DBMS Tutorial 06.02.2019

Topics

Joins : Iteration, Hash and Merge

Join Algorithms:

Iteration Join:
Explanation: Nested loop join iterates over each tuple in one relation and scans the entire other relation to find matching tuples based on a common attribute.
Example: Joining students and courses based on student ID.

Hash Join:
Explanation: Hash join partitions tuples from both relations into buckets based on a hash function applied to a common attribute and then joins tuples with matching hash values.
Example: Joining employees and departments based on department ID.

Merge Join:
Explanation: Merge join requires both input relations to be sorted on the join attribute and then iteratively merges the sorted lists to find matching tuples.
Example: Joining sales and products based on product ID.

These algorithms are used to efficiently combine data from multiple relations in a database.

Factors affecting performance

What is cost of operations such as selection, joins, etc?

Before any query can be executed, the data needs to be brought to main memory so that decisions can be made (which tuples satisfy selection conditions) or new resultant relations be created (such as joins or projections).

Bringing data file ie relation to main-memory (to carry out these comparisons) = cost (in terms of IOs)

IO cost might also include storing intermediate results.

How is the relation stored?

Cost is affected by how data is stored and retrieved-

1. Contiguously stored records : A block can be expected to be full of records from the same relation. Fetching a block will get us all the records that the block can hold. Also called a *clustered relation*.
2. Non-contiguously stored records : A block cannot be expected to be full of tuples from the same relation. In the worst case, fetching a block, ie, 1IO will get us only one record. When dealing with non-contiguous files we assume the worst case, ie, fetching 1 record = 1IO.
3. Clustering Index : Values with the same key are likely to be found together in the same block(s). For example a primary index.
4. Non-clustering index : All the values with the same key can't be expected to be found in the same block. For example a secondary index. Fetching such records costs 1IO per record.

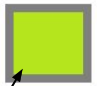
Should not be mixed up with *clustered file organization* we discussed in the first chapter. Clustered file organization means storing tuples of different relations together to speed up joins if frequent join operations are expected on those relations.

How is the relation stored?

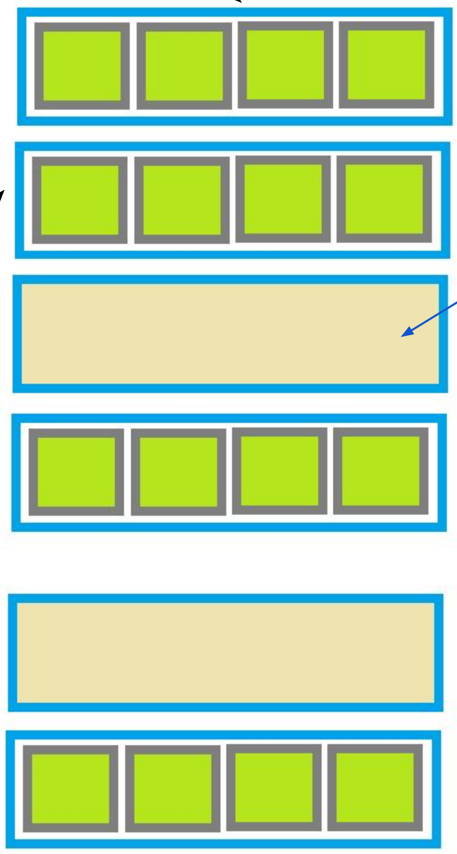
Cost is affected by how data is stored and retrieved-

1. Contiguously stored records : A block can be expected to be full of records from the same relation. Fetching a block will get us all the records that the block can hold. Also called a *clustered relation*.
2. Non-contiguously stored records : A block cannot be expected to be full of tuples from the same relation. In the worst case, fetching a block, ie, 1IO will get us only one record. When dealing with non-contiguous files we assume the worst case, ie, fetching 1 record = 1IO.
3. Clustering Index : Values with the same key are likely to be found together in the same block(s). For example a primary index.
4. Non-clustering index : All the values with the same key can't be expected to be found in the same block. For example a secondary index. Fetching such records costs 1IO per record.

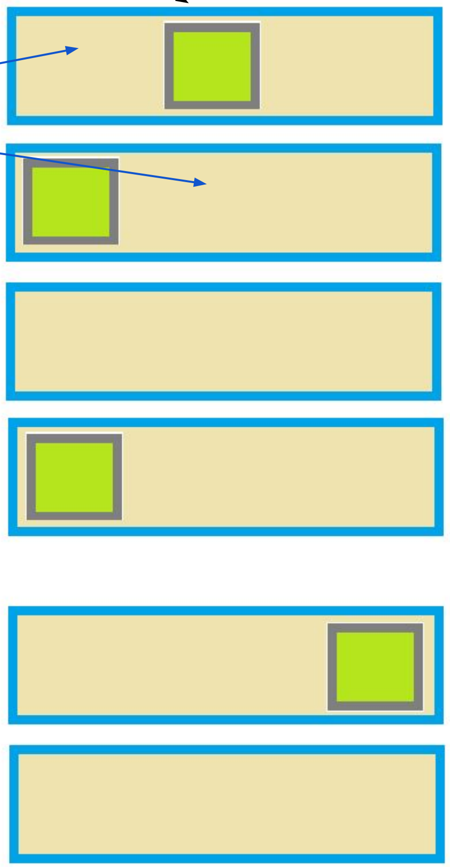
Contiguous and Non-contiguous



Represents a tuple of relation R. R has T(R) tuples. If stored contiguously this relation takes B(R) blocks.



