

数据库系统笔记：中大 2022 人工智能学院课程

1 内容大纲

- ER 模型：对数据进行建模 已学习
- Relational Model and Algebra 关系数据模型 已学习
- SQL 数据库查询语言 已学习
- 函数依赖和关系数据库设计 已学习
- 文件存储 已学习
- 索引及索引进阶 Tree and Hash 已学习
- 查询处理 已学习
- 查询优化 已学习
- Transactions 事务处理
- 并发协议
- 数据恢复

2 ER 图表

- 矩形：实体集。 强实体集可以被自带属性标识，
- 椭圆：属性 弱实体集（键下划虚线）依赖于强实体。
 - 普通椭圆：简单属性
 - 连接着椭圆的椭圆：复合属性，具有属性的属性
 - 双椭圆：多值属性 用双线标识多值
 - 虚线椭圆：派生属性，可用根据其它属性计算出来
- 下划线文本：键，唯一地指引一个实例的属性
 - 多个下划线文本：复合键 \Rightarrow 只有主键要用下划线标出
 - \rightarrow 一个实例可用有多个 key。可以唯一指代的属性集都为 key。
 - \rightarrow 最短的 key 称之为候选键，挑选一个候选键作为主键
- 菱形：关系 多元关系可以转化为二元关系
 - \rightarrow 关系也可以带有属性
 - 连接同一实体集：递归关系
- 约束
 - one-many 约束：由 one 指向 many
 - one-one 约束：双向箭头
 - many-many 约束：无箭头

- 参与约束:

- * 全部参与: 双线段, 表示至少由一种关联关系
- * 部分参与: 默认, 可以不存在关联关系

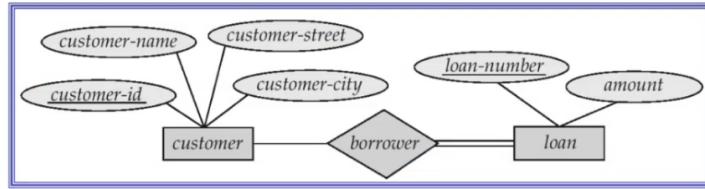


图 1: 参与约束

- 三角形: 自上而下的层级划分

3 函数依赖 FD

函数依赖的定义: $\alpha \rightarrow \beta$ 即 α 的值决定了 β 的值。

Trivial 无意义和 Non-Trivial:

- 无意义函数依赖 Trivial: $AB \rightarrow A$
- 有意义 Non-Trivial: $AB \rightarrow C$

FD 和 key 的关系

- α 是 R 的超键 $\Leftrightarrow \alpha \rightarrow R$
- α 是 R 的候选键 $\Leftrightarrow \alpha \rightarrow R$ 且没有 β s.t. $\beta \subset \{\alpha, \beta\} \rightarrow R$

闭包与最小依赖集: 见下图手写笔记。

4 关系型数据库设计: 3NF

数据库异常行为包括函数依赖下的冗余存储、异常更新、异常插入、异常删除, 如下图 ($position \rightarrow salary$)。

规范化是将关系型数据库分解的过程。 $R = R_1, R_2, \dots, R_n$ 。

- 要求分解无损、依赖保留且冗余信息小。
- **依赖保留:** FDs 存在于各个单个 relation, 以便于验证 FD。
- 即, 所有子表格 FD 并集的闭包与原来表格 FD 闭包一致

给定一组函数依赖 F , F 的闭包记作 F^+ , 它是所有由 F 逻辑上蕴涵的函数依赖的集合。换句话说, F^+ 包括 F 中的所有函数依赖以及可以通过 F 中的函数依赖使用推理规则推导出来

Closure of F :

闭包 可以由 F 推论得到的 All FD set

Armstrong's axioms : $\left\{ \begin{array}{l} \rightarrow B \subseteq \alpha \Leftrightarrow \alpha \rightarrow B \\ \rightarrow \alpha \rightarrow \beta \Leftrightarrow \gamma \alpha \rightarrow \gamma \beta \\ \rightarrow \alpha \rightarrow \beta, \beta \rightarrow \gamma \Rightarrow \alpha \rightarrow \gamma \end{array} \right.$

Closure of Attribute set:

可由 α 决定的所有属性值集合

最小属性依赖集 F_C : $\left\{ \begin{array}{l} F_C \subseteq F \\ F_C \text{ 没有冗余 } \end{array} \right.$

冗余 Redundancy

图 2: 闭包与最小依赖集手写笔记

first_name	last_name	address	department	position	salary
Dewi	Srijaya	12a Jin Lempeng	Toys	clerk	2000
Izabel	Leong	10 Outram Park	Sports	trainee	1200
John	Smith	107 Clementi Rd	Toys	clerk	2000
Axel	Bayer	55 Cuscaden Rd	Sports	trainee	1200
Winny	Lee	10 West Coast Rd	Sports	manager	2500
Sylvia	Tok	22 East Coast Lane	Toys	manager	2600
Eric	Wei	100 Jurong drive	Toys	assistant manager	2200
?	?	?	?	security guard	1500

↓

key

异常插入

冗余存储

异常更新

图 3: 数据库异常行为

的所有函数依赖。

计算一个属性集 X 关于一组函数依赖 F 的闭包 X^+ 的方法如下：

1. 初始化：设 X^+ 为 X 的初始值。
2. 迭代：重复以下步骤直到 X^+ 不再变化：
 - 对于每一个 F 中的函数依赖 $A \rightarrow B$ ，
 - * 如果 $A \subseteq X^+$ ，则将 B 添加到 X^+ 中。

当 X^+ 不再变化时，算法结束，此时 X^+ 就是 X 在 F 下的闭包。

不满足依赖保留的分解：

- 可能会出现信息丢失、存储冗余等问题。
- 如果不能通过自然连接而恢复依赖，则出现信息丢失。
- 在下图中， R_1 和 R_2 不存在依赖 $B \rightarrow C$ 。

$R = (A, B, C)$, $F = \{\{A\} \rightarrow \{B\}, \{B\} \rightarrow \{C\}, \{A\} \rightarrow \{C\}\}$. Key: A
 R 中存在 FD: $B \rightarrow C$, 由于 B 不是该表的 key, B 和 C 出现存储冗余信息。

A	B	C
1	2	3
2	2	3
3	2	3
4	3	4

R ₁	R ₂		
A	B	A	C
1	2	1	3
2	2	2	3
3	2	3	3
4	3	4	4

图 4: 错误分解

- $F_1 = \{\{A\} \rightarrow \{B\}\} \& F_2 = \{\{A\} \rightarrow \{C\}\}$, $(F_1 \cup F_2)^+ \neq F^+$ 。正确分解如图 3。

A	B	C
1	2	3
2	2	3
3	2	3
4	3	4

R ₁	R ₂		
A	B	B	C
1	2	2	3
2	2	3	4
3	2	3	4
4	3	3	4

图 5: 正确分解

Lossless Join Decomposition:

- 如果能利用分解后的表恢复原来的表，则该分解是无损的。
- 两个 schema 的交集属性至少构成其中一个 schema 的 key。
 - 外键必须引用另一个表的候选键。

1NF: 第一范式

- 第一范式: 表中所有属性的域没有多值属性。
- 根据关系模型的定义, 关系表 (relational tables) 永远是 1NF。
 - 关系模型要求每一个属性都是原子的。

1NF 是不够的, 不一定能满足无损、依赖保留和形式良好。

范式是对关系模式的规范化, 不是对分解方法的选择。

2NF: 第二范式

- 定义: 1NF 基础上, 所有非主属性完全函数依赖于任何一个候选键。
 - 2NF 确保所有非主属性完全依赖于任何一个候选键, 而不是部分依赖于候选键的一部分。即, 非主属性必须完全依赖于候选码。

