Join/连表的本质是把割开的关系重组起来，比如说xxx在一班，一班在3楼，把这两条关系组合起来，就可以得到“xxx在3楼”这个关系

连表时，有个原则，要尽量把小表（所占页数较少的表）放在左侧（此时这个小表也叫驱动表），后面会讲到，这会减少硬盘IO次数

We will focus on performing binary joins (two tables) using **inner equijoin** algorithms.
→ These algorithms can be tweaked to support other joins.
→ Multi-way joins exist primarily in research literature.

In general, we want the smaller table to always be the left table ("outer table") in the query plan.

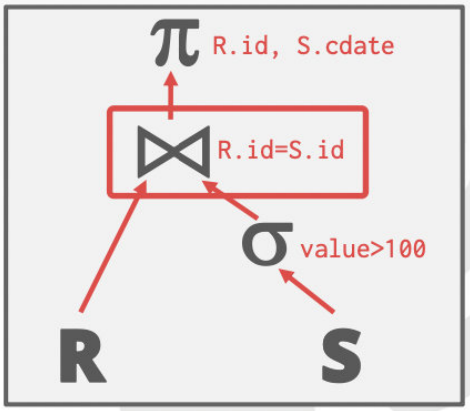
join算子的输出

R表的一条数据和S表的一条数据，它们的连接列match上了（也就是相等），之后这两条数据就可以组合成一条新的数据，输出给join算子的父算子

For tuple **r** ∈ **R** and tuple **s** ∈ **S** that match on join attributes, concatenate **r** and **s** together into a new tuple.

Output contents can vary:
→ Depends on processing model
→ Depends on storage model
→ Depends on data requirements in query

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



U-DB

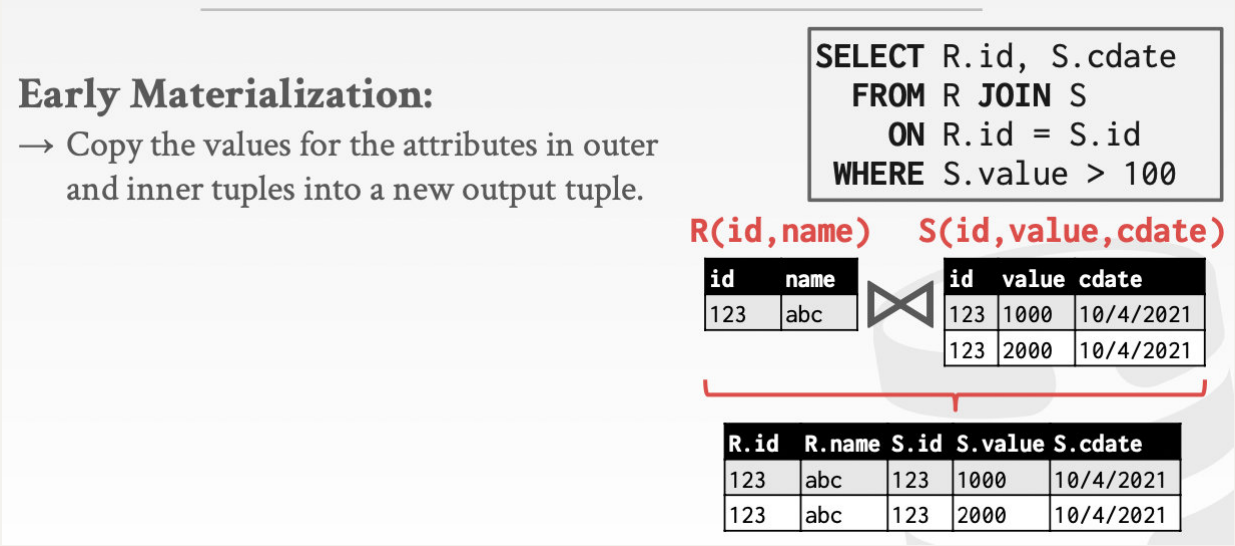
join算子输出的内容由以下三个因素决定：

- SQL处理模型
- 存储模型
- 整个SQL语句所需要的数据

join算子输出的内容有如下几种：

- 直接输出数据

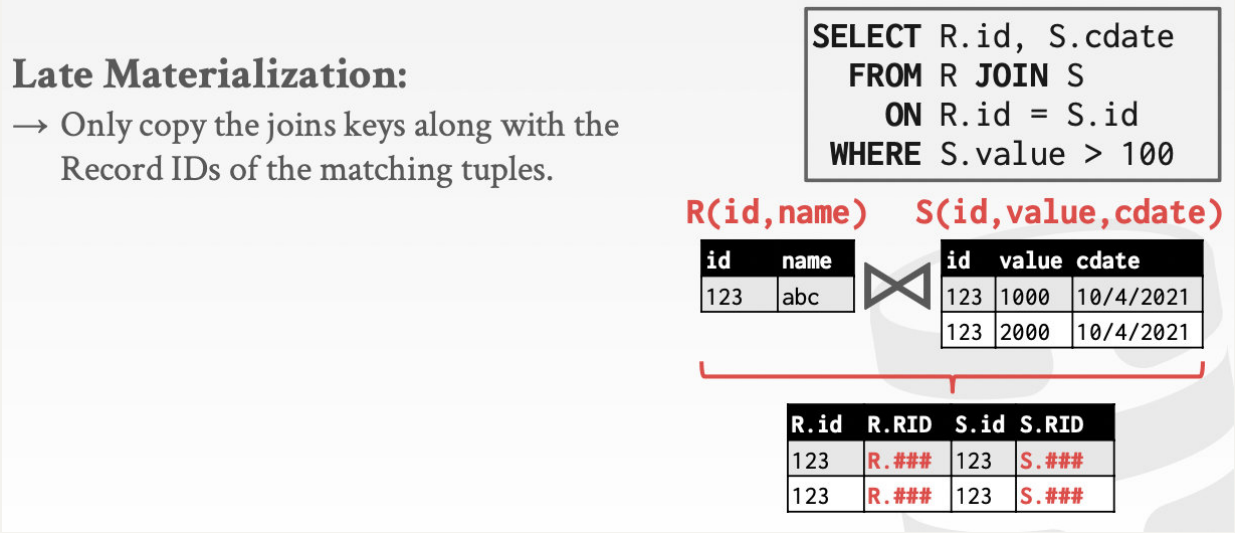
  这种情况属于早物化，被join的都是完整的tuple，因此join操作结束后输出的就是完整的一行数据



