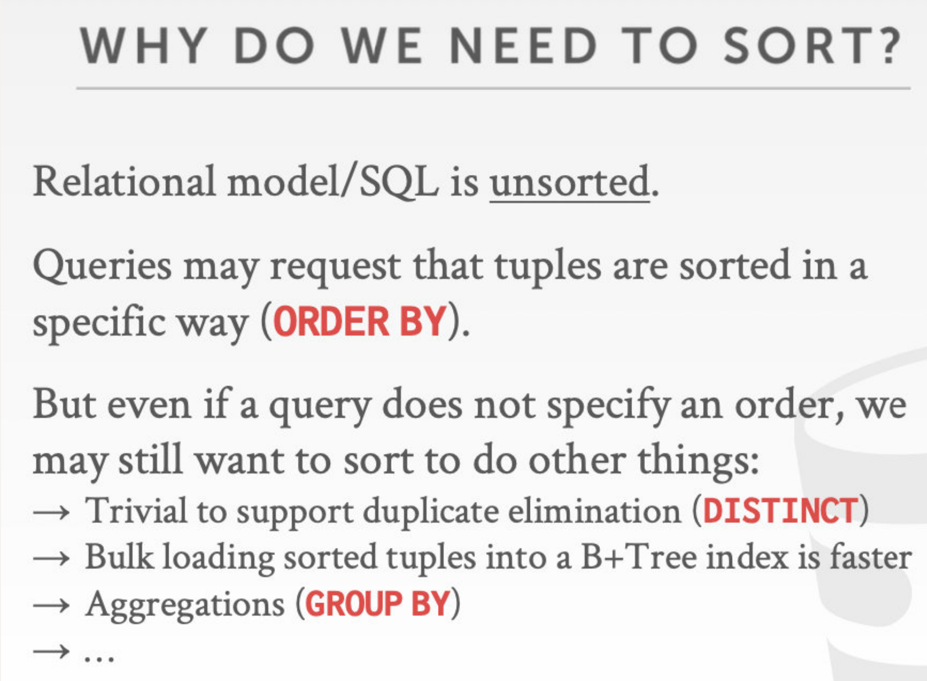
接下来的Lecture将研究算子与计划是如何执行的（Operator Excecution）

External Merge Sort

标准SQL语句的返回的数据是unsorted的，但如果在查询语句中通过一些关键字显示地指明，要求查询返回的数据有序，那么DBMS就要进行排序的工作



如果待排序的数据的量不大，可以全部放在内存当中，那么我们可以使用常见的快速排序，归并排序这些标准的排序算法来解决，但是问题在于，基于硬盘的DBMS所存储的数据量非常大，无法把全部的数据装载到内存中，那就无法直接使用前面的那些排序算法。我们需要多次的硬盘I/O，每次向内存装载部分数据，基于这一特性，我们使用了如下的策略来完成排序

External Merge Sort，外部归并排序

由于我们要排序的数据不在内存中，而是在外部的磁盘设备里，故因此得名，简称外排序

外排序的思路和基于分治算法的归并排序比较像，先把要排序的数据分成一块一块的，然后把每块放到内存里分别排好序，再把这些排好序的小块合并成更大的有序的块

Divide-and-conquer algorithm that splits data into separate **runs**, sorts them individually, and then combines them into longer sorted runs.

Phase #1 – Sorting

→ Sort chunks of data that fit in memory and then write back the sorted chunks to a file on disk.

Phase #2 – Merging

→ Combine sorted runs into larger chunks.

一个sorted run就意味着一次排序，排序后的结果是一列键值对，即KV，我们依据Key来排序，在排序时，相应的V有两种表达方式，分为早物化/晚物化，早物化是说Value是Key相对应的整个tuple，晚物化是说Value是Key相对应的tuple的record id，而不是tuple本身，我们根据这个id就可以在DBMS的主表里找到我们想要的信息。如果tuple很长，表很宽，那么我们在选择早物化时，每次调整表的信息时，开销就很大，晚物化解决了这种问题。

接下来以2阶（国内的教材一般翻译成2路）外排序为例来进行分析

We will start with a simple example of a 2-way external merge sort.

