参考链接

- Chap 9 | Query Processing - 时清川的个人主页
- notebook/docs/DB/db11.md at note1 · HobbitQia/notebook

查询处理 (Query Processing) 与查询优化 (Query Optimization) 笔记 (综合版)

第一部分：查询处理 (Query Processing)

1. 查询处理的基本步骤 (Basic Steps in Query Processing)

查询处理是将用户提交的查询语句（例如 SQL）转化为可以在数据库上高效执行的操作序列，并最终返回查询结果的过程。其基本步骤通常包括：

1. 解析与翻译 **(Parsing and Translation)**：
   - 语法分析 **(Parsing)**：检查查询语句的语法是否正确。
   - 语义检查 **(Semantic Checking)**：验证查询中涉及的表、列等对象是否存在，以及操作是否合法。
   - 翻译 **(Translation)**：将合法的查询语句翻译成内部表示形式，通常是关系代数表达式 (Relational Algebra Expression) 或其变体（如查询树）。
2. 优化 **(Optimization)**：
   - 对关系代数表达式进行优化，生成多种逻辑上等价的表达式。
   - 为这些逻辑表达式中的操作选择具体的实现算法，并考虑操作的执行顺序，从而生成多个候选的物理执行计划 (Physical Evaluation Plan / Execution Plan)。
   - 代价估算 **(Cost Estimation)**：为每个可能的执行计划估算其执行代价。
   - 选择代价最小的执行计划。
3. 求值 **(Evaluation)**：
   - 代码生成 **(Code Generation)** (可选)：某些系统可能会将最优的执行计划编译成可执行的机器码或字节码。
   - 执行引擎 **(Evaluation Engine)**：解释并执行选定的执行计划，访问数据，计算并返回结果。

一个执行计划 **(Evaluation Plan)** 精确定义了每个操作所使用的算法以及这些操作如何协调执行。例如，对于一个选择操作，是使用全表扫描还是索引扫描；对于连接操作，是使用哈希连接还是归并连接。

流水线 **(Pipeline)**：在执行计划中，某些操作的结果可以直接作为后续操作的输入，而无需等待前一个操作完全结束并将中间结果存盘，这种方式称为流水线，可以提高并行度和效率（也称为火山模型 - Volcano Model）。

大多数数据库系统提供 EXPLAIN (或类似) 命令来显示查询优化器选择的执行计划及其估算成本。有些系统还支持 EXPLAIN ANALYZE，它会实际执行查询并显示真实的运行时统计

信息。

2. 查询代价的衡量 **(Measures of Query Cost)**

查询代价通常主要关注磁盘访问 **(Disk Access)**，因为它往往是主要的性能瓶颈，尤其对于大数据量。CPU 代价在传统磁盘密集型系统中常被简化忽略，但随着内存计算和CPU密集型操作（如复杂计算、解压缩）的增多，CPU代价在现代系统中也越发重要。网络通信代价在分布式数据库中也是一个重要因素。

此处主要关注磁盘访问代价：

1. CPU 无法直接操作磁盘中的数据，数据必须先加载到内存。
2. 磁盘数据读/写 (Transfer) 速度的增长远慢于磁盘容量的增长。
3. 磁盘寻道 (Seek) 速度的增长也远慢于磁盘数据传输速度的增长。

衡量指标包括：

- 磁盘寻道次数 **(Number of Seeks,** S**)**：磁头移动到正确磁道所需的时间。
- 读取的磁盘块数 **(Number of Blocks Read)**。
- 写入的磁盘块数 **(Number of Blocks Written)**：通常写操作比读操作耗时，因为可能需要验证写操作是否成功。

为简化起见，通常使用磁盘块传输数量 **(Block Transfers)** 和寻道次数 **(Seeks)** 作为代价衡量：

- tT：传输一个磁盘块所需的时间 (Time to Transfer one Block)。
  - 对于高端磁盘 (Magnetic Disk, 4KB块)：tT≈0.1 毫秒 (ms)。
  - 对于固态硬盘 (SSD, 4KB块)：tT≈2–10 微秒 (μs)。
- tS：一次寻道所需的时间 (Time for one Seek)。
  - 对于高端磁盘：tS≈4 毫秒 (ms)。(tS/tT≈40)
  - 对于SSD：tS≈20–90 微秒 (μs)。(tS/tT≈10)
- 总磁盘访问代价 (Cost) = b×tT+S×tS，其中 b 是块传输数量。

