Query Evaluation

(This note is based on previous note “Relational Model”)

<Selection>

Here is some **definition about the selection operation**:

- Notation: $\sigma_p(r)$
- p is the selection **predicate**
- Defined by:

  $$\sigma_p(r) = \{t \mid t \in r \text{ and } p(t)\}$$

  in which p is a formula of propositional calculus of terms connected by: $\wedge$ (and), $\vee$ (or), $\neg$ (not)

  Each term is of the form:

  <attribute> *op* [<attribute> or <constant>]

  where *op* can be one of: $=, \neq, >, \geq. <. \leq$
- Selection example:

  $\sigma_{\text{branch-name='Perryridge'}}(account)$

Evaluation of Selection Operation [查询操作的评估]:

File scan – search algorithms that scan files and retrieve records that fulfill a selection condition.

[文件扫描-搜索算法，扫描文件并检索满足选择条件的记录]（磁盘顺序搜索）

< linear search>

对<u>任意</u>的文件均适用，以下为时间消耗需求：

- Cost estimate = b_r block transfers + 1 seek

- Average cost = $(b_r/2)$ block transfers + 1 seek

< binary search>

对<u>已排序</u>的文件适用，以下为时间消耗需求：

tT 是 transfer 时间，tS 是 seek 时间

- cost of locating the first tuple by a binary search on the blocks
  - $\lceil \log_2(b_r) \rceil * (t_T + t_S)$
- If there are multiple records satisfying selection
  - Add transfer cost of the number of blocks containing records that satisfy selection condition
  - Will see how to estimate this cost later

Index scan – search algorithms that use an index.

[索引扫描-使用索引的搜索算法]（根据索引键搜索）

< primary index on candidate key >

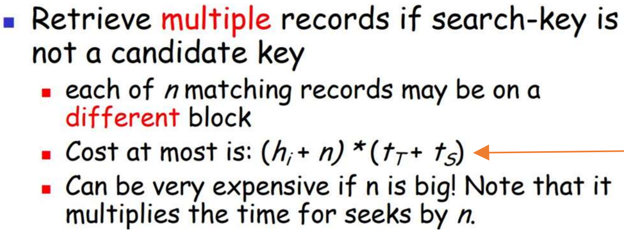
除非关系非常小，不然索引搜索都是<u>高效</u>的，以下为时间消耗需求：

- Retrieve a **single** record that satisfies the corresponding equality condition
  - $Cost = (h_i + 1) * (t_T + t_S)$

  where h_i denotes the **height** of the index

B+-tree index is at most $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$ (n 是每个节点指针的数量)

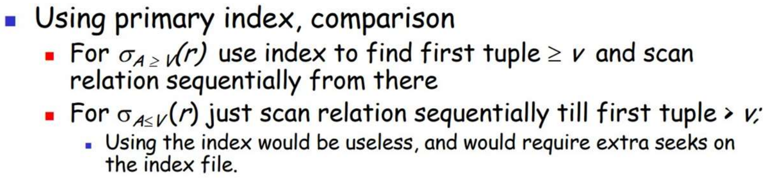
i.e. for a relation with 1,000,000 (1 million) different search keys, and with 100 index entries per node, hi = 4 calculation progress: Log[500000,100] - Wolfram|Alpha (wolframalpha.com)



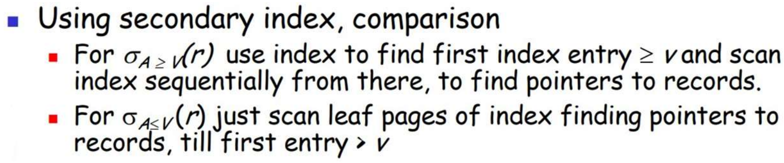
- Retrieve multiple records if search-key is not a candidate key
  - each of n matching records may be on a different block
  - Cost at most is: $(h_i + n) * (t_T + t_S)$
  - Can be very expensive if n is big! Note that it multiplies the time for seeks by n.

▲ Comparative Selections [比较查询]

更推荐线性扫描，由于是排序了的，只需要根据要求找到临界值再顺序检索就可以了。



- Using primary index, comparison
  - For $\sigma_{A \geq V}(r)$ use index to find first tuple ≥ v and scan relation sequentially from there
  - For $\sigma_{A \leq V}(r)$ just scan relation sequentially till first tuple > v;
    - Using the index would be useless, and would require extra seeks on the index file.



