Query Execution

Contains slides by
Hector Garcia-Molina

Overview

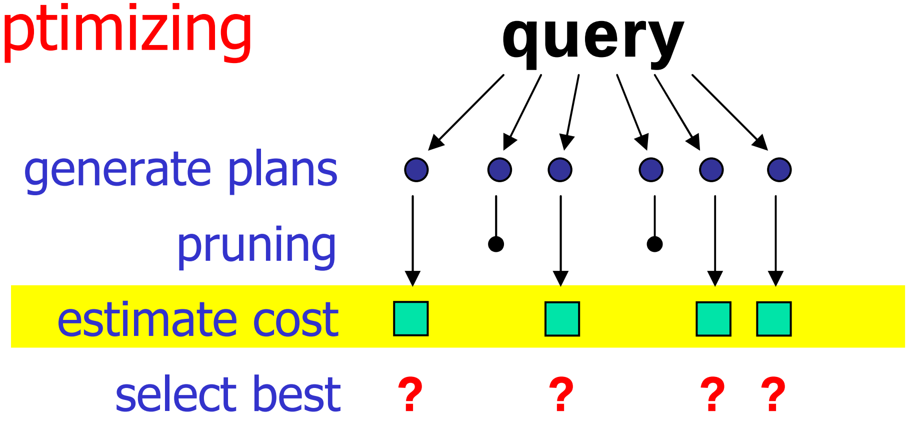
- ✓ Short about query processors
- ✓ Model for computing costs
- ✓ Cost of basic operations
- ✓ Implementation algorithms and their costs
  - ➢ tuple-at-a-time, unary operations
  - ➢ full-relation, unary operations
  - ➢ full-relation, binary operations

Query Processors – I

- ✓ So far, we have looked
  - ➢ hardware features such as disks and memory
  - ➢ data structures allowing fast lookup and efficient execution of basic operations
- ✓ SQL is a declarative language (specifies what to find, not how)
- ✓ A query processor must find a plan how to execute the query
  - ➢ query compilation
  - ➢ query execution
- ✓ There might be *several ways to implement a query* - the query compiler should find an appropriate plan
  - ➢ parsing – translating the query into a parsing tree
  - ➢ query rewrite – the parse tree is transformed into an expression tree of relational algebra (logical query plan)
  - ➢ physical plan generation – translate the logical plan into a physical plan
    - select algorithms to implement each operator
    - choose order of operations

Query Processors – II

- ✓ Making logical and physical query plans are often called query optimizing



- ✓ Next week, we look at how to generate and select a query plan, but first we must know *how to estimate the cost of each operator* performing a specific task in the entire operation:
  - which algorithm works best under the given circumstances?
  - how to pass data between operators?
  - ...

Cost Computation Model

Plan Operators

- ✓ A query consist of several operations of relational algebra
  - ➢ a physical query plan is implemented by a set of operators corresponding to the relational algebra operators
  - ➢ additionally, we need basic operators automatically used by other operators like reading (scanning) a relation, sorting a relation, etc.
- ✓ To choose a good query plan, we must be able to estimate the cost of each operator:
- ⇨ we will use the number of disk I/O's
  and we assume (if not specified otherwise) that
  - ➢ parameters to an operator must intially be retrieved from disk
  - ➢ output is consumed directly from memory (cost only dependent of output buffer size)
  - ➢ we can ignore other costs like CPU cycles, timing, ...

Cost Parameters

- ✓ Determining which mechanism to use, i.e., which has lowest costs, is dependent of several factors like
  - ➢ number of available memory blocks, M
  - ➢ existence of indexes (if so, what kind, size, overhead, ...)
  - ➢ layout on disk and disk characteristics
  - ➢ ...

- ✓ Additionally, for a relation R, we need
  - ➢ number of blocks to store all tuples, B(R)
  - ➢ number of tuples in R, T(R)
  - ➢ number of distinct values for an attribute a, V(R, a)
    (average of identical a-value tuples is then T(R)/V(R,a))

Factors Increasing Estimated Disk I/O Cost

- ✓ The actual disk I/O costs may be somewhat higher than our estimates:
  - ➢ if we use an index, the index itself may not be resident in memory: must retrieve index blocks
  - ➢ tuples where condition C holds, might fit on b blocks, but they might not start at the beginning of the first block – read b + 1 blocks
  - ➢ data on blocks might not be "compressed" – we leave room for data evolution
  - ➢ data might be sorted and grouped, and each "collection" may be stored on their own blocks – fragmentation
  - ➢ relation R is stored together with other relations – clustered file organization

- ✓ These factors can influence the costs of several algorithms later in the lecture, but we will *not* use them in our cost estimates

Factors Reducing Overall Time

- ✓ Extra buffers can speed up the overall processing time of an operation
  - ➢ if data is stored consecutively on disk, we can then retrieve or write more blocks at the same time – reducing the number of seeks and rotational delays
  - ➢ double buffering saves time waiting for disk I/O
  - ➢ parallel operations on multiple disks

- ✓ But, these mechanisms do not reduce the number of blocks that initially has to be moved between disk and memory – only average time per block

Cost of Basic Operators

Cost of Basic Operators – I

- ✓ The cost of reading a disk block is 1 disk I/O
- ✓ The cost of writing a disk block is 1 disk I/O
  (we assume that verifying the write operation is free →read I/O = write I/O)
- ⇨ updates cost 2 disk I/Os

- ✓ One of the fundamental operations is to read a relation R - must read (scan) all blocks which contain records for R
  → cost dependent on storage
  - ➢ *clustered* relation, all records stored together – B(R) disk I/Os
  - ➢ *scattered* relation, records on different blocks – max T(R) disk I/Os
    (we must in a worst case scenario read T(R) blocks – all tuples on different blocks)
  - ➢ we will *assume clustered relations* if not specified otherwise
    (relations that is a result of other operators is almost always clustered)

Note:
- clustered *file organization* – interleaves tuples of different relations
- clustered *relation* – records of a relation is stored on as few blocks as possible
- clustering *index* – index on attribute sorting a clustered relation on disk

Cost of Basic Operators – II

- ✓ Sorting is another important operation – sort-scan reads a relation R and returns R in sorted order
  - use an index having a list of sorted pointers, e.g., B-trees, sequential index files – cost is dependent of operation, storage, available memory, ...
  - if relation fits in memory, use an efficient main-memory sorting algorithm – cost B(R) disk I/Os
  - if relation is too large to fit in main memory, we must use a sorting algorithm making several passes over data → two-phase multiway merge sort (TPMMS) is often used

Cost of Basic Operators: TPMMS – I

✓ Two-Phase, Multiway-Merge Sort (TPMMS)

- phase 1: sort main-memory sized pieces of the relation
  - fill all available memory with blocks containing the relation
  - sort the records in memory
  - write the sorted list back to disk
  - repeat until all blocks are read and all records are sorted in sub-lists
  - ⇨ cost 2B(R), i.e., all blocks are both read and written
- phase 2: merge all sorted sub-lists into one sorted list
  - read first block of all sub-lists into memory and compare first element in each block
  - place smallest element in new list
  - ⇨ cost B(R) (result is consumed directly from memory)