注意事项：

- 我们通常忽略CPU代价以简化模型，但实际系统会考虑。
- 在代价公式中，我们常常忽略将最终结果写入磁盘的代价，因为输出可能直接通过流水线传递给父操作或应用程序。
- 许多算法可以通过使用额外的内存缓冲区来减少磁盘I/O。可用内存量取决于并发查询和操作系统进程，这在执行时才知道。因此，我们经常使用最坏情况估计 **(Worst Case Estimates)**，假设只有操作所需的最小内存可用。
- 所需数据可能已存在于缓冲区 (Buffer Resident)，从而避免磁盘I/O，但这很难在代价估算时精确考虑。

3. 选择操作 **(Selection Operation)**

选择操作 (σcondition(r)) 根据给定条件筛选关系 r 中的元组。

A. 文件扫描 **(File Scan)**

不使用索引来查找和检索满足选择条件的记录。

- 算法 **A1 (**线性搜索**, Linear Search)**：扫描文件的每一个块，并测试块中所有记录是否满足选择条件。
  - 适用于任何选择条件、文件顺序或无论有无索引。
  - 最坏代价 (Worst Cost) = br×tT+tS (br 是关系 r 的块数)。
  - 如果选择条件是针对键属性的，并且找到记录后即可停止：平均代价 (Average Cost) 约为 (br/2)×tT+tS。
- 二分查找 **(Binary Search)**：仅当文件是根据选择条件中的属性排序的，并且选择条件是等值比较时适用。由于数据块通常不连续存储（除非是聚簇组织），二分查找在普通文件上意义不大，它需要更多的寻道次数，通常不如索引搜索。

B. 索引扫描 **(Index Scan)**

利用索引来加速查找满足条件的元组。选择条件必须在索引的搜索键上。
设 hi 为索引的高度（例如，B+树从根到叶节点的层数，若根是第0层，叶是第 hi 层，则需要 hi+1 次访问来到达叶节点）。

- 算法 **A2 (**主 **B+**树索引**/**聚簇 **B+**树索引，键上的等值查找**, Primary B+-tree index / Clustering B+-tree index, equality on key)**：检索满足条件的单个记录。
  - 代价 (Cost) = (hi+1)×(tT+tS) (遍历索引的 hi 层内部节点 + 读取1个叶节点 + 读取1个数据块)。
- 算法 **A3 (**主 **B+**树索引**/**聚簇 **B+**树索引，非键上的等值查找**, Primary B+-tree index / Clustering B+-tree index, equality on nonkey)**：检索多个记录，这些记录通常在连续的块上。
  - 设 b 为包含匹配记录的数据块数。
  - 代价 (Cost) = (hi+1)×(tT+tS)+(b−1)×tT (遍历索引并读第一个数据块 + 顺序读剩下的 b−1 个数据块)。简化为 hi×(tT+tS)+tS+b×tT。
- 算法 **A4 (**辅助 **B+**树索引，键上的等值查找**, Secondary B+-tree index, equality on key)**：
  - 代价 (Cost) = (hi+1)×(tT+tS) (同A2，但最后的数据块访问可能不是顺序的)。
- 算法 **A4' (**辅助 **B+**树索引，非键上的等值查找**, Secondary B+-index on nonkey, equality)**：
  - 每个匹配的 n 条记录可能位于不同的块上。这些记录的 n 个指针可能存储在 m 个索引叶块中。
  - 代价 (Cost) = (hi+m+n)×(tT+tS) (遍历索引内部节点 + 读 m 个叶块 + 读 n 个数据块)。这可能非常昂贵。

C. 涉及比较的选择 **(Selections Involving Comparisons)**

例如 σA≤V(r) 或 σA≥V(r)。

- 算法 **A5 (主 B+树索引/聚簇 B+树索引，比较操作, Primary B+-index / Clustering B+-index, comparison)**：(假设关系在属性 A 上有序)
  - 对于 σA≥V(r)：使用索引找到第一个值 ≥V 的元组，然后顺序扫描关系。代价同 A3。
  - 对于 σA≤V(r)：可以直接顺序扫描关系直到第一个元组 >V，不一定使用索引(除非 V 很小)。如果使用索引，找到 V 然后反向扫描或从头扫描到 V。
- 算法 **A6 (辅助 B+树索引，比较操作, Secondary B+-tree index, comparison)**：
  - 对于 σA≥V(r)：使用索引找到第一个索引项 ≥V，然后顺序扫描索引叶节点以找到指向记录的指针。
  - 对于 σA≤V(r)：扫描索引叶节点找到所有 ≤V 的项的指针。
  - 每条记录都需要一次I/O来获取。线性文件扫描可能更便宜，如果满足条件的记录很多。

