W5

Query Evaluation part1

查询评估中的单个基本操作，包括**选择**、**投影**和**外部归并排序。**

Selection

选择操作 $\sigma_p(r)$ 是从关系 (r) 中选择出符合谓词 (p) 的元组。

评估方式：

1. File Scan

存储的块连续分布，系统会逐个读取文件的所有数据块

- **linear search**：逐个扫描所有记录并检查是否符合选择条件。
  - **cost**： br transfers + 1 seek

    读取整个关系 (r)，需要传输 (br) 个块。磁盘查找次数为 (1)。
  - **平均cost**： (br/2) transfers + 1 seek
- **Binary Search**：仅当在有序文件上进行相等比较时适用
  - **成本**：二分查找需要 ($\log_2(br)$) 次磁盘查找，以及同样次数的块读取。
  - **in time** : $\log_2(br) * (T_t + T_s)$
  - 如果多个记录满足条件：加上 读取这些块的cost
  - br 不大的话不值得用（适合处理大规模数据）

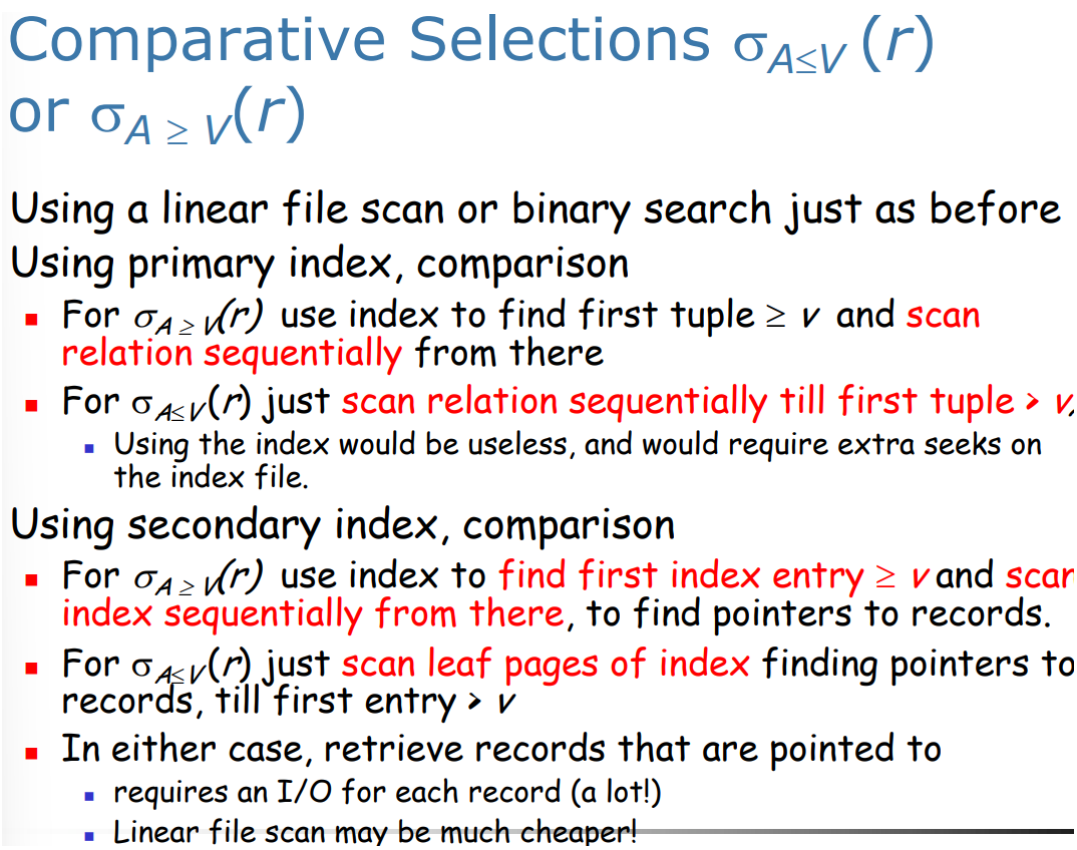
2. Index Scan

如果在Selection condition上有索引，则可以利用索引进行高效的选择操作（等值）：

- **主索引在候选键上**：如果索引是基于主键的索引，查询效率较高。
  - **cost**：(h + 1) trans + (h + 1) seeks， h为索引高度），1是读取record
  - **time**: $(h+1) * (T_t + T_s)$
  - B+树的最大高度为 $log_{n/2}(K)$, n是node中的pointer的数量，K是key的数量
- **主索引在非键上**：可能有多个记录, b 表示包含匹配记录的块数量
  - **Cost** = (h + b) block transfers + (h +1) seeks,
  - **In time:** $h * (T_t + T_s) + T_s + T_t * b$
- **次级索引在候选键上**：非键上的次级索引，可能有多个record, n 代表可能分布在不同块中的记录数量
  - **cost**: (h + 1) trans + (h + 1) seeks， h为索引高度），1是读取record
  - **time**: $(h+1) * (T_t + T_s)$
- **次级索引在非键上**: 非键上的次级索引，可能有多个record, n 代表可能分布在不同块中的记录数量
  - **cost**：(h + n) trans + (h + n) seeks
  - 如果n很大开销会很大