- 2NF 通过消除部分函数依赖，可以减少数据冗余，提高数据的一致性和完整性。
- 形式化： R 是在 2NF 中当且仅当对于每个函数依赖 $X \rightarrow \{A\}$ 在 F^+ 中，至少满足以下三个条件的其中一条：
 - $A \in X$ (函数依赖是平凡的)：自己依赖自己，无函数依赖
 - X 不是某个候选键的真子集：被依赖属性非为部分候选键
 - A 是一个主属性
- 向 2NF 的转换：
 - 如果存在非主属性对候选码的部份依赖，则这个属性和主关键字这一部分应该分离作为一个新表。

假设关系模式 $R(A, B, C, D)$, (A, B) 是主键，函数依赖集为 $F = \{A \rightarrow C, (A, B) \rightarrow D\}$ 。

- 主键是 (A, B) 。
- 非主属性是 C 和 D 。
- 函数依赖 $A \rightarrow C$ 表明 C 部分依赖于主键 (A, B) 的一部分 A 。
- 函数依赖 $(A, B) \rightarrow D$ 表明 D 完全依赖于主键 (A, B) 。

结论：关系模式 R 不满足 2NF，因为存在部分函数依赖 $A \rightarrow C$ 。

转换为 2NF：

- 分离成为两个关系模式 $R_1(A, C)$ 和 $R_2(A, B, D)$ 。
- R_1 和 R_2 的 FDs，非主属性都完全依赖于主键。

3NF：第三范式

- 定义：在 1NF 的基础上，任何非主属性不依赖于非键。
- 理解：在 2NF 的基础上消除传递依赖。
- 3NF 同样减少数据冗余和提高数据的一致性。
- 形式化： R 是在 3NF 中当且仅当对于每个函数依赖 $X \rightarrow \{A\}$ 在 F^+ 中，至少满足以下三个条件的其中一条：
 - $A \in X$ (函数依赖是平凡的)。
 - X 是 R 的超键。(相比起 2NF 的更进一步。)
 - A 是一个主属性。
- 3NF 的分解：假设有一个关系模式 $R(A, B, C, D)$, 其函数依赖集为 $F = \{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$ 。
 - 确定候选键：计算属性闭包，发现 A 是候选键，因为 $A^+ = \{A, B, C, D\}$ 。
 - 识别非主属性：非主属性是 B 、 C 和 D 。
 - 检查传递依赖：存在传递依赖 $A \rightarrow B \rightarrow C$ ，因为 $A \rightarrow B$ 且 $B \rightarrow C$ ，但 $A \not\rightarrow C$ 。
 - 分解关系模式：将 R 分解为两个关系模式： $R_1(A, B, D)$ 和 $R_2(B, C)$

一个值得注意的自环传递例子：

- $R = (B, C, E)$, $F = \{\{E\} \rightarrow \{B\}, \{B, C\} \rightarrow \{E\}\}$
- 找出候选键：BC 和 EC。
- 满足任何非主属性不依赖于非键。
- 该关系模式符合 3NF 范式。

BCNF: Boyce-Codd 范式

- BCNF 旨在消除非平凡的多值依赖和非主属性对候选键的部分依赖。
- 形式化： R 是在 BCNF 中当且仅当对于每个函数依赖 $X \rightarrow \{A\}$ 在 F^+ 中，至少满足以下两个条件的其中一条：
 - $A \in X$ (函数依赖是平凡的)。
 - X 是 R 的超键。
- 就是说 → 对于任意一个 non-trivial 函数依赖，左侧为超键。非候选键不能决定其它属性。

5 内存层级和文件组织 377

磁盘和主存

- DBMS 将信息存储在磁盘上。
- 数据才磁盘上的存储单元为块 (block)，又称为页 (page)。
- 当前使用的数据存储在主存，主要数据存储在二次存储，旧数据存储在磁带。

文件组织

- 文件组织引入：
 - 在数据库存储中，一个文件相当于一个表，即 relation 实例。
 - 表格中的一行，文件中的一个元组即一个记录。
 - 一个表格中的记录大小是固定的。
- 如何组织记录存放在文件里？

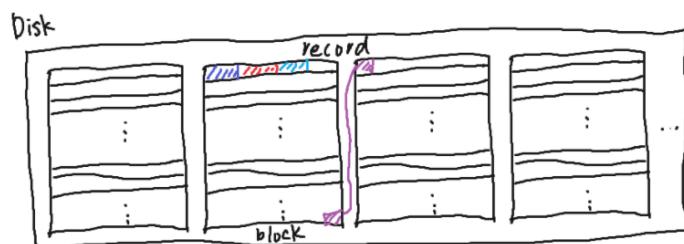


图 6: 文件组织

- 固定长度记录：存储记录为连续的固定区域存储。同个文件的记录可能跨 block 存储。删除记录需要内存空间移位并且利用 free list 回收空间。

Free list：空闲记录链表，在文件首部指向第一个删除记录的位置。

- 可变大小记录：存储变长几里路，并且允许记录中某个字段重复出现。

如何存储可变大小记录？

- 方法一：用字节字符标识记录结尾。
 - 删除困难且插入困难。
- 方法二：预先分配最大的可能长度，插入记录时用符号 \perp 填充末尾部分
- 方法三：指针方法：设置两种结构来组织文件：锚定块和溢出块。
 - 对于具有重复属性的记录类型很有效。
 - 锚定块 *Anchor*: 包含链的第一条记录
 - 溢出块 *Overflow*: 溢出字段

文件中记录的组织方式

- Heap 堆：无序存放，有空间就可以存放
 - 需要记录的信息：每个文件存放在哪些页面、页面的剩余空间、每个页面存放了记录的位置信息。
处理好文件和页面之间的关系
- Sequential 有序：按某个搜索键的值的顺序存放
 - 删除：用指针跳过该记录
 - 插入：找到插入位置 → 如果有可用空间就插入 || 如果没有可用空间就插入到溢出块中。
 - 时不时重新组织文件以恢复顺序
- Hashing 哈希：计算哈希值并指向块
 - 设计适合文件大小的哈希函数。比如假设我们为 R 分配 1200 个页面，我们可以用设计以某个合适属性值的模 1200 作为哈希寻址函数。
 - 如果该页面已满，就创建一个溢出页面并在那里创建记录。
 - 不适用于范围搜索。

时间成本分析：

- 以磁盘读写 page 的总次数作为衡量单位。
- 假设 B 为文件的数据 pages 数量：
- Hash 扫描所有文件的成本较高。
- Hash 查询特定文件的成本最低，其次是有序存储。
- 范围查找 Hash 成本最高，有序查找最低，
- 插入操作有序的成本非常高。
- 插入和删除 Hash 都很方便，删除的话直接删除比有序删除方便很多（不重排）。

	Heap File	Sorted File	Hashed File
Scan all records	B	B	1.25B (80% occupancy)
Equality Search	0.5B	$\log_2 B$	1
Range Search	B	$\log_2 B$ + (no of pages with matches)	1.25B
Insert	2 (load only last page, write out)	Search + B (0.5B adjust)	2 (find page, write out)
Delete	Search+1 (find page, write out)	Search + B (0.5B adjust)	2 (find page, write out)

图 7: 文件存储方式时间成本分析

数据字典: 即系统目录, 存放元数据。

- 关系表的信息:
 - names of relations 关系名称
 - names and types of attributes of each relation 属性值名称和类型
 - names and definitions of views 视图名称和定义
 - integrity constraints 约束
- 用户账号信息, 包括密码。
- 统计和描述性数据:
 - number of tuples in each relation 关系元组数量
- 物理的文件组织信息:
 - How relation is stored (sequential/hash/...) 关系存储方式
 - Physical location of relation: 物理存储地址
 - * operating system file name or ↓
 - * disk addresses of blocks containing records of the relation
- 索引信息 (index)