D. 复杂选择的实现 (Implementation of Complex Selections)

- 合取 **(Conjunction)**: σθ1∧...∧θn(r)
  - 算法 **A7 (使用单个索引的合取选择, Conjunctive selection using one index)**：
    - 选择一个 θi 和相应的索引扫描算法，使得 σθi(r) 的代价最小。
    - 将获取的元组读入内存后，再检查其他 θj(j𠔃=i) 条件。
  - 算法 **A8 (使用组合索引的合取选择, Conjunctive selection using composite index)**：
    - 如果存在合适的多键组合索引 (Composite Index)，则使用它。
  - 算法 **A9 (通过标识符交集实现的合取选择, Conjunctive selection by intersection of identifiers)**：
    - 如果多个条件属性都有索引(且索引提供记录指针/RID)。分别查找满足每个条件的记录标识符集合，然后取这些集合的交集。最后根据交集中的标识符获取记录。如果某些条件没有索引，则在内存中应用这些测试。
- 析取 **(Disjunction)**: σθ1∨...∨θn(r)
  - 算法 **A10 (通过标识符并集实现的析取选择, Disjunctive selection by union of identifiers)**：
    - 如果所有条件都有可用的索引，则对每个条件使用其索引获取记录标识符，然后取这些标识符集合的并集(注意去重)。
    - 否则，通常使用线性扫描。
- 否定 **(Negation)**: σ¬θ(r)
  - 通常使用线性扫描。

E. 位图索引扫描 (Bitmap Index Scan)

对于取值可能性较少的列(低基数列)，位图索引非常有效。PostgreSQL 中的位图索引扫描算法：

- 目的：弥合二级索引扫描(当匹配记录数未知时可能低效)和线性文件扫描之间的差

距。
- 位图结构：通常为关系中的每个页面（块）使用1位来创建一个位图。
- 步骤：
  1. 索引扫描与位图构建：使用常规索引（如B+树）查找满足选择条件的记录ID。对于每个找到的记录ID，在位图中设置其对应数据页面（块）的位为1。
  2. 线性文件扫描：然后，执行线性文件扫描，但只读取那些在位图中对应位被设置为1的页面（块）。
- 性能：
  - 当只有少数位被设置时（高选择性），性能类似于索引扫描。
  - 当大多数位被设置时（低选择性），性能类似于线性文件扫描（但避免了读取完全不相关的页面）。
  - 相对于最佳选择，其性能通常不会太差，具有较好的鲁棒性。

4. 排序 **(Sorting)**

如果关系数据量太大无法一次性装入内存 (External Sort)，通常使用外排序-归并 **(External Sort-Merge)** 算法。也可以通过已有的有序索引来按顺序读取元组，但这可能导致对每个元组进行一次磁盘块访问，效率较低。

过程 **(Procedure)**

设 M 为内存大小（以页/块为单位），关系 r 占用 br 个块。

1. 创建有序的归并段 **(Create Sorted Runs)**：
   - 重复以下步骤直到关系末尾：
     - 读取 M 个关系块到内存。
     - 对内存中的块进行内部排序 (e.g., Quicksort)。
     - 将排序后的数据写回磁盘，形成一个归并段 (Run) Ri。
   - 此阶段会生成 N0=⌈br/M⌉ 个初始归并段。
2. 合并归并段 **(Merge the Runs)**：
   - 如果 N0<M (归并段数量小于可用内存页数)：
     - 可以进行 N0 路归并 (N-way merge)。为每个归并段分配一个输入缓冲区（通常1块），外加一个输出缓冲区（通常1块）。
   - 如果 N0≥M (归并段数量大于等于可用内存页数)：
     - 需要多趟 (Multiple Passes) 合并。每一趟，将 k=M−1 (因为需要一个块作为输出缓冲区) 个归并段合并成一个更大的归并段。
     - 一趟归并将运行段数量减少约 M−1 倍。
     - 重复此过程，直到所有归并段合并成一个。总共需要的归并趟数 P=⌈logM−1(N0)⌉。