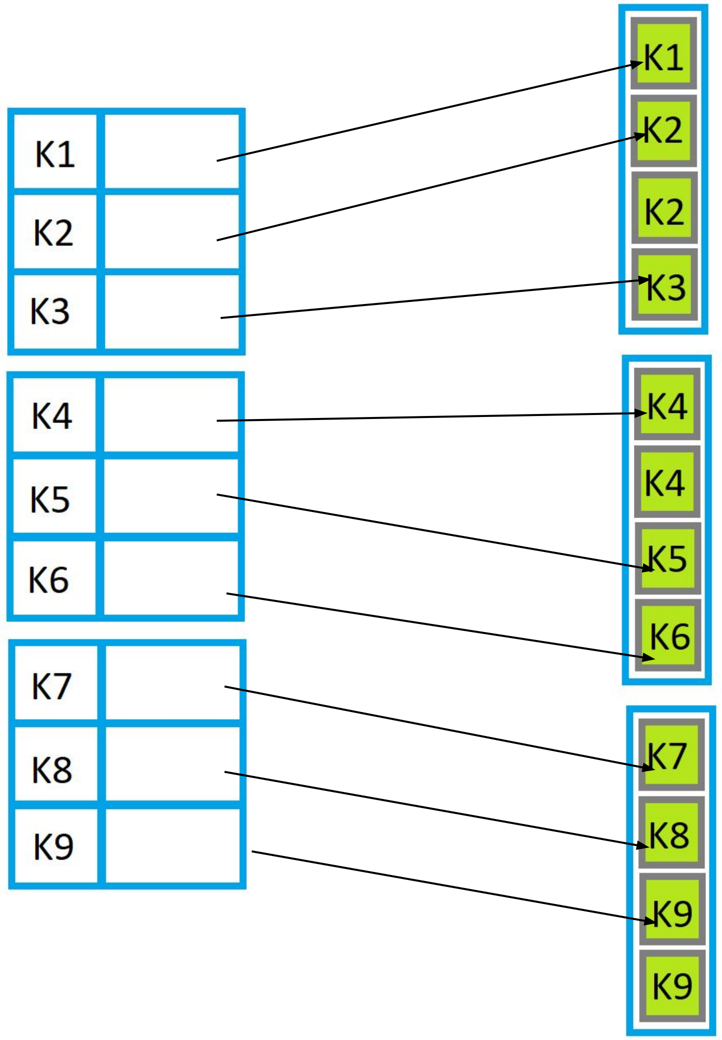
R stored contiguously: all blocks holding R tuples contain only tuples of R. If we fetch such a block we get all the tuples of R a block can hold, fetching such a file takes B(R) IOs.



R stored non-contiguously : all blocks holding R tuples contain only one tuple of R, retrieving such a file would mean T(R) IOs.

Clustering Index

A primary index, can be dense or sparse.



Non-Clustering Index

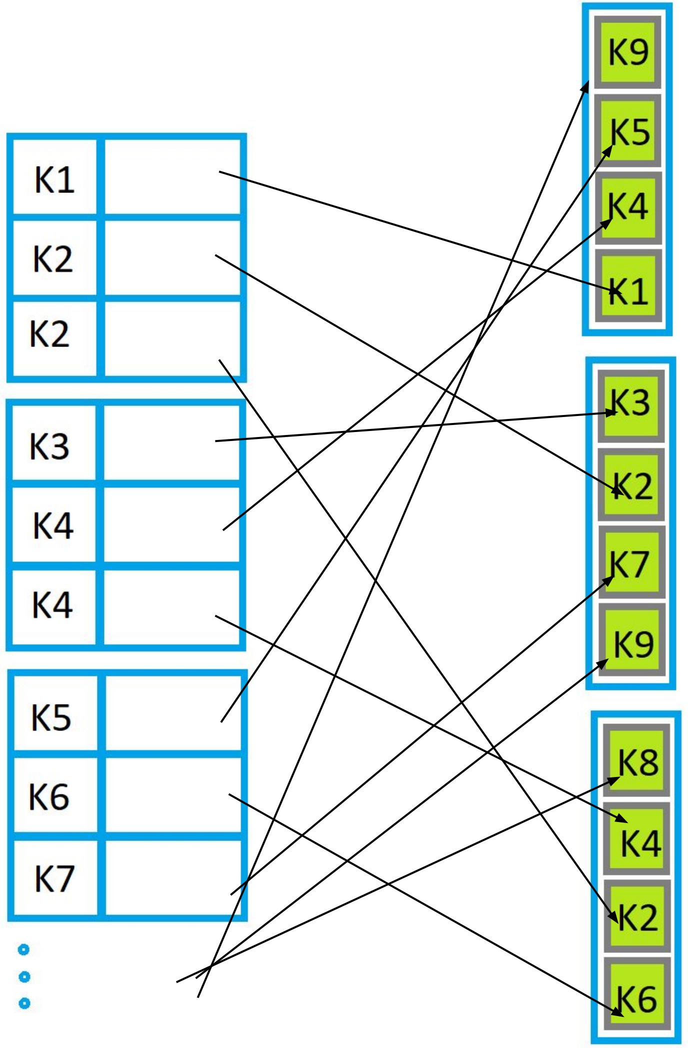
A secondary index.

