻塞
消息轮次	少	多
适用	同机房、故障少	跨机房、要容错

讲义的一句话总结很好记：**“2PC 是 Paxos 的退化情形（F=0）”**。Multi-Paxos 通过“先选 leader，之后跳过 propose 阶段”来摊薄成本。

24.4 机制：CAP 与 PACELC

字母	含义
C	一致性 ≈ **线性一致性（linearizability）**：写完成后所有后续读都能看到
A	可用性：所有活着的节点都能响应请求
P	分区容忍：丢包/断连时系统仍能正确工作

三者不可兼得，只能选两个。PACELC 补上了“无分区时”的维度：Partition 时在 A 与 C 之间选；Else（正常运行时）在 Latency 与 C 之间选。

24.5 机制：分布式 Join 的四种场景

场景	数据布局	做法
1	一表在每个节点都有副本，另一表分区	各节点本地 join，结果汇总
2	两表都按 **join key** 分区	各节点本地 join，汇总
3	两表按**不同** key 分区，其中一表小	**broadcast join**：小表广播到所有节点，退化为场景 1
4	两表都**不**按 join key 分区	**shuffle join**：先按 join key 重分区，退化为场景 2；磁盘不够则失败

两个优化：**semi-join**（先传“可能匹配的 key 集合”，用 Bloom filter 近似，减少数据传输）、**shuffle 算子**（通用的重分区 exchange，也可用于负载再平衡）。

24.6 关键数据结构与不变量

结构	不变量
2PC 日志	每个节点持久记录每阶段的决定（崩溃后可恢复）
Paxos acceptor	只能对**更高**逻辑时间戳的提议投 Agree
Shuffle 分区	同 key 同分区；分区数决定并行度

24.7 经典陷阱

1. **以为 2PC 能容忍协调者故障**：它阻塞。
2. **把 CAP 的 C 理解成 ACID 的 C**：CAP 的 C 是线性一致性，不是“约束满足”。
3. **忘了 2PC 的日志**：参与者崩溃后不知道自己投过什么票。
4. **场景 4 直接开干**：不估算 shuffle 的数据量，磁盘爆掉直接失败。
5. **忽略 semi-join 的机会**：把整表搬来再过滤，网络成本高一个数量级。

第 25 章 存储专题：从页布局到 LSM 的选型矩阵

> 本章把第 3–6 章的内容压成“可决策”的形式：给定负载，选哪种页布局、哪种存储模型、哪种替换策略。事实依据同前四章（Fall 2026 notes 03–06）。

25.1 三种页布局的取舍

维度	槽式页（slotted）	日志结构（log-structured）	索引组织（index-organized）
更新方式	原地（in-place）	追加（append-only）	原地（B+ 树叶内）
点查	索引 → RID → 一页	逐层探 SSTable	索引即数据，一次下降
范围扫描	需回表，随机 I/O	多层 merge	**顺序 I/O，最优**

维度	槽式页（slotted）	日志结构（log-structured）	索引组织（index-organized）
写放大	低	**高**（compaction）	中（页分裂）
碎片	有（删除留洞）	无（顺序追加）	有（半满页）
典型	通用行存	KV / 写密集	InnoDB 主键表

判据：写入量 ≫ 读取量且能容忍读放大 → LSM；范围扫描为主且要聚簇 → 索引组织；通用 OLTP → 槽式页 + 二级索引。

25.2 元组编码的四个决策

决策	选项	影响
定长 vs 变长	全定长 / 变长 + 偏移	定长可随机定位，变长省空间
对齐	字对齐（word aligned）	未对齐会触发多次内存访问
NULL	bitmap / 特殊值 / per-attr flag	bitmap 最省；列存用特殊值
大值	内联 / overflow page / 外部文件	German Strings 折中（≤16B 内联）

25.3 替换策略的量化对照

策略	元数据/页	命中率（扫描型负载）	命中率（倾斜点查）	实现复杂度
LRU	时间戳（8B）+ 链表指针	**差**（被冲干净）	一般	低
CLOCK	1 bit	差	一般	低
LFU	计数器	中	好	中（需分桶）
LRU-K	K 个时间戳	好（K 大时抗扫描）	好	高（+ ghost cache）
ARC	4 个链表	好	**好**	**高**（P1 的实现难点）