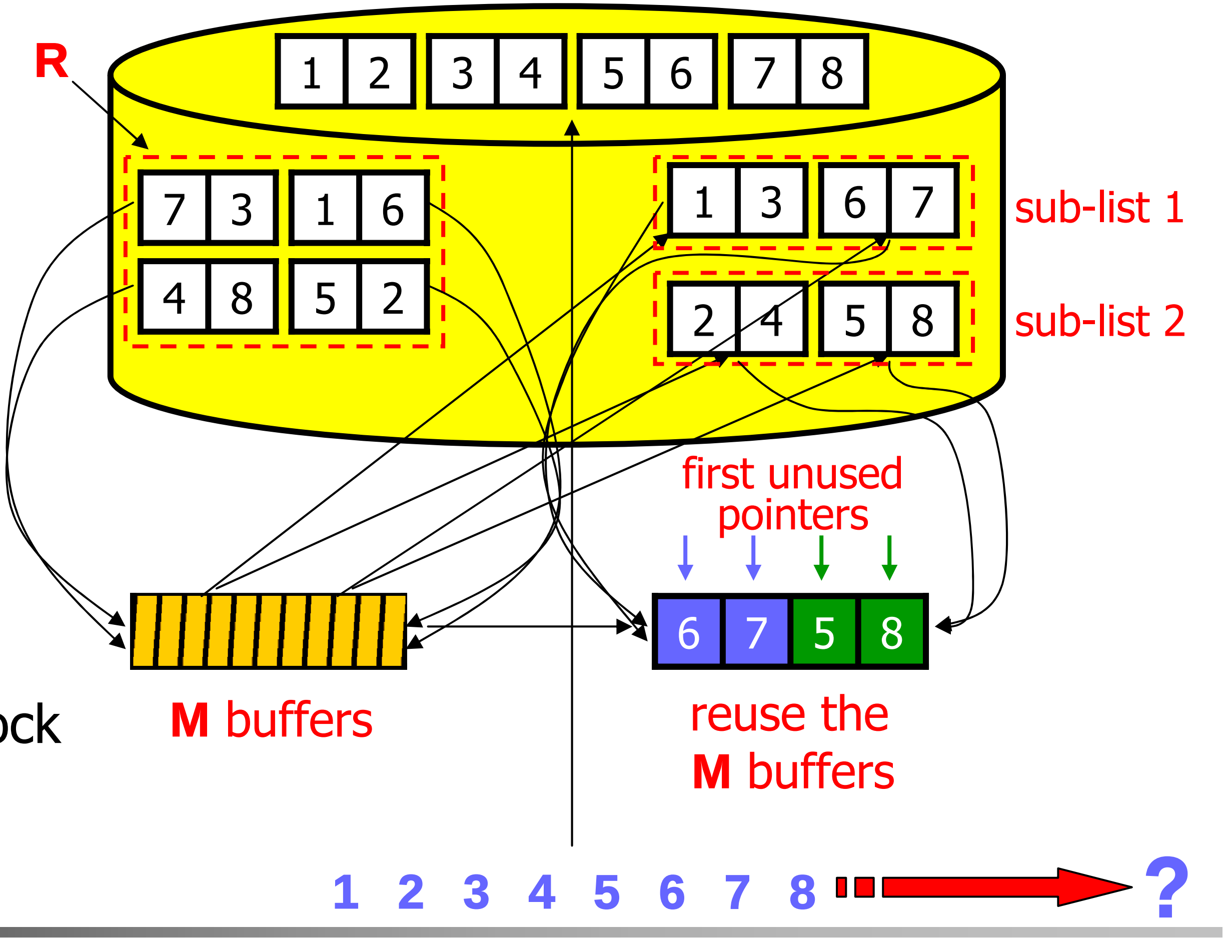
⇨ total cost 3B(R)

Cost of Basic Operators: TPMMS – II

- ✓ Example:
  M=2, B(R)=4, T(R)=8
  - fill memory
  - sort
  - write back sub-list
  - repeat
  - read first block of all sub-lists
  - compare first *unused* element
  - output the smallest element, fetch new block if necessary
  - repeat two last steps

Note 1:
optionally (and usually), we may write the result back to disk, but we assume the result is given to another operator or returned as final result – cost **3B(R)**

Note 2:
if R is not clustered, cost **T(R)+2B(R)**



Cost of Basic Operators: Hash Partitioning – I

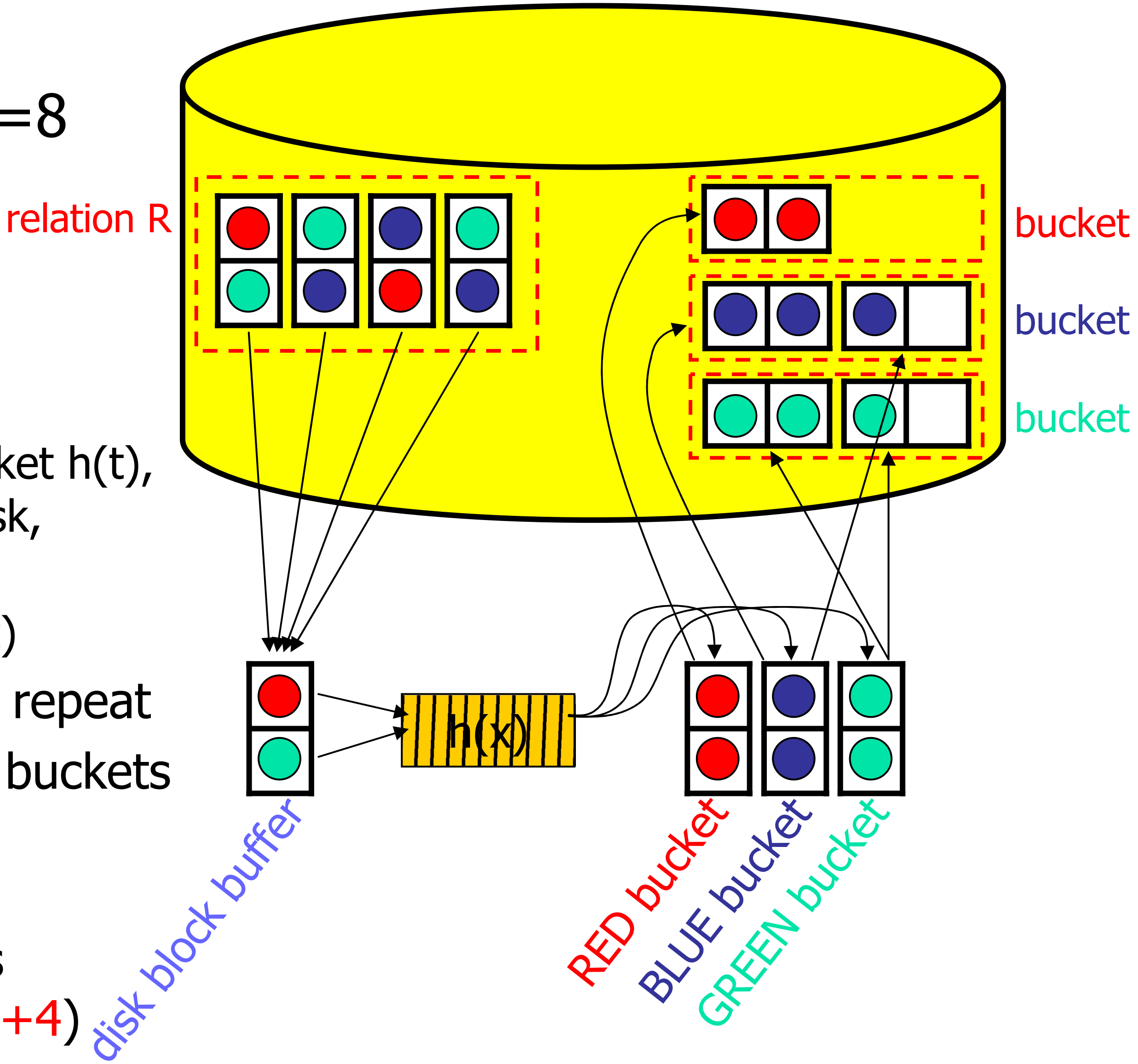
- ✓ Splitting the relation in sub-groups using hashing is also used for several operators if the data set is too large to fit in memory
  - ➢ hash function mapping tuples that should be considered together into same bucket
  - ➢ if M available buffers:
    use M-1 buffers for buckets, 1 for reading disk blocks
  - ➢ algorithm:

```
FOR each block b in relation R {
        read b into buffer M
        FOR each tuple t in b {
                IF NOT room in bucket h(t) {
                        copy bucket h(t) to disk
                        initialize new block for bucket h(t) }
                copy t into bucket h(t) }}
FOR each non-empty bucket { write bucket to disk }
```