Comparative Selection

范围查找 $\sigma_{A\leq V}$ or $\sigma_{A\geq V}$



Conjunctive Selection

多个条件合取，关键在于第一个条件选择成本最小的的，降低整体成本，在接着别的条件。

Projection Operation

projection会删掉没有被选择属性的列，关键在于移除重复项，可以通过hashing 或 sorting去重

- **基于排序的投影**：
  1. 对关系 (r) 进行排序。
  2. 在排序后的结果中去掉重复的元组。
- **基于哈希的投影**：
  1. 使用哈希分区，将相同哈希值的元组放在同一个分区中。
  2. 在每个分区内进行去重。

Sorting

排序的用处：

- order by
- 有些操作（join/ set operation/ aggregation）需要先排序

External Merge Sort

External Sort-Merge 是一种处理大规模数据集排序的常用算法，尤其是在数据无法一次性装入内存的情况下。

- M 指内存中能内存中块的数量

外部排序的过程：

Partion + Merge

1. create sorted runs

初始 i = 0，对关系循环执行以下过程直到结束

1. 读取 M blocks 进内存
2. sort in-memory blocks
3. 将已排序的块写回run file R_i
4. i ++

最终 i 的值为N，也就是N个run file

数据被分成多个块（partion），每个块大小等于内存缓冲区的大小，对每个块进行排序，将已排序的块写回磁盘。

2. merge the runs

内存划分：M-1blocks input, 1 block output

- 在排序后的多个块之间进行合并。
- 每次合并将一部分数据读入内存进行处理，直到所有块都合并成一个有序文件。

当 N < M

每个run file 读取的第一个块

1. 读入所有页的第一个块，查找最小值（第一个record）。
2. 将该值写入 output buffer，并将其从 source input buffer 中删除。
3. 如果输出缓冲区已满，将其写入磁盘。
4. 如果input buffer空了，读下一个块
5. 如果所有 input buffers 都空了，则将 output buffer 的其余部分写回到磁盘，结束。

当 $N >= M$

in one pass merge M-1 runs

每次pass 减少 M-1 个runs，创建更长的runs，直到所有runs merge为一个

cost

Assume relation in b_r blocks, M memory size, number of run file $\lceil b_r/M \rceil$.
Buffer size b_b (read b_b blocks at a time from each run and b_b blocks for output writing; before we assumed b_b=1).

Cost of Block Transfer

- Each time can merge $\lfloor (M-b_b)/b_b \rfloor$ runs;
- So total number of merge passes required: $\lceil \log_{\lfloor M/bb \rfloor -1} \lceil b_r/M \rceil \rceil$.
- Block transfers for initial run creation as well as in each pass is $2b_r$ (read/write all b_r blocks).
- Thus total number of block transfers for external sorting (For final pass, we don't count write cost):
  $2b_r + 2b_r \lceil \log_{\lfloor M/bb \rfloor -1} \lceil b_r/M \rceil \rceil - b_r = b_r (2 \lceil \log_{\lfloor M/bb \rfloor -1} \lceil b_r/M \rceil \rceil + 1)$

Cost of seeks

- During run generation: one seek to read each run and one seek to write each run $2\lceil b_r / M \rceil$
- During the merge phase: need $2\lceil b_r / b_b \rceil$ seeks for each merge pass
- Total number of seeks:
  $2\lceil b_r / M \rceil + 2\lceil b_r / b_b \rceil \lceil \log_{\lfloor M/bb \rfloor -1} \lceil b_r/M \rceil \rceil - \lceil b_r / b_b \rceil =$
  $2\lceil b_r / M \rceil + \lceil b_r / b_b \rceil (2\lceil \log_{\lfloor M/bb \rfloor -1} \lceil b_r/M \rceil \rceil - 1)$

Part2：Join

Nested-Loop Join

最基础的暴力算法，泛用但很慢，对于r的每一行比较s的每一行。联接两个表本质上是每个表中记录的双 for 循环

```
for record r in R:
    for record s in S:
        if join_condition(r, s):
            add <r, s> to result buffer
```

r is called the **outer relation** and s the **inner relation** of the join

成本:

- 在最坏情况下，内存只能容纳每个关系的一个块。

  Let **nr** be the **number of tuples in relation r**, **br** and **bs** be the **number of blocks** of r and s.