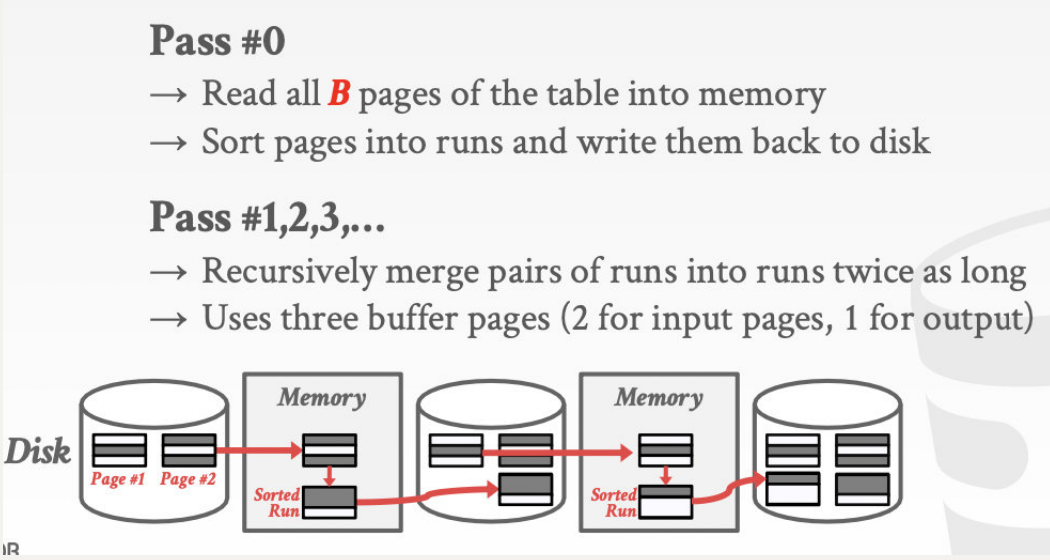
→ "2" is the number of runs that we are going to merge into a new run for each pass.

Data is broken up into ***N*** pages.

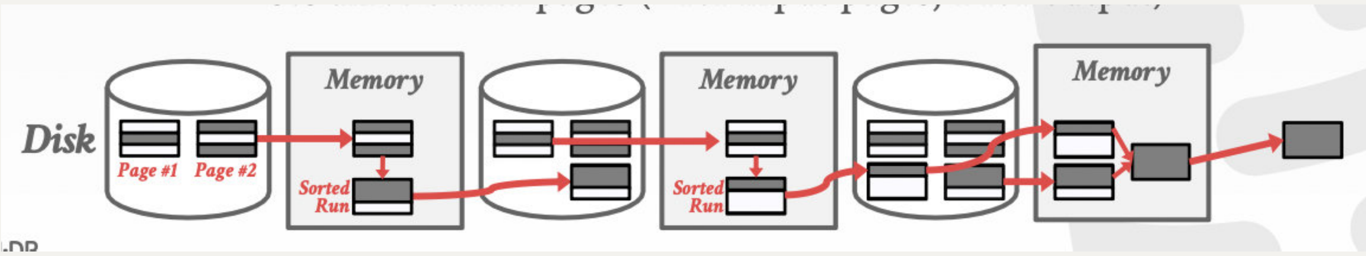
The DBMS has a finite number of ***B*** buffer pool pages to hold input and output data.

假设待排序的数据有N个页大小，缓存池有B个页大小

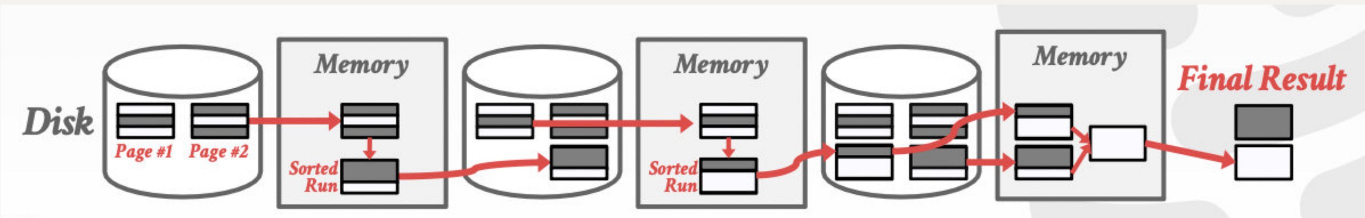
我们先把硬盘上的一个页（下图page 2）读入内存，之后在内存中对其进行排序，得到排完序之后的中间结果后，将其写回硬盘，此时，硬盘里面有原本的page1，page2，以及排完序后的page 2，之后再把page 1读入内存，进行同样的操作，此时，硬盘里有原本的page 1，page 2，排完序后的page 1和page 2



