Chapter 8: Query Execution

Zhaonian Zou

Massive Data Computing Research Center
School of Computer Science and Technology
Harbin Institute of Technology, China
Email: znzou@hit.edu.cn

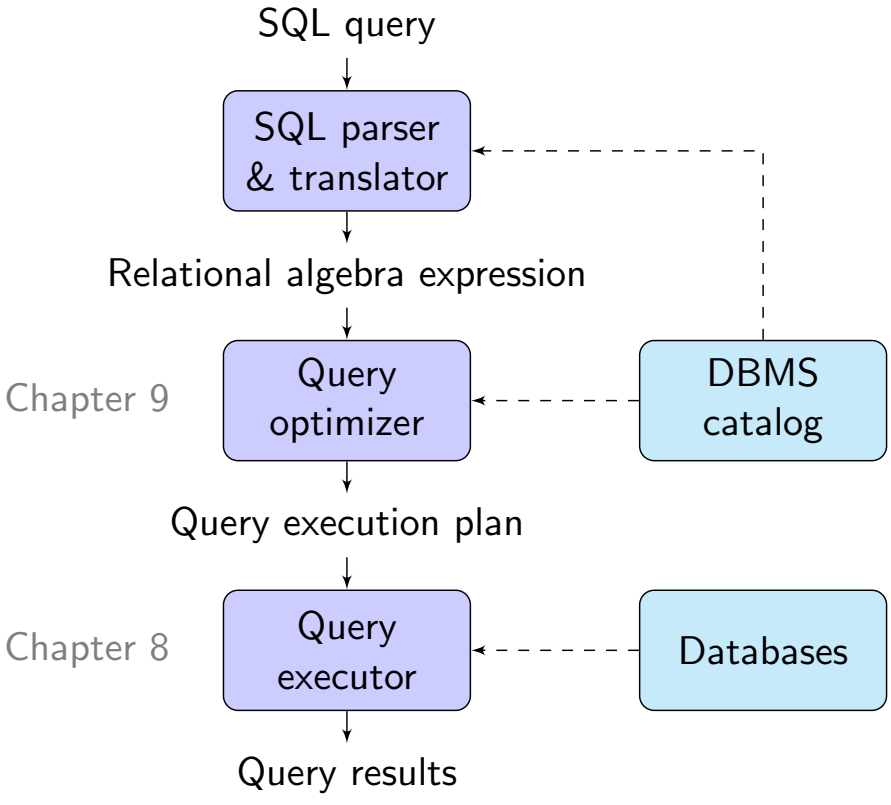
Spring 2019

Outline[1]

[1] Updated on April 9, 2019

8.1 Overview of Query Processing

Overview of Query Processing



SQL Parser & Translator

The SQL parser and transaltor transform an SQL query to an relational algebra expression

Example (Query Translation)

- Relation: instructor(ID, name, dept_name, salary)
- SQL query:

```
SELECT ID, salary FROM instructor WHERE salary < 75000;
```

- Relational algebra expression:

$$\Pi_{\texttt{ID,salary}}(\sigma_{\texttt{salary}<75000}(\texttt{instructor}))$$

Query Optimizer

The query optimizer transforms an initial relational algebra expression to an optimized relational algebra expression and finally to a physical query execution plan

Example (Query Optimization)

Initial expression	Alternative expression	Query execution plan
$\Pi_{ID,salary}$	$\sigma_{salary<75000}$	$\Pi_{ID,salary}$
$\sigma_{salary<75000}$	$\Pi_{ID,salary}$	$\sigma_{salary<75000}$; use index
instructor	*instructor*	*instructor*