- Using secondary index, comparison
  - For $\sigma_{A \geq V}(r)$ use index to find first index entry ≥ v and scan index sequentially from there, to find pointers to records.
  - For $\sigma_{A \leq V}(r)$ just scan leaf pages of index finding pointers to records, till first entry > v

▲ Conjunctive Selections [连接查询]

推荐使用多键索引，如若使用单键索引那么算法将十分重要

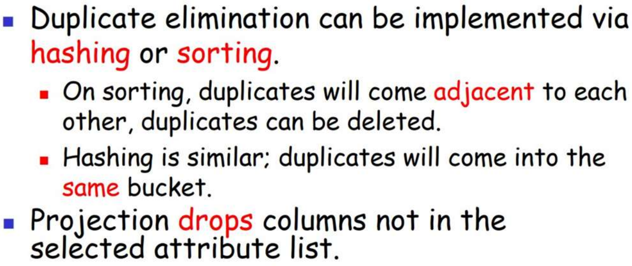
▲ Disjunctive Selections [分隔查询]

使用线性扫描或者索引扫描（如果某些条件有可用的索引），对每个条件使用相应的索引并取所有获得的记录指针集的并集，然后从文件中获取记录。

▲ Selections With Negation [否定查询]

使用线性扫描或者索引扫描

▲ Duplicate Elimination and Evaluating Projection [消除重复&投射评估]



- Duplicate elimination can be implemented via hashing or sorting.
  - On sorting, duplicates will come adjacent to each other, duplicates can be deleted.
  - Hashing is similar; duplicates will come into the same bucket.
- Projection drops columns not in the selected attribute list.