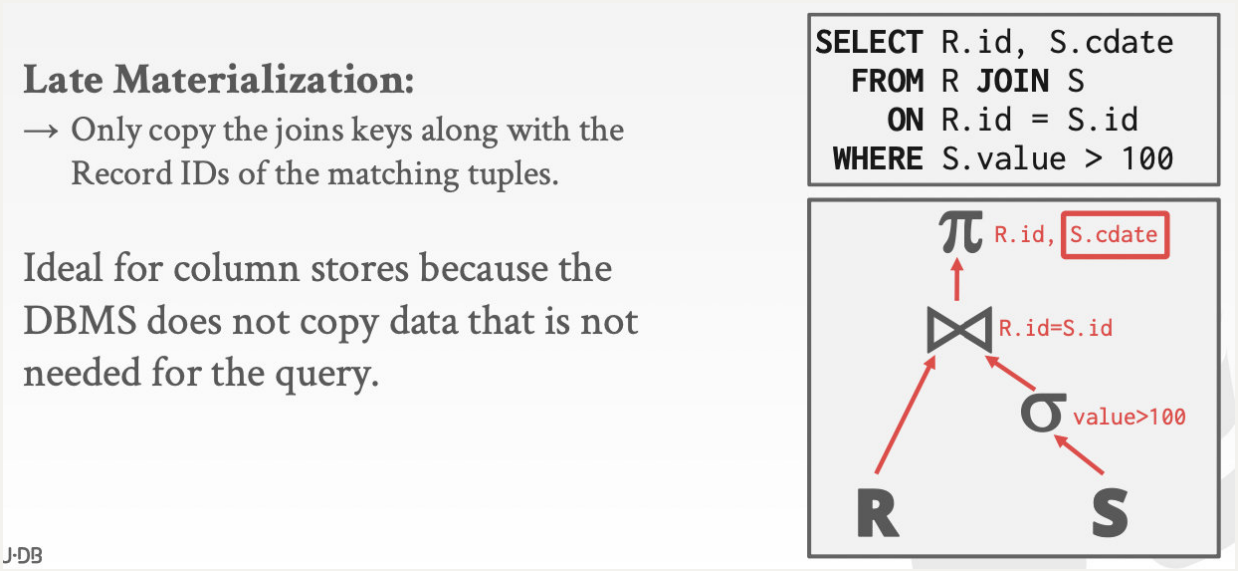
  这样处理的好处在于，join操作输出的结果是完整的，如果join的父算子的输出算子（project），那么就可以直接将感兴趣的字段输出，无需再回到原始的表中找数据（这个操作简称回表）

- 输出record id

  这种属于晚物化，join后得到的一条数据里只含有在相对应的原始表中的record id，而不是全部的字段



上级的父算子如果需要原始的表中的其他字段，就需要回表



这种输出方式符合列存储的思想，DBMS在对join后得到的表执行查询时无需拷贝不需要的数据

join操作的开销

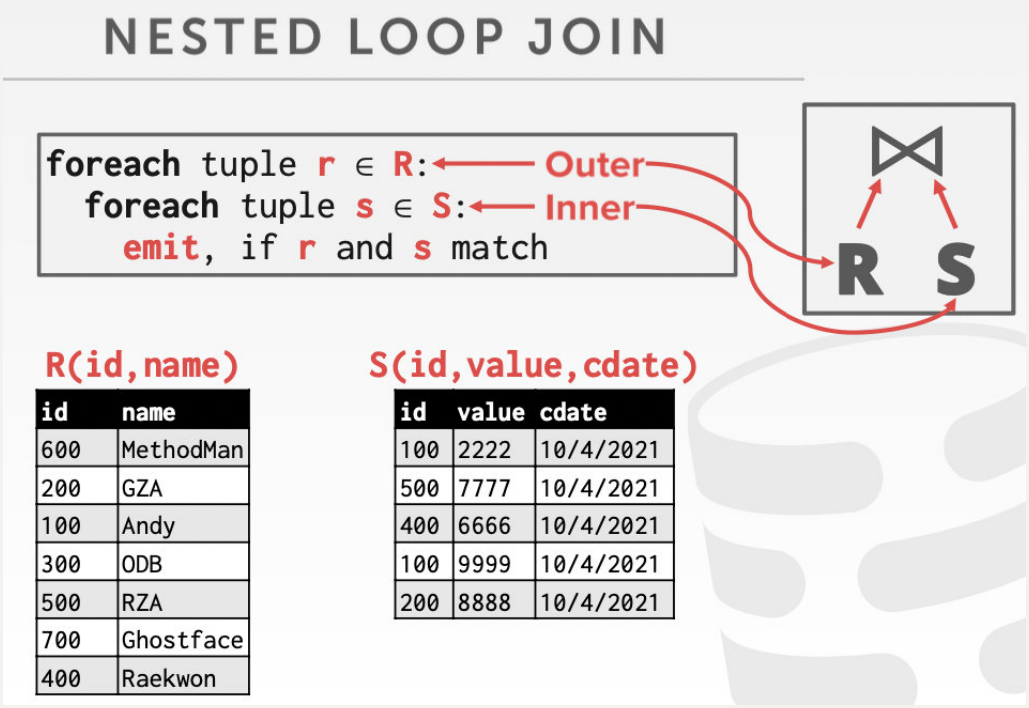
假设R表有M个页大，含有m条数据，S表有N个页大，含有n条数据，我们首先通过分析硬盘I/O次数进而分析join操作的开销

此外，还要注意一个和笛卡尔积相关的问题，join算子在SQL语句中最为常见，也最耗时，最容易出问题，因此对其进行的优化必须格外小心，join操作有时可以通过笛卡尔积来完成，先对两个表进行笛卡尔积，然后用谓词来筛选，但这非常低效，因为笛卡尔积导致的中间结果非常巨大，所以说除了笛卡尔积以外，DBMS的设计者更倾向于采用下面的几种join算法

Nested Loop join

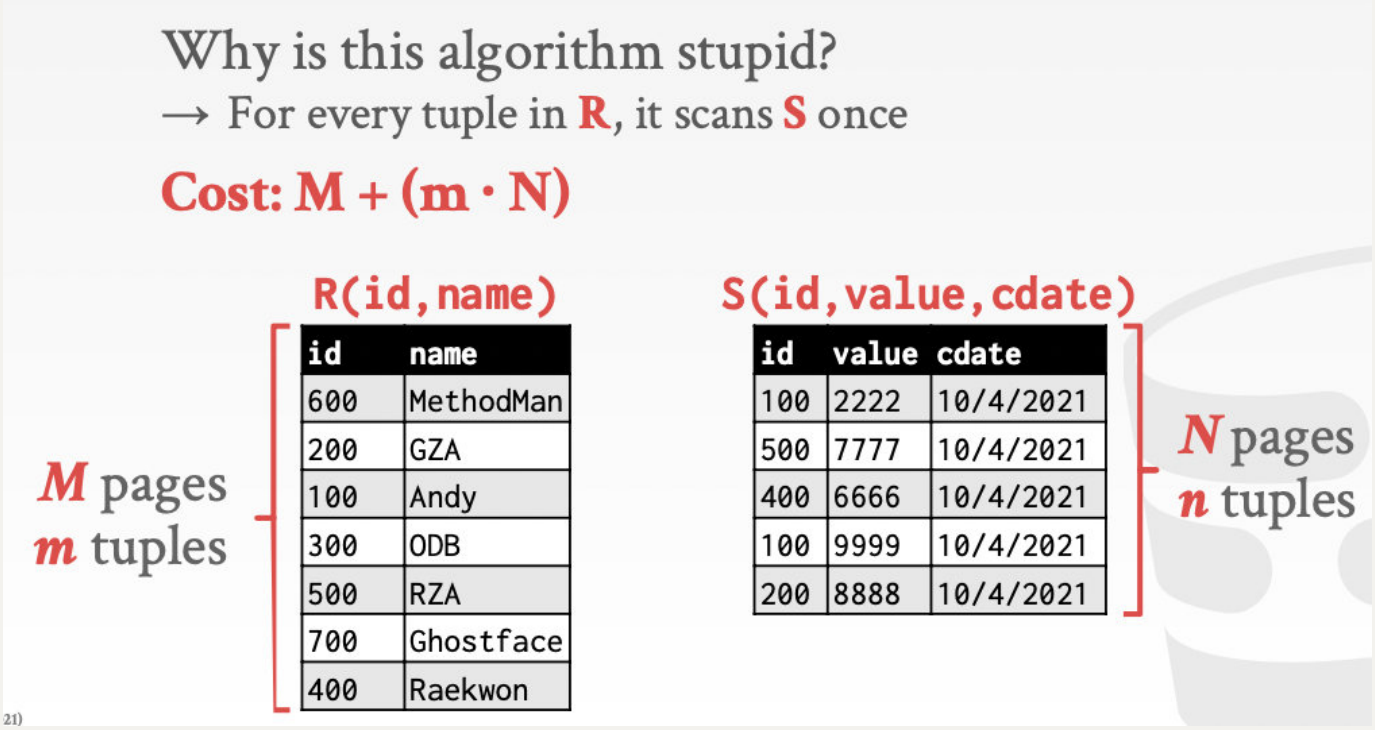
嵌套循环join，伪代码如下，外层循环是遍历R表的所有行，对于R表的每一行，再开一个内层循环，遍历S表的所有行，看r和s能不能连上