代价分析 **(Cost Analysis)**

- 块传输 **(Block Transfers)**：

    - 初始创建归并段：读 br，写 br (共 2br)。
    - 每趟归并：读 br，写 br (共 2br)。
    - 如果不计算最后一次写入（结果可能直接流水线）：总块传输 ≈br×(2×⌈logM−1(br/M)⌉+1)。
- 寻道次数 **(Seeks)** (高级版本，每个输入/输出流使用 bb 块缓冲区)：
    - 可同时归并的流数量 k=⌊M/bb⌋−1。
    - 初始创建归并段：2×⌈br/M⌉ 次 (假设每次读写 M 块都是连续的，需要一次寻道)。
    - 每趟归并：大约 2×⌈br/bb⌉ 次寻道 (读写所有数据，每次 bb 块)。
    - 总寻道次数 ≈2⌈br/M⌉+⌈br/bb⌉×(2⌈logk(⌈br/M⌉)⌉−1) (最后一次归并的写寻道可能不计)。
- bb 的权衡：
    - 增大 bb（为每个归并流分配更多缓冲块）可以减少每趟归并的寻道次数。
    - 但会减少可同时归并的流的数量 k，可能增加归并的总趟数 P=⌈logk(N0)⌉。
    - 总块传输（考虑 bb，最后写不计）：br(2⌈log⌊M/bb⌋−1(⌈br/M⌉)⌉+1)。

5. 连接操作 **(Join Operation)**

连接操作 (r⋈conditions) 是将两个关系 r 和 s 中满足连接条件的元组合并起来。

A. 嵌套循环连接 **(Nested-Loop Join)**

```
for each tuple tr in r do // 外层关系 (Outer Relation)
  for each tuple ts in s do // 内层关系 (Inner Relation)
    if tr and ts satisfy the join condition then
      add tr ts to result
```

- 代价 (最坏情况，内存仅够各放一块)：nr×bs+br 次块传输；nr+br 次寻道。
- 如果较小关系 (设为 s) 能完全放入内存，则代价为 br+bs 次块传输和 2 次寻道。

B. 块嵌套循环连接 **(Block Nested-Loop Join)**

```
for each block Br of r do
  for each block Bs of s do
    for each tuple tr in Br do
      for each tuple ts in Bs do
        if tr and ts satisfy the join condition then
          add tr ts to result
```

- 最坏情况代价：br×bs+br 次块传输；2×br 次寻道。
- 内存优化：若内存有 M 块，1 块用于输出，1 块用于内层关系 s 的当前块，则 M−2 块可用于缓存外层关系 r 的块。

- 代价：⌈br/(M−2)⌉×bs+br 次块传输；2×⌈br/(M−2)⌉ 次寻道。
- 如果 r 能完全放入 M−2 块内存中，则代价为 br+bs 次块传输，2次寻道。

C. 索引嵌套循环连接 **(Indexed Nested-Loop Join)**

如果内层关系的连接属性上有索引。

- 对外层关系 r 中的每个元组 tr，使用索引查找 s 中满足连接条件的元组。
- 代价：br(tT+tS)+nr×Clookup，其中 Clookup 是对 s 进行一次索引查找并获取所有匹配元组的代价。

D. 归并连接 **(Merge Join / Sort-Merge Join)**

1. 排序 **(Sort)**：如果关系未在连接属性上排序，则先排序。
2. 归并 **(Merge)**：同时扫描两个已排序的关系，合并满足连接条件的元组。

- 代价 (若已排序)：br+bs 次块传输；⌈br/bb⌉+⌈bs/bb⌉ 次寻道。若未排序，需加上排序代价。
- 内存优化分配：若总内存为 M 块，分配给 r,s 的缓冲块数 xr,xs (xr+xs≤M−1) 最优分配近似为 xr=(M−1)br+bsbr。
- 混合归并连接 **(Hybrid Merge-Join)**：如果一个关系已排序，另一个关系在连接属性上有二级B+树索引。合并已排序关系与B+树的叶节点条目。

E. 哈希连接 **(Hash Join)**

适用于等值连接和自然连接。

1. 分区阶段 **(Partitioning Phase / Build Phase)**：
   - 选择哈希函数 h1 和分区数 np (通常 np≈M−1)。
   - 对较小关系 (构建输入 Build Input, 设为 s) 的每个元组，计算连接属性的哈希值，放入对应哈希桶 si，写回磁盘。
   - 对较大关系 (探测输入 Probe Input, r) 做同样操作，得到 ri。
   - 目标是每个 si 能放入内存。np≥⌈bs/M⌉。通常取 np=⌈bs/M⌉×f (fudge factor ≈1.2)。
   - 代价：2(br+bs) 次块传输。
2. 探测阶段 **(Probing Phase / Match Phase)**：
   - 对每个分区 i：读入 si 到内存，为其连接属性构建内存哈希表 (用不同哈希函数 h2)。然后逐一读取 ri 的元组，用 h2 探测内存哈希表。
   - 代价：br+bs 次块传输。