  - ⇨ cost 2B(R) – read all data and write it back partitioned (NB This cost includes writing to memory!)

Cost of Basic Operators: Hash Partitioning – II

- ✓ Example
  M=4, B(R)=4, T(R)=8
  - ➢ initialize buffers
  - ➢ read block b
  - ➢ for each tuple t in b
    - calculate h(t)
    - if not room in bucket h(t), write bucket to disk, initialize new
    - put t in bucket h(t)
  - ➢ read next block and repeat
  - ➢ write all non-empty buckets to disk
  - ➢ cost 2B(R) disk I/Os (actually 4+5, not 4+4)



Query Execution – I

- ✓ Having looked at some basic operators, we now begin studying algorithms for the different relational algebra operators
- ✓ Mainly, three classes of algorithms:
  - ➢ sorting-based
  - ➢ hash-based
  - ➢ index-based
- ✓ Additionally, the cost and complexity can be divided into different levels
  - ➢ one-pass algorithms – data fits in memory, reading data only once from disk
  - ➢ two-pass algorithms – data too large to fit in memory, read data, process, write back, read again
  - ➢ n-pass algorithms – recursive generalizations of two-pass algorithms for methods needing several passes over the entire data set

Query Execution – II

- ✓ In addition to several classes and levels of algorithms, there are also different groups of operators:
  - tuple-at-a-time, unary operations:
    - selection (σ)
    - projection (π)
  - full-relation, unary operations:
    - grouping (γ)
    - duplicate-elimination (δ)
  - full-relation, binary operations:
    - set and bag union ($\cup$)
    - set and bag intersection ($\cap$)
    - set and bag difference (–)
    - joins ($\bowtie$)
    - products (×)

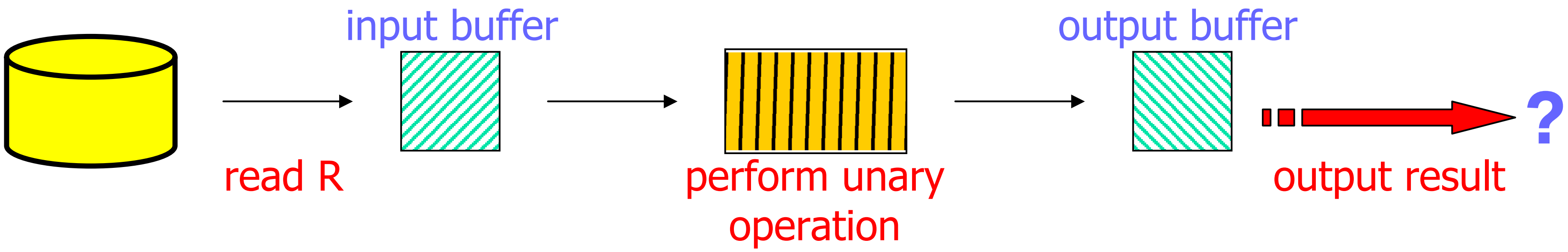
we will now look at several ways to implement these operators using different algorithms and number of passes

Unary, Tuple-at-a-Time Operations

Note that only summaries will be lectured from here on!

Tuple–at–a–Time Operators – I

✓ Both selection (σ) and projection (π) have obvious algorithms – regardless of whether the relation fits in memory or not:



- read the blocks of relation R one at a time
- perform the operation on each tuple
- move the selected or projected tuples to the output buffer

Tuple–at–a–Time Operators – II

- ✓ Memory requirement is only $M \geq 1$ for the input buffer – output buffer is assumed to be part of consuming operator (or application)

- ✓ The cost of performing a scan in number of disk I/Os is dependent on how relation R is provided
  - in memory – 0
  - on disk, typically
    - B(R) disk I/Os if R is clustered
    - T(R) disk I/Os if R is not clustered (max)

Tuple–at–a–Time Operators – III

- ✓ Selection (σ) can greatly benefit from an index on R.a
  - single value queries, e.g., $\sigma_{a\,=\,v}(R)$
    - *clustering* index: cost #"a=v"-records/records_per_block disk I/Os , average B(R)/V(R,a) disk I/Os
    - *index on non-clustered relation*:
      cost #"a=v"-records disk I/Os , average T(R)/V(R,a) disk I/Os
      (can be less if several records is on same block)
    - index on key attribute: 1 disk I/Os (V(R,a) = T(R), B(R) > T(R))
  - range queries, e.g., $\sigma_{a\,<\,v}(R)$
    - *clustering* index: cost #"a<v"-records/records_per_block disk I/Os
    - *index on non-clustered relation*: cost #"a<v"-records disk I/Os
      (can be less if several records is on same block)
    - index on key attribute:
      - non-clustered relation: #"a<v"-records disk I/Os
      - clustered relation: #"a<v"-records/records_per_block disk I/Os
  - complex queries, e.g., $\sigma_{a\,<\,v \text{ AND } C}(R)$
    - cost can further be reduced if we can compare pointers before retrieving blocks

Tuple–at–a–Time Operators – IV

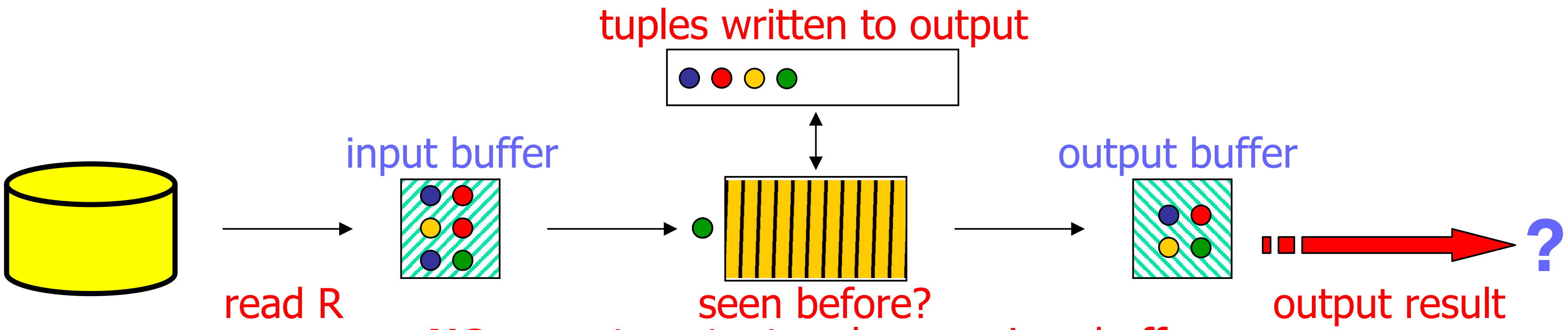
- ✓ **Worst-case** example: T(R) = 20.000, B(R) = 1000, $\sigma_{a = v}(R)$
  - ➢ no index
    - R clustered – retrieve all blocks → 1000 disk I/Os
    - R not clustered – each tuple on different blocks → 20.000 disk I/Os
  - ➢ clustering index (R clustered) – retrieve B(R) / V(R, a)
    - V(R, a) = 100 → 1000 / 100 = 10 disk I/Os
    - V(R, a) = 10 → 1000 / 10 = 100 disk I/Os
  - ➢ index, R not clustered – retrieve T(R) / V(R, a)
    - V(R, a) = 100 → 20.000 / 100 = 200 disk I/Os
    - V(R, a) = 10 → 20.000 / 10 = 2000 disk I/Os
      (even more than retrieving the whole file if R is clustered)
  - ➢ V(R, a) = 20.000, i.e., a is a key → 1 disk I/O