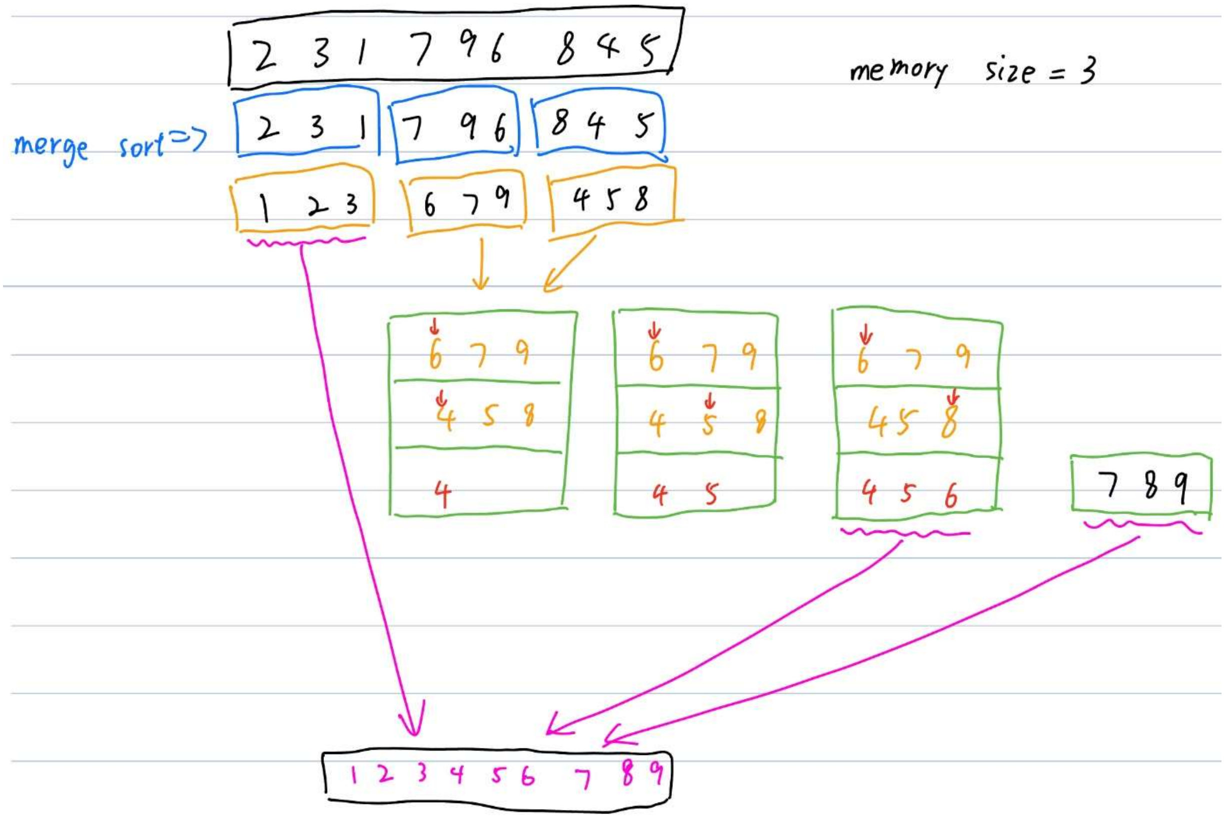
一般去重消耗更大

▲External Sort-Merge [外部排序归并算法]

由于磁盘空间有限，不能使用标准的 merge sort，故采取使用时间复杂度换取空间复杂度的方法。

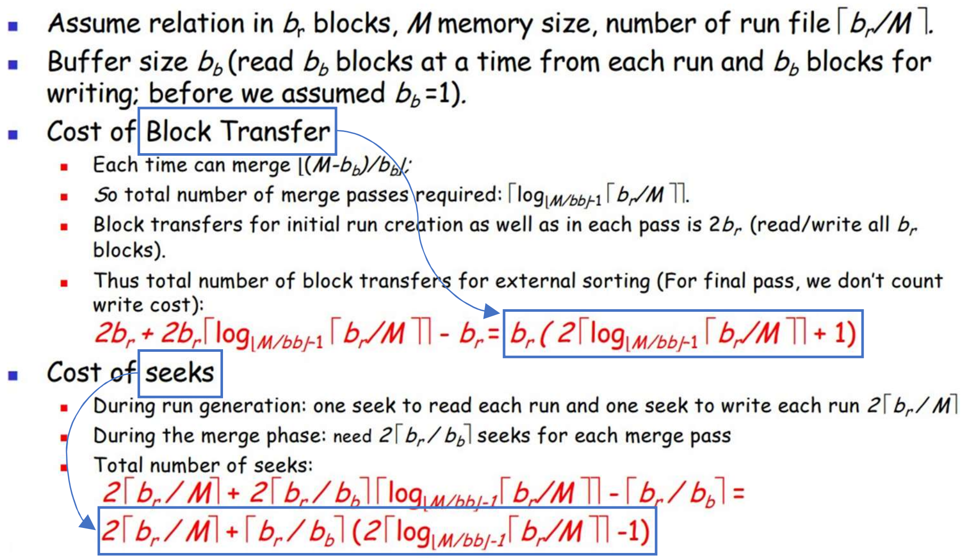
具体做法：将所需排序数据分为不同的块（磁盘最大可接受空间），然后再额外加一块作为产出对比结果后的缓存作用。

具体流程举例：



Reference: CPT201 外部排序归并算法（external sort-merge）与 merge join - 知乎 (zhihu.com)

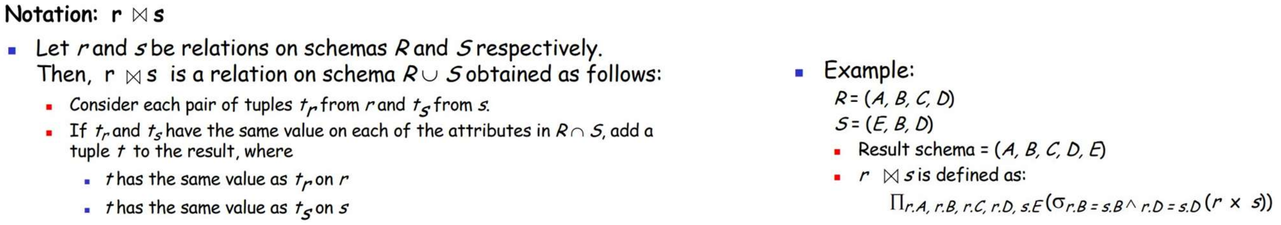
Continue-Cost Analysis:



- 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 writing; before we assumed b_b=1).
- Cost of Block Transfer
  - Each time can merge $\lfloor (M-b_b)/b_b \rfloor$;
  - So total number of merge passes required: $\lceil \log_{\lfloor M/b_b \rfloor - 1} \lceil b_r/M \rceil \rceil$.
  - 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):
    $2b_r + 2b_r \lceil \log_{\lfloor M/b_b \rfloor - 1} \lceil b_r/M \rceil \rceil - b_r = \boxed{b_r (2 \lceil \log_{\lfloor M/b_b \rfloor - 1} \lceil b_r/M \rceil \rceil + 1)}$
- 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:
    $2\lceil b_r / M \rceil + 2\lceil b_r / b_b \rceil \lceil \log_{\lfloor M/b_b \rfloor - 1} \lceil b_r/M \rceil \rceil - \lceil b_r / b_b \rceil =$
    $\boxed{2\lceil b_r / M \rceil + \lceil b_r / b_b \rceil (2 \lceil \log_{\lfloor M/b_b \rfloor - 1} \lceil b_r/M \rceil \rceil - 1)}$

<Join>

▲ Natural-Join Operation [自然连接]

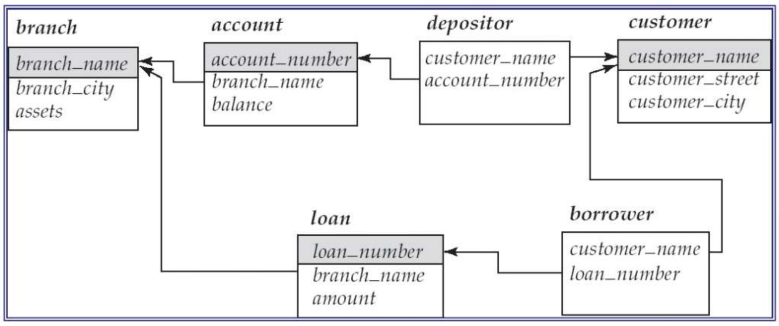


Notation: $r \bowtie s$

- Let r and s be relations on schemas R and S respectively. Then, $r \bowtie s$ is a relation on schema $R \cup S$ obtained as follows:
  - Consider each pair of tuples t_r from r and t_s from s.
  - If t_r and t_s have the same value on each of the attributes in $R \cap S$, add a tuple t to the result, where
    - t has the same value as t_r on r
    - t has the same value as t_s on s

- Example:
  $R = (A, B, C, D)$
  $S = (E, B, D)$
  - Result schema = (A, B, C, D, E)
  - $r \bowtie s$ is defined as:
    $\Pi_{r.A,\, r.B,\, r.C,\, r.D,\, s.E} (\sigma_{r.B = s.B \wedge r.D = s.D} (r \times s))$

Banking example:



- Number of records of *customer*: 10,000
- Number of blocks of *customer*: 400
- Number of records of *depositor*: 5,000
- Number of blocks of *depositor: 100*

▲Nested-Loop Join [嵌套循环连接]

Can be used independently of everything (like the linear search for selection)

```
for each tuple $t_r$ in $r$ do begin
  for each tuple $t_s$ in $s$ do begin
     test pair ($t_r$, $t_s$) to see if they satisfy the join condition $\theta$
     if they do, add $t_r \cdot t_s$ to the result.
  end
end
```

(r is called the outer relation and s the inner relation of the join)

最简单，但消耗较大，cost:

- In the worst case, if there is enough memory only to hold one block of each relation, n_r is the number of tuples in relation r, the estimated cost is:
  - $n_r * b_s + b_r$ block transfers, plus
  - $n_r + b_r$ seeks
- If the smaller relation fits entirely in memory, use that as the inner relation.
  - Reduces cost to $b_r + b_s$ block transfers and 2 seeks
- But in general, it is much better to have the smaller relation as the outer relation
- The choice of the inner and outer relation strongly depends on the estimate of the size of each relation.