25.4 行存 vs 列存：一句话判据

- 查询**命中列比例 < ~20%** 且行数巨大 → 列存（I/O 省下来的比“拼元组”贵得多）。
- 查询以**整行读写**为主、并发点更新多 → 行存。
- 既要又要 → **PAX**（Parquet/ORC/Arrow），或 HTAP 双格式（fractured mirror / delta store）。

25.5 LSM 的三个必背代价

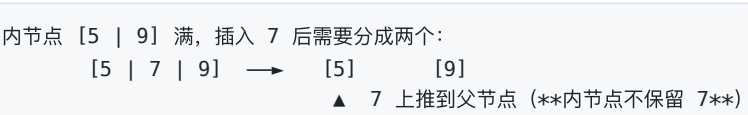
1. **写放大**：一次逻辑写可能触发多次 compaction 物理写。
2. **读放大**：要探 MemTable + 多层 SSTable（靠 Bloom + min/max 剪枝）。
3. **空间放大**：旧版本与墓碑占空间，直到 compaction 回收。

第 26 章 索引专题：B+ 树的分裂/合并与索引选择

26.1 插入导致的分裂（叶子）



26.2 内节点分裂：push up 而非 copy up



26.3 删除：先借，后并

删除后叶子只剩 1 个 key（半满阈值 2）：
① 向左兄弟借：兄弟有 3 个 → 挪 1 个过来，**更新父节点分隔 key**
② 借不到 → 与兄弟合并，**删除父节点中对应的分隔 key 与指针**
③ 父节点因此不足半满 → 递归向上（可能一直合并到根）
④ 根只剩 1 个孩子 → 树高降低，root_page_id 更新

26.4 阶数与容量公式（一页纸要能算）

设页大小 `P` 字节，内节点每条目 `key_size + ptr_size`，叶子每条目 `key_size + value_size`，页头 `H`：

量	公式
内节点最大 key 数	`m_inner = ⌊(P − H) / (K + ptr)⌋ − 1`
叶子最大条目数	`m_leaf = ⌊(P − H) / (K + V)⌋`
扇出（fanout）	`m_inner + 1`
树高（N 条记录）	`⌈log_fanout(N / m_leaf)⌉ + 1`

例（BusTub 风格，8 KB 页，页头 64 B）：`K=8B, ptr=8B` → 每节点 508 项 → `m_inner = 507`，fanout = 508；`V=8B` → `m_leaf = 508`。四层树可索引约 `508^3 × 508 ≈ 6.7×10^10` 条记录。

26.5 哈希索引 vs B+ 树索引

查询	B+ 树	哈希
等值	✓	✓（更快、常数小）
范围	✓	✗
前缀 / 部分键	✓（组合索引前缀）	✗（要完整 key）

查询	B+ 树	哈希
有序输出（ORDER BY）	✓（聚簇时最优）	✗
最坏复杂度	O(log n)	O(n)（冲突严重时）

26.6 索引选择的经验规则

1. **等值 + 高选择性** → 索引几乎总赢。
2. **范围 + 结果集占比大（> ~10–20%）** → 顺序扫描往往更快（回表随机 I/O 太贵）。
3. **组合索引**：只能吃**最左前缀**；`(a,b,c)` 上 `WHERE b=1` 一般用不上（除非 skip scan）。
4. **覆盖索引（index include columns）**：查询所需列都在索引里 → 免回表。
5. **部分索引（partial index）**：`CREATE INDEX … WHERE` 缩小索引体积与维护成本。
6. **索引不是越多越好**：每个索引都要在写路径上维护，且占空间、增并发冲突。