- 总代价 **(无递归)**：约 3(br+bs) 块传输。加上部分填充块开销可能为 3(br+bs)+4np。寻道次数约 2(⌈br/bb⌉+⌈bs/bb⌉)+2np。
- 递归分区 **(Recursive Partitioning)**：若某构建分区 si 仍大于内存 (M>bs 时一般不需递归)，则需对该 si 和对应 ri 再次分区。
- 溢出处理 **(Overflow Handling)**：因数据倾斜或哈希函数不佳导致某 si 无法装入内存。可进一步分区，或对溢出分区使用块嵌套循环连接。

- 混合哈希连接 **(Hybrid Hash-Join)**：当内存较大但构建输入仍大于内存时。第一个构建分区 s0 直接保留在内存中，对应的 r0 在分区时直接进行探测，无需写出和读回 s0 和 r0。

F. 复杂连接 **(Complex Joins)**

- 合取条件连接 **(**$r \bowtie \theta 1 \wedge \theta 2 s$**)**: 可先执行一个简单连接 $r \bowtie \theta 1 s$，然后在结果上检查 θ2。
- 析取条件连接 **(**$r \bowtie \theta 1 \vee \theta 2 s$**)**: 可计算 $(r \bowtie \theta 1 s) \cup (r \bowtie \theta 2 s)$。

6. 其他操作 **(Other Operations)**

- 去重 **(Duplicate Elimination)**：
  - 排序：排序后删除相邻重复元组。可在外排序的归并段生成和合并阶段进行。
  - 哈希：将元组哈希到桶中，每个桶内去重。
- 投影 **(Projection)**：先对每个元组执行投影，然后去重。
- 聚合 **(Aggregation)**：
  - 排序或哈希：按 GROUP BY 属性将元组分组，然后对每组应用聚合函数。
  - 优化：在排序的归并段生成和合并阶段计算部分聚合值。AVG 通过维护 SUM 和 COUNT 实现。
- 集合操作 **(Set Operations:** $\cup, \cap, -$**)**：
  - 可用排序归并的变体或哈希连接的变体实现。
  - 例如，使用哈希：分区两个关系，然后对每个分区对，构建一个关系的内存哈希表，探测另一个关系。
    - $r \cup s$: 将 si 中不在 ri 哈希表中的元组加入，最后输出哈希表内容。
    - $r \cap s$: 输出 si 中在 ri 哈希表中能找到的元组。
    - $r - s$: 从 ri 的哈希表中删除在 si 中找到的元组，最后输出哈希表剩余内容。
- 外连接 **(Outer Join)**：
  - 可先计算内连接，然后补充未匹配的元组并填充NULL。
  - 或修改连接算法：
    - 归并外连接：合并时，若某关系元组无匹配，则输出该元组并补NULL。
    - 哈希外连接：若 r 是探测输入，输出未匹配的 r 元组补NULL。若 r 是构建输入，探测时标记匹配的 r 元组，最后输出未标记的 r 元组补NULL。

7. 表达式求值策略 **(Evaluation of Expressions)**

- 物化 **(Materialization)**：逐步执行操作，每个操作的结果完整生成并存储在临时关系（磁盘或内存）中，作为后续操作的输入。
  - 优点：总是适用。
  - 缺点：写回和读出中间结果的代价高。
  - 双缓冲 **(Double Buffering)**：可用于输出，一个缓冲区满时写入磁盘，同时另一个缓冲区被填充，以重叠计算和I/O。
- 流水线 **(Pipelining)**：同时评估多个操作，一个操作产生的元组立即传递给其父操作进行处理，无需等待子操作完成。

    - 优点：极大降低代价，无需存储临时关系。
    - 缺点：不适用于某些操作（如排序、标准哈希连接的构建阶段）。
    - 实现方式：
        - 需求驱动 **(Demand-Driven / Lazy / Pull Evaluation)**：顶层操作请求元组，该请求向下一级传递，直到叶操作产生元组。操作间通过迭代器 (Iterator) 模型实现，每个操作有 open(), next(), close() 方法，next() 调用间维护状态。
        - 生产驱动 **(Producer-Driven / Eager / Push Evaluation)**：底层操作主动生成元组并推送给父操作。操作间需要缓冲区，若缓冲区满则子操作等待。
    - 流水线友好算法：如索引嵌套循环连接。混合哈希连接的部分阶段也可流水线。双流水线连接 (Double-pipelined join) 技术允许更早输出结果。