Note:
we must add any disk
I/Os for index blocks

Unary, Full-Relation Operations

Duplicate Elimination (δ): One-Pass – I

- ✓ Duplicate elimination (δ) can be performed by reading one block at a time, and for each tuple we
  - ➢ copy it to the output buffer if first occurrence
  - ➢ ignore it if we have seen a duplicate
- ✓ To be able to perform this operation, we must keep one copy of all tuples in memory for comparison



seen before?

NO, copy to output and comparison buffer

YES, do nothing

repeat for all blocks holding R

Duplicate Elimination (δ): One-Pass – II

- ✓ Memory requirement is M = 1 + B(δ(R))
  - ➢ input buffer – 1
  - ➢ buffers to hold all distinct tuples for comparison – B(δ(R))

- ✓ If M is too low, we will pay significantly due to thrashing

- ✓ Another important aspect here choice of main-memory data structure holding comparison tuples
  - ➢ searching sequentially – O(n^2)
  - ➢ hashing – O(n)
  - ➢ binary tree – O(n log$n$)

  } will need some more memory, but usually insignificant

- ✓ Number of disk I/Os is B(R)

Duplicate Elimination (δ): Two-Pass Sorting

- ✓ To perform duplicate elimination in two passes, we use an algorithm similar to Two-Phase, Multiway-Merge Sort (TPMMS)
  - read M blocks into memory
  - sort these M blocks and write sub-list to disk
  - however, instead of sorting the sub-lists, copy first tuple, eliminate duplicates in front of sub-lists

- ✓ Total cost is 3B(R) disk I/Os
  - 2 for first phase of TPMMS
  - 1 for duplicate elimination of first tuples of the sub-lists

- ✓ Memory requirement
  - M buffers can make M block long sub-lists (except last which may be smaller)
  - $B(R) \leq M^2$ → $\sqrt{B(R)} \leq M$
  - if $B(R) > M^2$ → more than M sub-lists, the algorithm will not work (cannot hold the first block of all sub-lists)

Duplicate Elimination (δ): Two-Pass Hashing

- ✓ Hash-based partitioning can be used for duplicate elimination in two passes
  - ➢ partition the relation as described before
  - ➢ duplicate tuples will hash to same bucket
  - ➢ read each bucket into memory and perform the one-pass algorithm removing duplicates

- ✓ Total cost is 3B(R) disk I/Os
  - ➢ 2 for partitioning the relation into hash buckets
  - ➢ 1 for duplicate elimination on each bucket

- ✓ Memory requirement:
  - ➢ M buffers to make M - 1 partitions (buckets)
  - ➢ $B(R) \leq M(M - 1) \approx B(R) \leq M^2 \rightarrow \sqrt{B(R)} \leq M$
  - ➢ each partition can be at most M long – algorithm will not work otherwise (must be able to read whole bucket into memory)

Duplicate Elimination (δ) : Cost and requirement summary for $\delta(R)$:

Algorithm	**Memory Requirement**	**Disk I/Os**
One-Pass	$M \geq 1 + B(\delta(R))$	B_R
Two-Pass Sorting	$M \geq \sqrt{B_R}$	$3B_R$
Two-Pass Hashing	$M \geq \sqrt{B_R}$	$3B_R$

Grouping (γ) : One-Pass

- Grouping (γ) gives us tuples consisting of *grouping attributes* and one or more *aggregated attributes*
- One-pass grouping:
  - one main-memory entry per group
  - scan tuples of R, reading one block at a time
  - modify aggregated values using the read value for each tuple belonging to group
    - `MAX` and `MIN`: compare stored aggregated value, change if necessary
    - `COUNT`: add one to the aggregated value for each tuple belonging to group
    - `SUM`: add value of tuple attribute to the aggregated value
    - `AVG`: store `COUNT` and `SUM`, calculate `AVG = SUM/COUNT` in the end
- Requirements and costs are similar to duplicate elimination
  - B(R) disk I/Os
  - $M = 1 + B(\gamma(R))$ memory buffers
    - input buffer – 1
    - buffers to hold all grouping elements – $B(\gamma(R))$
  - as with duplicate elimination one should use a fast main-memory data structure holding grouping elements (hashing, binary trees, ..)

Grouping (γ) : Two-Pass Sorting

- ✓ Two-pass grouping can be performed as duplicate elimination in two passes (based on TPMMS)
  - read M blocks into memory
  - sort these M blocks on grouping attribute(s) and write sub-list to disk
  - read first block of all sub-lists, for each smallest, unused sort key *v*
    - compute required aggregates for all *v* tuples
    - if buffer becomes empty, fetch new block from corresponding sub-list
    - repeat until all *v* tuples are used
    - output tuple with sort key *v* and associated aggregate values
  - repeat until all sub-lists are empty

- ✓ Total cost is 3B(R) disk I/Os
- ✓ Memory requirement is
  - M buffers can make M block long sub-lists (except last which may be smaller)
  - $B(R) \leq M^2$ → $\sqrt{B(R)} \leq M$
  - if $B(R) > M^2$ → more than M sub-lists, the algorithm will not work (cannot hold the first block of all sub-lists)

Grouping (γ) : Two-Pass Hashing

- ✓ Hash-based partitioning can be used for grouping in two-passes
  - ➢ partition the relation as described before, but use only grouping attributes as search key in hash function
  - ➢ duplicate tuples will hash to same bucket
  - ➢ read each bucket into memory and perform the one-pass algorithm removing duplicates

- ✓ Total cost is $3B(R)$ disk I/Os

- ✓ Memory requirement:
  - ➢ M buffers to make M - 1 partitions (buckets)
  - ➢ $B(R) \leq M(M - 1) \approx B(R) \leq M^2 \rightarrow \sqrt{B(R)} \leq M$
  - ➢ each partition can be longer than M and still use one pass per bucket
    - need only 1 record per group in the bucket
    - the algorithm will still work if records for all the groups in the bucket
    - ⇨ $B(R)$ might therefore be larger than M^2, but $B(R) \leq M^2$ is a good estimate

Grouping (γ) :
Cost and requirement summary for $\gamma(R)$:

Algorithm	**Memory Requirement**	**Disk I/Os**
One-Pass	$M \geq 1 + B(\gamma(R))$	B_R
Two-Pass Sorting	$M \geq \sqrt{B_R}$	$3B_R$
Two-Pass Hashing	$M \geq \sqrt{B_R}$	$3B_R$

Binary, Full-Relation Operations

Binary, Full–Relation Operations

- ✓ A binary operation takes two relations as arguments:
  - ➢ union: $R \cup S$
  - ➢ intersection: $R \cap S$
  - ➢ difference: $R - S$

  it is a difference between the set- and bag-versions of these operators – we will look at both, but unless specified otherwise, we assume a *bag-version*

  - ➢ joins $R \bowtie S$
  - ➢ products: $R \times S$

  we will look at *natural join*, the other operators can be implemented similarly

- ✓ In the operations needing a comparison (search), we usually implement a *main-memory search structure*, like binary trees or hashing, which also need resources. However, we will not be counting these buffers in our requirement estimation

Union (∪) : One-Pass

- ✓ **Bag** union (∪) can be computed using a very simple one-pass algorithm - R ∪ S:
  - ➢ read and copy every tuple of relation R to the output buffer
  - ➢ read and copy every tuple of relation S to the output buffer
- ✓ Total cost B(R) + B(S) disk I/Os
- ✓ Memory requirement 1 (read block directly to output buffer)

- ✓ **Set** union must remove duplicates
  - ➢ read smallest relation into M-1 buffers, say S, and copy every tuple to output
  - ➢ read the blocks holding R one-by-one into one buffer, and for each tuple see if it exists in S → if not, copy to output
- ✓ Memory requirement is now 1 + (M-1) = M, B(S) < M

Union (∪) :
Two-Pass Sorting

- ✓ **Bag** union works perfectly using the simple one-pass algorithm regardless of size of relations (just output all R and S blocks)
- ✓ **Set** union must remove duplicates
  - perform phase 1 of TPMMS on both R and S (make sorted sub-lists)
  - use one buffer for each sub-list of R and S
  - repeatedly, find first remaining tuple of all sub-lists
    - output tuple
    - discard duplicates from the front of the list

- ✓ Total cost is $3B(R) + 3B(S)$ disk I/Os
- ✓ Memory requirement
  - M buffers can make M block long sub-lists (in total)
  - $B(R) + B(S) \leq M^2 \rightarrow \sqrt{(B(R)+B(S))} \leq M$
  - if $B(R) + B(S) > M^2$ → more than M sub-lists, the algorithm will not work

Union (∪) : Two-Pass Hashing

- ✓ **Set** union two-pass hashing algorithm
  - ➢ Partition both R and S into M-1 buckets using same hash function
  - ➢ for all buckets, perform union on buckets *i* separately – $R_i \cup S_i$ – using one-pass set union
    - read smallest relation into M-1 buffers, say S_i, and copy every tuple to output
    - read the blocks holding R_i one-by-one into one buffer, and for each tuple see if it exists in S_i → if not, copy to output

- ✓ Total cost: 3B(R) + 3B(S) disk I/Os
  - ➢ 2 for partitioning the relations
  - ➢ 1 for performing union on different buckets

- ✓ Memory requirement: M buffers
  - ➢ M buffers can make M-1 buckets for each relation
  - ➢ for each bucket pair, R_i and S_i, either $B(R_i) \leq M-1$ or $B(S_i) \leq M-1$
  - ➢ approximately $\min(B(R), B(S)) \leq M^2$ → $\sqrt{\min(B(R), B(S))} \leq M$
  - ➢ if the smaller bucket of R_i and S_i does not fit in M-1 buffers, the algorithm will not work

Union ($\cup$) :
Cost and requirement summary for **R** $\cup$ **S:**

If $B_S \le B_R$:

BAG-version :

Algorithm	Memory Requirement	Disk I/Os
One-Pass	$M \ge 1$	$B_R + B_S$

SET-version :

Algorithm	Memory Requirement	Disk I/Os
One-Pass	$B_S \le M - 1$	$B_R + B_S$
Two-Pass Sorting	$M \ge \sqrt{B_R + B_S}$	$3B_R + 3B_S$
Two-Pass Hashing	$M \ge \sqrt{B_S}$	$3B_R + 3B_S$

Intersection (∩) : One-Pass

- ✓ **Bag** intersection (∩) can be implemented using a tuple counter:
  - ➢ read smallest relation, S, into M-1 buffers, but store only distinct tuples and the counter
  - ➢ read the blocks of R one-by-one and for each tuple see if it exists in S
    - ▪ if not, do nothing
    - ▪ otherwise and if counter > 0, copy to output and decrement counter
- ✓ Total cost B(R) + B(S) disk I/Os
- ✓ Memory requirement 1 + (M-1) = M, B(S) < M
  (additionally, we may need more memory to hold counters)

- ✓ **Set** intersection
  - ➢ read S into M-1 buffers and R block-by-block
  - ➢ if tuple t from R exists in S, output
- ✓ Same costs and memory requirement as bag-version
  (except set-version does not need to hold counters)

Intersection (∩) : Two-Pass Sorting

- ✓ Two-pass sorting intersection use an algorithm similar to TPMMS:
  - perform phase 1 of TPMMS on both R and S (make sorted sub-lists)
  - **bag**-version:
    output tuple t the minimum number of times it appears in R and in S
  - **set**-version:
    output tuple t if it occurs in both R and S

- ✓ Total cost is $3B(R) + 3B(S)$ disk I/Os
- ✓ Memory requirement
  - M buffers can make M block long sub-lists (totally)
  - $B(R) + B(S) \leq M^2 \rightarrow \sqrt{B(R)+B(S)} \leq M$
  - bag-version also needs room for counters
  - if $B(R) + B(S) > M^2$ → more than M sub-lists, the algorithm will not work

Intersection (∩) : Two-Pass Hashing

- ✓ Two-pass hashing intersection algorithm
  - ➢ Partition both R and S into M-1 buckets using same hash function
  - ➢ for all buckets, perform intersection on buckets *i* separately – $R_i \cap S_i$ – using either **bag**- or **set**-version of one-pass intersect
- ✓ Total cost: 3B(R) + 3B(S) disk I/Os
  - ➢ 2 for partitioning the relations
  - ➢ 1 for performing intersection on different buckets
- ✓ Memory requirement: M buffers
  - ➢ M buffers can make M-1 buckets for each relation
  - ➢ for each bucket pair, R_i and S_i, either $B(R_i) \leq M\text{-}1$ or $B(S_i) \leq M\text{-}1$
  - ➢ approximately $\min(B(R), B(S)) \leq M^2 \rightarrow \sqrt{}\min(B(R), B(S)) \leq M$
  - ➢ bag-version also needs room for counters
  - ➢ if the smaller bucket of R_i and S_i does not fit in M-1 buffers, the algorithm will not work

Intersection ($\cap$) : Cost and requirement summary for **R $\cap$ S:**

If $B_S \leq B_R$:

Algorithm	**Memory Requirement**[1]	**Disk I/Os**
One-Pass	$B_S \leq M - 1$	$B_R + B_S$
Two-Pass Sorting	$M \geq \sqrt{B_R + B_S}$	$3B_R + 3B_S$
Two-Pass Hashing	$M \geq \sqrt{B_S}$	$3B_R + 3B_S$