第 27 章 查询执行专题：从火山模型到算子选择

27.1 四种执行模型的对照

模型	每次交互的数据量	函数调用次数	适合	代表
Iterator（Volcano）	1 元组	最多（每元组一次）	OLTP、需要早停	多数行存
Materialization	全部结果	最少	OLTP 小结果	早期系统
Vectorized	一批（如 1024）	中	OLAP 扫描	BusTub、DuckDB
Compiled（JIT）	一批	无（生成代码）	OLAP 极致性能	HyPer、ClickHouse 部分路径

27.2 Join 算法决策树

```
两表 join
 ├─ 谓词是等值?
 │   ├─ 否 → Nested Loop (block NLJ)
 │   └─ 是
 │       ├─ build 侧装得下内存?
 │       │    ├─ 是 →  (In-memory) Hash Join
 │       │    └─ 否 → Grace Hash Join (分区)
 │       ├─ 右表 join key 上有索引?  → Index NLJ (外表小、内表索引探测便宜时)
 │       └─ 两侧已按 join key 有序?  → Sort-Merge Join (省排序)
 └─ 输出需要按 join key 有序?  → Sort-Merge 更划算
```

27.3 排序的三种场景

场景	算法	代价
数据装得下	内存快排	O(N log N) CPU，0 额外 I/O

场景	算法	代价
装不下	外部归并排序	`2N × (1 + ⌈log_{B−1}⌈N/B⌉⌉)` I/O
Top-N	堆	一次扫描 + O(N log n)

27.4 聚合的两条路与选择

判据	选排序聚合	选哈希聚合
输出需要有序	✓（省一次排序）	✗
输入已按 group key 有序	✓	✗
追求速度	✗	✓
内存不足	外部排序天然可溢出	需分区溢出（更复杂）

27.5 流水线断点清单（写出执行计划时要能标出来）

`ORDER BY`、聚合、join 的 build 侧、`DISTINCT`（去重）、`UNION`（去重）、递归 CTE、窗口函数的排序阶段。**除这些之外**的算子都可以流水线化。

第 28 章 优化器专题：代价、基数与连接顺序

28.1 逻辑 vs 物理

	逻辑计划	物理计划
表达	关系代数（σ π ⋈）	具体算法 + 访问路径
与数据格式	无关	相关（排序、压缩、分区）
映射	一对多（无一一对应）	多对一

28.2 代价模型的四项成本

成本	量级	难点
Disk I/O	块传输次数	可测
CPU	小但难估	谓词求值、比较、哈希
Memory	DRAM 占用	与并发度耦合
Network	消息数（分布式）	与分区/ shuffle 相关

28.3 基数估计的三条假设与三个修正

假设	修正手段
均匀	直方图（等深/端偏）、sketch
独立	多列统计、相关性采样
包含	检查 join 键是否为主键/唯一键

28.4 Join 顺序的搜索空间

n（表数）	左深树数量	全 bushy 树数量级
3	6	小
5	120	数百
10	3.6M	10^10 级

所以工业界普遍**只搜左深树 + 动态规划**（System R 风格），或用 **top-down + memo + 分支裁剪**（Volcano/Cascades 风格）。

28.5 三条“优化器会犯的错”（也是面试常问）

1. **选错索引**：把高选择性谓词估成低选择性（倾斜列没有直方图）。
2. **选错 join 顺序**：中间结果估计过小，导致 build 侧溢写。
3. **谓词下推越界**：对外连接/子查询做过度改写。

第 29 章 事务专题：五种协议的异常覆盖矩阵

29.1 协议 × 异常

协议	脏读	不可重复读	幻读	写偏斜	备注
2PL（基础）	可能	✗	可能	✗	会级联回滚
Strict 2PL	可能（S 锁早放）	✗	可能	✗	—
Strong Strict 2PL	✗	✗	✗（配索引锁）	✗	工业界默认
TSO / OCC	✗	✗	需索引锁	✗	冲突多时 abort 率高
MVCC + SI	✗	✗	部分	**可能**	读写不阻塞

29.2 死锁策略对照

策略	机制	优点	缺点
超时	等太久就 abort	实现最简单	可能误杀、恢复慢
waits-for 检测	建图找环，选 victim	精准	需要周期扫描
Wait-Die	老的等、小的死	无环	年轻事务反复重启
Wound-Wait	老的杀、小的等	无环、重启少	老事务会“伤害”年轻事务