其中，因为R表是小表，所以被放在外层循环中，也同时被放在了join算子的左侧，被称为outer table（对于基于硬盘的DBMS来说，所谓的“小表”一般是指所占的文件页少的，R表虽然行数多，但它比较窄，所占用的页数少，那么遍历R表所需的硬盘IO次数少）

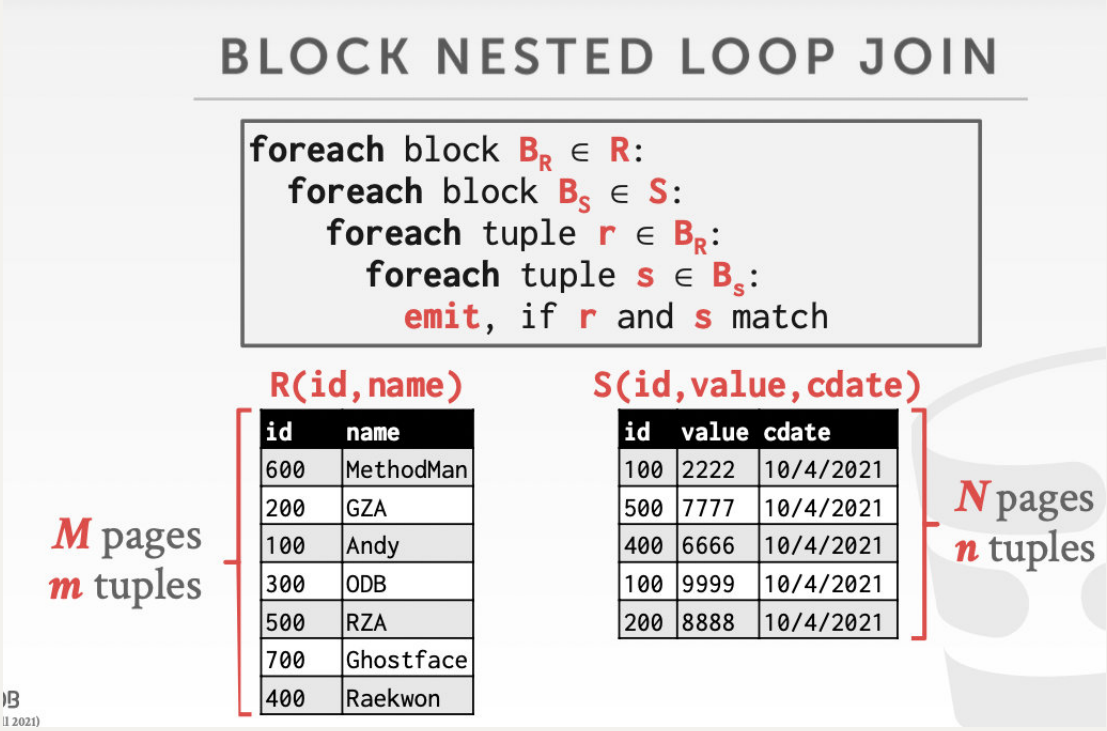


这种策略又名"stupid nested loop join"，因为它十分低效，当扫描S的整张表时，缓存池完全用不上：比如说S有3个页大，缓存池有1个页大，扫描S全表的时候会把缓存池灌满，然后不断地淘汰，然后当外层循环遍历到R表的下一条数据的时候，又要开始扫描S全表，但S表的第一个页早已被踢出缓存池