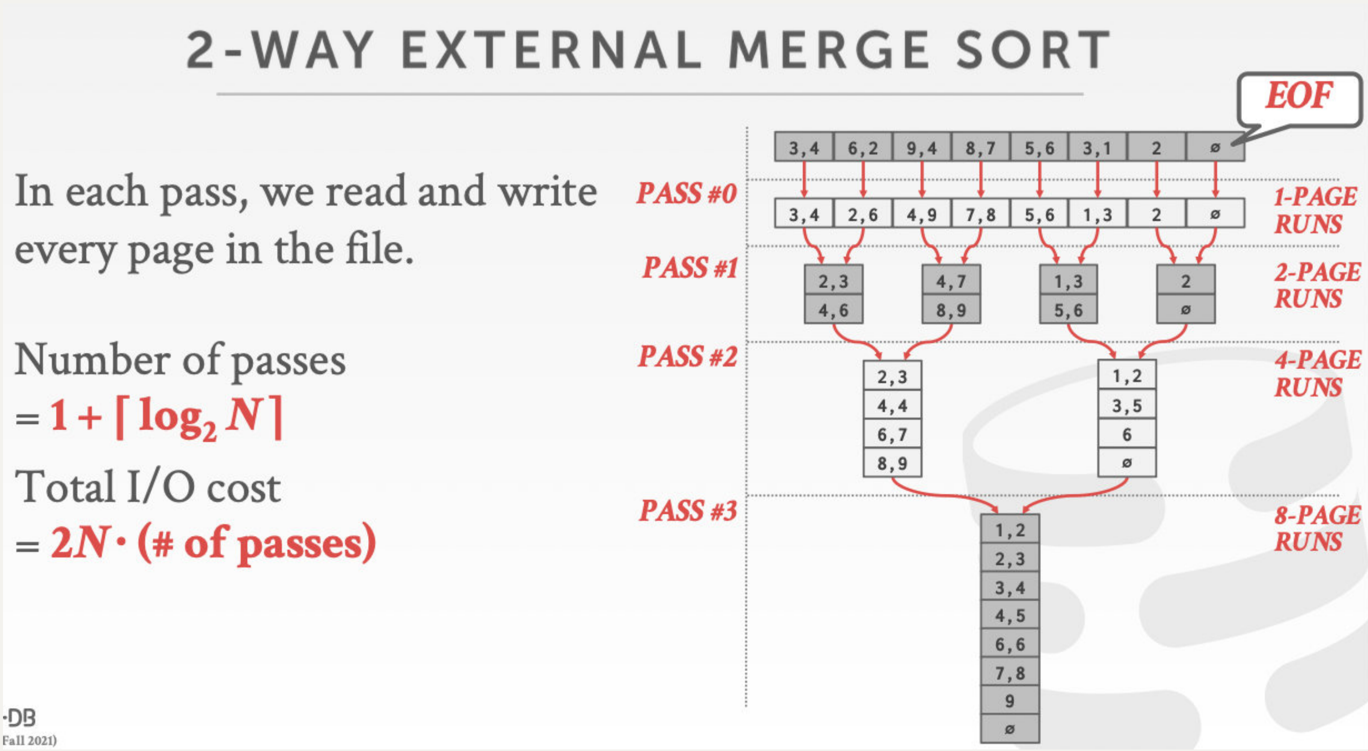
之后在内存中开辟3页大小的内存空间，把硬盘里的排好序的page1和page2都读进内存，然后做merge sort中的merge操作，直到填满内存中排序用的那个页，之后把这个页写回硬盘，之后继续完成前面的merge操作，等到再次填满内存中用于排序的页时，再次将这个页写回硬盘，这便彻底完成了对排好序的page1和page2的merge操作