8.2 External Sort

Uses of Sorting in a DBMS

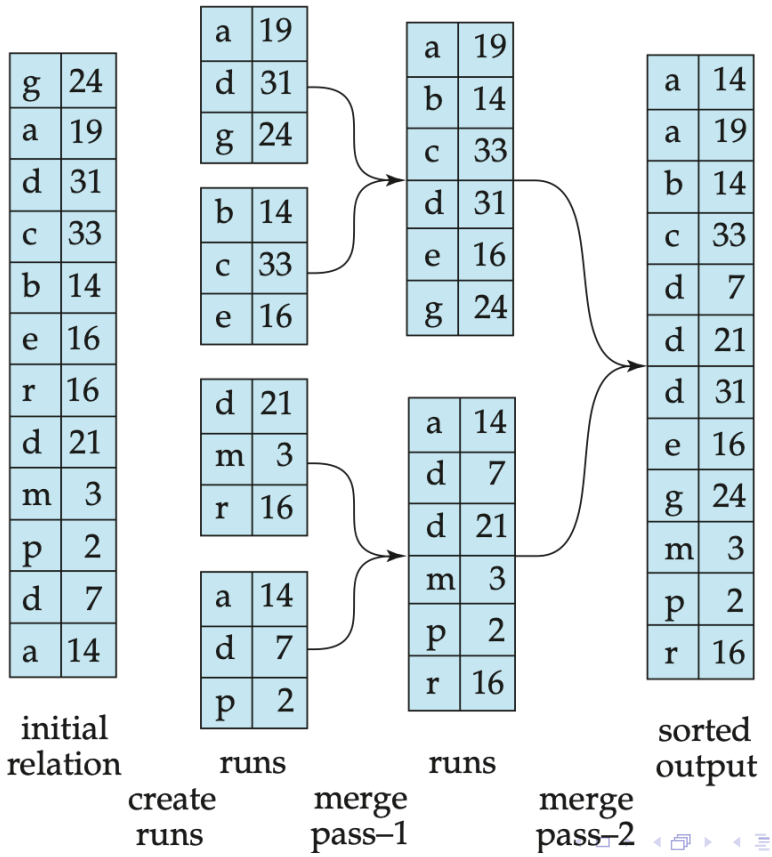
Sorting a collection of records on some search key (a single attribute or an ordered list of attributes) is a very useful operation in a DBMS

- Users may want answers in some order using `ORDER BY`
- Sorting records is the first step in bulk loading a B+ tree
- Sorting is useful for eliminating duplicate records
- The Sort-Merge-Join algorithm requires a sorting step

When the data to be sorted is too large to fit into available main memory, we need an external sorting (外存排序) algorithm, which seeks to minimize the cost of disk accesses

8.2 External Sort
External Merge Sort

External Merge Sort (外存归并排序)



Two-Pass Multiway External Merge Sort (两趟多路外存归并排序)

Example (Two-Pass Multiway External Merge Sort)

- $R =$ | 2 5 | 2 1 | 2 2 | 4 5 | 4 3 | 4 2 | 1 5 | 2 1 | 3 |, so $N = 17$ (# of records, here each integer represents a record)
- $M = 3$ (# of pages available in the buffer pool)
- $B = 2$ (only B tuples fit on a block)
- Requirement: $N \leq BM^2$

Pass 0: Read in M pages at a time and sort internally to produce $\lceil N/M \rceil$ sublists (or runs) of M pages each (except for the last sublist, which may contain fewer pages)

- Read | 2 5 | 2 1 | 2 2 | and write sublist $R_1 =$ | 1 2 | 2 2 | 2 5 |
- Read | 4 5 | 4 3 | 4 2 | and write sublist $R_2 =$ | 2 3 | 4 4 | 4 5 |
- Read | 1 5 | 2 1 | 3 | and write sublist $R_3 =$ | 1 1 | 2 3 | 5 |

Two-Pass Multiway External Merge Sort (Cont'd)

Pass 1: Read in one page for each of the $\lceil N/B \rceil$ sorted sublists and do a $\lceil N/B \rceil$-way merge of the records in the loaded pages

Sublist	In memory	Waiting on disk
R_1	1 2	2 2 | 2 5
R_2	2 3	4 4 | 4 5
R_3	1 1	2 3 | 5