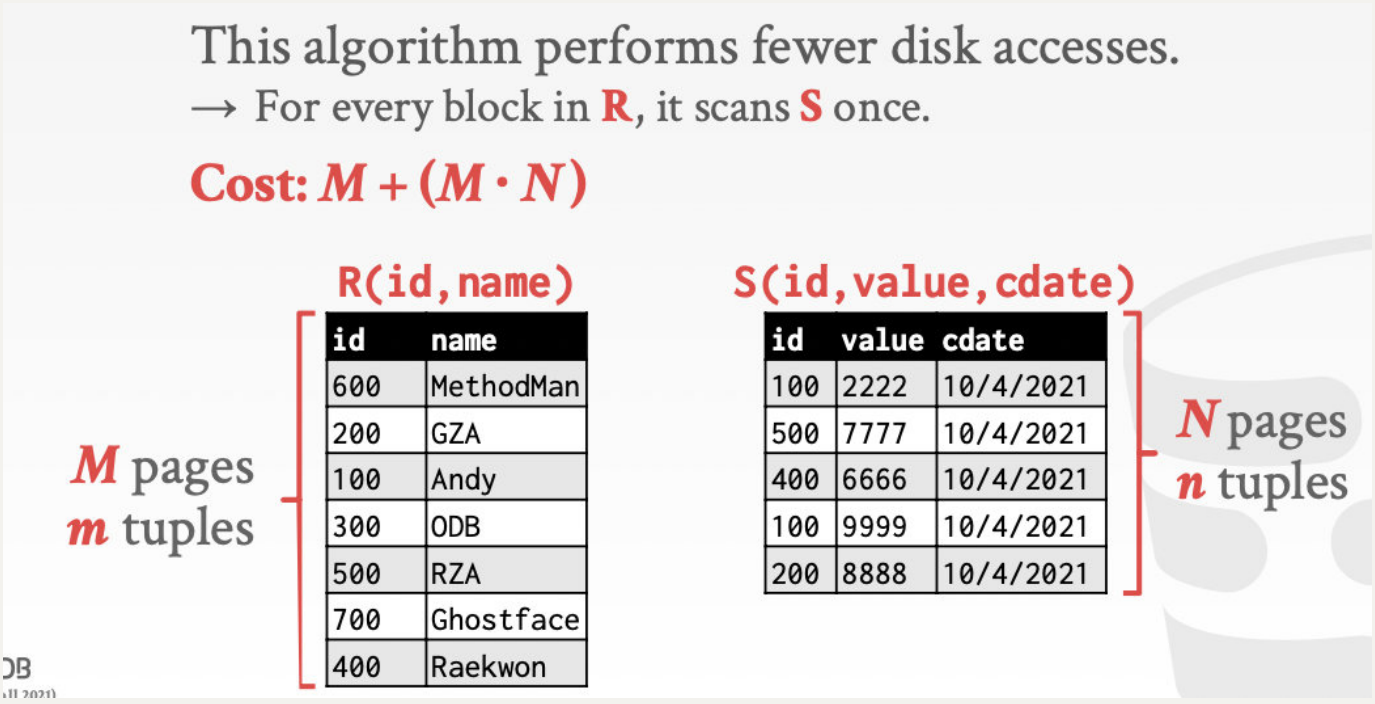
我们从硬盘IO的角度分析这种策略的开销，得到如下结果：



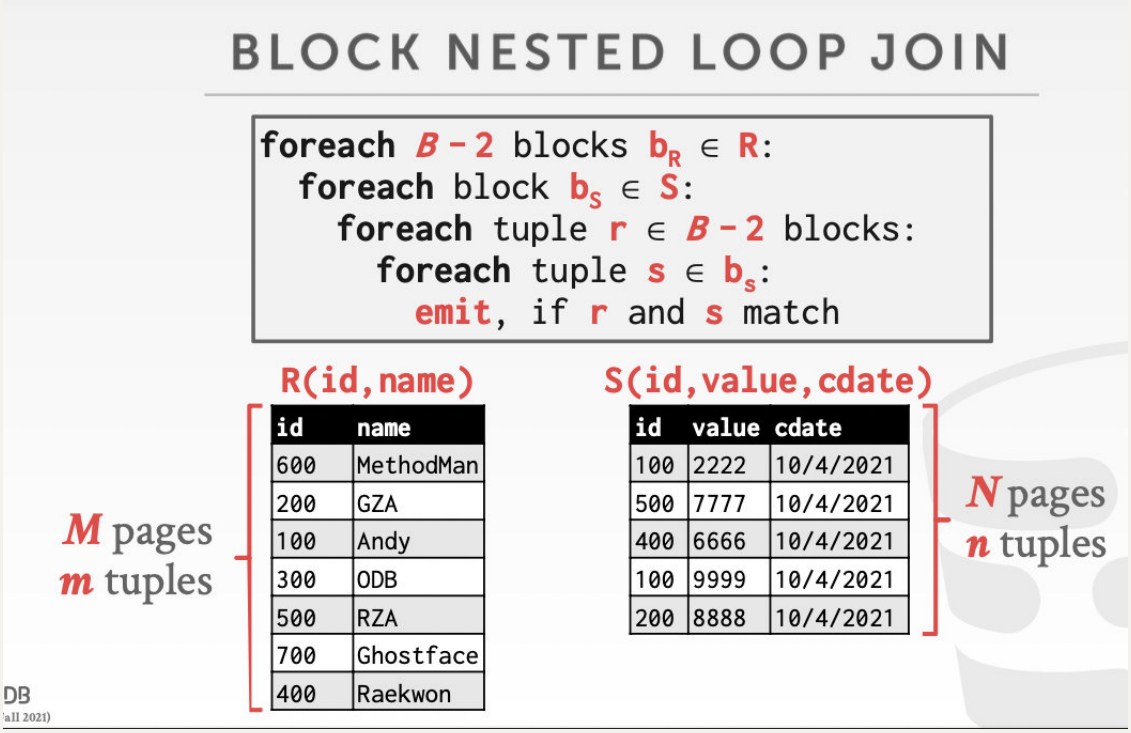
这种最原始的策略无法充分利用缓存池，基于这一点，我们可以对其做出优化，如下所示



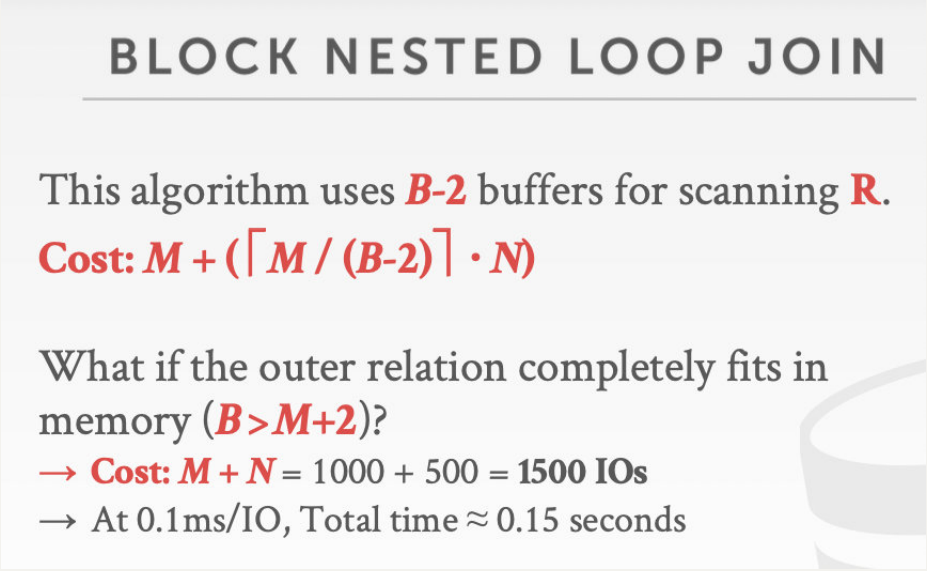
之前是对于R表的每条tuple，都需要完整地遍历一遍S表，现在变成：对于R表的每一页，其中的tuples共享地遍历S表，一边遍历一边检测有没有发生join操作的可能，这样的话，遍历S表的次数就由上图中的m减少到了M，这种优化使得性能大大地提升



刚才的策略是在缓存池当中拿出一个页放R表，一个页放S表，一个页做join的输出缓存，如果可供join使用的缓存池很大，有B个页，我们拿出来1个页用作输出缓存，剩下的B-1个页里拿出B-2个页缓存outer table（也就是R表），1个页缓存inner table（也就是S表），之后执行如下描述的算法：



该算法的开销如下：



可以看到，这个算法的思想就是把缓存池尽可能多地给outer table使用，从而减少遍历inner table表的次数，从而减少开销，毕竟给inner table用的缓存池再大，在遍历inner table时，都会面临缓存重刷的问题