6 索引的相关概念 405

索引的引入

- ID 排序 + 二分搜索: $\log(n)$ 。
- ID 排序 + 索引条目: $< ID, page.No > [Search-key | pointer]$

- 每个 page 的第一条记录有一个条目，从而反映条目区间。
- 对索引使用二分查找。
- 可以创建多级索引 Multilevel Index。
- 两种基础索引类型：顺序索引、哈希无序索引。

顺序索引

- 顺序索引也成为树索引。
- 获取一个记录所需的 page 访问次数为 `tree.height+1`

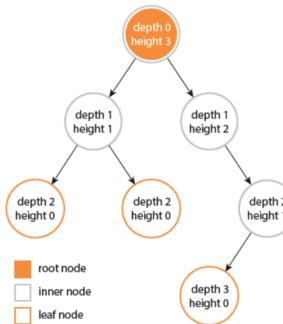


图 8: tree index

主索引和辅助索引

- 主索引 Primary index: 文件实际存放也是按照 search key 排序的。其基于主键或唯一标识符构建索引，具有唯一性。
- 辅助索引 Secondary index: 也称为二级索引。文件的实际存放不按照 search key 排序。其基于非主属性构建索引，不具有唯一性。
 1. 当检索许多记录时，辅助索引昂贵，因为磁盘不连续寻址耗时长。
 2. 辅助索引常用于优化不涉及主键的查询。

对于非排序属性（如账号、余额等），可以通过额外建立索引生成二级索引，从而提高查询效率。这样可以在不需要重新组织数据的情况下，实现对这些属性的有效查询。

稀疏索引和密集索引

- 稀疏索引 Sparse Index: 仅包含某些 search key。
稀疏检索：找到最后一个比 K 小 key 的 index，然后向后遍历寻找 K。
- 密集索引 Dense Index: 包含所有 search key。

哈希索引

- 哈希索引维护哈希函数 $F(\text{search key}) = \text{location_of_record}$
- 严格来说，哈希索引是二级索引（辅助索引）。

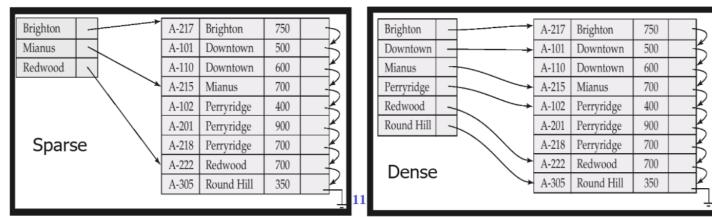


图 9: 稀疏索引和密集索引

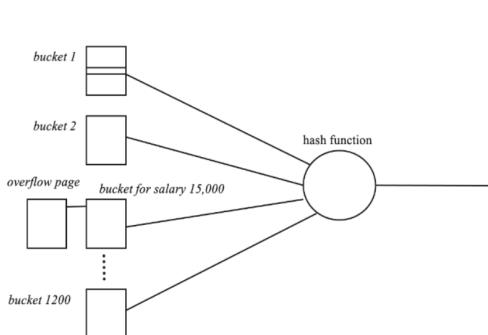


图 10: 哈希索引结构

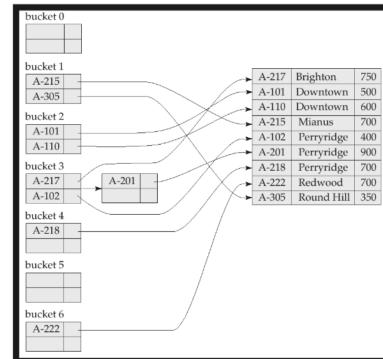


图 11: 哈希索引示例

索引的评价指标

- 成本定义为最坏情况下的磁盘打开页数。
- 高效支持不同查询：例如，辅助索引（包括哈希）在范围检索表现很差，因为磁盘经常需要跨区寻址。
- 更新时间成本小：当文件被修改时，索引也必须更新。
- 额外存储成本小（即 `index` 本身的大小）：索引的大小应该比数据文件小得多。

索引的更新

- **删除：**索引的删除思路符合直觉。
 - 如果删除的记录为索引的唯一记录，则对应的索引也要删除。
 - 对于稀疏索引，如果删除记录的 `search key` 在 `index` 存在，则将 `index` 里的的记录替换为所删除的下一条记录。如果下一条记录已经在 `index` 中存在，则直接删除。
- **插入：**同样符合直觉。
 - 对于密集索引：如果 `search key` 没有出现在 `index` 则插入。
 - 对于稀疏索引：
 1. 如果该索引为每个 `page` 维护有索引条目则不需要修改。

2. 否则要创建新 page。当有新的 block (page) 创建的时候, 将 block 的第一个 search key 加入 index。

7 索引进阶: B^+ 树和动态哈希

7.1 B^+ 树:

B^+ 树引入

- B^+ 树是一种特殊的平衡多路搜索树。

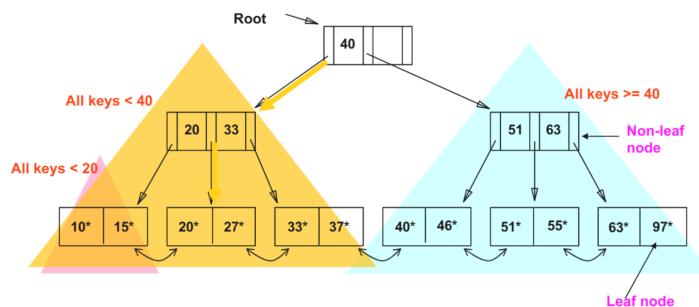


图 12: B^+ 树示意图

- B^+ 树是所有商业 DBMS 中优序索引的默认实现。

B^+ 树的主要特征

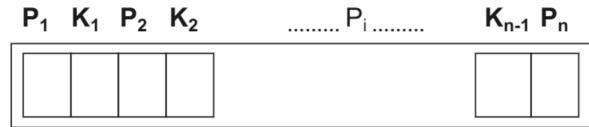
- 所有叶子节点都在同一层。
- 非叶子节点只存储索引信息。
- 叶子节点存储实际数据信息。
- 叶子节点之间有指针连接: 所有的叶子节点通过指针相互连接, 形成一个双向链表。这种结构支持高效的范围查询和顺序扫描。

B^+ 树的性质

- 从根到叶子的所有路径都具有相同的长度 (a balanced tree)
- n 称为扇出 fanout (每个节点最多可有拥有指针数):
 1. 每个非叶节点都有数量在 $\lceil n/2 \rceil$ 到 n 之间的指针
 2. 每个叶节点存储数量为 $\lceil (n-1)/2 \rceil$ 和 $n-1$ 之间的数据值
 3. 值 $\lceil (n-1)/2 \rceil$ 称为 order 阶数 (数据值的最小数量)

1. 和 2. 都是在设定了子节点下限，从而维护树的平衡。

特别例子：如果根不是叶节点，则至少有两个子节点。如果根是叶节点，可以有 0 到 $(n - 1)$ 个数据值。



Search keys: k_1, k_2, \dots, k_{n-1}
Pointers: P_1, \dots, P_n

图 13: B^+ 树节点示意图

B^+ 树查询

- 从所有记录中寻找 $search_key = a$ 的记录：

从根节点开始：

- 在当前节点内部寻找 P_i ，满足 $K_{i-1} \leq a < k_i$ 。
- 跳到 P_i 指向的子节点。重复以上步骤直到叶节点。如果在叶结点中找不到，则该树不包含 $search_key = a$ 的记录。