K1 ----------->

K3 ----------->

K5 ------------>

.
.
.



Join Algorithms

Apart from storage and retrieval, which algorithm are we using to perform the join will also affect the cost. We will look at :

1. Iteration Join
2. Merge Join
3. Hash Join
4. Index Join

Which algorithm we use depends on how the data is stored, how much main memory we have available, if an index is available on the join attribute, etc.

Running Example

Two Relations R and S

T(R) = 60000

T(S) = 2000

Block size = 4000 bytes

S(R) = 100 bytes

S(S) = 500 bytes

Main Memory = 126 Buffers

Iteration Join R ⋈ S

Simplest Case :

1. Relations not Contiguous, ie, 1 IO will fetch one record.
2. We read tuples from R one by one and for each tuple we read the whole of S to find matches.

For each T(R) that we read (ie. bring to memory), we read the whole of S, ie T(S) IOs.

$$\text{Cost} = \underbrace{\boxed{T(R)}}_{\text{Cost of Reading R}} + \underbrace{\boxed{T(R)xT(S)}}_{\text{Cost of Reading S}} = T(R)\ (1+T(S))$$

Iteration Join $R \bowtie S$

Simplest Case :

1. Relations not Contiguous, ie, 1 IO will fetch one record.
2. We read tuples from R one by one and for each tuple we read the whole of S to find matches.

For each T(R) that we read (ie. bring to memory), we read the whole of S, ie T(S) IOs.

Cost = T(R) + T(R)xT(S)

= T(R) (1+T(S))

=60000(1+2000)

=120060000 IOs

Iteration Join R $\bowtie$ S

Simplest Case :

1. Relations not Contiguous, ie, 1 IO will fetch one record.
2. We read tuples from R one by one and for each tuple we read the whole of S to find matches.

For each T(R) we read, we read the whole of S, ie T(S) IOs.

Cost = T(R) + T(R)xT(S)
= T(R) (1+T(S))
=60000(1+2000)
=120060000 IOs

We are utilizing only two main memory buffers at any given time using this simple method, ie, this approach only makes sense if we have only two buffers available. But this also means an iteration join is possible with even with just two buffers and no assumptions on how the file is stored!

Iteration Join : Improvement 1/3

Since we have 126 buffers in main memory why not use them.

We can fill up 125 buffers with tuples from R and using the remaining buffer for S.

Case 1: Relations, R and S not contiguous

block size : 4000

1. One IO will still fetch only one record, but now we can keep 125 buffers worth of R tuples in memory.
2. Since S(R) = 100bytes, one buffer can hold 40 records, and 125 buffers can hold 5000 records. 4000/100
3. So at one time we can have 5000 R tuples in memory, and we can join these tuples with S by reading all of S one tuple at a time into the remaining one buffer. T(s) + chunk = 2000 + 5000 = 7000
4. Once we are done with step three, we repeat this step for the next 5000 tuples of R.

60000/5000

Since T(R) = 60000, R can be divided into 12 such chunks. Reading these chunks takes T(R) IO, and for each of these we read all of T(S) tuples, ie 12 times T(S).

? Cost = T(R) + 12* T(S) OR $12 * \left(\frac{T(R)}{12} + T(S)\right)$ OR 12 * (5000 + 2000)

= 60,000 + 12x2000 = 84000 IOs

60,000 / 5000 = 12

12 * 7000 = 84000

Iteration Join : Improvement 2/3

Since we have 126 buffers in main memory why not use them.

We can fill up 125 buffers with tuples from R and using the remaining buffer for S.