Output: | 1 1 | 1 | (Red integers indicate the outputted records)

Sublist	In memory	Waiting on disk
R_1	2	2 2 | 2 5
R_2	2 3	4 4 | 4 5
R_3	2 3	5

Output: | 1 1 | 1 2 | 2 2 |

Two-Pass Multiway External Merge Sort (Cont'd)

Pass 1:

Sublist	In memory	Waiting on disk
R_1	2 2	2 5
R_2	3	4 4 | 4 5
R_3	3	5

Output: 1 1 | 1 2 | 2 2 | 2 2

Sublist	In memory	Waiting on disk
R_1	2 5	
R_2	3	4 4 | 4 5
R_3	3	5

Output: 1 1 | 1 2 | 2 2 | 2 2 | 2

Two-Pass Multiway External Merge Sort (Cont'd)

Pass 1:

Sublist	In memory	Waiting on disk
R_1	5	
R_2	3	4 4 | 4 5
R_3	3	5

Output: 1 1 | 1 2 | 2 2 | 2 2 | 2 3 | 3

Sublist	In memory	Waiting on disk
R_1	5	
R_2	4 4	4 5
R_3	5	

Output: 1 1 | 1 2 | 2 2 | 2 2 | 2 3 | 3 4 | 4

Two-Pass Multiway External Merge Sort (Cont'd)

Pass 1:

Sublist	In memory	Waiting on disk
R_1	5	
R_2	4 5	
R_3	5	

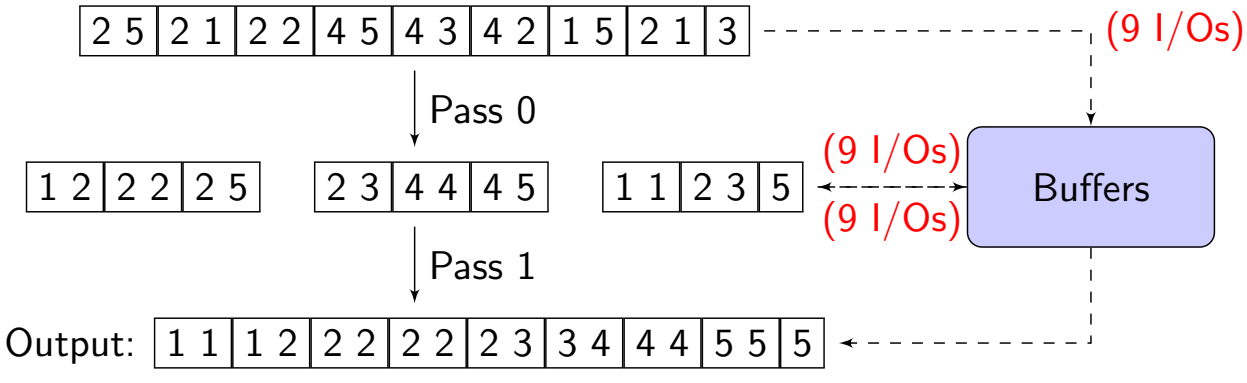
Output: | 1 1 | 1 2 | 2 2 | 2 2 | 2 3 | 3 4 | 4 4 |