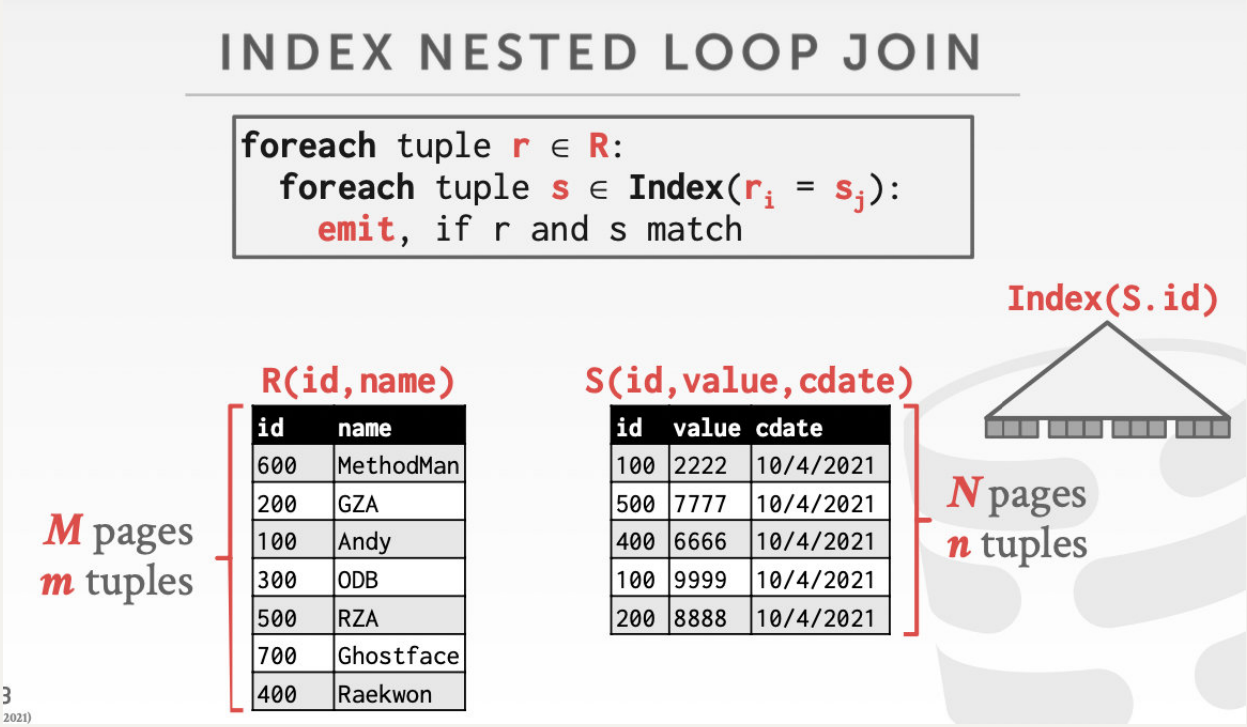
在前面介绍的各种嵌套循环join当中，无论怎么优化都会不止一次遍历右侧的表，这本质上是因为我们没有构建相应的索引，只能通过暴力地遍历去探测有没有可以join的tuple。因此就有了如下的优化方式：我们以inner table的参与join的那一列字段为key构建索引，这称为index nested loop join或lookup join

Why is the basic nested loop join so bad?

→ For each tuple in the outer table, we must do a sequential scan to check for a match in the inner table.

We can avoid sequential scans by using an index to find inner table matches.

→ Use an existing index for the join.

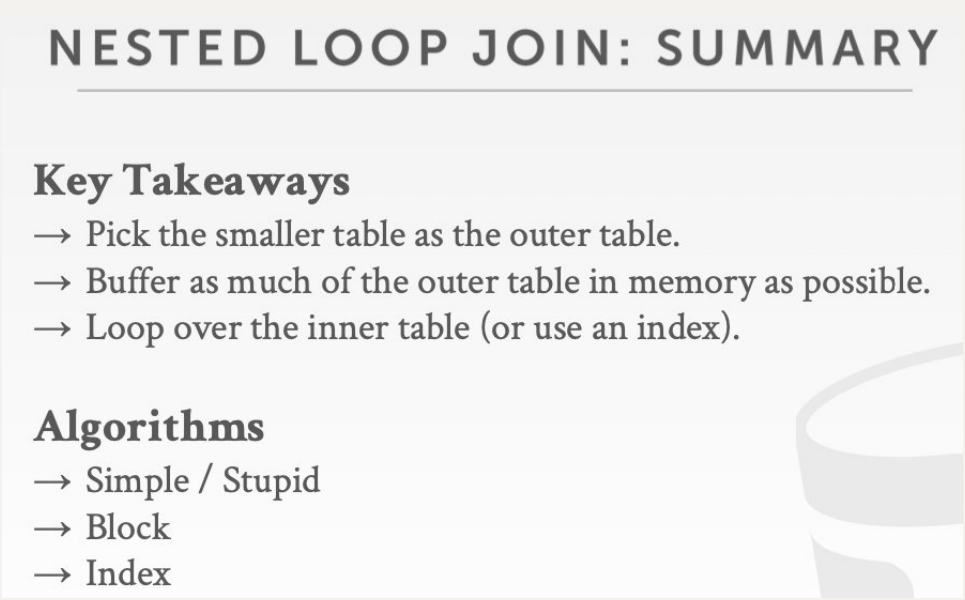


Lookup join的开销如下：

Assume the cost of each index probe is some constant C per tuple.

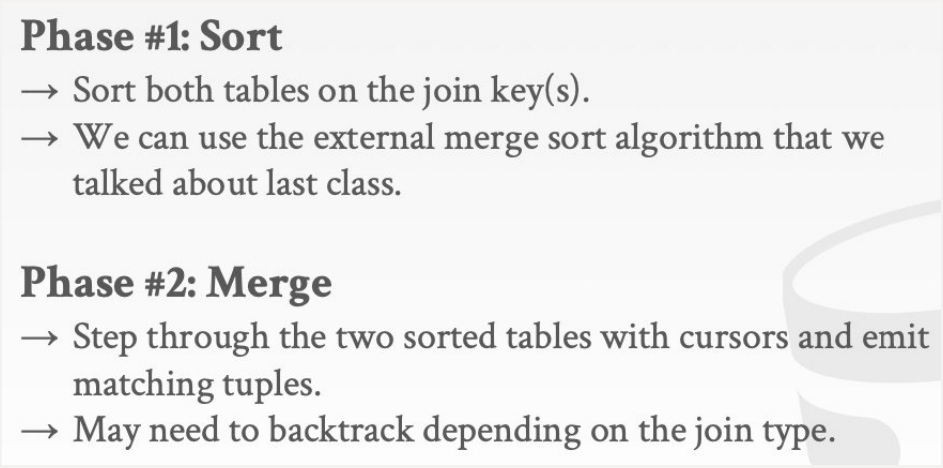
Cost: $M + (m \cdot C)$

总计一下嵌套循环join的优化策略如下：



Sort-Merge Join

又名merge join，它分为如下两个阶段



先给参与join的两个表按照连接列的字段进行排序，然后对这两个排好序的表进行merge，伪代码如下：

```
sort R,S on join keys
cursor_R ← R_sorted, cursor_S ← S_sorted
while cursor_R and cursor_S:
  if cursor_R > cursor_S:
    increment cursor_S
  if cursor_R < cursor_S:
    increment cursor_R
  elif cursor_R and cursor_S match:
    emit
    increment cursor_S
```

merge阶段维护两个指针，它们先分别指向R表和S表的第一行数据，然后比较这两个指针所指向的两行数据的连接列字段的大小，之后如上图所示走向不同的分支，但是这段伪代码其实少描述了一种情况：在某些时刻指针会有回溯操作（相应的场景在slides中有提及），DBMS的具体实现中是包含这一点的

这种策略的开销如下（还是从硬盘IO的角度来分析）：

sort的开销是外排序算法的开销，merge阶段的开销就是把两个表都遍历一遍的开销，总的开销就是它们加起来

Sort Cost (R): $2M \cdot (1 + \lceil \log_{B-1} \lceil M / B \rceil \rceil)$

Sort Cost (S): $2N \cdot (1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil)$