B^+ 树插入

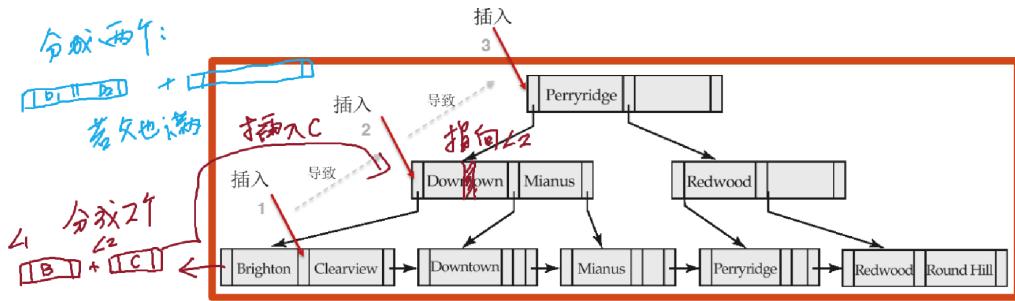
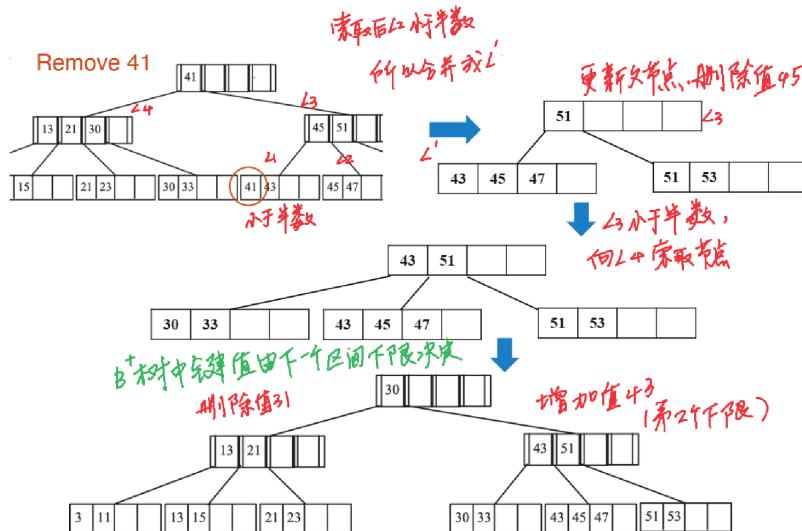
- 用键值 k 进行查询，得到叶节点 L 。

- 将 k 插入到 L 中：
 - 如果 L 有空位，则插入完成。
 - 如果 L 无空位：将 L 分裂成 L_1 和 L_2 ：
 - 将 keys 平分（奇数则左侧多一个）。
 - 更新 L 的父节点：插入 L_2 的最左侧数值。（中间键值往父节点插入）
 - 如果父节点也无空位，则由下往上递归更新节点。

B^+ 树删除

- 用 k 值查询，得到其所在叶节点 L 。

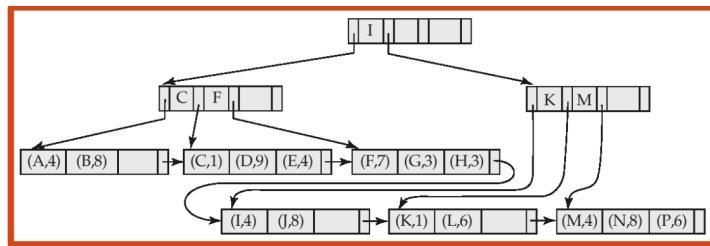
- 将 k 从 L 中移除：
 - 如果 L 有超过半数 key，则结束。
 - 否则，从右侧兄弟节点索取一个 key。
 - 如果右侧节点少于 $\lceil (n - 1)/2 \rceil + 1$ 个节点，则合并两个节点：
 - 合并后，从 L 的父节点删除指向 L 的键值。
 - 合并也可能发生连锁反映。

图 14: B^+ 树插入图 15: B^+ 树删除

B^+ 树文件组织结构

- 可以用 B^+ 树索引解决索引退化问题。
- 数据文件退化问题（碎片化）用 B^+ 树来组织文件存储。要求：
 - 叶节点存储数据本身。
 - 叶节点存储的最大记录数小于非叶节点的最大指针数。
 - (上面两条都是 B^+ 树本身的要求)

在文件组织和数据库管理中，索引退化（Index Degradation）是指索引结构在长时间运行和频繁的插入、删除操作后，逐渐失去其优化效果，导致性能下降的问题。索引退化会影响查询性能、增加磁盘 I/O 操作次数，并可能导致其他性能问题。

图 16: B^+ 树文件组织结构

7.2 动态哈希：

静态哈希: Buckets 数量是固定的。将搜索键值固定映射到某个 Bucket。随着数据增长，由于过多的溢出导致该 index 的性能退化。

动态哈希

- Buckets 数目可以动态调整。
- 关键：哈希结果和数据位置中间加一层指针：
 - 使用指向 buckets 的指针目录，通过将目录加倍来增加 buckets 数量（见下）。
 - 仅拆分溢出的存储 bucket。
 - 只需要对指针进行整理，成本小很多。

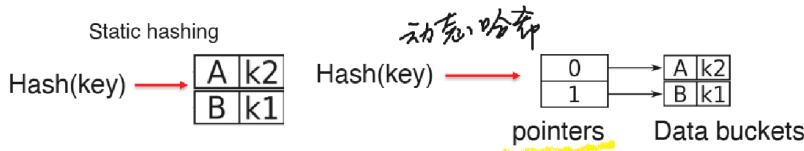


图 17: 静态与动态哈希对比

动态哈希函数的调整

- 掩码哈希: $f(value) = value \% (2^{DEPTH})$
- 通过全局深度 $GLOBAL\ DEPTH$ 和局部深度 $LOCAL\ DEPTH$ 动态分裂桶。
 - 全局深度：将一个元素映射到一个桶时需要看几位。
 - 局部深度：桶中所有元素相同尾数序列长度。
 - $Global\ depth = \max\{Local\ depth\}$
 - $Local\ depth = Global\ depth - \log_2(lines)$, 其中 $lines$ 为连接线段数量。

动态哈希删除

- 如果删除导致 bucket 变空，则将其与其镜像（譬如 001 和 101）合并，合并后 local depth-1。

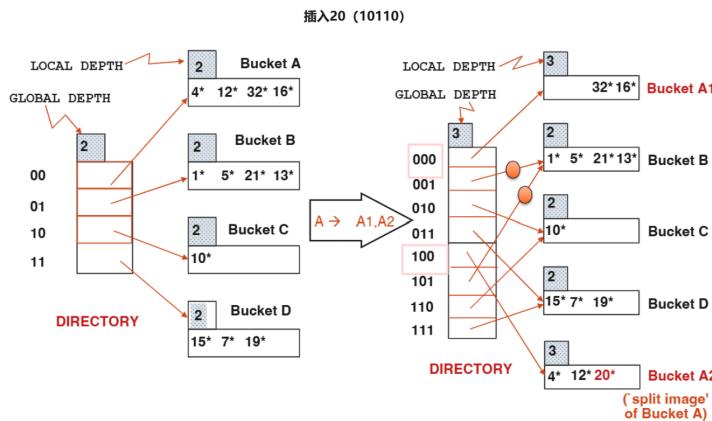


图 18: 掩码哈希动态调整

- 当因为一系列删除操作，导致指针目录的任意指针与其镜像指针均指向同一个 bucket，则将整体指针目录缩减一半，global depth-1。

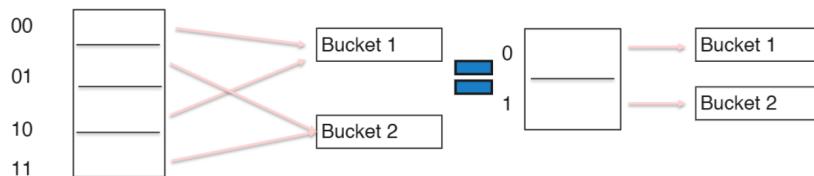


图 19: 动态哈希全局目录缩减

8 其它索引：多键访问

多键访问引入

- 如何加快具有多个条件的查询记录？

```
Select loan-number
From account
Where branch-name<"Perryridge" and Balance = 1000
```