One IO can now fetch 40 records of R or 8 records of S

Case 2: Relations, R and S, Contiguous

S(R) = 100bytes, therefore 40 records/block, and B(R) = 1500 blocks 60,000/40 = 1500 blocks

4000/500=8

S(S) = 500bytes, therefore 8 records/block, B(S) = 250 blocks 2000/8 = 250 blocks

1. We can still keep 125 buffers or blocks of R in memory. (125 blocks = 125*40 = 5000 records)
2. Once we have these 125 buffers of R, we can read the whole of S one ***block*** at a time to join with the records in these 125 buffers.
3. Once we are done with step 2, we repeat this step for the next 125 tuples of R. Since B(R) = 1500, we have to repeat step 2 12 times. 1500 / 125 = 12

Cost = B(R) + 12* B(S) OR $12 * \left(\frac{B(R)}{12} + B(S) \right)$

= 1500 + 12* 250 = 4500

Iteration Join : Improvement 3/3

What if we divided S in chunks of records/blocks that fill 125 buffers of memory and read R repeatedly.

When contiguous, think and compute in blocks B(·); when non-contiguous, think in tuplesT(·) .

Case 1: Non Contiguous relations: ?

125 buffers will hold 1000 T(S) tuples at a time, it will take 1000 IOs. Then we will read the whole of R once for each 1000 S tuples. (B(s)/ no. buffer) = 250/125 = 2; 2000/2=1000

Cost = T(S) + 2* T(R) = 2 (1000 + 60000) = 122000 (worse compared to the earlier 84000 IOs)

Case 1: Contiguous relations: One I/O fetches a block. Keep 125 locks of R in memory

125 buffers will still hold 1000 T(S) tuples at a time. But now we can fetch these 1000 tuples in 125IOs.

Cost = B(S) + 2* B(R) = 2 (125 + 1500) = 3250 (better compared to 4500 IOs)

250/2

B(S)250 / 125 = 2

Merge Join

Assumption: The join is performed on sorted (on join attribute), contiguous relations. Therefore if relations not sorted, first sort and only then join.

The algorithm linearly reads both relations, and compares the tuples. So the cost of the join is = cost of reading the relations = B(R) + B(S)

But if the files are not sorted, we need to sort first using Multiway Merge Sort. Cost of the sort (2 IOs per phase per block) is added to the join cost.

NOTE: if the relations are not contiguous, then the cost of the first phase of sort will be = $\underbrace{T(R) + T(S)}_{\text{Reading}} + \underbrace{B(R) + B(S)}_{\text{Writing}}$

The rest of the phases will 2 IO per phase per block.

Merge Join

Case 1 : R and S are sorted (and therefore contiguous as well!)

Cost = Cost of reading R and S = B(R) + B(S) = 1750 IOs

Case 2 : R and S are not sorted but are contiguous

1. Sort R
   a. Phase 1: We get c(1500/126) = 12 sublists
   b. Phase 2 : Merge 12 sublists (available buffers 125, therefore sort can be completed in the second phase)
   c. Cost = 2*2*1500 = 6000
2. Sort S
   a. Phase 1: We get c(250/126) = 2
   b. Phase 2 : Merge 2 sublists
   c. Cost = 2*2*250 = 1000
3. Join the sorted Files
   a. Cost = B(R) + B(S) = 1750

Total Cost = Cost of Sort + Cost of Join = 7000 + 1750 = 8750 IOs

Merge Join : An improvement on Case 2

Case 2 : R and S are not sorted but are contiguous

1. Sort R
   a. Phase 1: We get c(1500/126) = 12, Cost of Phase 1 for R = 2*1500 3000
2. Sort S
   a. Phase 1: We get c(250/126) = 2, Cost of Phase 1 for S = 2*250 500
3. Read sublists and Join
   a. Cost = 1500+250 cost of joining

After Phase 1 is finished for both, we have in total 12+2 sublists of R and S respectively on Disk. Since we have in total 126 buffers in main memory, and altogether we need only 14 for input, we can perform the join right after phase 1 (if we don't need the sorted files for future use).

Total Cost = Cost of creating sublists + Cost of Join = 3500 + 1750 = 5250 IOs

Skipping the last phase and doing the Join instead can only be done once the total number of sublists of both relations together ≤ Memory buffers (This might take more than 1 phase or might not even be possible always).

Hash Join

1. Hash R tuples into G buckets numbered $G_0, G_1, \dots G_k$
2. Hash S tuples into H buckets numbered $H_0, H_1, \dots H_k$
3. For i = 0 to k do : match tuples in G_i, H_i buckets

- The hash key will be the attribute on which the Join is to be performed.
- The tuples with the same key will go to the same numbered bucket.
- We only need to compare buckets with the same number, ie G_1 with H_1, G_2 with H_2.. etc

Cost of creating Buckets : First phase

= **Reading** the whole relation once, hashing and **writing** back into buckets

= 2 x (B(R) + B(S))

Cost of Join : Second phase

= Reading the buckets

= B(R) + B(S)

NOTE: if the relations are not contiguous, then the cost of the first reading of tuples will be equal to the number of tuples, therefore Cost of bucketizing in such a situation = **T(R) + T(S)** + **B(R) + B(S)**

Hash Join Example

Let's assume our Hash function creates 50 buckets. Lets calculate the IO for R and S, assuming they are contiguous (whether the relations are initially sorted or unsorted has no effect on cost of Hash Join).