8. 内存查询处理与缓存感知算法 **(Query Processing in Memory & Cache Conscious Algorithms)**

- 查询编译 **(Query Compilation)**：将查询计划编译为机器码或字节码 (如通过LLVM, Java JIT)，以减少解释执行的开销（如元数据查找、表达式计算开销 ）。
- 列式存储 **(Column-Oriented Storage)**：利于仅访问查询所需列，提高缓存效率，并支持向量化操作 (SIMD)。
- 缓存感知算法 **(Cache Conscious Algorithms)**：设计算法以最小化CPU缓存未命中 (Cache Misses)，充分利用已加载到缓存的数据。
    - 排序：创建适合L3缓存大小的初始归并段。
    - 哈希连接：创建适合内存的分区后，进一步子分区使构建方的子分区及其哈希索引能放入L3缓存。
    - 数据布局：常一起访问的属性物理上存放在一起。
    - 多线程并行：一个线程因缓存未命中阻塞时，其他线程仍可执行。

第二部分：查询优化 (Query Optimization)

查询优化的目标是找到一个与原始查询等价且执行代价最小的执行计划。

1. 引言 **(Introduction)**

对于一个给定的查询，存在多种评估方式：

- 等价表达式 **(Equivalent Expressions)**：通过关系代数的等价规则，可以将一个查询表达式转换为多种逻辑上等价但形式不同的表达式。
- 每个操作的不同算法 **(Different Algorithms for each Operation)**：为逻辑表达式中的每个关系代数操作选择具体的实现算法。

优化步骤通常包括：

1. 生成逻辑等价表达式。
2. 对表达式进行注解，形成不同的查询计划。
3. 基于估算的代价选择最便宜的计划。

2. 关系表达式的转换 **(Transformation of Relational Expressions / Generating Equivalent Expressions)**

如果两个关系代数表达式在任何合法的数据库实例上都产生相同的元组集合（对于SQL，是多重集 Multiset），则称它们是等价的 **(Equivalent)**。

关键等价规则 **(Key Equivalence Rules)**

1. 选择操作 **(Selection)**：
   - $\sigma\theta1 \wedge \theta2(E) \equiv \sigma\theta1(\sigma\theta2(E))$ (选择的串联/分解)。
   - $\sigma\theta1(\sigma\theta2(E)) \equiv \sigma\theta2(\sigma\theta1(E))$ (选择的交换律)。
   - $\sigma\theta(E1 \times E2) \equiv E1 \bowtie \theta E2$。
   - $\sigma\theta1(E1 \bowtie \theta2 E2) \equiv E1 \bowtie \theta1 \wedge \theta2 E2$。
   - 选择下推：$\sigma\theta(E1 \bowtie E2) \equiv \sigma\theta(E1) \bowtie E2$ (若 θ 只涉 E1 属性)。
   - $\sigma\theta1 \wedge \theta2(E1 \bowtie E3) \equiv (\sigma\theta1(E1)) \bowtie E(\sigma\theta2(E3))$ (若 $\theta1$ 只涉 E1, $\theta2$ 只涉 E3)
2. 投影操作 **(Projection)**：
   - $\Pi L1(\Pi L2(...\Pi Ln(E)...)) \equiv \Pi L1(E)$ (投影的串联)。
   - 投影下推：$\Pi L(E1 \bowtie \theta E2) \equiv \Pi L(\Pi L1 \cup Attr(\theta1)(E1) \bowtie \theta \Pi L2 \cup Attr(\theta2)(E2))$，其中 L1,L2 是 E1,E2 中最终输出 L 所需的属性，$Attr(\theta i)$ 是 θ 中涉及 Ei 的属性。
3. 连接操作 **(Join)**：
   - $E1 \bowtie \theta E2 \equiv E2 \bowtie \theta E1$ (连接的交换律)。
   - $(E1 \bowtie E2) \bowtie E3 \equiv E1 \bowtie (E2 \bowtie E3)$ (自然连接的结合律)。Theta连接的结合律有特定条件。
4. 集合操作 **(Set Operations)** ($\cup, \cap, -$)：
   - $\cup, \cap$ 是可交换和可结合的。
   - 选择可分配于 $\cup, \cap, -$ (如 $\sigma\theta(E1-E2) \equiv \sigma\theta(E1)-\sigma\theta(E2)$ 或 $\sigma\theta(E1)-E2$)。
   - 投影可分配于 $\cup$。
5. 外连接 **(Outer Join)**：
   - 全外连接可交换。左右外连接不可交换，但 E1 LEFT OUTER JOIN E2≡E2 RIGHT OUTER JOIN E1。
   - 外连接通常不可结合。
   - 选择下推到外连接：$\sigma\theta1(E1 \text{ LEFT OUTER JOIN}\theta E2) \equiv (\sigma\theta1(E1)) \text{ LEFT OUTER JOIN}\theta E2$ (若 $\theta1$ 只涉 E1)。
   - 如果选择条件 $\theta1$ 对 E2 的属性是空值拒绝 (Null-Rejecting)，则 $\sigma\theta1(E1 \text{ LEFT OUTER JOIN}\theta E2) \equiv \sigma\theta1(E1 \bowtie \theta E2)$。
6. 聚合 **(Aggregation)**：
   - $\sigma\theta(GGA(E)) \equiv GGA(\sigma\theta(E))$ (若选择条件 θ 只涉及分组属性 G)。