->next->



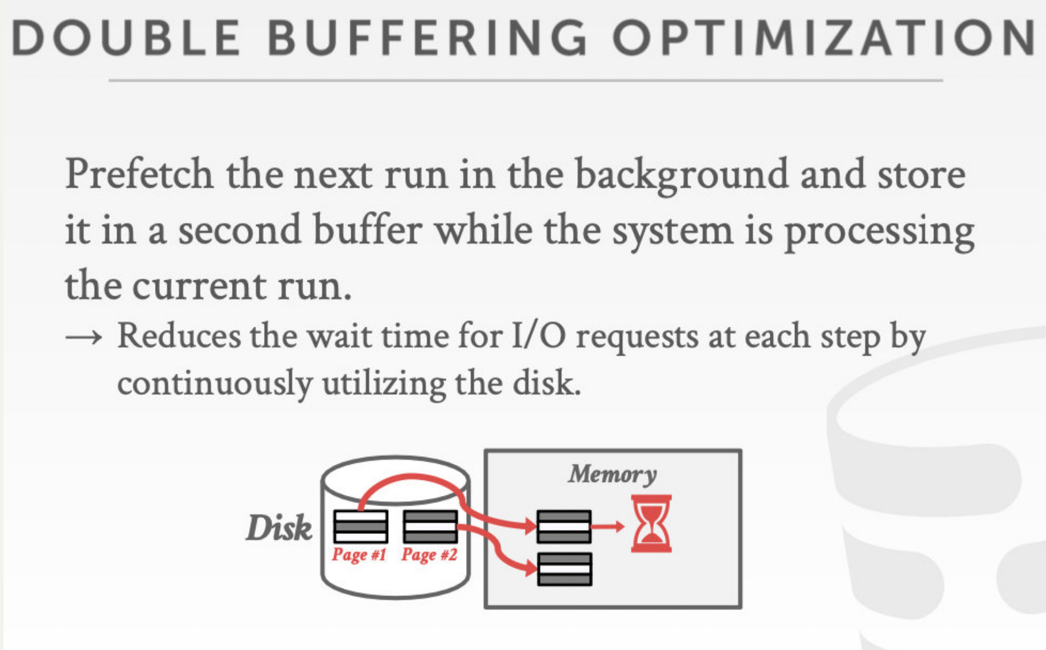
推演到更广泛的情况（不只是像上面的例子只涉及两个page的排序），如下所示：



2阶外排序的情况下，只需要缓存池有3个页大即可，两个用于存储待排序的输入数据，另一个用于存放排序后的中间结果，如果硬件资源充足（计算机的内存足够大），可以在2阶外排序的基础上做一些优化

- Double Buffering Optimization

  对当前的run进行排序时，把接下来要处理的run提前读进来（因为内存充足），这样的话，当前的run排完序后，接下来要处理的run已经被读进来了，无需再等待硬盘的I/O，从而提升了效率，这种优化策略有一点pipeline的思想在里面



- General External Merge Sort

简单的说，就是采用n阶外排序，n=B-1（B是缓存池中所拥有的页数）

GENERAL EXTERNAL MERGE SORT

Pass #0

→ Use ***B*** buffer pages

→ Produce $\lceil N / B \rceil$ sorted runs of size ***B***

Pass #1,2,3,…

→ Merge ***B*-1** runs (i.e., K-way merge)

Number of passes = $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$

Total I/O Cost = $2N \cdot$ **(# of passes)**

EXAMPLE

Determine how many passes it takes to sort 108 pages with 5 buffer pool pages: ***N*=108, *B*=5**

→ **Pass #0:** $\lceil N / B \rceil = \lceil 108 / 5 \rceil$ = 22 sorted runs of 5 pages each (last run is only 3 pages).

→ **Pass #1:** $\lceil N' / B\text{-}1 \rceil = \lceil 22 / 4 \rceil$ = 6 sorted runs of 20 pages each (last run is only 8 pages).

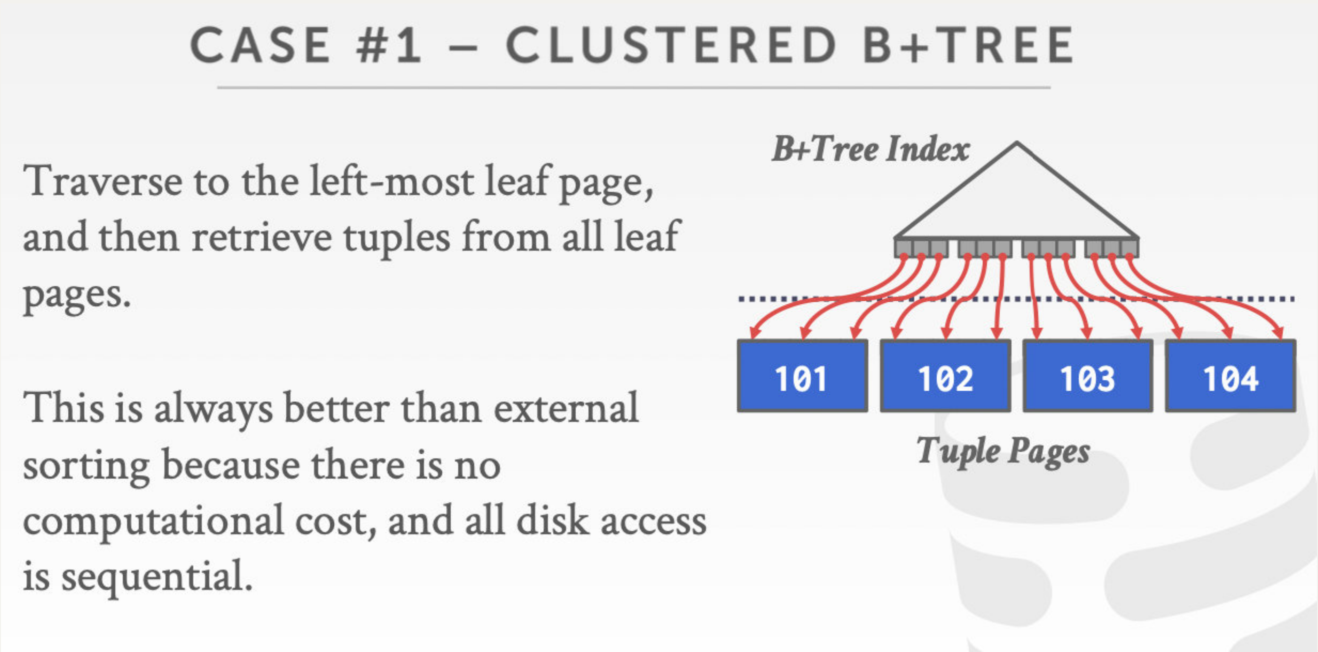
→ **Pass #2:** $\lceil N'' / B\text{-}1 \rceil = \lceil 6 / 4 \rceil$ = 2 sorted runs, first one has 80 pages and second one has 28 pages.