[1] **BAG**-version additionally needs memory buffers for tuple counters

Difference (–) : One-Pass – I

- ✓ **Bag** difference (–) can be implemented using a tuple counter:
  - ➢ read smallest relation, S, into M-1 buffers, but store only distinct tuples and the counter
  - ➢ S – R (tuples in S that do not exist in R):
    - read the blocks of R one-by-one and for each tuple existing in S, decrement associated counter
    - at the end, output tuples of which counter > 0 – counter number of times
  - ➢ R – S (tuples in R that do not exist in S):
    - read the blocks of R one-by-one and for each tuple, see if it exists in S
    - if no, copy the tuple to output
    - if yes, look at counter
      - o counter > 0, decrement counter
      - o counter = 0, output tuple

- ✓ Total cost B(R) + B(S) disk I/Os
- ✓ Memory requirement 1 + (M-1) = M, B(S) < M (additionally, we may need more memory to hold counters)

Difference (–) : One-Pass – II

- **Set** difference
  - read smallest relation, S, into M-1 buffers and R block-by-block
  - S – R:
    - if tuple t from R exists in S, delete t from S in memory
    - otherwise, do nothing
    - at the end, output all remaining tuples of S
  - R – S:
    - if tuple t from R exists in S, do nothing
    - otherwise, output t
- Same costs and memory requirement as bag-version (except set-version does not need to hold counters)

Difference (–) : Two-Pass Sorting

- ✓ Two-pass sorting difference uses an algorithm similar to TPMMS:
  - ➢ perform phase 1 of TPMMS on both R and S (make sorted sub-lists)
  - ➢ R – S:
    - **bag**-version:
      output tuple t the number of times it appears in R minus the number of times it appear in S
    - **set**-version:
      output tuple t if it occurs in R but not in S
  - ➢ S – R similarly (blocks from all sub-lists are in memory)

- ✓ Total cost is $3B(R) + 3B(S)$ disk I/Os
- ✓ Memory requirement
  - ➢ M buffers can make M block long sub-lists (totally)
  - ➢ $B(R) + B(S) \leq M^2 \rightarrow \sqrt{(B(R)+B(S))} \leq M$
  - ➢ if $B(R) + B(S) > M^2 \rightarrow$ more than M sub-lists, the algorithm will not work

Difference (–) : Two-Pass Hashing

- ✓ Two-pass hashing difference algorithm
  - partition both R and S into M-1 buckets using same hash function
  - for all buckets, perform difference on buckets *i* separately – R_i - S_i – using either **bag**- or **set**-version of one-pass difference

- ✓ Total cost: 3B(R) + 3B(S) disk I/Os
  - 2 for partitioning the relations
  - 1 for performing difference on different buckets

- ✓ Memory requirement: M buffers
  - M buffers can make M-1 buckets for each relation
  - for each bucket pair, R_i and S_i, either $B(R_i) \leq M-1$ or $B(S_i) \leq M-1$
  - approximately $\min(B(R), B(S)) \leq M^2$ → $\sqrt{\min(B(R), B(S))} \leq M$
  - **bag**-version also needs room for counters
  - if the smaller bucket of R_i and S_i does not fit in M-1 buffers, the algorithm will not work

Difference (–) : Cost and requirement summary for **R – S:**

If $B_S \leq B_R$:

Algorithm	**Memory Requirement**[1]	**Disk I/Os**
One-Pass	$B_S \leq M - 1$	$B_R + B_S$
Two-Pass Sorting	$M \geq \sqrt{B_R + B_S}$	$3B_R + 3B_S$
Two-Pass Hashing	$M \geq \sqrt{B_S}$	$3B_R + 3B_S$

[1] **BAG**-version additionally needs memory buffers for tuple counters

Natural Joins (⋈) : One-Pass

- ✓ Natural join (⋈) concatenates tuples from relation R(X,Y) with those tuples in S(Y,Z) where R.Y = S.Y
- ✓ One-pass algorithm:
  - ➢ read smallest relation, S, into M-1 buffers
  - ➢ read relation R block-by-block, and for each tuple t, concatenate t with matching tuples in S
→ move resulting joined tuples to output
- ✓ Total cost B(R) + B(S) disk I/Os
- ✓ Memory requirement 1 + (M-1) = M, B(S) < M

Natural Joins (⋈) :
Nested-Loop Joins – I

- ✓ Nested-loop joins can be used for relations of any size
- ✓ *Tuple-based* algorithm:

```
FOR each tuple s in relation S
    FOR each tuple r in R
        IF r and s join, concatenate to output
```

- ✓ Worst case of cost T(R)T(S) disk I/Os
  (can at least manage B(S) + B(S)B(R), more memory)

- ✓ Memory requirement 2 (hold R block and S block)

Natural Joins (⋈) :
Nested-Loop Joins – II

- *Block-based:*
  - use all tuples in a block
  - keep as much as possible of the smallest relation, S, in memory, i.e., M-1 blocks
  - algorithm:

```
FOR each M-1 sized partition p of relation S {
    read p into memory
    FOR each block b of R {
        read b into memory
        FOR each tuple t in b {
            find tuples in p that join with t
            join each of these with t to output }}}
```

actually only one pass through the tuples in R

- Total cost B(S) + [B(S)/(M-1)*B(R)] disk I/Os
  (Read S once, read R once for each partition of S)
- Memory requirement 2 (hold R block and S block)

Natural Joins (⋈) : Two-Pass Sorting – I

- ✓ There are several ways sorting can be used in join
- ✓ *Simple algorithm*, R ⋈ S:
  - ➢ sort R and S separately using TPMMS on join attribute(s), and write back to disk
  - ➢ join (merge) the sorted R and S, by repeatedly
    - ▪ if R or S buffers empty, fetch block(s) from disk
    - ▪ find tuples which have least value *v* for joining attribute (also on following blocks)
      - o if *v*–value tuples exist in both R and S, join R tuples with S tuples, write joined tuples to output
      - o otherwise, discard all *v*–value tuples

- ✓ Total cost: 5B(R) + 5B(S) disk I/Os
  - ➢ 4 for TPMMS
  - ➢ 1 of merging the sorted R and S
- ✓ Memory requirement: M buffers
  - ➢ must use TPMMS on both relations $B \leq M^2$, i.e., $B(R) \leq M^2$ AND $B(S) \leq M^2$
  - ➢ if there exists a collection of *v*–value tuples that does not fit in M memory blocks, the algorithm does not work

Natural Joins (⋈) :
Two-Pass Sorting – II

- ✓ *Sort-join algorithm*, R ⋈ S:
  - make M-sized, sorted sub-lists of R and S separately using first phase of TPMMS on join attribute
  - bring first block of each sub-list into memory
  - join the sorted R and S, by repeatedly
    - find tuples which have least value *v* for joining attribute (also on following blocks)
      - if *v* –value tuples exist in both R and S, join R tuples with S tuples, write joined tuples to output
      - otherwise, discard all *v* –value tuples
    - if a buffer is empty, retrieve new block (if any) from disk

- ✓ Total cost: 3B(R) + 3B(S) disk I/Os
  - 2 for first phase of TPMMS (making sub-lists)
  - 1 of merging the sorted R and S (join operation)
- ✓ Memory requirement: M buffers
  - must use first phase of TPMMS on both relations $B \leq M^2$, i.e., $B(R) + B(S) \leq M^2$ (cannot have more than M sub-lists)
  - the algorithm does not work if
    - there exists a collection of *v* –value tuples that does not fit in M memory blocks
    - there are more than M sub-lists totally