- 策略一：先后查询

先检查元组的 branch-name 然后在检查 balance

- 策略二：交集查询

先分别获取符合两种要求的指针，然后寻找指针的交集并获得记录。

位图索引

- 位图索引 (Bitmap) 旨在对多个键同时进行高效查询。
- 引入:
 - 记录按照顺序编号为 (R_0, R_1, R_2, \dots) , 给定 n 则检索 R_n 的效率很高。
 - 仅对取值个数有限的属性特别适用。
- 最简单的形式: 为属性 (搜索键) 的每个值都建有一个位图。
 - 位图长度 (bit 个数) 为 record 数量。
 - 每个位图代表某个属性的某个值在记录中的取值情况。

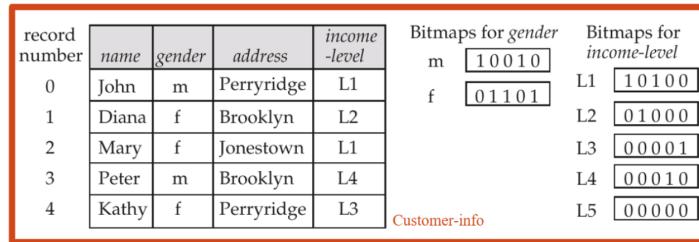


图 20: 位图索引

9 查询过程

查询过程的基本步骤

1. 解析和转化
 - 将 SQL 查询语句转化为关系代数。
 - 解析器检查语法和表名。
2. 求解
 - 查询执行引擎采用查询评估计划, 执行该计划并返回查询答案。
 - 通过关于查询市场的统计数据制定更优的查询计划。

优化引入和成本评估

- Need special evalution-plan to solve algebra expression.
 - 关系代数表达式有许多等效表达式。
 - 一个简单的关系运算可以有很多种算法实现。
 - 例子:

```
select B,D
from R,S
```

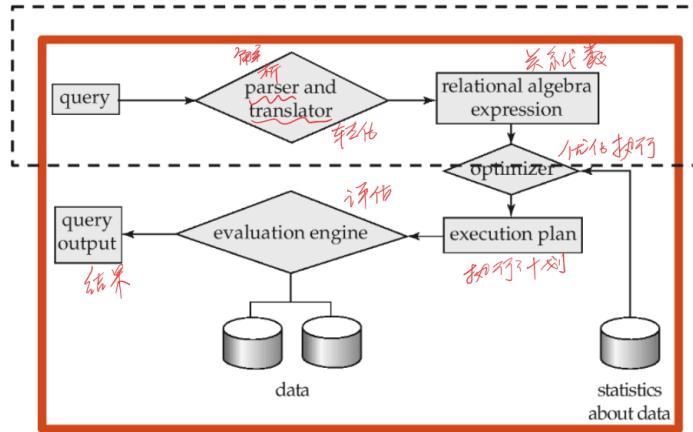


图 21: 查询架构

`where R.A='C' and S.E=2 and R.C=S.C`

具有两种查询计划如下图所示。

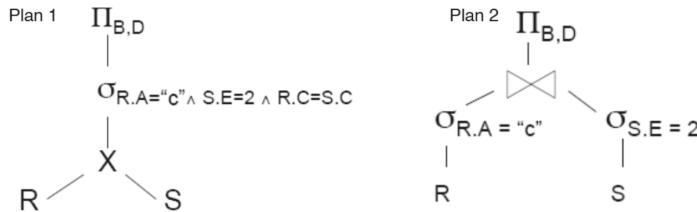


图 22: 等价查询计划例子（查询语句见笔记）

- 把时间成本简单计算为：在磁盘和主存之间读写 page 的总数量。
 - 忽略顺序 I/O 和随机 I/O 之间的成本差异（忽略寻道）。
 - 忽略 CPU 成本。
 - 忽略将最终结果写入磁盘的成本。
- 时间成本页依赖于主存中缓冲区的大小，更多的内存可以减少磁盘访问的需求。

选择操作的实现方式：检索满足 $\sigma_{A=V}(r)$ 的一些可能算法：

- 算法一：线性搜索 *linear search*
 - 访问每个 page 并测试所有记录是否满足条件。
 - Cost estimate（扫描的磁盘页数）= b_r
 - * b_r 标识包含关系表 r 中的记录的页数

- * 如果选择条件是关于键 key 的，则一旦访问到目标记录查询便可停止。

- * 平均成本为 $\frac{b_r}{2}$

- 算法二：二分查找

binary search

- 在检索文件排序所依据的属性上适用。

- 假设该表的文件 page 是连续存储在磁盘上的。

- 成本估算：

- * 定位一个元组的成本 $\lceil \log_2(b_r) \rceil$

- * 如果有多条记录满足选择条件，还需要算上跨越这些记录的 page 数。

⇒ 假设选择条件是 **作用在搜索键上** 的：算法三 → 五

- 算法三：在候选键上的主索引

primary tree index on candidate key, equality

- 检索满足相等条件的单个记录：Cost= $HT_i + 1 // HT_i$ 为树的高度。

- 算法四：在非键上的主索引

primary tree index on non-key, equality

- 检索满足条件的多个记录，记录在连续的页面上。

- Cost = $HT_i +$ 包含检索记录的页数

- 算法五：在搜索键上的二级索引

equality on search-key of secondary index

- 如果搜索键是候选键，即检索单个记录：Cost= $HT_i + 1$ 。

- 否则需要检索多个记录：Cost = $HT_i +$ 包含检索记录的页数。

- * 记录可能位于不同的页面，成本昂贵。

- * 最坏的情况：每条检索的记录都需要一页访问。

如何区分搜索键和候选键：

主索引：文件按照 search key 存放

- 搜索键指的是查询的属性。

二级索引：不按照 search key 存放，不唯一

- 候选键是关系表的属性，具有唯一性。

⇒ **涉及比较式的查询**：算法六 → 七

→ 实现 $\sigma_{A \leq V}(r)$ 或 $\sigma_{A \geq V}(r)$ 同样可以用线性扫描、二分搜索和索引。

- 算法六：主索引下的比较查询

primary index, comparison

- Relation is sorted on A.

- For $\sigma_{A \geq V}(r)$ 使用索引查找第一个满足并访问之后所有页面。

- For $\sigma_{A \leq V}(r)$ 从第一个开始访问直到检索大于 V 则停止。

- 算法七：二级索引下的比较查询

secondary index, comparison

- Relation is not sorted on A.