29.3 MVCC 版本存储三选一

方案	写成本	读成本	空间
Append-only	低（追加）	可能要遍历链	大
Time-travel table	中（拷贝旧版本）	低（一次跳转）	中
Delta	**最低**（只存差）	**高**（要回放）	**最小**

29.4 隔离级别的“实际语义”提醒

- 商业系统里 `REPEATABLE READ` 常常是 **snapshot isolation**（MySQL InnoDB、PostgreSQL 的 RR 都是快照读）→ **不阻止写偏斜**。
- 想要真正可串行化：`SERIALIZABLE` + SSI（serializable snapshot isolation）或索引锁 + 2PL。
- 隔离级别是**每事务**设定的，同一系统里可以混用。

第 30 章 恢复专题：WAL 的四条规则与 ARIES 的时序

30.1 WAL 的四条规则（可当检查表用）

1. 修改数据页前，对应的**日志记录必须已落盘**。
2. 提交返回前，`<COMMIT>` 记录必须已落盘（可 group commit）。
3. 数据页只有在 `pageLSN ≤ flushedLSN` 时才能写盘。
4. checkpoint 必须记录足够状态（ATT + DPT）以支持从该点恢复。

30.2 ARIES 三阶段的时序图



30.3 三种 checkpoint 的成本对照

类型	运行期影响	恢复速度	实现复杂度
一致性（blocking）	**大**（停业务）	快	低
稍好（slightly better）	中（停新事务）	快	中
Fuzzy	小（不停）	稍慢（要处理 checkpoint 期间的更新）	高

30.4 日志结构存储的恢复差异

LSM 的“恢复”主要不是 ARIES 的 redo/undo，而是**重放 MemTable/WAL**：写路径先写 WAL（或 binlog）再进 MemTable，崩溃后重放未 flush 的部分；SSTable 本身不可变，无需 undo。**共同点**：都依赖“先写日志、后写数据”这一条铁律。

第 31 章 分布式/并行专题：分区、复制与 2PC

31.1 分区策略对照

策略	扩容迁移量	倾斜风险	范围查询
Range	中（分裂/合并区间）	中	**好**（相邻区间连续）
Hash（`% n`）	**全部**	低	差
一致性哈希	~`1/n`	中（虚拟节点可缓解）	差
Rendezvous	仅受影响 key	低	差

31.2 复制：两种经典形态

形态	写路径	读路径	一致性
单主（primary-backup）	全写主，再同步/异步复制	可读从（可能旧）	强（同步）或最终（异步）

形态	写路径	读路径	一致性
多主 / leaderless	任意节点写，靠 quorum 收敛	读 quorum	可调（R + W > N 时强）

31.3 分布式事务：2PC 的三种失败点

失败点	后果	处理
参与者在 Prepare 前崩	协调者可安全 abort	简单
参与者投票后崩	协调者据日志决定	恢复后补发决定
协调者发完 Prepare 后崩	参与者**阻塞**	等协调者恢复；或协议升级（3PC/Paxos）

31.4 NewSQL 的三件套

组件	作用	代表
分片（sharding）+ 自动再平衡	水平扩展	CockroachDB、TiDB
共识复制（Raft/Paxos）	每分片高可用	同上
分布式事务（2PC over 共识组 / MVCC + 时间戳）	跨分片原子性	Spanner（TrueTime）、CockroachDB（HLC）

第 32 章 BusTub 项目 0–4 精讲

> 依据：`fall2026/project0/` … `project4/` 页面、`assignments.html`、syllabus、FAQ（均 2026-11 访问）。**详细的文件级清单、命令与踩坑见同目录《15445_数据库系统_项目上手速查.md》**，本章只讲“目标—关键类—顺序—测试”。