Block size = 4000 bytes
S(R) = 100 bytes
S(S) = 500 bytes

1. Bucketize R.
   a. Divide the 60000 tuples into 50 buckets, reading one block at a time.
   b. We will end up with 60000/50 = 1200 records per bucket, OR 1200/40 = 30 blocks per bucket.
   c. Cost = 2xB(R) = 3000 B(R) = 1500 : 60000/40
2. Bucketize S.
   a. Divide the 2000 tuples into 50 buckets, reading one block at a time. Block size 4000 / 500 bytes S(s) = 8
   b. We will end up with 2000/50 = 40 records per bucket, OR 40/8 = 5 blocks per bucket.
   c. Cost = 2xB(S) = 500 B(S) = 250 block (IOs):
      S : 4000/500 = 8; B(S): 2000/8 = 250 blocks,
3. Join
   a. Keep one whole bucket of either relation (if not enough space we use the smaller one)
   b. Here Rs buckets are made up of 30 blocks each and Ss buckets are 5 blocks each
   c. Since we have 126 buffers, we can fit both buckets together and compare.
   d. G_0 will be joined with H_0, G_1 with H_1, $G_2$with H_2, and so on…
   e. Cost = B(R) + B(S) = 1750

Total cost = 5250 IOs 3000 + 500 + 1750

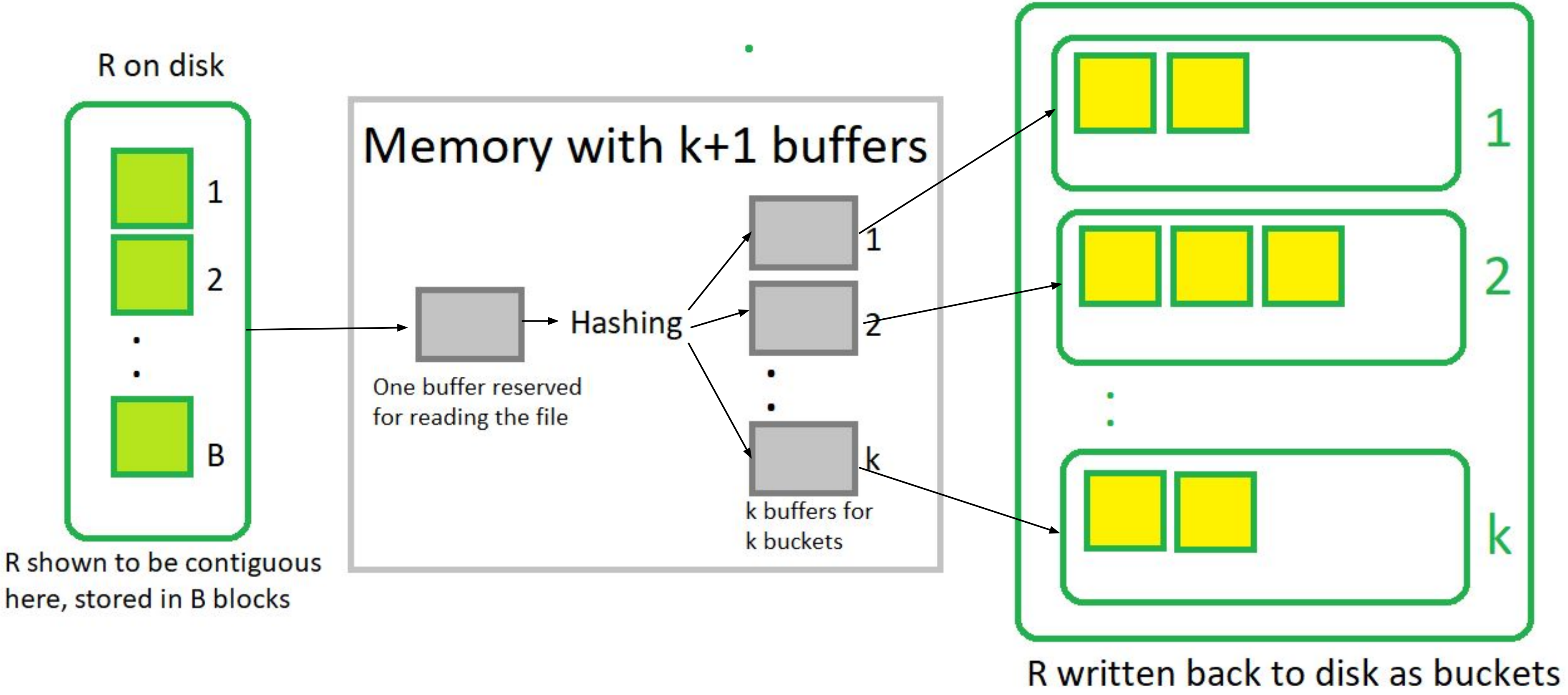
Number of Buckets : Hash Join

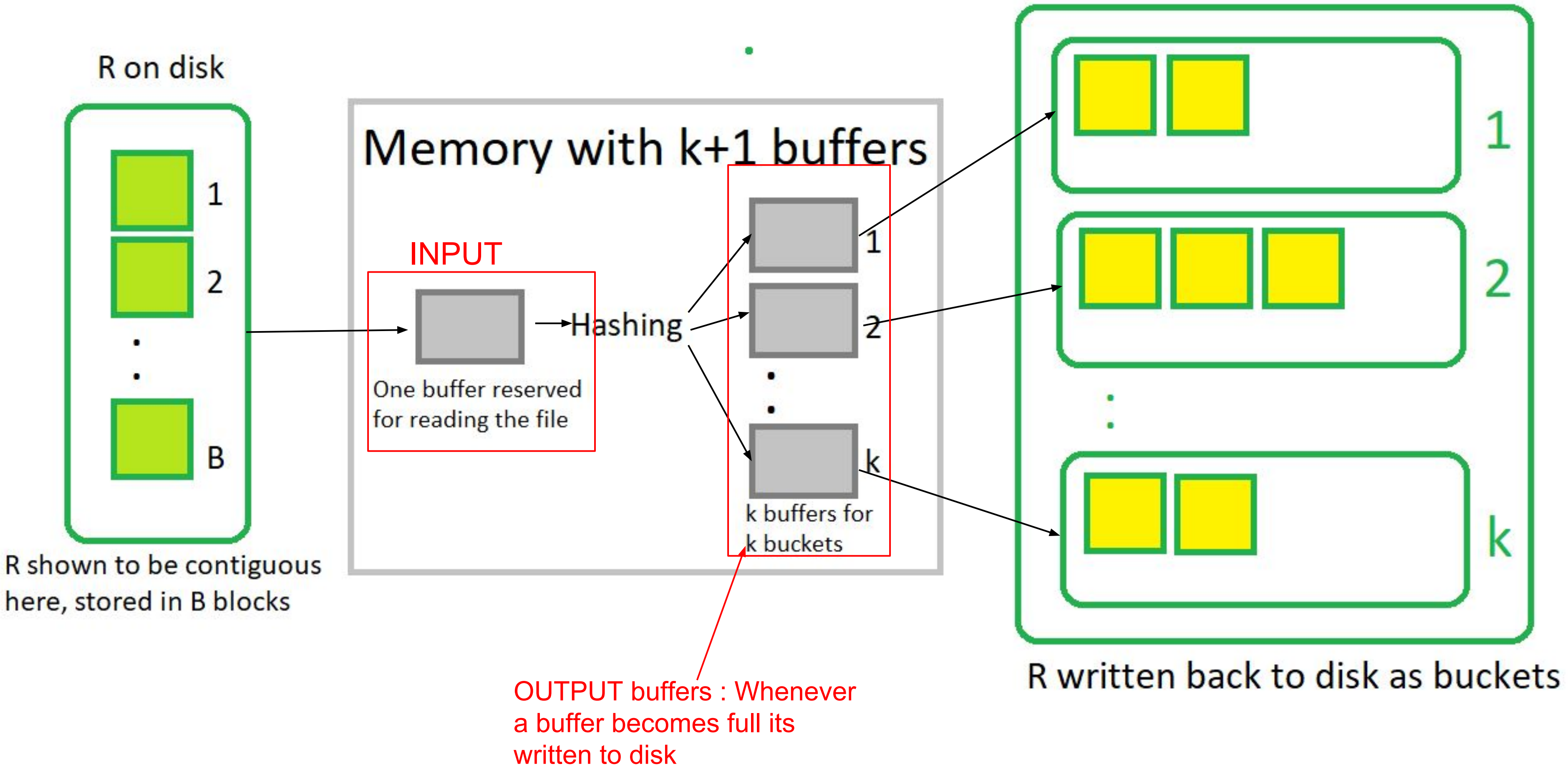
Bucketizing : Number of buckets can be at most M-1

Where M is the number of buffers available in Memory. Of course we can have fewer buckets than M-1 if we want, but not more than M-1.

When reading the blocks of a relation to memory to bucketize, we reserve one buffer for **input**, and the rest can be used for **output.** The output buffers represent the buckets. The relation is read into the input buffer, the key is hashed, and the tuples are moved to the output buffers depending on the hash value. When any of the output buffer becomes full, its written to disk in the corresponding bucket.

The next slide shows a relation R stored contiguously on disk in B blocks, being bucketized using memory which has k+1 buffers available.


R on disk
1
2
B
R shown to be contiguous here, stored in B blocks
Memory with k+1 buffers
Hashing
One buffer reserved for reading the file
1
2
k
k buffers for k buckets
1
2
k
R written back to disk as buckets


R on disk
1
2
B
R shown to be contiguous here, stored in B blocks
Memory with k+1 buffers
INPUT
Hashing
One buffer reserved for reading the file
1
2
k
k buffers for k buckets
OUTPUT buffers : Whenever a buffer becomes full its written to disk
1
2
k
R written back to disk as buckets

Size of Buckets : Hash Join

The number of buckets was decided in Phase 1 (bucketizing), size of the buckets becomes important for Phase 2 (Join)

The size of the bucket decides if the join can be completed in two phases or will take more than two.