- For $\sigma_{A \geq V}(r)$ 使用索引查询第一个索引条目并往后访问，根据条目得到满足条件记录的指针，根据指针访问记录。

- For $\sigma_{A \leq V}(r)$ 从头扫描索引的叶页面（leaf page），从中查找指向记录的指针，直到第一个条目大于 V。

- ⇒ **复杂（关系运算）** 查询：算法八 → 十二
- **交集** $\sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(r)$ 查询：算法八 → 十
- 算法八：单键索引的交集查询 *conjunctive selection using one index*
 - 对于其中一个条件 θ_i , 选择算法一 → 七中的低成本算法进行索引, 将元组提取到主存后测试记录对其他条件的满足性, 不满足则丢弃。
 - 算法九：多键索引的交集查询 *conjunctive selection using multiple-key index*
 - 如果可行, 也可以选择使用的符合(多键)索引。
 - 算法十：标识符的交集索引 *conjunctive selection by intersection of identifiers*
 - 对每个条件使用相应的索引, 并对所有获得的记录指针集合取交集。
 - 如果某些条件没有合适索引, 则在读到内存后再对他们进行该条件的测试。
- **并集** $\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r)$ 查询：算法十一
- 算法十一：标识符的并集索引 *conjunctive selection by union of identifiers*
 - 如果所有条件都有创建索引, 则使用。对每个条件使用相应索引, 并对所有获得的指针集取并集, 然后从文件中获取记录。
 - 如果只有部分条件或者没有条件有可用索引, 使用线性扫描。
- **取否** $\sigma_{\neg\theta}(r)$ 查询：算法十二 *Negation*
- 对文件使用线性扫描。
 - 如果 $\neg\theta$ 记录很少, 并且有关于且适用 θ 的索引, 则使用索引查找 $\neg\theta$ 的记录。
- 外部排序** *External Sorting*
- 引入:
 - 在执行选择操作时, 有时候需要对记录排序。如升序查询 `asc` 和降序查询 `desc`。
 - 数据体量大, 分 `pages` 存储, 无法把全部记录装到主存后进行快排。
 - 需要使用外部排序从而最小化磁盘读写, 对不装进内存的关系表进行外部排序。
 - 主存体量 $\geq n + 1$: 同时合并 n 个排序好的文件需要占用 $n + 1$ 个主存页。
 - 主存外部排序例子: 主存具有 3 个页的 buffer
 - 双指针合并排序好的 `page1` 和 `page2`, 得到 `mergePage`。
 - 将 `mergePage` 作为 `page1`, 继续合并 `page3`。
 - *Question:* 如果文件(`pages`)未排序, 该如何排序?
 - 情况一: 文件页数小, 放入主存用任意排序算法排序。
 - 情况二: 文件页数大, 无法放入主存进行排序:
 1. 将数据逐批带入主存, 每一批都是 M 页, 使用主存排序算法排序后写回。
 2. 迭代的对上述排序文件进行 $M - 1$ 路合并。(需要 1 页写出排序结果)
 - 未排序大文件例子: $M = 4page$, 1 个大文件, 每个文件有 100 个 page
 1. 分 $100/M = 25$ 批, 每次读取 $M = 4$ 页进行排序, 得到 a_1 至 a_{25} 个文件。
 2. 对上述 25 个文件进行 $M - 1 = 3$ 路合并, 每次合并三个, 迭代直到结束。

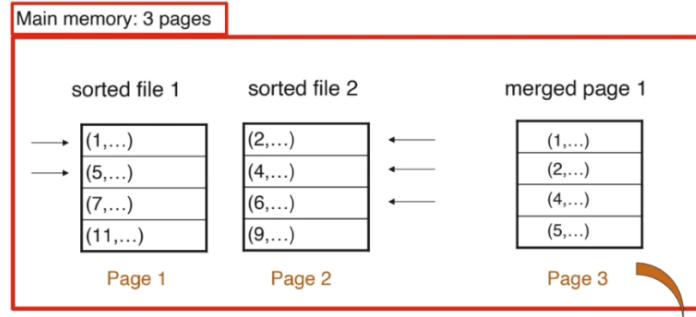


图 23: 外部排序

(10,...)	(12,...)	(110,...)	(112,...)
(5,...)	(34,...)	(44,...)	(75,...)
(17,...)	(56,...)	(17,...)	(20,...)
(12,...)	(19,...)	(18,...)	(201,...)

图 24: 未排序大文件

sorted file 1	sorted file 2	sorted file 1	merged page 1
(1,...)	(2,...)	(12,...)	(6,...)
(5,...)	(4,...)	(14,...)	(7,...)
(7,...)	(6,...)	(17,...)	(9,...)
(11,...)	(9,...)	(19,...)	

图 25: 合并排序好的子文件

- 外部排序算法总结:

→ M : 主存大小 (page); R : 关系表; b_r : 存储整个 R 需要的页数

- 阶段一: 读取关系表中的 M 页进入主存并排序, 从而得到 N 个子表 R_i 。

- 阶段二: 如果 $N \leq M - 1$, 则直接 N-way 合并。如果 $N \geq M$, 则连续合并 $M - 1$ 个文件, 一个 pass 将文件个数减少 $M - 1$ 倍, 同时长度增大 $M - 1$ 倍, 直到所有文件合并为一个文件。成本为 $b_r(2\lceil\log_{M-1}(b_r/M)\rceil + 1)$ 。

10 连接算法

连接算法引入

- 连接算法即 Join algorithms。
- 连接算法的成本只简单考虑 I/O 次数, 即 page 读写次数。

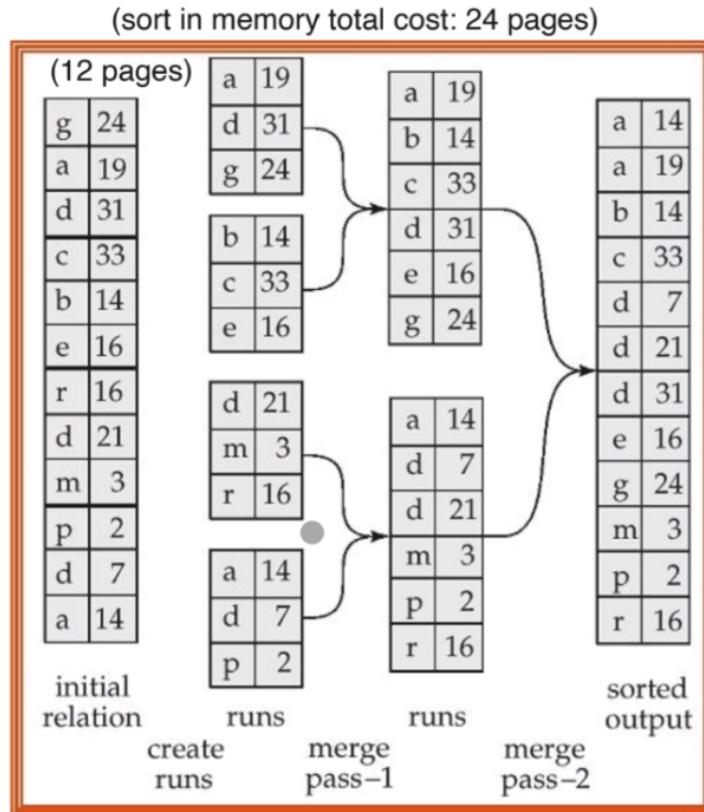


图 26: 整体外部排序例子

- Notation:

- r, s : 两个连接的关系表。
- n_r, n_s : r 和 s 的记录数。
- b_r, b_s : r 和 s 的页数。
- M : 内存的可用页数。

连接算法复杂度分析

- 块嵌套循环连接:

- 最坏成本: $b_r * b_s + b_r$ 。（主存的页面大小 = 3 页）
 - 对于 r 的每一个页面都遍历一次 s 。也就是要把 r 里面的每一个页面依次主存，每一次把 s 的所有页面再依次读入主存，即 $b_r * b_s$ 。最后一项 b_r 是写输出成本。
- 最好成本: $b_r + b_s$ 可以把整个 r 都都进去主存。
- 一般情况: 使用 $M - 2$ 内存页面存储外关系 r 的 pages，剩下的 2 页一页用于读取 s ，一页用于写输出。总成本为 $\lceil b_r / (M - 2) * b_s \rceil + b_r$ 。