Merge Cost: $(M + N)$

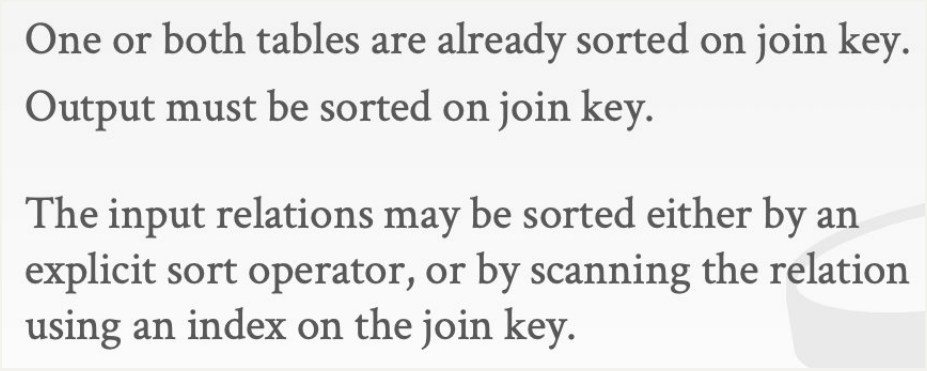
Total Cost: Sort + Merge

但是这种策略存在退化的问题，和指针回溯的操作有关，极端情况就是要join的两列里所有字段的值都相等，这会退化成最原始的Nested Loop join

The worst case for the merging phase is when the join attribute of all the tuples in both relations contains the same value.

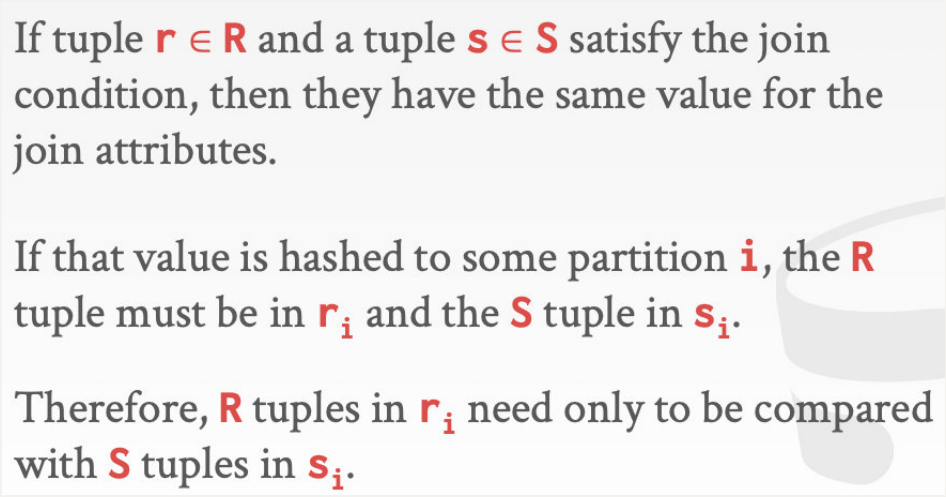
Cost: (M · N) + (sort cost)

在要参与join运算的表都是已经排好序的情况下（或者是通过join key的索引来扫描表），merge join的效率是最高的，开销最低，硬盘IO次数只是两个表的页数之和，此外，如果我们期望join的结果是排好序的，那么merge join也非常合适，因为在这个算法内部实现里面已经完成了排序，在这种场景下使用别的join算法都还需要额外再执行一遍外排序



Hash Join

在前面介绍index nested loop join时，是以B+树为索引来举例的，B+树点查询的时间复杂度是O(log N)，随着其中存储的KV增多，点查询速度会变慢，虽然B+树的好处是支持高效的区间查找，可以按照K递增/递减的顺序遍历叶子节点中的KV，但在index nested loop join场景下，我们要进行的是一次次的点查询，前后两次查询中的key大概率毫无关系，所以说我们也用不到B+树的这个特性。与B+树索引相对的是哈希索引，不管哈希表里存储了多少KV，哈希索引的开销始终都是常熟量级O(1)，点查询执行的飞快，这非常符合我们的期望，因此不妨将B+树索引替换成哈希索引



Hash Join的策略是给outer table构建哈希索引，对inner table进行遍历

原始的hash join算法分为两个阶段

Phase #1: Build

→ Scan the outer relation and populate a hash table using the hash function h_1 on the join attributes.

Phase #2: Probe

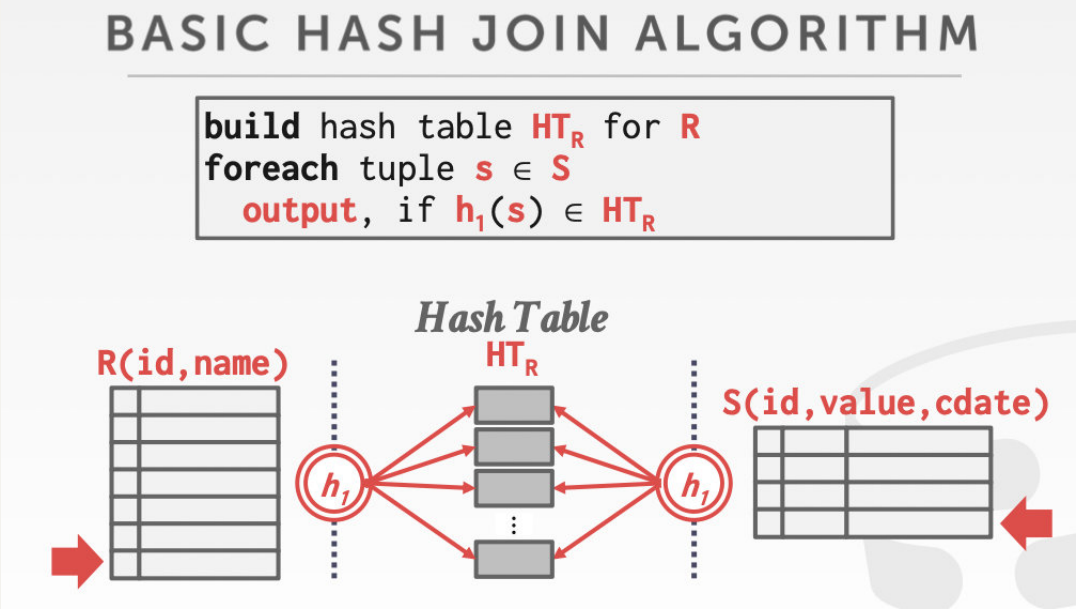
→ Scan the inner relation and use h_1 on each tuple to jump to a location in the hash table and find a matching tuple.

- 阶段1 Build，构建哈希表

  扫描outer table，使用哈希函数h1构建哈希表，以要join的字段为Key

- 阶段2 Probe，点查询

  去inner table以哈希函数h1进行查询，如下所示，以S表的每一行中要 join的字段为key去阶段1中的哈希表里查询，如果在哈希表里找到了能match的KV，那就可以完成相应的join操作



哈希表中每个KV中的Key是要join的连接列的字段，Value的选择也存在早物化/晚物化的差别：

Approach #1: Full Tuple

→ Avoid having to retrieve the outer relation's tuple contents on a match.
→ Takes up more space in memory.