→ **Pass #3:** Sorted file of 108 pages.

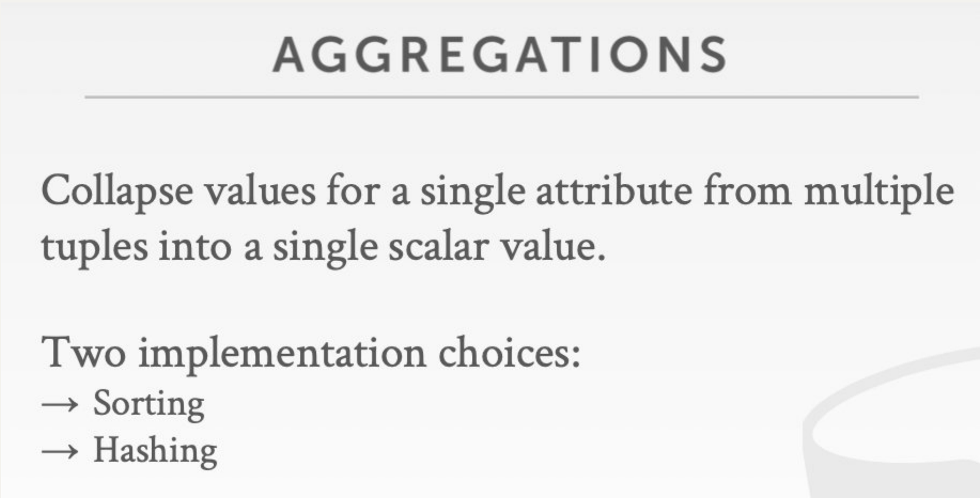
$$1+\lceil \log_{B-1}\lceil N / B \rceil \rceil = 1+\lceil \log_4 22 \rceil = 1+\lceil 2.229\ldots \rceil$$

$$= \textbf{4 passes}$$

除此之外，由于B+树的叶子节点本身就是天然有序的，所以当我们使用B+树来作为我们感兴趣的KV的索引时，就无需排序了，B+树分为聚簇和非聚簇的，聚簇的B+树（前面关于B+树的介绍中的聚簇索引）就和前面所介绍过的早物化的概念差不多