  $nr * bs + br$ **block transfer**. 读取r中的所有块，对于r中的每一个tuples，读取s中的所有块。

  $nr + br$ **seeks**. 查找r的每一个块。
- 如果关系小到可以直接放进内存，成本将减少为$br + bs$ 块传输和2次搜索。

Block Nested Loop Join

NLJ效率太低，对于每一个单独记录对另一个表的记录I/O。因此加上buffer使之更高效，利用缓冲区来帮助我们降低 I/O 成本。可以通过在block级别而不是记录级别进行操作来改进这一点：在进入下一个块之前，处理当前页面上记录的所有连接.

```
for rblock in R:
    for sblock in S:
            for rtuple in rpage:
                    for stuple in spage:
                        if join_condition(rtuple, stuple):
                            add <r, s> to result buffer
```

在最坏情况下，成本为`br * bs + br`块传输和`2 * br`搜索。

如果较小的关系能够完全放入内存，成本将减少为`br + bs`块传输和2次搜索。

Nested Loop Join Animations | CS186 Projects (gitbook.io)

Indexed Nested-Loop Join

使用index（即在一个数据结构中查找）减少冗余操作。

如果连接是以下情况则可以用索引查找代替文件扫描：

- 等值连接（*Equi-join*）或自然连接，
- 在内部关系的连接属性上有可用索引

在最坏情况下，缓冲区仅有足够的空间用于外部关系的一个页面。

计算成本：`br + nr * c`块传输和`br + nr * c`搜索，其中`c`为遍历索引和获取所有匹配元组的成本（取决于索引本身）。

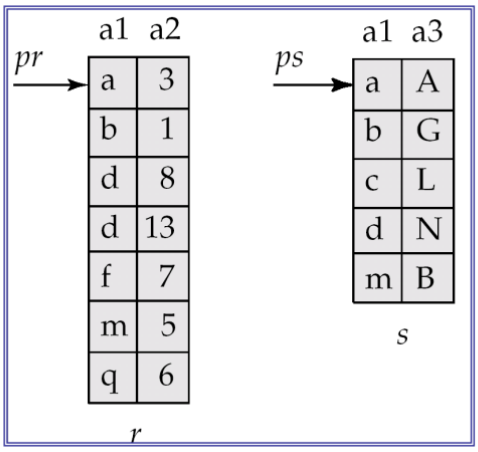
如果两个关系都有索引，把更少tuples的作为外循环

B+树的最大高度：log(N/2) K

Merge-Join

先对两个关系进行排序，然后类似于归并排序中的归并，对于每个范围（具有相同索引的值组）检查匹配项并生成所有匹配项。

仅适用于equi-join连接和nutural join



假设每个块只需要读取一次（fit the memory），成本为$br + bs$块传输和 $\lceil br/bb \rceil + \lceil bs/bb \rceil$搜 索操作（`bb`为分配给每个关系的块数）。

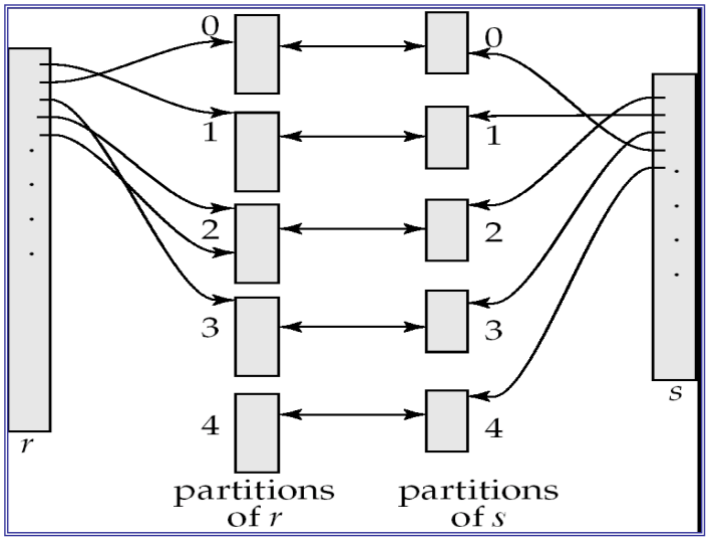
如果关系未排序，还需要加上**排序成本。**

由于seek 远比 transfer 贵，分配更多缓冲区是合理的

Hash-Join

仅适用于equi-join连接和nutural join

使用哈希函数将两个关系的元组分区（partition），然后在分区上进行连接



- **算法步骤**:
  1. 使用哈希函数`h`将关系`s`进行分区。
  2. 将关系`r`也进行相似的分区。
  3. 对每个分区 i
     - 将分区`si`加载到内存中，构建in-memory的哈希索引。这个hash function和分区的hash function不同。
     - 从磁盘中读取`ri`，并查找每个匹配的元组`ts`。

Relation s is called the **build input** and r is called **probe input**

- 哈希连接的成本为`3(br + bs) + 4 * nh`块传输
- `2(⌈br / bb⌉ + ⌈bs / bb⌉) + 2 * nh`搜索。