32.1 五个项目一览（Fall 2026 实际形态，与历史学期不同）

项目	主题（本学期实际）	关键类/文件	依赖	测试入口
P0	**Robin Hood 哈希集合**（不是往年的 trie/skiplist）	`src/include/primer/robin_hood_hash_set.h`、`src/primer/robin_hood_hash_set.cpp`	无	`make robin_hood_hash_set_test` → `./test/robin_hood_hash_set_test`
P1	**ARC 替换策略** + Disk Scheduler + BufferPoolManager + PageGuard	`buffer/arc_replacer`、`storage/disk/disk_scheduler`、`storage/page/page_guard`、`buffer/buffer_pool_manager`	P0 的 C++ 功底	`arc_replacer_test` / `disk_scheduler_test` / `page_guard_test` / `buffer_pool_manager_test`
P2	B+ 树（页类 + 增删查 + 迭代器 + 并发）+ **tombstone buffer**	`b_plus_tree_page` / `_internal_page` / `_leaf_page`、`b_plus_tree.cpp`、`index_iterator`	P1 的 page guard	`b_plus_tree_insert_test`、`_delete_test`、`_concurrent_test`、`_sequential_scale_test`

项目	主题（本学期实际）	关键类/文件	依赖	测试入口
P3	执行器 + 优化规则（扫描/增删改/索引扫描 → 聚合/连接 → 排序/窗口）	`seq_scan`、`insert/update/delete`、`index_scan`、`aggregation`、`nested_loop_join`、`nested_index_join`、`hash_join`、`external_merge_sort`、`limit`、`window_function`、`optimizer/seqscan_as_indexscan`、`optimizer/nlj_as_hash_join`	P1+P2	SQLLogicTest（`make sqllogictest` 后跑 `test/sql/*.slt`）
P4	并发控制（**规格页截至 2026-11 未发布**，只知 release Nov 09 / due Dec 06）	以发布为准	P1–P3	以发布为准

32.2 实现顺序（经验顺序，非官方）

- P0：先写“探测循环”的骨架（`Insert` / `Contains` / `Remove` 共用一个 `FindSlot`），再补 `Clear`、move 语义（`Capacity()==0 && Size()==0` 且仍可 `Clear()`）、`LoadFactor` / `MaxProbeDistance`。最后处理并发：全局 latch 能过功能测试，但**过不了性能测试**（要求相对串行加速 > 1.2），需要细粒度或 CAS。
- P1：`ArcReplacer`（四种 RecordAccess 分支最容易写错）→ `DiskScheduler`（单 worker + Channel/promise）→ `PageGuard`（RAII + move）→ `BufferPoolManager`（pin/unpin、脏页写回、`CheckedRead/WritePage`）。**先让 `GetPinCount` 通过**，再用它验证 guard。
- P2：页类（`Init` / `Lookup` / `Insert` / `Remove` / `Split` / `MoveHalfTo`）→ 点查 → 插入（含根分裂与 header page 更新）→ 删除（借/并/降高）→ 迭代器 → 并发（乐观 crabbing）。
- P3：`SeqScan` / `Insert` / `Delete` / `Update` → `IndexScan` + `SeqScan→IndexScan` 规则 → 聚合 → NLJ → NestedIndexJoin → HashJoin + 规则 → ExternalMergeSort + Limit → 窗口函数。**每步都先跑对应编号的 SQLLogicTest**。
- P4：等规格。

32.3 评分点与“测试读法”

- 每个项目满分 100，评分四条：**用例通过 + 无内存泄漏 + 风格/lint 通过 + design document 完整**（P1/P2/P3 页面都写了这四条）。
- **隐藏测试**：P1/P2 明确“本地测试只是子集”；**P0 与 P3 明确“无隐藏测试”**（P3 的 `.slt` 全给）。
- 读测试的方法：先看测试**构造了什么状态**（容量、pin 数、页数），再看**断言了什么不变量**，最后才看期望值。P1 的 `arc_replacer_test` 大量用例其实在测“四种 RecordAccess 分支后，MRU/MFU/ghost 的长度与 victim 是谁”。