1. **Condition For Join:** one bucket should completely fit inside the memory with one block to spare. It is enough if it's the smaller bucket.
2. If neither bucket can fit into the main memory with a block to spare, we need to re-bucketize. This makes further smaller buckets from the existing buckets.
3. Step 2 will be repeated till one bucket can fit completely in memory with one block to spare for the corresponding bucket for the other relation. Each additional bucketizing is one extra phase.

Each additional phase increases the IO cost by : $2(B(R) + B(S))$

Memory Requirement for Hash Join

Thc two-pass hash-join algorithm will work as long as approximately $\min(B(R), B(S)) \leq M^2$.

In general for n passes : $\min(B(R), B(S)) \leq M^n$.

The first n-1 phases will be for bucketizing and the last one for Join.

Please note in the case of Merge SORT: $\max(B(R), B(S)) \leq M^n$. decides how many phases the sorting will take, the join itself is one phase extra. However in some cases, when the total number of sublists from both relations become less than or equal to the memory buffers available, we can skip the last phase of sort and do the join instead. This optimization is not always possible, and sometimes we might need to do all n phases of sorting before we can join.

Memory Requirement for Hash Join

Let's assume in our Hash join example from the last slide, **the number of main memory buffers available are only 6**. The rest of the specifications are the same. Now only 5 buffers are available for bucketizing.

Number of phases or passes : min(B(R), B(S)) ≤ M^n. ie 250 ≤ 6^n. This gives n = 4, ie we will need four passes to perform this join. The first three will be bucketizing and the last will be the join.

1. Phase 1 : Bucketizing
   a. R : Divide the 1500 blocks into 5 buckets, reading one block at a time, we get 300 blocks per bucket.
   b. S : Divide 250 blocks into 5 buckets, we get 50 blocks per bucket.
   c. Cost = 2 (B(R) + B(S)) = 3500
2. Phase 2 : re-Bucketizing
   a. R: Further divide each of the 5 buckets into another 5, size of new buckets = 300/5 = 60 blocks per new bucket.
   b. S : Further divide each of the 5 buckets into another 5, size of new buckets = 50/5 = 10 blocks per new bucket.
   c. Cost = 2 (B(R) + B(S)) = 3500
3. Phase 3 : re-re-Bucketizing
   a. R: new bucket size = 60/5 = 12 blocks / bucket.
   b. S: new bucket size = 10/5 = 2 blocks / bucket. **Finally we have a bucket small enough to fit in main memory with a block to spare!**
   c. Cost = 2 (B(R) + B(S)) = 3500
4. Phase 4 : Join
   a. **Only Ss buckets can be kept in the memory** while the corresponding R Bucket is read one block at a time and joined with the whole of the S bucket.
   b. Cost = B(R) + B(S) = 1750

Total cost = 3x2x(B(R)+B(S)) + (B(R) + B(S)) = 3x3500+1750 = 12,250 IOs.

A One Pass Join

If M>min(B(R),B(S)), the smaller relation can completely fit in the memory with one or more buffers to spare.

This means we can keep one relation entirely in memory, and compare and join with the other, by reading the other relation block by block into the spare buffers. This gives a **one pass join**.

In other words:

M-1 = min(B(R),B(S)) , ie,

M = min(B(R),B(S)) +1

is the minimum memory requirement for a trivial one pass join.

Cost of One Pass Join = B(R) + B(S)

Generally for Contiguous Relations

Cost of Merge Join:

Sorting: 2* (B(R) +B(S)) per phase

Join : B(R) + B(S)

Cost of Hash Join:

Bucketizing : 2* (B(R) +B(S)) per phase

Join : B(R) + B(S)

Cost of a One Pass Join:

Join : B(R) + B(S)

Exercise 1

Relation R : B(R) = 10,000

Relation S : B(S) = 4000

Calculate the number of phases and the IOs for a Hash Join if R and S are contiguous for:

a) M = 100
b) M = 50
c) M = 30

Exercise 1 Ans

Relation R : B(R) = 10,000, Relation S : B(S) = 4000

Calculate the number of phases and the IOs for a Hash Join if R and S are contiguous for:

min(B(R),B(S)) = B(S) = 4000 (the smaller relation decides the number of phases for Hash Join)

a) M = 100

$M^2 = 100^2 = 10000 > 4000$, therefore 2 phases

$\text{IOs} = \underset{\text{Bucketizing}}{2x(B(R)+B(S))} + \underset{\text{Join}}{B(R)+B(S)} = 42{,}000$

b) M = 50

$M^3 = 50^3 = 125{,}000 > 4000$, therefore 3 phases

$\text{IOs} = \underset{\text{Bucketizing}}{2x2x(B(R)+B(S))} + \underset{\text{Join}}{B(R)+B(S)} = 70{,}000$

c) M = 30

$M^3 = 30^3 = 27{,}000 > 4000$, therefore 3 phases

$\text{IOs} = \underset{\text{Bucketizing}}{2x2x(B(R)+B(S))} + \underset{\text{Join}}{B(R)+B(S)} = 70{,}000$