- Assuming worst case memory availability cost estimate is
  - with *depositor* as outer relation:
    - 5,000 ∗ 400 + 100 = 2,000,100 block transfers,
    - 5,000 + 100 = 5,100 seeks
  - with *customer* as the outer relation
    - 10,000 ∗ 100 + 400 = 1,000,400 block transfers and 10,400 seeks
- If smaller relation (*depositor*) fits entirely in memory, the cost estimate will be 500 block transfers and 2 seeks

▲Block Nested-Loop Join [模块嵌套循环连接]

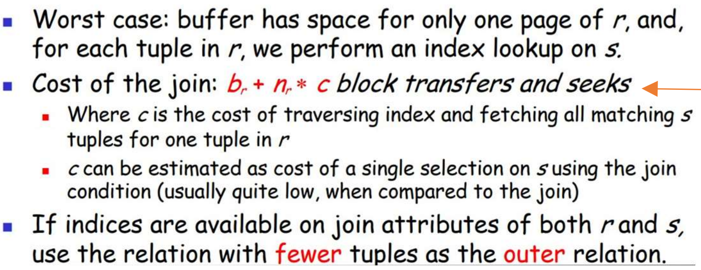
```
for each block $B_r$ of $r$ do begin
   for each block $B_s$ of $s$ do begin
      for each tuple $t_r$ in $B_r$ do begin
         for each tuple $t_s$ in $B_s$ do begin
            Check if ($t_r$, $t_s$) satisfy the join condition
            if they do, add $t_r \cdot t_s$ to the result.
         end
      end
   end
end
```

前后交替扫描内循环，充分利用缓冲区中剩余的块，减少磁盘访问次数，cost:

- Worst case estimate: $b_r * b_s + b_r$ block transfers and $2 * b_r$ seeks
  - Each block in the inner relation s is read once for each *block* in the outer relation (instead of once for each tuple in the outer relation).
- Best case (when smaller relation fits into memory): $b_r + b_s$ block transfers plus 2 seeks.

▲Indexed Nested-Loop Join [索引嵌套循环连接]

检索索引可以避免文件扫描，cost:

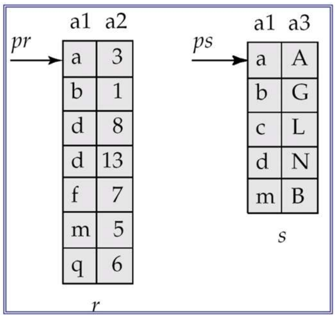


- Worst case: buffer has space for only one page of *r*, and, for each tuple in *r*, we perform an index lookup on *s*.
- Cost of the join: $b_r + n_r * c$ *block transfers and seeks*
  - Where *c* is the cost of traversing index and fetching all matching *s* tuples for one tuple in *r*
  - *c* can be estimated as cost of a single selection on *s* using the join condition (usually quite low, when compared to the join)
- If indices are available on join attributes of both *r* and *s*, use the relation with fewer tuples as the outer relation.

i.e. 设置 customer 有 1000tuples，故 hi=4，然后需要再进行一次访问才能找到实际数据 (4+1)

- Cost of indexed nested loops join
  - 100 + 5,000 * (4+1) = 25,100 block transfers and seeks.
  - The number of block transfers is less than that for block nested loops join
  - But number of seeks is much larger
  - In this case using the index doesn't pay (this is specially so because the relations are small)

▲Merge-Join [合并连接]

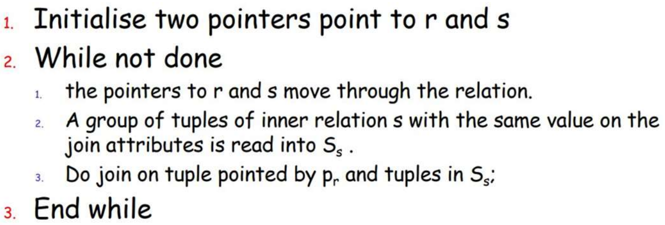


a1	a2
a	3
b	1
d	8
d	13
f	7
m	5
q	6

r

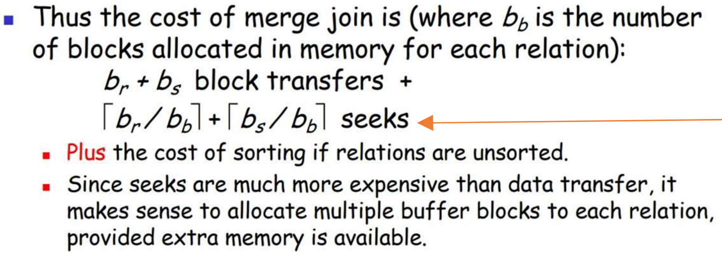
a1	a3
a	A
b	G
c	L
d	N
m	B

s



1. Initialise two pointers point to r and s
2. While not done
   1. the pointers to r and s move through the relation.
   2. A group of tuples of inner relation s with the same value on the join attributes is read into S_s .
   3. Do join on tuple pointed by p_r and tuples in S_s;
3. End while