32.4 政策红线（必须如实遵守）

项目	能否用 coding agent	额外要求
P0	否（作业与 P0 一律禁止）	必须满分，否则退课
P1	**禁止**	需 design doc（含 AI Usage 段，如实写 None）

项目	能否用 coding agent	额外要求
P2	**禁止**	需 design doc
P3	允许	**一页 reflection**（FAQ 口径：P3/P4 都要）
P4	允许	**一页 reflection**（syllabus 明确要求）

> 无论是否用 AI，**code ownership** 是硬要求：DRC（笔试，两次）与 CRI（每人至少一次）都可能问到你**几周前写的代码**。判断标准很朴素：**能不能不看注释、在纸上讲清楚每个 latch 的获取顺序与不变量**。做不到，就不要把代码交给工具生成。

第 33 章 与已有笔记的衔接

已有笔记	本课对应	衔接点（可复用的心智模型）
《CSAPP 深入理解计算机系统》	第 3–4、10 章	cache line / 局部性 → B+ 树节点大小与页内二分；`atomic`/CAS → spin latch；虚拟内存 → page table 与 TLB shootdown
《CS162 操作系统实现》	第 4、10、17–22 章	页表/缺页 → buffer pool 的 pin；mutex/条件变量 → latch；WAL 与文件系统 journal 是同一套“先写日志”的思路
《CS425 分布式系统》	第 23–24、31 章	复制/一致性/共识 → 2PC 与 Paxos；CAP/PACELC → 分布式事务的取舍
《CS144 计算机网络》	第 23–24 章	RPC 语义、超时重传 → 分布式事务的消息重传与幂等

第 34 章 复习清单与自测题

34.1 复习清单（按“能不能在白纸上画出来”自查）

#	应该能独立画出/推出	对应章节
1	槽式页布局 + 插入/删除/变长更新的三种情形	3, 25
2	buffer pool 结构图（page table / pin / dirty / replacer）+ 一次缺页的完整流程	4, 25
3	CLOCK 手转一圈的过程；ARC 四个列表与 p 的调整	4, 25
4	B+ 树插入分裂（叶子 copy up / 内节点 push up）与删除借并	8, 26
5	用给定页大小算 fanout 与树高	26
6	外部归并排序的趟数与 I/O 公式，并代入数字	11, 27
7	四种 join 的代价公式与“谁该当外表”	12, 27
8	火山 vs 向量化的取舍；流水线断点清单	13, 27
9	优先图判可串行化；2PL 的两个阶段	17, 18
10	Wait-Die / Wound-Wait 的方向（能举一个例子）	18, 29
11	MVCC 版本链 + 可见性判断 + 写偏斜的例子	20, 29

#	应该能独立画出/推出	对应章节
12	WAL 四条规则；ARIES 三阶段与五种 LSN	21, 22, 30
13	2PC 的消息流与阻塞点；CAP/PACELC	24, 31

34.2 自测题（先自己写，再看右列）

#	题目	答案要点
1	页 8 KB，key 8 B，指针 8 B，页头 64 B，求内节点 fanout 与树高（10^9 条记录，叶子每条 16 B）。	内节点每项 16 B → `(8192-64)/16 = 508` 项 → key 数 507，fanout 508；叶子 `8128/16 = 508` 条。树高 ≈ `⌈log_508(10^9/508)⌉ + 1 ≈ 4`。
2	`B = 100`，`N = 12,000,000` 页，外部归并排序共几趟?	排序阶段 `⌈N/B⌉ = 120,000` 个 run；归并 `⌈log_99 120000⌉ ≈ 3`；总趟数 `1 + 3 = 4`（注意底数是 B−1）。
3	为什么 B+ 树叶子分裂用 copy up，内节点用 push up?	叶子必须保留全部 key（数据在叶子）；内节点 key 只是路标，上推后父节点持分隔符即可。
4	一个事务在 RR 隔离级别下读到 `x=1,y=1`，随后写 `x=-1`；另一个写 `y=-1`。会出什么问题?	写偏斜：SI 下两事务改不同对象，无 WW 冲突，结果违反 `x+y>=0`，不可串行化。
5	ARIES 的 redo 为什么要重放“将要回滚的事务”?	“Repeating history”：先把数据库恢复到崩溃瞬间的状态，undo 阶段再统一回滚；否则 pageLSN 与日志不一致，无法判断哪些修改已落盘。
6	`flushedLSN=100`，某页 `pageLSN=120`，能否把该页写盘?	不能。必须先 flush 日志到至少 120（WAL 规则）。
7	Bloom filter 说“不在”，能否跳过 B+ 树查找? 说“在”呢?	“不在”可以跳过（无假阴性）；“在”不能确信（有假阳性），仍需查。
8	两表都按 `id` 分区、join on `id`，属于哪种分布式 join 场景? 代价是什么?	场景 2：各节点本地 join 后汇总；代价 ≈ 本地 join + 结果网络传输，无 shuffle。
9	P1 中把 `pin_count` 减到 0 但忘记 `SetEvictable(true)`，会发生什么?	该帧永远不可驱逐 → 池子可用帧逐渐耗尽 → OOM/超时（P1 经典 bug）。