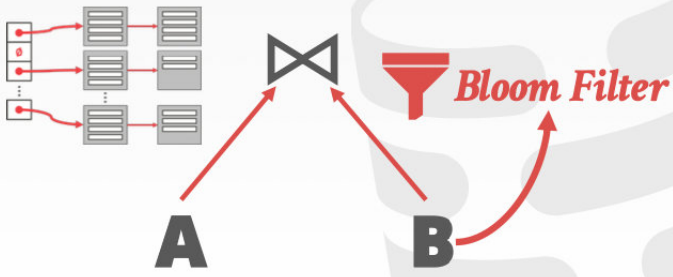
Approach #2: Tuple Identifier

→ Could be to either the base tables or the intermediate output from child operators in the query plan.
→ Ideal for column stores because the DBMS does not fetch data from disk that it does not need.
→ Also better if join selectivity is low.

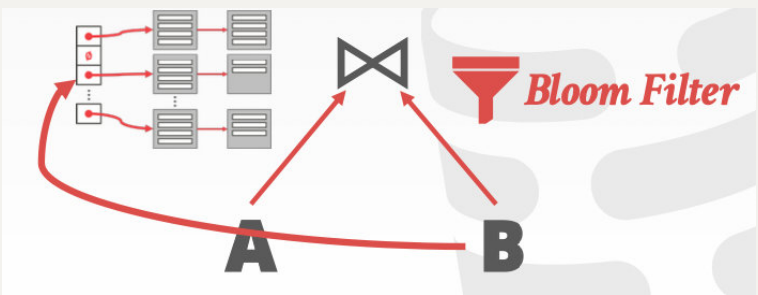
原始的哈希join中，每次都使用inner table中一行的join key去哈希表里查询，但使用有些join key去查询的时候，根本没有对应的哈希表项，这种无效的查询增大了开销，我们不妨使用布隆过滤器，给outer table构建哈希表的时候顺便构建一个布隆过滤器，inner table在去哈希表中查询前，先去查布隆过滤器，判断本次查询在哈希表中能否找到相应的表项，如果能通过布隆过滤器断定哈希表里没有对应的表项，便可以确定这是一次无效的查询，于是让此次查询提前结束

Create a Bloom Filter during the build phase when the key is likely to not exist in the hash table.

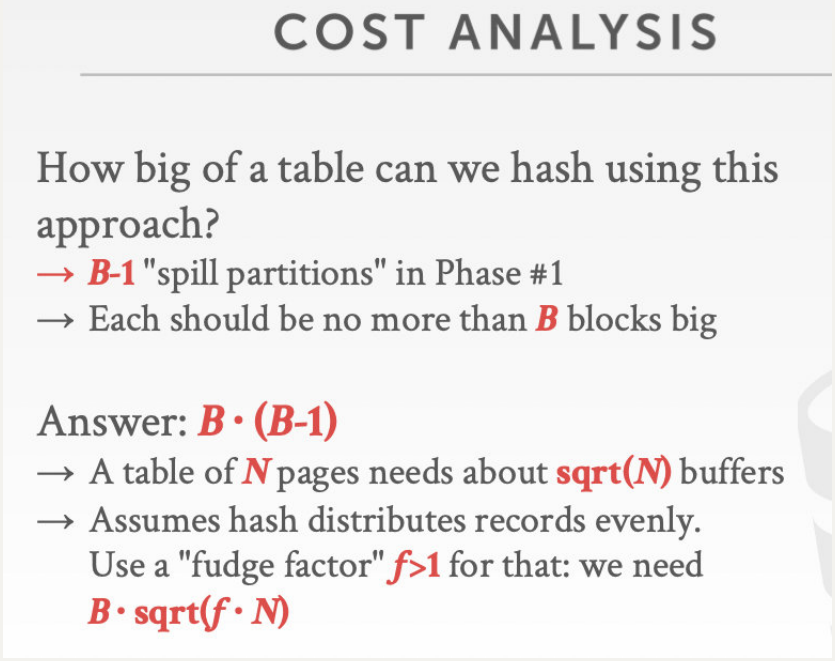
→ Threads check the filter before probing the hash table. This will be faster since the filter will fit in CPU caches.
→ Sometimes called ***sideways information passing.***



->next->



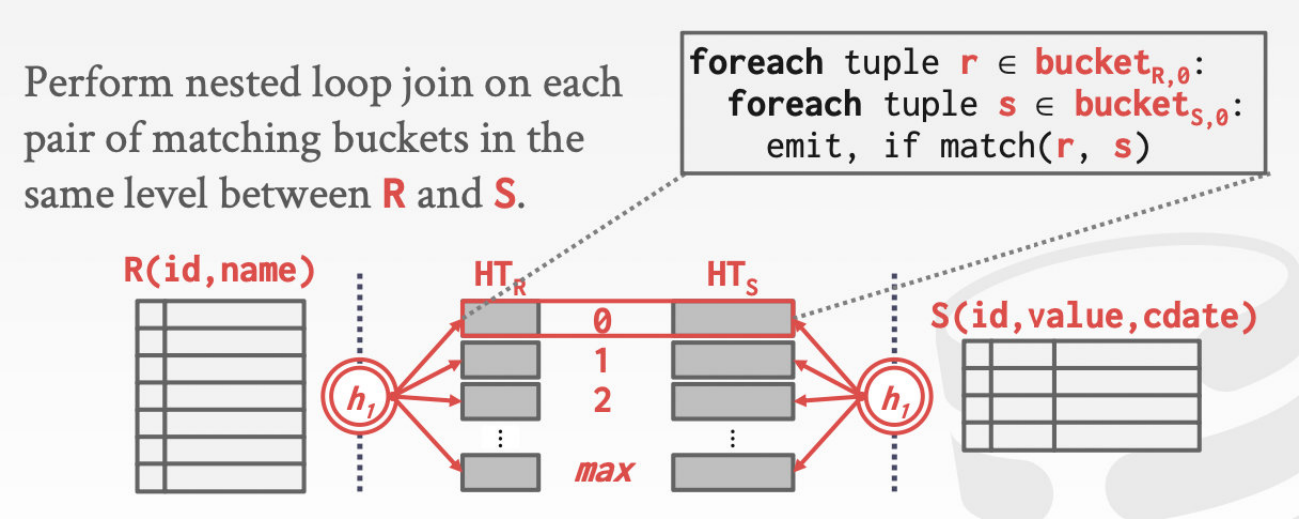
这种策略的开销如下



如果给outer table构建的哈希表太大了（因为outer table太大），内存里放不下，那么我们就要把哈希表的部分内容驱逐到硬盘里，因此就有了相应的驱逐策略：

Grace Hash Join

做两套哈希表，也就是对前面的例子中的S表和R表都构建相应的哈希索引，并且使用相同的哈希函数。我们把哈希表存在硬盘里，查询索引时，把硬盘中两个表相对应的哈希桶都取出，因为我们使用的是相同的哈希函数，所以相同的（也就是可以match的）join key所在的哈希桶号是一样的，之后我们对刚刚从硬盘里取出的两个哈希桶里的KV数据做nested loop join