等价表达式的枚举 **(Enumeration of Equivalent Expressions)**

通过反复应用等价规则系统地生成等价表达式。为避免空间和时间浪费，优化器会：

- 共享公共子表达式：当 E1 由 E2 生成时，若其子树相同，则用指针共享。

- 动态规划/记忆化 **(Memoization)**：存储已优化的子表达式的最佳计划，避免重复计算。

3. 代价估算的统计信息 **(Statistical Information for Cost Estimation)**

- nr: 关系 r 中的元组数量。
- br: 包含关系 r 元组的块数量。
- lr: 关系 r 中元组的平均大小。
- fr: 关系 r 的块因子 (fr=⌊blockSize/lr⌋)。
- V(A,r): 关系 r 中属性 A 的不同值的数量。
- SC(A,r): 关系 r 中属性 A 的选择基数，即满足 A=value 的平均记录数，估算为 nr/V(A,r)。
- min(A,r),max(A,r): 属性 A 的最小值和最大值。
- 直方图 **(Histograms)**：更精确描述属性值分布。
  - 等宽直方图 **(Equi-width)**：将属性值域划分为等宽的桶。
  - 等深直方图 **(Equi-depth / Equi-height)**：调整桶的宽度，使每个桶包含大致相同数量的元组。

选择基数估算 **(Selection Cardinality Estimation)**

- σA=v(r): 若 A 是键，大小为1；否则为 nr/V(A,r)。
- σA≤v(r): 均匀分布假设下为 nr×max(A,r)−min(A,r)v−min(A,r)。无统计信息则为 nr/2 或 nr/C。
- 复杂选择（合取、析取、否定）基于独立性假设和各子条件选择率估算。

连接基数估算 **(Join Size Estimation)**

- r×s: nr×ns 元组。
- r⋈s (连接属性 A from r, B from s):
  - 若 A 是 r 主键，B 是 s 外键引用 A: 大小为 ns。
  - 一般情况: min(V(A,r)nr×ns,V(B,s)nr×ns) 或 nr×ns/max(V(A,r),V(B,s))。

其他操作及不同值数量估算 **V(A,result)**

- 投影 ΠL(r): 大小 nr (除非去重)。V(A,ΠL(r))=V(A,r) if A∈L。
- 聚合 GGF(r): 大小 V(G,r)。V(\text{agg_attr}, \text{result}) 估算为 \min(V(\text{agg_attr}, r), V(G,r)) (对MIN/MAX)，或 V(G,r) (对其他聚合)。
- 选择 σθ(r): V(A,σθ(r)) 若 θ 为 A=v 则为1；若为 A op v 则为 V(A,r)×selectivity(θ)；其他情况 min(V(A,r),nσθ(r))。
- 连接 r⋈s: 若 A 只来自 r, V(A,r⋈s)=min(V(A,r),nr⋈s)。若 A 为连接属性, V(A,r⋈s)=min(V(A,r),V(A,s))。