Natural Joins ($\bowtie$) : Two-Pass Hashing

- ✓ Two-pass hashing natural join algorithm
  - partition both R and S into M-1 buckets using same hash function
  - for all buckets, perform natural join on buckets *i* separately – $R_i \bowtie S_i$ – using one-pass join:
    - read smallest relation, S, into M-1 buffers
    - read relation R block-by-block, and for each tuple t, join t with matching tuples in S → move resulting tuples to output

- ✓ Total cost: 3B(R) + 3B(S) disk I/Os
  - 2 for partitioning the relations
  - 1 for performing join on different buckets

- ✓ Memory requirement: M buffers
  - M buffers can make M-1 buckets for each relation
  - for each bucket pair, R_i and S_i, either $B(R_i) \leq M-1$ or $B(S_i) \leq M-1$
  - approximately $\min(B(R), B(S)) \leq M^2$ → $\sqrt{\min(B(R), B(S))} \leq M$
  - if the smaller bucket of R_i and S_i does not fit in M-1 buffers, the algorithm will not work

Natural Joins (⋈) : Two-Pass Hybrid Hashing – I

- ✓ If we have more memory on the first pass – partitioning the relations – we can save some disk I/Os
- ✓ Two-pass *hybrid* hashing natural join algorithm
  - create *k* buckets, *k* << M
  - partition the smaller relation, S, but
    - keep entire first bucket in memory
    - partition buckets 2 .. *k* as normally
      - put tuples in corresponding bucket
      - if block full, write to disk
      - at the end, write all non-empty buckets to disk
  - partition the larger relation, R, but
    - tuples going to bucket R_1 are joined with corresponding tuples of S_1 which is kept in memory
    - remaining tuples are partitioned normally using the disk to hold the buckets
  - make a second pass using the algorithm described previously on buckets *i* separately – $R_i \bowtie S_i$ – using one-pass join on buckets 2 .. *k*

Natural Joins (⋈) :
Two-Pass Hybrid Hashing – II

- ✓ Total cost: $3B(R) + 3B(S) – 2B(R_1) – 2B(S_1)$ disk I/Os
  - two-pass hash joins take 3 disk I/Os per block
  - we save 2 disk I/Os for each block belonging to first bucket
  - approximate cost:
    - assume we can make the size of a bucket M (available memory)
      → k = B(S)/M for both R_1 and S_1 (we save about 2k reads, subtract 2/k)

      → 3 (B(R)+B(S)) – (2/k)(B(R) + B(S)) = (3 – 2/k)(B(R)+B(S)) =
      (3 – (2M/B(S)) (B(R)+B(S))

- ✓ Memory requirement: M buffers
  - M buffers must hold entire S_1 and k buckets, $M > B(S_1) + (k-1)$
  - for each bucket pair, R_i and S_i, i > 1, either $B(R_i) \leq M-1$ or $B(S_i) \leq M-1$
  - approximately $\min(B(R), B(S)) \leq M^2$ → $\sqrt{\min(B(R), B(S))} \leq M$
  - if the smaller bucket of R_i and S_i does not fit in M-1 buffers,
    the algorithm will not work

Natural Joins (⋈) : Index-Based

- ✓ Index natural join algorithm – R(X,Y) ⋈ S(Y,Z) :
  - ➢ assume index on join attribute Y for relation S
  - ➢ read each block of relation R, and for each tuple
    - find tuples in S with equal join attribute using the index on S
    - read corresponding blocks and output join of these tuples

- ✓ Total cost: ? disk I/Os
  - ➢ if R is clustered, we need B(R) disk I/Os, otherwise, up to T(R) to read all R-tuples
  - ➢ additionally, for *each tuple in R* we need to read corresponding S-tuples:
    - if index is clustered and sorted on Y: B(S) / V(S,Y)
    - if S in not sorted on Y: T(S) / V(S,Y)
    - we will use an average T(S) / V(S,Y)
  - ⇨ thus, reading tuples of S is the dominant cost: T(R)T(S) / V(S,Y)

Natural Joins (⋈) : Zig-Zag Index-Based

- ✓ Zig-zag index join algorithm – R(X,Y) ⋈ S(Y,Z) :
  - ➢ assume sorted index on join attribute Y for both relation R and S
  - ➢ for each value of Y in index of R
    - find tuples in S with equal search key using index on S
      - o if no equal tuples exist, just proceed
      - o if we have a match on join attribute, retrieve corresponding disk blocks from both relations, and output join tuples

- ✓ Total cost: ? disk I/Os
  - ➢ if both R and S are clustered and sorted on Y, we can be able to perform the join in B(R) + B(S) disk I/Os
  - ➢ complicating factors adding I/Os
    - fractions of R and S with equal Y value do not fit in memory
    - blocks containing several different tuples must be read several times
  - ➢ relations are not clustered, ...?

Natural Joins (⋈) : Cost and requirement summary for **R ⋈ S**:

If $B_S \leq B_R$:

Algorithm	Memory Requirement	Disk I/Os
One-Pass	$B_S \leq M - 1$	$B_R + B_S$
Tuple-Based Nested-Loop	$2 \leq M$	worst case $T_R T_S$, can do at $B_S + B_S B_R$
Block-Based Nested-Loop	$2 \leq M$	$B_S + [(B_S /M\text{-}1) \text{ x } B_R]$
Simple Two-Pass Sorting	$\sqrt{B_R} \leq M$	$5\ B_R + 5\ B_S$
Sort-Join	$\sqrt{B_R + B_S} \leq M$	$3\ B_R + 3\ B_S$
Hash-Join	$\sqrt{B_S} \leq M$	$3\ B_R + 3\ B_S$
Hybrid Hash-Join	$\sqrt{B_S} \leq M$	$(3–2M/B_S)(B_R + B_S)$
Index Join	$2 \leq M$	$B_R + (T_R B_S / V_{S,Y})$
Zig-Zag Index Join	$B(T_R/V_{R,a})+B(T_S/V_{S,a}) \leq M$	$B_R + B_S$

Natural Join Example – I

- ✓ Example:
  - ➢ T(R) = 10.000, T(S) = 5.000
  - ➢ V(R,a) = 100, V(S,a) = 10
  - ➢ Both R and S are clustered
  - ➢ 4 KB blocks (no block header)
  - ➢ both R and S records are 512 B (including header)
  - ➢ clustering index on attribute a for both R and S

⇨ B(S) = 5.000 / 8 = 625
B(R) = 10.000 / 8 = 1250

Natural Join Example – II

- ✓ Example (cont.):
  B(S) = 625, B(R) = 1250, V(R,a) = 100, V(S,a) = 10, T(R) = 10.000, T(S) = 5.000

  - **What is the minimum memory requirement for R(x,a) ⋈ S(a,y)?**

  - One-Pass:
    $\min(B(R), B(S)) \leq M - 1$ → $1 + 625 = 626$

  - Tuple-Based Nested-Loop:
    $2 \leq M$ → 2

  - Block-Based Nested-Loop:
    $2 \leq M$ → 2

  - Simple Two-Pass Sorting:
    $\sqrt{\max(B(R), B(S))} \leq M$ → $\sqrt{1250} = 35.35 \approx 36$

  - Sort-Join:
    $\sqrt{B(R) + B(S)} \leq M$ → $\sqrt{625 + 1250} = 43.30 \approx 44$

Natural Join Example – III

- ✓ Example (cont.):
  B(S) = 625, B(R) = 1250, V(R,a) = 100, V(S,a) = 10, T(R) = 10.000, T(S) = 5.000