第 35 章 参考来源

课程主页与政策（访问日期 2026-11）

1. CMU 15-445/645 课程主页（Fall 2026）：https://15445.courses.cs.cmu.edu/
2. 排课与讲义页：https://15445.courses.cs.cmu.edu/fall2026/schedule.html
3. 作业页：https://15445.courses.cs.cmu.edu/fall2026/assignments.html
4. 大纲（评分、迟交、AI 政策）：https://15445.courses.cs.cmu.edu/fall2026/syllabus.html
5. FAQ（DRC/CRI、AI 政策、环境）：https://15445.courses.cs.cmu.edu/fall2026/faq.html

Fall 2026 讲义（notes，逐讲下载并提取文本）

1. `notes/01-relationalmodel.pdf`、`02-modernsql.pdf`：https://15445.courses.cs.cmu.edu/fall2026/notes/
2. `notes/03-storage1.pdf`、`04-bufferpool.pdf`、`05-storage2.pdf`、`06-storage3.pdf`（同上目录）
3. `notes/07-hashtables.pdf`、`08-indexes1.pdf`、`09-indexes2.pdf`、`10-indexconcurrency.pdf`

4. `notes/11-sorting.pdf`、`12-joins.pdf`、`13-queryexecution1.pdf`、`14-queryexecution2.pdf`
5. slides：`fall2026/slides/01-…` 至 `16-optimization2.pdf`：https://15445.courses.cs.cmu.edu/fall2026/slides/

Fall 2025 归档（本学期未发布主题的补位依据）

1. `fall2025/notes/15-optimization1.pdf`、`16-optimization2.pdf`
2. `fall2025/notes/17-concurrencycontrol.pdf`、`18-twophaselocking.pdf`、`19-timestampordering.pdf`、`20-multiversioning.pdf`
3. `fall2025/notes/21-logging.pdf`、`22-recovery.pdf`、`23-distributed1.pdf`、`24-distributed2.pdf`
4. 归档目录：https://15445.courses.cs.cmu.edu/fall2025/

项目与代码

1. BusTub 仓库与 README：https://github.com/cmu-db/bustub
2. 项目说明：`fall2026/project0/`（Robin Hood 哈希集合）、`project1/`（ARC + Disk Scheduler + BPM）、`project2/`（B+ 树）、`project3/`（Query Execution）；`project4/` 截至 2026-11 仅有目录列表，无正文。
3. 作业 PDF：`fall2026/files/hw2-15445-fall26.pdf`、`hw3-15445-fall26.pdf`、`hw4-15445-fall26.pdf`

教材

1. Silberschatz, Korth, Sudarshan. *Database System Concepts*, 7th Edition.（章节号见 1.4 的讲题地图）

> 说明：本笔记不含任何课程未公开的评分测试内容；讲义引用只做要点转述与结构化重组，不复刻原文。若课程后续发布第 18–25 讲讲义或 P4 规格，应以其为准并替换第 17–24、32 章中标注为“Fall 2025 归档”的部分。