4. 执行计划的选择 **(Choice of Evaluation Plans)**

基于代价的连接顺序选择 **(Cost-Based Join-Order Selection)**

- 动态规划 **(Dynamic Programming)**：
  - 对 n 个关系的连接，考虑所有子集的最佳连接计划。
  - 浓 **bushy trees**: 时间 O(3n)，空间 O(2n)。
  - 左深连接树 **(Left-Deep Join Trees)**：右操作数总是基表。时间 O(n2n)，空间 O(2n)。
- 相关因素：
  - 有趣的排序顺序 **(Interesting Sort Orders)**：某个连接算法（如归并连接）产生的有序输出可能对后续操作（如另一个归并连接、GROUP BY、ORDER BY）有利，即使该算法本身代价较高。动态规划需考虑为每个子集保留不同有趣排序顺序的最佳计划。

启发式优化 **(Heuristic Optimization)**

1. 尽早执行选择 (下推选择)。
2. 尽早执行投影 (下推投影)。
3. 优先执行选择性最强（结果最小）的选择和连接。
4. 优先考虑左深连接树。
5. 避免笛卡尔积。

查询优化器的结构

- 许多优化器优先考虑左深连接树。
- 一些优化器先进行启发式重写（如去相关、下推），然后对每个查询块进行基于代价的连接顺序优化。
- 计划缓存 **(Plan Caching)**：缓存已优化的计划，以便在相同或相似查询（即使常量不同）再次提交时重用。
- 优化代价预算 **(Optimization Cost Budget)**：如果优化本身过于耗时，可能提前终止并选择当前找到的最佳计划。

5. 高级查询优化技术 **(Advanced Query Optimization Techniques)**

A. 嵌套子查询优化 **(Optimizing Nested Subqueries)**

- 相关子查询 **(Correlated Subquery)**：内部查询引用外部查询的属性（相关变量）。直接的嵌套循环求值（相关执行 Correlated Evaluation）效率低。
- 去相关 **(Decorrelation)**：将相关子查询转换为等价的连接操作。
  - EXISTS → 半连接 (Semijoin, ⋉)。
  - IN → 半连接或普通连接+去重。
  - NOT EXISTS → 反半连接 (Anti-semijoin, ▷)。
  - NOT IN → 反半连接或外连接+IS NULL。
  - 标量聚合子查询 (Scalar Aggregate Subqueries) → 外连接 + GROUP BY。

B. 物化视图 **(Materialized Views)**

预先计算并存储的视图。

- 增量视图维护 **(Incremental View Maintenance)**：根据基表的差量 (Differential: ir,dr) 更新视图，而非完全重新计算。
  - 连接 $V=R \bowtie S$：插入 $iR \Rightarrow Vnew=Vold \cup (iR \bowtie S)$。
  - 投影 $\Pi A(R)$：需维护元组计数。
  - 聚合：类似维护计数或总和。
- 查询重写 **(Query Rewriting)**：优化器判断能否用物化视图回答用户查询，并重写查询。
- 物化视图选择 **(Materialized View Selection)**：选择哪些视图进行物化以获得最大收益，是数据库调优的一部分，通常有工具辅助。

C. 其他高级技术

- **Top-K** 查询优化：例如 SELECT ... ORDER BY ... LIMIT K。可使用特殊算法，或估算范围后转换为选择查询。
- 更新优化 **(Optimization of Updates)**：
  - 万圣节问题 **(Halloween Problem)**：UPDATE R SET A = A * 1.1 WHERE A > 100。若使用 A 的索引扫描并立即更新，可能导致同一元组被多次更新。
  - 解决方案：延迟更新（先收集所有要更新的元组，再统一应用）；或仅在更新影响选择条件中的属性时才延迟。
- 连接消除/最小化 **(Join Minimization/Elimination)**：若某连接是多余的（如连接到仅用于检查存在性的表，且有外键约束保证存在性），则可消除。
- 多查询优化 **(Multi-Query Optimization)**：同时优化一批查询，通过识别和共享公共子表达式或扫描来降低总代价。
- 参数化查询优化 **(Parametric Query Optimization)**：查询中包含参数（如 WHERE R.A < ?），其值在编译时未知。
  - 方案1：运行时优化。
  - 方案2：生成多个计划，每个对应参数值的某个范围，运行时选择。
  - 方案3：计划缓存，若优化器认为一个计划对多数参数值都适用。
- 自适应查询处理 **(Adaptive Query Processing)**：
  - 运行时根据实际的中间结果大小或数据分布调整执行策略（如在嵌套循环和哈希连接间切换 ）。
  - 若统计信息与实际情况偏差过大，可暂停执行，重新优化，再继续。

这份笔记整合了您提供的材料以及PDF文档中的核心内容，希望能为您的教学提供一个更全面、详细的参考。