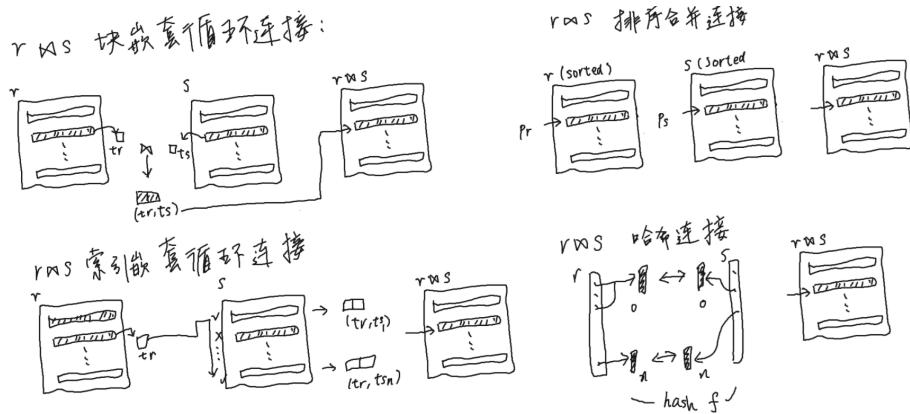


图 27: 四种连接算法图示

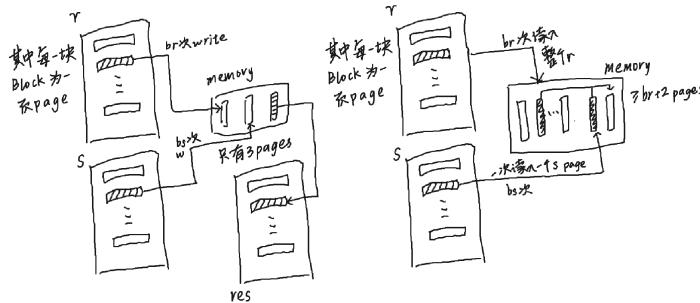


图 28: 块嵌套循环连接时间复杂度分析

- 索引嵌套循环连接:

- (...)

11 查询优化

Intro: 查询优化不只是计算每个运算的成本，并且需要考虑不同操作执行步骤间在输入输出的交互。

系统目录信息: 存储描述数据的数据

- 对于关系表 R : n_R 标识元组数量, b_R 标识页面数量。
- 额外信息:
 - f_R 一个页面能容纳的元组数量, $b_R = \lceil n_R / f_R \rceil$ 。
 - $V(A, R)$: R 中属性 A 的不同值数量, 即 $\pi_A(R)$ 的大小。

块嵌套循环连接例子： $b_{depositor} = 100$, $b_{customer} = 400$. depositor 为外关系。
 → 最坏情况： $100 \times 400 + 100 = 41000$ 次
 → 最佳情况： $100 + 400 = 500$ 次
 → 一般情况：当 $M = 52.0$ 时， $\frac{100}{52.0} \times 400 + 100 = 900$ 次

图 29: 块嵌套循环连接例子

- 对于每个索引 i , 系统存储以下信息:

- HT_i : 索引 i 的层数, 即索引树的高度。
 - * 对于关系表 R , 根据属性 A 构建的树, 高度 $HT_i = \lceil \log_{f_i}(V(A, R)) \rceil$ 。
 - * 对于哈希索引, 如果我们设定了溢出桶, 则 $HT_i = 1.x$; 否则 $HT_i = 1$ 。
- 额外信息:
 - * f_i : 内节点的扇出数 (fanout)。
 - * LB : 叶节点占用的页面数量。

以上包括关系表的信息、页面的信息、属性的信息以及索引的信息。

用以上信息来刻画选择大小估计表达式。

选择大小估计 Selection Size Estimation

一次操作的输出大小决定了该操作的成本以及后续操作的成本。其准确估计对于优化很重要。

- Equality selection $\sigma_{A=v}(R)$ 例如: $\sigma_{rating=8}(\text{SAILORS})$
 - 定义 $SC(A, R)$: 属性 A 在关系表 R 上的选取数目。
 - A 上满足一个等式的记录的平均数: $SC(A, R) = \frac{n_R}{V(A, R)}$
 - $\lceil SC(A, R)/f_R \rceil$: 如果这些记录按属性 A 排序, 存储这些记录所需的页/块数。
 - 如果记录在 A 上没有排序, 则每条记录可能驻留在不同的页面中。
 - 特殊情况: 如果 A 是关键属性, 则 $SC(A, R) = 1$ 。

Selections Involving Comparisons

选择形式: $\sigma_{A < v}(R)$ (同理可分析 $\sigma_{A > v}(R)$ 的情况) 令 S 表示满足条件的元组的估计数量: $\min(A, R)$ 和 $\max(A, R)$ 可以从目录中获得。

假设属性 A 的值是均匀分布的:

$$S = \begin{cases} 0, & \text{if } v < \min(A, R) \text{ 或 } v > \max(A, R) \\ n_R \cdot \frac{v - \min(A, R)}{\max(A, R) - \min(A, R) + 1}, & \text{otherwise} \end{cases}$$

Example:

$$\sigma_{rating < 2}(\text{SAILORS}) = \# \text{ records in sailors} \cdot \frac{2 - 1}{10 - 1 + 1} = \# \text{ records in sailors}/10$$

使用直方图能得到更准确的估计。本质上是概率统计的一个过程，暂不展开。

等价关系表达式：有很多表达式的结果是等价的，我们称其为等价表达式。

- 选择操作的交际可以结构为单独选择，且选择是可交换的。
- 在序列投影操作中只需要保留最后一个，其他的投影操作可以省略。
- 选择条件可以嵌入到叉积或连接里面。
- 连接操作是可交换、元素可结合、选择条件可结合的。
- 在以下两个条件中，选择操作可以前置到 theta join 操作上。
 - 当 θ_1 只涉及 E_1 表中的属性， θ_2 只涉及 E_2 表中的属性：

$$\sigma_{\theta_1 \wedge \theta_2}(\mathbb{E}_1 \bowtie_{\theta} \mathbb{E}_2) = (\sigma_{\theta_1}(\mathbb{E}_1)) \bowtie_{\theta} (\sigma_{\theta_2}(\mathbb{E}_2))$$

在数据库中，Theta Join（θ 连接）是一种基于条件的连接操作，用于将两个表按照特定的条件组合在一起。与等值连接（Equi-Join）不同，Theta Join 可以使用任意比较运算符，而不仅仅是等于运算符（=）。

目前查询优化器包含一下两种主要方法：

- 启发式算法优化：尽早执行选择，尝试利用检索，删除无用属性
- 基于成本的优化：估算所有计划，选择最优计划。

启发式算法优化

- 尽早执行选择操作：当 θ_1 只涉及 E_1 表中的属性， θ_2 只涉及 E_2 表中的属性：

$$\sigma_{\theta_1 \wedge \theta_2}(\mathbb{E}_1 \bowtie_{\theta} \mathbb{E}_2) = (\sigma_{\theta_1}(\mathbb{E}_1)) \bowtie_{\theta} (\sigma_{\theta_2}(\mathbb{E}_2))$$
- 尽早执行投影操作： $\pi_a(R \bowtie S) = \pi_a((\pi_{a1}R) \bowtie (\pi_{a2}S))$

基于成本的优化

- n 个连接操作 ($n+1$ 个关系表) 可能得到的连接树数量为： $(2n)!/n!$ 。
 - 连接树的形状数量为： $(2n)!/[n!(n+1)!]$ 。
 - 每棵树共有 $(n+1)!$ 种排列方式。
- 子序列会重复评估，将所有可能子序列写进内存后直接提取。
 \Rightarrow 思路：所有执行计划都可以用 $S_1 \bowtie (\mathbf{S} - S_1)$ 来表示，递归求解。
 1. 枚举 \mathbf{S} 中的所有非空子集 S_1
 2. $\text{cost} = \text{cost}(S_1) + \text{cost}(\mathbf{S} - S_1) + \text{cost}(S_1 \bowtie (\mathbf{S} - S_1))$
 3. 选择 cost 最少的 plan
- 成本仅计算将中间结果协会磁盘的 cost
- 这一部分没看懂。以后搞懂。

12 Transactions 事务处理

事务是访问并可能更新各种数据项的程序执行单元。

事务的性质: ACID, 原子、一致（顺序无关）、隔离（过程不可见）、持久（保持）

目录与索引

1	内容大纲.....	1
2	ER 图表.....	1
3	函数依赖 FD	2
4	关系型数据库设计：3NF	2
5	内存层级和文件组织 377	6
6	索引的相关概念 405	8
7	索引进阶：B ⁺ 树和动态哈希	11
7.1	B ⁺ 树：	11
7.2	动态哈希：	14
8	其它索引：多键访问.....	15
9	查询过程.....	16
10	连接算法.....	20
11	查询优化.....	22
12	Transactions 事务处理.....	25
A	作业二.....	27
B	Tutorial 7: Indexing&B ⁺ tree&Hashing	29

A 作业二

苏睿熹 22330100

Part 1. Convert the ERD into the corresponding database schema:

(Underline the primary keys and foreign keys.)