Exercise 2

Relation R : B(R) = 10,000

Relation S : B(S) = 4000

Calculate the number of phases and the IOs for a Merge Join if R and S are contiguous. Optimize whenever possible.

a) M = 101
b) M = 50
c) M = 30

Exercise 2 Ans 1/3

Relation R : B(R) = 10,000; Relation S : B(S) = 4000

Calculate the number of phases and the IOs for a Merge Join if R and S are contiguous. Optimize whenever possible.

max(B(R),B(S)) = B(R) = 10000, the bigger relation decides the number of phases for Merge Sort.

a) M = 101

$M^2 = 101^2 = 10201 > 10000$, therefore 2 phases for sorting

1st Phase :

No. of Sublists for R : c(10000/101) =100

No. of Sublists for S : c(4000/101) = 40

2nd Phase :

No. of Sublists for R : c(100/100) =1

No. of Sublists for S : c(40/100) = 1

Sorting complete. We can do the Join now in B(R) + B(S) IOs. Note we weren't able to optimize this Join by skipping last phase of sorting as the total number of sublists weren't ≤ M after any intermediate phase. So we completed both the phases of Sort.

$$\text{IOs} = \underbrace{2\text{x}2\text{x}(B(R)+B(S))}_{\text{Sorting}} + \underbrace{B(R)+B(S)}_{\text{Join}} = 70{,}000$$

Exercise 2 Ans 2/3

b) M = 50

$M^3 = 50^3 = 125{,}000 > 10{,}000$, therefore 3 phases

1st Phase :

No. of Sublists for R : c(10000/50) =200

No. of Sublists for S : c(4000/50) = 80

2nd Phase :

No. of Sublists for R : c(200/49) =4

No. of Sublists for S : c(80/49) = 2

Here we have a possibility for optimization as the total sublists = 4+2 $\leq$ M. Therefore we can skip the last phase of sort and perform the Join instead.

$$\text{IOs} = \underbrace{2\text{x}2\text{x}(B(R)+B(S))}_{\text{Sorting}} + \underbrace{B(R)+B(S)}_{\text{Join}} = 70{,}000$$

Exercise 2 Ans 3/3

c) M = 30

$M^3 = 30^3 = 27{,}000 >$ ~~4000~~ 10,000, therefore 3 phases

1st Phase :

No. of Sublists for R : c(10000/30) =334

No. of Sublists for S : c(4000/30) = 132

2nd Phase :

No. of Sublists for R : c(334/29) =12

No. of Sublists for S : c(132/29) = 5

Here we have a possibility for optimization as the total sublists = 12+5 ≤ M. Therefore we can skip the third and last phase of sort and perform the Join instead.

IOs = 2x2x(B(R)+B(S)) + B(R)+B(S) = 70,000

Exercise 3

Relation R : B(R) = 10,000, T(R) = 100,000

Relation S : B(S) = 4000, T(S) = 32,000

M = 100. Calculate IOs:

a) for Hash Join, if the relations are not contiguous
b) for Hash Join, if the relations are sorted
c) for Merge Join, if the relations are sorted
d) And Minimum memory requirement for One Pass Join if the relations are Contiguous.

Exercise 3 Ans 1/2

Relation R : B(R) = 10,000, T(R) = 100,000

Relation S : B(S) = 4000, T(S) = 32,000

M = 100. Calculate IOs:

a) for Hash Join, if the relations are not contiguous

$M^2 = 100^2 = 10000 > 4000$, therefore 2 phases

1. IOs for 1st phase, ie, Bucketizing:

   Reading = T(R)+T(S) = 100,000+32,000

   Writing = B(R)+B(S) = 10,000+4000

   Total for 1st Phase = 146000

2. IOs for 2 Phase , ie, Join = B(R)+B(S) = 14000

Total IOs = 146000+14000

Exercise 3 Ans 2/2

b) for Hash Join, if the relations are sorted

Sorted ⇒ Contiguous

$M^2 = 100^2 = 10000 > 4000$, therefore 2 phases

Total IOs = 2 (B(R)+B(S)) + B(R)+B(S) = 42000

c) for Merge Join, if the relations are sorted

IOs = Reading the relations for Join = B(R)+B(S) = 14000

d) And Minimum memory requirement for One Pass Join if the relations are Contiguous.

For One pass : M = min(B(R),B(S)) +1

Therefore we need M = 4001 for a one pass join.

IOs = reading the relations = B(R) + B(S) = 14000