  - **What is the minimum memory requirement for R(x,a) $\bowtie$ S(a,y)?**

  - Hash-Join:
    $\sqrt{\min(B(R), B(S))} \leq M$ → $\sqrt{625} = 25$

  - Hybrid Hash-Join:
    $\sqrt{\min(B(R), B(S))} \leq M$ → $\sqrt{625} = 25$

  - Index Join:
    $2 \leq M$ → 2

  - Zig-Zag Index Join:
    $B(T(R)/V(R,a))+B(T(S)/V(S,a) \leq M$ → 10.000/100/8 + 5.000/10/8 =
    12,5 + 62,5 ≈ 13 + 63 = 76

Natural Join Example – IV

- ✓ Example (cont.): **R(x,a) ⋈ S(a,y)**
  - ➢ assume now available memory M = 101 blocks

    T(R) = 10.000, T(S) = 5.000, B(R) = 1250, B(S) = 625, M = 101
  - ➢ **what is the cost in disk I/Os for the different algorithms?**
  - ➢ One-Pass:
    B(R) + B(S) → 1250 + 625 = 1875

    (but one-pass cannot be performed, because memory requirement is 626)
  - ➢ Tuple-Based Nested-Loop:
    min(B(R), B(S)) + B(S)B(R) → 625 + 625 * 1250 = 781875
  - ➢ Block-Based Nested-Loop:
    min(B(R), B(S)) + [(min(B(R), B(S)) / (M-1)) * max(B(R), B(S))]
    → 625 + (625/(101-1) * 1250) = 9375
  - ➢ Simple Two-Pass Sorting:
    5 B(R) + 5 B(S) → 1250 * 5 + 625 * 5 = 9375
  - ➢ Sort-Join:
    3 B(R) + 3 B(S) → 1250 * 3 + 625 * 3 = 5625

Natural Join Example – II

- ✓ Example (cont.): **R(x,a) ⋈ S(a,y)**

  T(R) = 10.000, T(S) = 5.000, B(R) = 1250, B(S) = 625, M = 101, V(R,a) = 100, V(S,a) = 10

  - **what is the cost in disk I/Os for the different algorithms?**
  - Hash-Join:
    3 B(R) + 3 B(S) → 1250 * 3 + 625 * 3 = 5625
  - Hybrid Hash-Join:
    (3–2M/min(B(R), B(S)))(B(R) + B(S)) → (3 – (2*101)/625) * (1250 + 625) = 5019
  - Index Join:
    - index on S: B(R) + (T(R)B(S) / V(S,a)) → 1250 + (10.000 * 625 / 10) = 626250
    - index on R: B(S) + (T(S)B(R) / V(R,a)) → 625 + (5.000 * 1250 / 100) = 63125
  - Zig-Zag Index Join (index on both R and S):
    B(R) + B(S) → 625 + 1250 = 1875

Natural Join Example – II

- ✓ Example summary:

  T(R) = 10.000, T(S) = 5.000, B(R) = 1250, B(S) = 625, M = 101

Algorithm	Minimum Memory	Disk I/Os
One-Pass	626	1875
Tuple-Based Nested-Loop	2	781875
Block-Based Nested-Loop	2	9375
Simple Two-Pass Sorting	36	9375
Sort-Join	44	5625
Hash-Join	25	5625
Hybrid Hash-Join	25	5019
Index Join	2	626250 (S-index) 63125 (R-index)
Zig-Zag Index Join	76	1875

Which Algorithm Should I Choose?

- ✓ One-Pass algorithms are great if one of the arguments (relations) fits in memory
- ✓ Two-Pass algorithms must be used if we have large relations
  - Hash-based algorithms
    - require less memory compared to sorting approaches – only dependent of the smallest relation – often used
    - assume approximately equal bucket size (good hash function) – in real life there will be a small variation, must assume smaller bucket sizes
  - Sort-based algorithms
    - produce a sorted result, which can be used in successive operators again using sort-based algorithms
  - Index-based algorithms
    - excellent for selections and for joins if both have clustered indexes
- ✓ They all benefit from optimized disk block layout reducing seeks and rotational delays, more buffers, ...

Further Extensions and Other Factors Influencing Cost

N–Pass Algorithms

- ✓ Our algorithms so far make one or two passes over the entire data set
- ✓ If a relation gets really big, this is not sufficient
- ✓ Example: B(R) = 1.000.000
  - ➢ TPMMS require that $B(R) < M^2$ → M > 1000
  - ➢ if 1000 blocks not available, TPMMS does not work
  - ⇨ must add more passes over the data set
- ✓ Sort-based algorithms:
  - ➢ if R fits in memory, sort
  - ➢ if not, partition R into M groups and recursively sort each R_i
  - ➢ merge the sub-lists
  - ➢ total cost: $(2k - 1)B(R)$, k is the number of passes needed
  - ➢ we need $\sqrt[k]{B(R)}$ memory buffers, i.e., $B(R) \leq M^k$
- ✓ There exists a similar recursive approach using hashing

Buffer Management

- ✓ The *buffer manager* controls and manages available memory
  - ➢ if we get too few memory buffers for an algorithm to work properly, we will pay a significant penalty due to "thrashing"
  - ➢ when a new buffer is needed, the buffer manager replaces an old one according to an appropriate *replacement policy* (often based on reference locality in space and time)
  - ➢ the query optimizer will select a set of physical operators that will be used to execute the query
    - the amount of available memory might vary from query to query
    - must make an algorithm selection each time
    - "wrong" selection may lead to "thrashing" or "degradation" (e.g., change algorithm from one-pass to two-pass)

Parallel Algorithms

- ✓ Database operations can in general benefit from parallel processing

- ✓ Tuple-at-a-time operations:
  - ➢ if there are p processors, divide relation R into p equal partitions and distribute
  - ➢ each processor performs the operation on its own subset of the tuples
  - ➢ processing time: 1/*p* compared to a single-processor system (but we must add time for shipping data to remote machines)
  - ➢ same amount of disk I/Os in total (but more fragmentation)

Parallel Algorithms

- ✓ Full relation operations (join):
  - ➢ if there are *p* processors, partition relation R and S using the same hash function on both R ans S' join attributes, hash into *p* buckets, i.e., all join tuples are sent to same bucket
  - ➢ ship bucket R_i and S_i to processor *i*
  - ➢ perform join on each processor on each pair of buckets using any of the uniprocessor joins we have looked at
  - ➢ total cost:
    - perform hash-partitioning on main machine, but ship full bucket-blocks to corresponding remote machine – B(S) + B(R)
    - store bucket on disk on local or remote machine – B(S) + B(R)
    - perform any two-pass join algorithm – 3B(R) + 3B(S)
    - ⇨ total number of disk I/Os: **5B(R) + 5B(S)**

    - However, only 1/p of all blocks is at each machine – p partitions are retrieved in parallel → time: B(R) + B(S) + (4B(R) + 4 B(S))/p
    - Additionally,
      - o each bucket may now be small enough to fit in memory → does not need any of the remote site disk I/Os: B(R) + B(S)
      - o at least one of the buckets may fit in memory → store and retrieve the larger bucket, say R: B(R) + B(S) + 2B(R)/p

Summary

- ✓ Model for computing costs
  → counting number of disk I/O according to available memory

- ✓ Cost of basic operations
  - table scans
  - sorting
  - bucket-partitioning

- ✓ Implementation algorithms and their costs
  - tuple-at-a-time, unary operations
  - full-relation, unary operations
  - full-relation, binary operations