对于同一种情况（如关系的映射），课件中可能讨论了多种做法，如果课件中没有明确说哪种会最优的，皆以 ppt 出现的第一种做法为准。

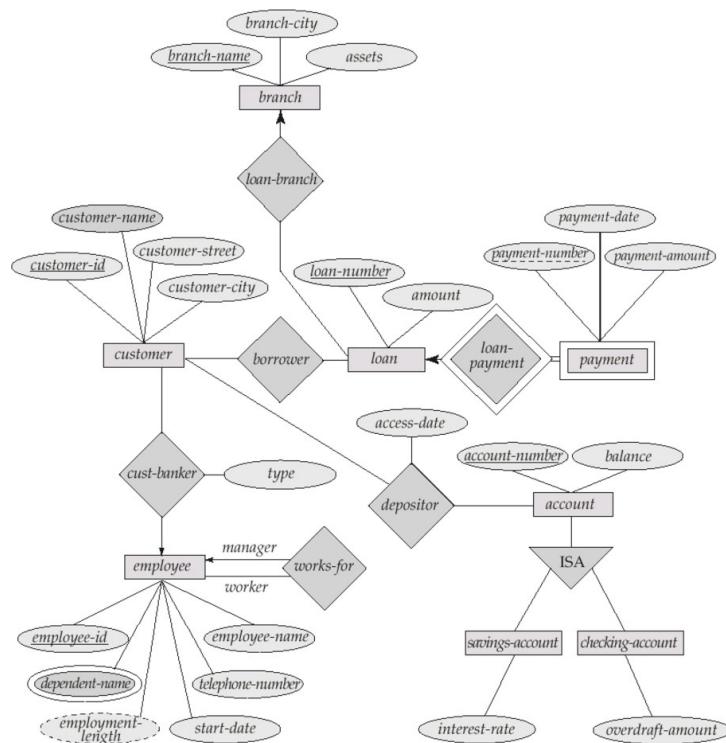


图 30: Part 1 ERD

1. 为强实体集构建相应的 relations:

⇒ BRANCH: branch-name || branch-city || assets

- CUSTOMER: customer-id || customer-name || customer-street || customer-city
- EMPLOYEE: employee-id || start-date || telephone-number || employee-name

由于 employment-length 为衍生属性，所以不创建相应属性以避免数据不一致。

由于 dependent-name 为多值属性，所以多创建一张表 DnTable。

⇒ DNTABLE: employee-id || dependent-name

- LOAN: loan-number || amount
 \Rightarrow ACCOUNT: account-number || balance

2. 为弱实体集创建相应的 relations:

- PAYMENT: loan-number || payment-number || payment-date || payment-amount

3. 构建 1-to-1 关系: 图中没有。

4. 构建 1-to-many 关系:

\Rightarrow LOAN: loan-number || amount || branch-name

\Rightarrow PAYMENT:

loan-number || payment-number || payment-date || payment-amount || loan-number

\Rightarrow CUSTOMER:

customer-id || customer-name || customer-street || customer-city || employee-id || type

\Rightarrow EMPLOYEE:

employee-id || start-date || telephone-number || employee-name || manager-id

5. 构建 many-to-many 关系:

\Rightarrow BORROWER: customer-id || loan-number

\Rightarrow DEPOSITOR: account-number || customer-id || access-date

6. 构建层级模型:

\Rightarrow SAVINGACC: account-number || interest-rate

\Rightarrow CHECKINGACC: account-number || overdraft-amount

在上述过程中, 由箭头 \Rightarrow 指向的关系表为最终构建结果。

Part 2. 3NF and BCNF Decomposition:

Schema $R = (A, B, C, D, E)$ with $F = \{AB \rightarrow C, A \rightarrow D\}$

1. DO the 3NF Decomposition

- 所有的候选键位: A , B 和 E 。
- 不满足 2NF, 因为存在部分依赖 $A \rightarrow D$ 。进行 2NF 分解:
 - $- R \Rightarrow R_1 = (A, B, C), R_2 = (A, D), R_3 = (A, B, E)$
 - 不存在传递依赖, 满足 3NF。

2. DO the BCNF Decomposition

- $R \Rightarrow R_1 = (A, B, C), R_2 = (A, D), R_3 = (A, B, E)$
- R 是在 BCNF 中当且仅当对于每个函数依赖 $X \rightarrow \{A\}$ 在 F^+ 中, 至少满足以下两个条件的其中一条:
 - $A \in X$ (函数依赖是平凡的)。
 - X 是 R 的超键。
- 分解后满足 BCNF。
- 不会丢失函数依赖。

B Tutorial 7: Indexing & B^+ tree & Hashing

内容回顾: 索引的目的是为了加速对需要数据的获取速度。其中比较重要的概念是搜索键, 即用于记录搜索的属性。索引分为顺序索引和哈希索引两种。

索引的特征可分为三项: 主要/辅助, 密集/稀疏, 单层/多层。Primary 指的是搜索键顺序刚好和文件排列顺序一致, 密集指的是一条 index 只对应一个记录, 多层索引即可以为索引创建索引。

Exercise 1:

- ❖ 假设一所学校保存了一份包含学生记录的档案: Student (sid:4 bytes, sname: 10 bytes, dept-id: 4 bytes), dept-id is the department(院系) id where a student belongs to.
- ❖ There exist 10,000 student records and 50 departments. A page is 128 bytes and a pointer is 4 bytes. The data file is sorted sequentially on sid.
- ❖ Q1: Given the data file only, what's the cost of finding students in a particular department (e.g., CSE)?
- ❖ Q2: How to improve?

Q1: 由于数据文件是用 sid 进行排序的, 所以要检索特定 department 的记录, 需要自上而下的检索一遍所有页面。一条学生记录占用 18bytes, 一共有 10000 个学生记录。一个页面最多可以存储记录数量为: $\lfloor 128/18 \rfloor = 7$, 又因为 $\lceil 10000/7 \rceil = 1429$, 所以一共要读取 1429 次磁盘页面。

如果记录可以跨页存储, 则只需要 $\lceil 10000 * 18/128 \rceil = 1407$ 次。

Q2: 构造一个关系实体, 存储 department 和 pointer 之间的对应关系 (department, pointer)。每一条记录为 8bytes。顺序索引和哈希索引的不同开销分析如下:

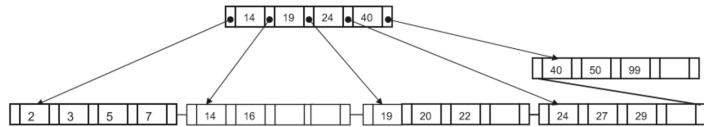
假设每个 department 刚好有 200 个 students。

- 顺序索引:
- 哈希索引:

Exercise 2: B^+ tree:

插入 8:

❖ Given a B^+ -tree:



❖ Draw the tree after

1. Inserting 8
2. Deleting 2
3. Deleting 3