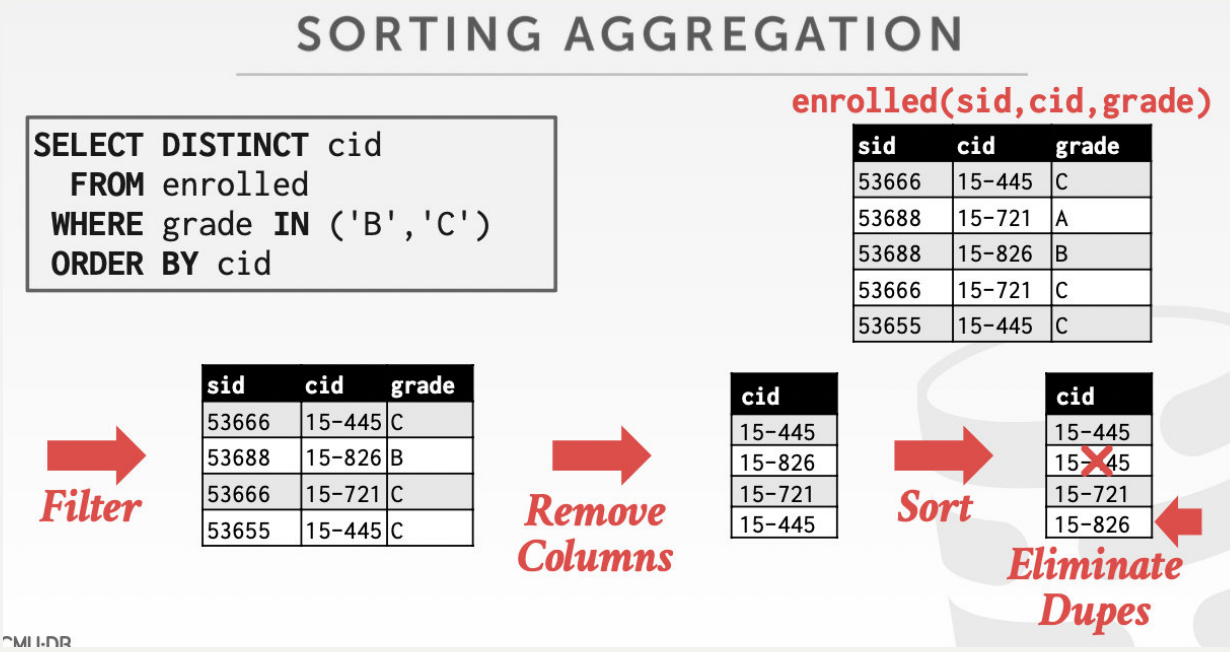
Aggregations



聚集操作有两种实现的方法，一是排序聚集，二是哈希聚集

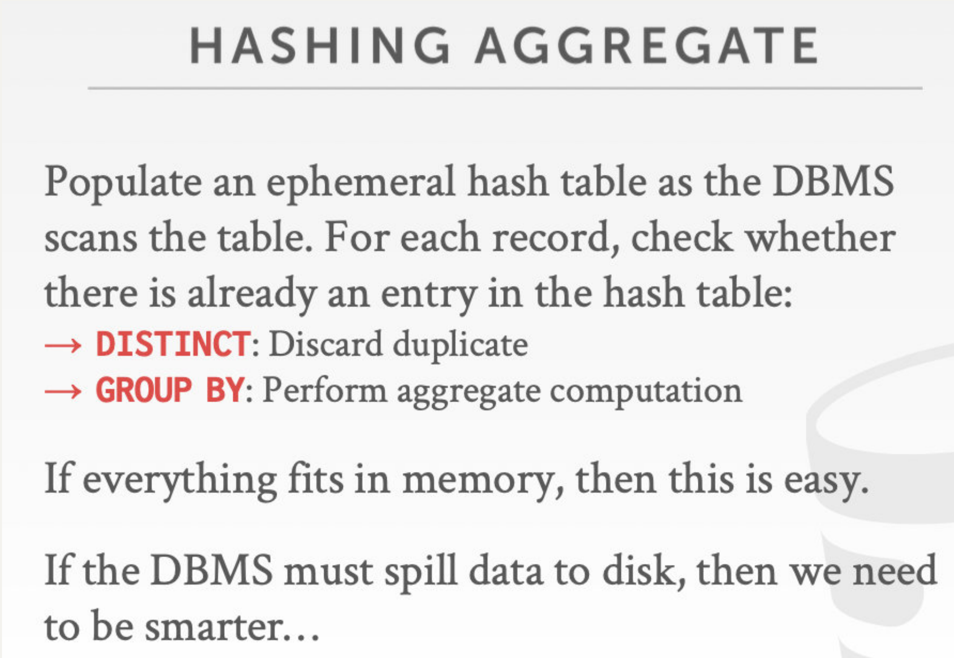
- 排序聚集

排序聚集的场景一般如下所示，由于SQL语句中有order by关键字，所以说最后要进行排序操作：



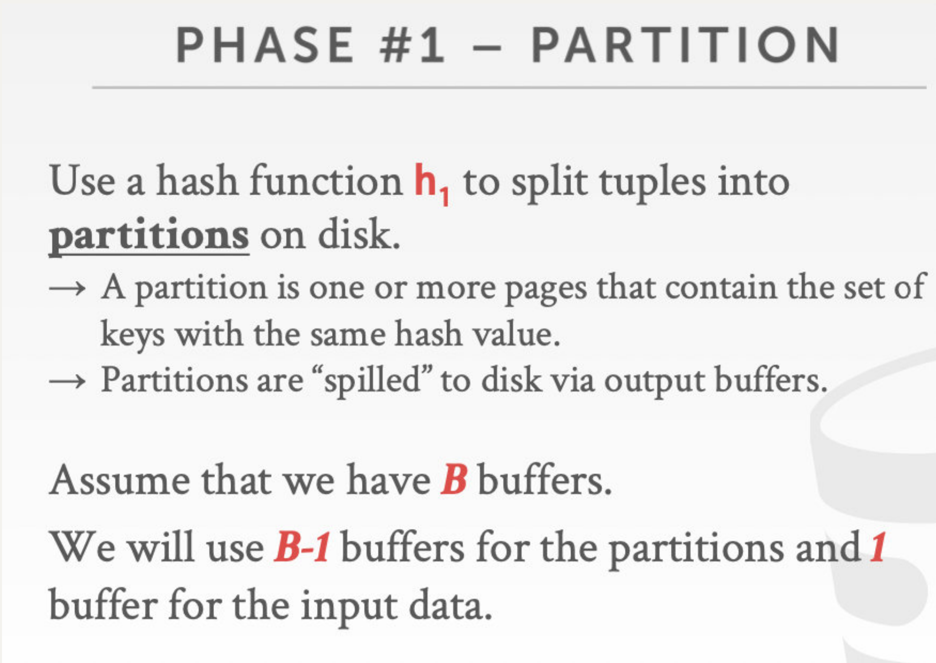
- 哈希聚集

  如果我们仅仅是想实现某类数据的聚集，不需要在此基础上再进行排序（因为排序往往都会有不小的开销），那可以使用哈希聚集



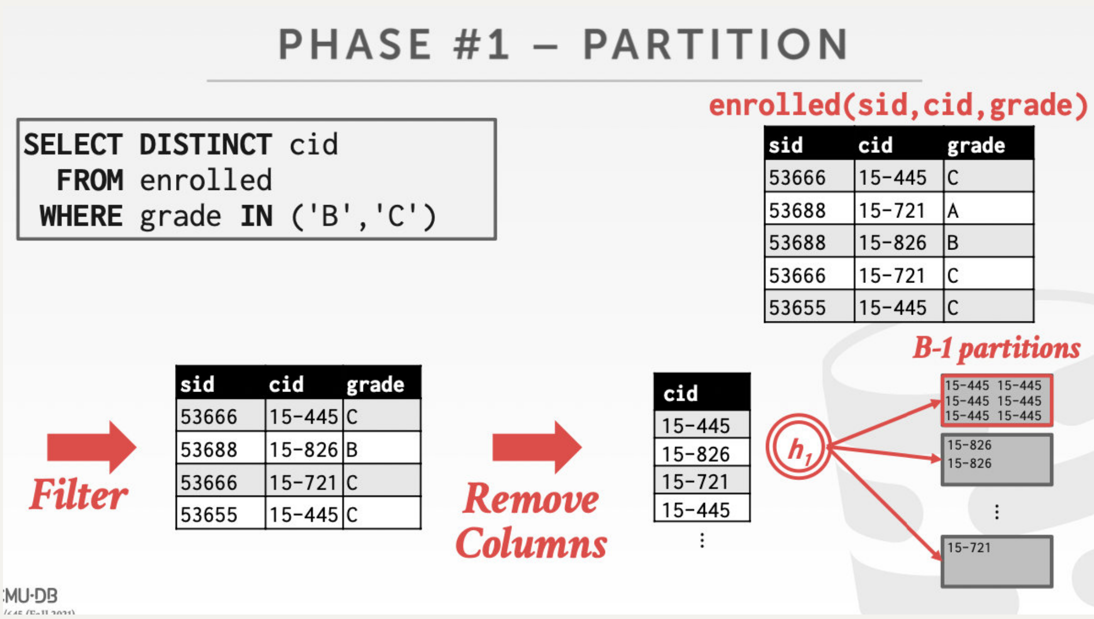
  和外排序一样，我们要处理的数据量比内存容量要大，因此使用外部哈希聚集策略，外部哈希策略有两个阶段，

  - 阶段1 partition

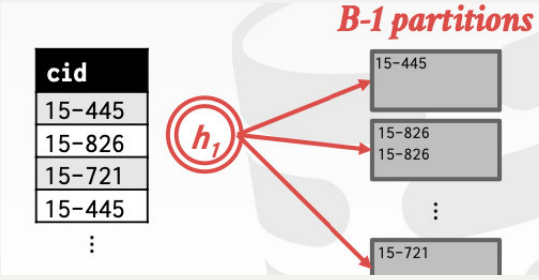


我们先以聚集操作所感兴趣的tuple里的字段为key，做一个哈希表，但是哈希表要分区之后落在硬盘上，因为我们要操纵的数据量比内存容量要大

以如下场景为例，SQL语句中没有order by，因此无需排序，我们只需先完成下图中的过滤，投影（remove columns）操作，然后使用我们感兴趣的cid这一列的数据构建哈希表，相同的值会落在同一个哈希桶里，之后将哈希表以哈希桶为分区落盘，