只能用于等价连接和自然连接



- Thus the cost of merge join is (where b_b is the number of blocks allocated in memory for each relation):

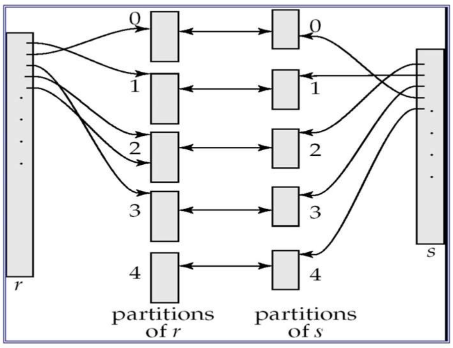
  $b_r + b_s$ block transfers +

  $\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$ seeks
  - Plus the cost of sorting if relations are unsorted.
  - Since seeks are much more expensive than data transfer, it makes sense to allocate multiple buffer blocks to each relation, provided extra memory is available.

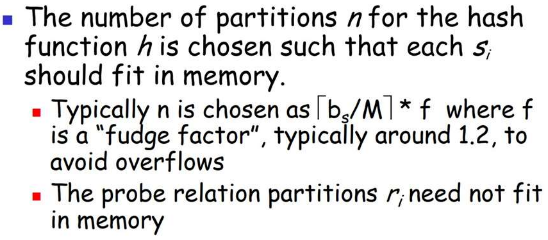
▲Hash-Join [哈希连接]

只能用于等价连接和自然连接

思路：利用哈希函数将数据计算后在公共区域分区，后根据哈希值如果相同可以互相连接

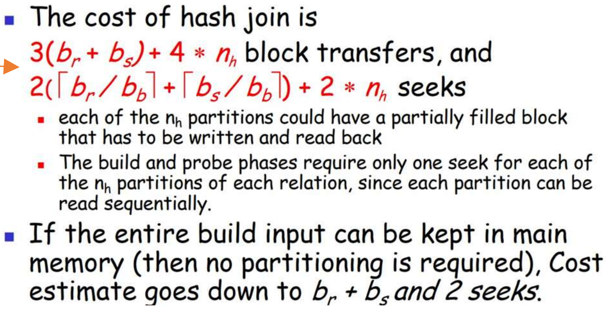


关于分区 n 的计算：

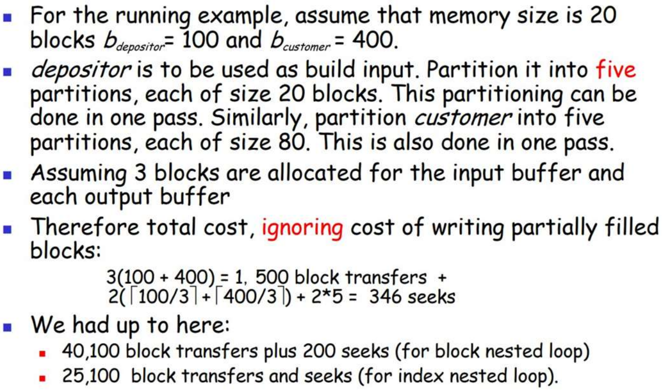


- The number of partitions n for the hash function h is chosen such that each s_i should fit in memory.
  - Typically n is chosen as $\lceil b_s/M \rceil * f$ where f is a "fudge factor", typically around 1.2, to avoid overflows
  - The probe relation partitions r_i need not fit in memory

Cost & example:

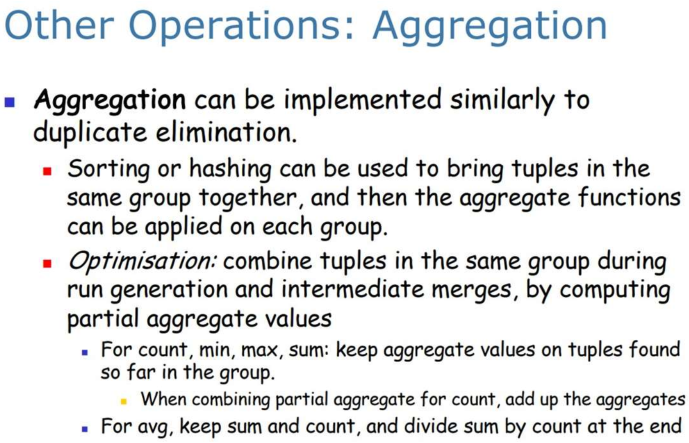


- The cost of hash join is $3(b_r + b_s) + 4 * n_h$ block transfers, and $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) + 2 * n_h$ seeks
  - each of the n_h partitions could have a partially filled block that has to be written and read back
  - The build and probe phases require only one seek for each of the n_h partitions of each relation, since each partition can be read sequentially.
- If the entire build input can be kept in main memory (then no partitioning is required), Cost estimate goes down to $b_r + b_s$ and 2 seeks.



- For the running example, assume that memory size is 20 blocks $b_{depositor} = 100$ and $b_{customer} = 400$.
- *depositor* is to be used as build input. Partition it into five partitions, each of size 20 blocks. This partitioning can be done in one pass. Similarly, partition *customer* into five partitions, each of size 80. This is also done in one pass.
- Assuming 3 blocks are allocated for the input buffer and each output buffer
- Therefore total cost, ignoring cost of writing partially filled blocks:

  $3(100 + 400) = 1,500$ block transfers +
  $2(\lceil 100/3 \rceil + \lceil 400/3 \rceil) + 2*5 = 346$ seeks
- We had up to here:
  - 40,100 block transfers plus 200 seeks (for block nested loop)
  - 25,100 block transfers and seeks (for index nested loop).

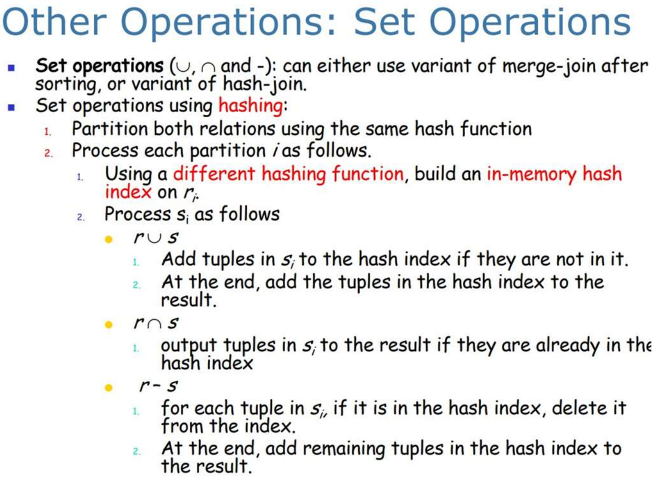


Other Operations: Aggregation

- **Aggregation** can be implemented similarly to duplicate elimination.
  - Sorting or hashing can be used to bring tuples in the same group together, and then the aggregate functions can be applied on each group.
  - *Optimisation:* combine tuples in the same group during run generation and intermediate merges, by computing partial aggregate values
    - For count, min, max, sum: keep aggregate values on tuples found so far in the group.
      - When combining partial aggregate for count, add up the aggregates
    - For avg, keep sum and count, and divide sum by count at the end

聚合函数：sum/avg/count/min/max

聚合的实现与重复消除类似，可以使用排序或哈希将同一组中的元组放在一起，然后在每个组上应用聚合函数。



Other Operations: Set Operations

- **Set operations** ($\cup$, $\cap$ and -): can either use variant of merge-join after sorting, or variant of hash-join.
- Set operations using hashing:
  1. Partition both relations using the same hash function
  2. Process each partition *i* as follows.
     1. Using a different hashing function, build an in-memory hash index on r_i.
     2. Process s_i as follows
        - $r \cup s$
          1. Add tuples in s_i to the hash index if they are not in it.
          2. At the end, add the tuples in the hash index to the result.
        - $r \cap s$
          1. output tuples in s_i to the result if they are already in the hash index
        - $r - s$
          1. for each tuple in s_i, if it is in the hash index, delete it from the index.
          2. At the end, add remaining tuples in the hash index to the result.

先用哈希函数将两个关系进行分区，根据不同的要求对分组进行操作（∩ ∪ -）