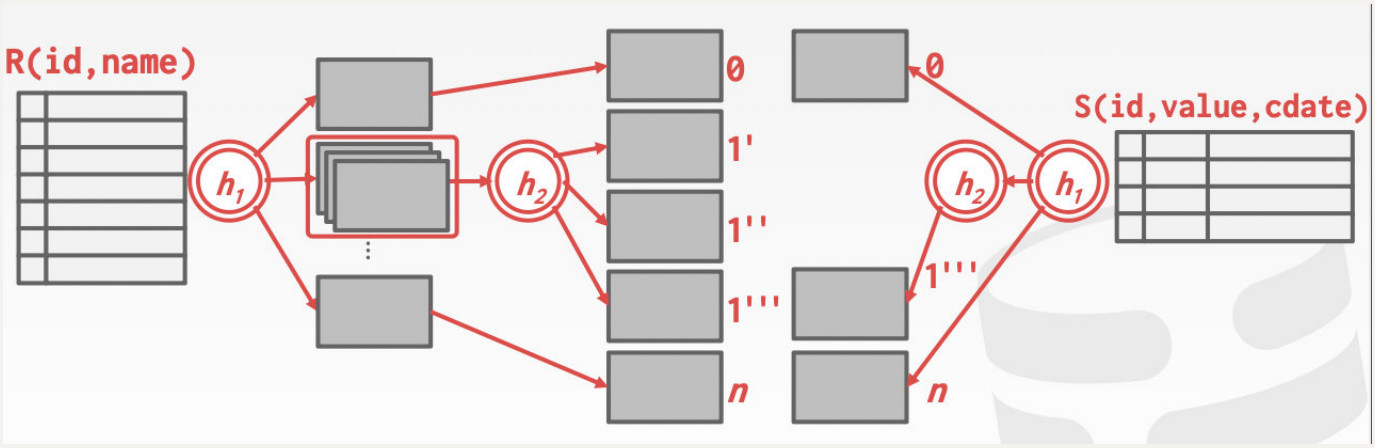
这个策略提出的基础是认为：虽然无法把整个哈希表放进内存里，但可以把哈希表的某个哈希桶放入内存，如果单个的哈希桶太大，也放不进内存，那该怎么办呢？如果使用哈希函数h1构建的哈希表里的哈希桶太大，那我们就把这个大的哈希桶存储硬盘，使用哈希函数h2，对这个哈希表再进行一次哈希操作来进行分区，递归地进行这个过程，直到切成足够小的块

If the buckets do not fit in memory, then use **recursive partitioning** to split the tables into chunks that will fit.

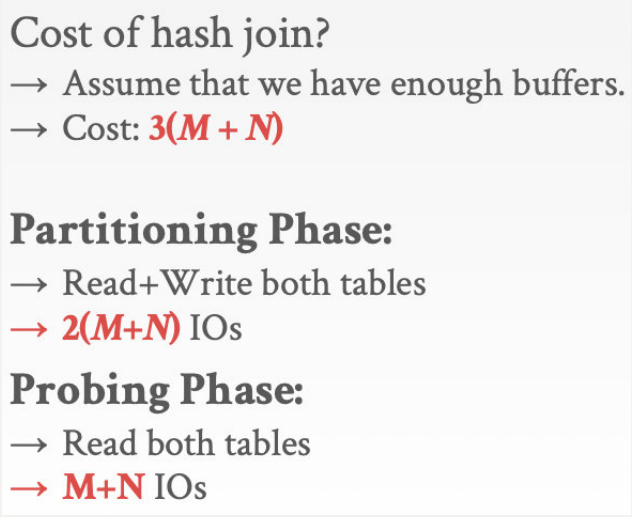
→ Build another hash table for $bucket_{R,i}$ using hash function h_2 (with $h_2 \neq h_1$).

→ Then probe it for each tuple of the other table's bucket at that level.

结合实际场景来理解，如下所示，最终我们会依次把下图的0号页，1'号页，1''号页，1'''号页,...,n号页读入内存，和S表的哈希桶的页来进行匹配操作。还要特别注意一下，R表原来的1号哈希桶因为太大被再次哈希拆分成了三个小的哈希桶，在执行匹配操作时，S表的1号哈希桶也要再次以h2函数进行一遍哈希操作，拆分成更小的块，然后依次执行匹配操作



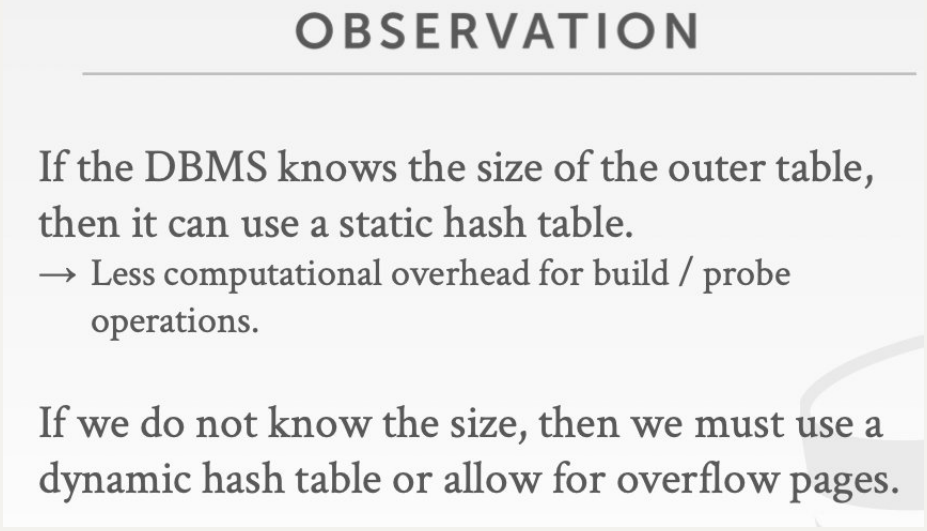
grace hash join的开销如下（在输出join结果之前的开销）：



- 构建哈希表阶段的开销是2(M+N)次硬盘IO，其中有(M+N)次IO是把R表和S表读进内存，因为哈希表的空间复杂度是O(N)，所以说给R表和S表构建的哈希表大约有(M+N)个页大，把这两个哈希表写入硬盘的IO开销便是(M+N)次IO，因此建立哈希表阶段总的开销是2(M+N)次硬盘IO
- 匹配阶段的开销是(M+N)次IO，因为我们只需把两个哈希表各自的哈希桶都读进内存

Summary

如果DBMS知道outer table的大小，就可以为它构建静态的哈希表（也就是简单的，不可扩容的），否则就必须构建可扩容的哈希表



最终总结一下各种join算法的开销

Algorithm	IO Cost	Example
Simple Nested Loop Join	$M + (m \cdot N)$	1.3 hours
Block Nested Loop Join	$M + (M \cdot N)$	50 seconds
Index Nested Loop Join	$M + (M \cdot C)$	Variable
Sort-Merge Join	$M + N +$ (sort cost)	0.75 seconds
Hash Join	$3(M + N)$	0.45 seconds

如果是面对OLAP型的负载，join两个大表，那么大多数情况下哈希join最为合适，但也有一些特例，如果数据是“倾斜的”，也就是说为其构建哈希表会导致严重的哈希碰撞，sort-merge join效率好的多，此外，如果要求join的输出结果必须有序，sort-merge join也是最优选择，DBMS的优化器会结合实际场景在sort-merge join和hash join之间做选择