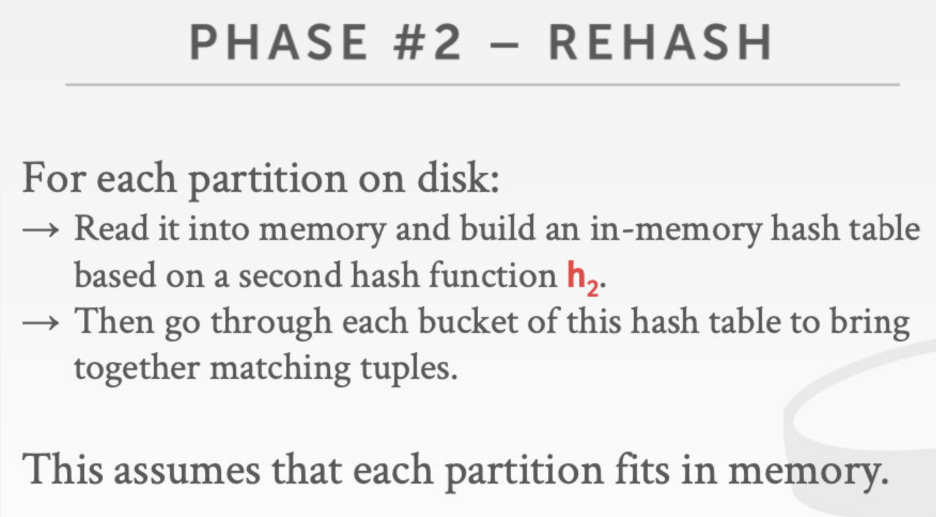


落盘的时候可以做提前优化，因为我们的SQL语句的目的是去重，所以在落盘的时候就可以不把重复的字段写入硬盘，提前完成去重

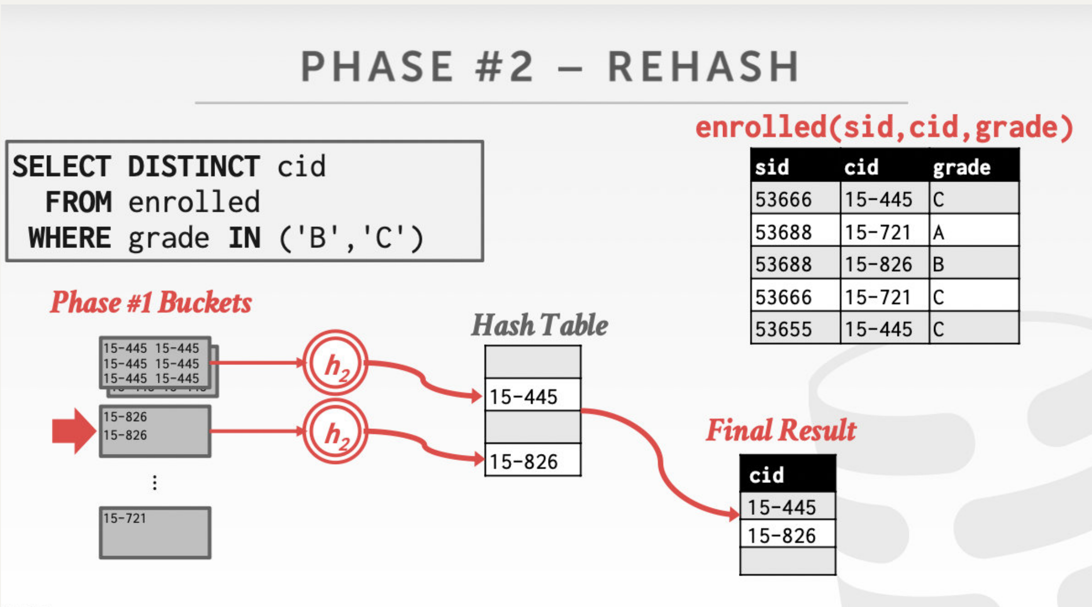


- 阶段2 rehash

  完成阶段1之后，存储在硬盘中的哈希表有可能太大，没做完去重操作，并且同一个哈希桶中可能有哈希碰撞，也就是Key不同，但进了同一个哈希桶，之后我们进行阶段2，rehash



我们把硬盘中一个个哈希桶中的数据以页为单位往内存里读，读进去之后做第二次哈希，第二次哈希就可以彻底去重并且把阶段1中哈希碰撞的值区分出来，我们把第二次哈希的结果放到最终的哈希表里



前面讨论的都是去重，很多聚集操作并不是以去重为终点，而是在去重之后再进行一些计算得出一些额外的统计值，这种情况下，在rehash阶段，还需额外记录一些动态变化的临时结果