Sublist	In memory	Waiting on disk
R_1	5	
R_2	5	
R_3	5	

Output: | 1 1 | 1 2 | 2 2 | 2 2 | 2 3 | 3 4 | 4 4 | 5 5 | 5 |

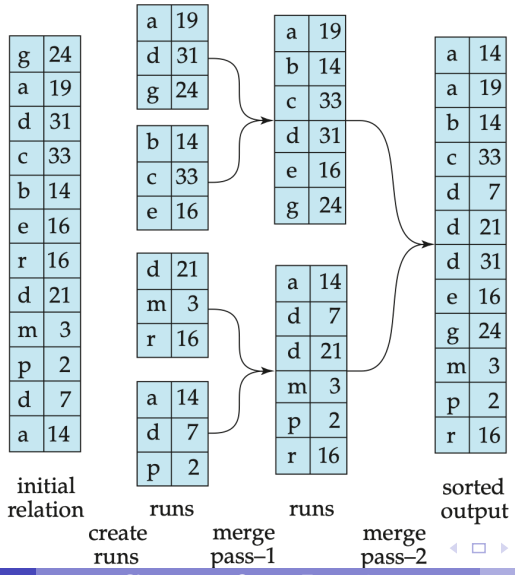
Two-Pass Multiway External Merge Sort (Cont'd)

- I/O cost: $3N/B$
  - In pass 0, each data record is read and written only once, so the number of I/Os is $2N/B$
  - In pass 1, each sublist is scanned only once, so the number of I/Os is also N/B
- Memory requirement: $N \leq BM^2$
  - There are no more than M sublists
  - Each sublist contains at most BM records



Multi-Pass Multiway External Merge Sort (多趟多路外存归并排序)

- If $N > BM^2$, merge sort cannot be done in two passes
- In this situation, we need a multi-pass multiway external merge sort algorithm
- The I/O cost is mN/B, where m is the number of passes



Optimization 1: Double Buffering (双缓冲)

Problem: When all the records in an input block of a run have been consumed, an I/O request is issued for the next block of records in the run, and the execution is forced to suspend until the I/O is complete

Sublist	In memory	Waiting on disk
R_1	1 2	2 2 | 2 5
R_2	2 3	4 4 | 4 5
R_3	1 1	2 3 | 5

Basic Idea

- Keep the CPU busy while an I/O request is being carried out
- Current hardware supports such overlapped computation

Optimization 1: Double Buffering (Cont'd)

Method (Double Buffering)

- Allocate extra pages to each input buffer
- When all the records in a block have been consumed, the CPU can process the next block of the run by switching to the second, "double", block for this run
- Meanwhile, an I/O request is issued to fill the empty block

Sublist	In memory		Waiting on disk
R_1	1 2	2 2	2 5
R_2	2 3	4 4	4 5
R_3	1 1	2 3	5

block currently merged double block

Optimization 1: Double Buffering (Cont'd)

Sublist	In memory		Waiting on disk
R_1	1 2	2 2	2 5
R_2	2 3	4 4	4 5
R_3	1 1	2 3	5

Output: 1 1 | 1

Sublist	In memory		Waiting on disk
R_1	2	2 2	2 5
R_2	2 3	4 4	4 5
R_3	5	2 3	5

Output: 1 1 | 1 2 | 2 2

Optimization 2: Blocked I/O

In passes $i = 1, 2, \ldots$ runs are read and written in units of b consecutive blocks, where $b \geq 1$ is called a blocking factor

- Cons: In each pass, $\lfloor M/b \rfloor$ runs are merged. When b is large, the number of passes is also large.
- Pros: Issuing a single request to read (or write) b consecutive pages can be much cheaper than reading (or writing) the same number of pages through independent I/O requests

Sublist	In memory		Waiting on disk
R_1	1 2	2 2	2 5 ...
R_2	2 3	4 4	4 5 ...
R_3	1 1	2 3	5 5 ...

8.3 Execution of Operations of Relational Algebra

8.3 Execution of Operations of Relational Algebra
8.3.1 Execution of Selection

Execution of Selection $\sigma_C(R)$

- Scanning-based selection
- Hash-based selection
- Index-based selection

Notation

- $B(R)$: the number of blocks of R
- $T(R)$: the number of tuples of R
- M: the number of main-memory buffers available
- $V(R, A)$: the number of unique values of attribute(s) A in R

Scanning-based Selection

Algorithm (Scanning-based Selection)

1. Read the blocks of R one at a time into an input buffer
2. Check the condition C on each tuple
3. Move the selected tuples to the output buffer

I/O Cost

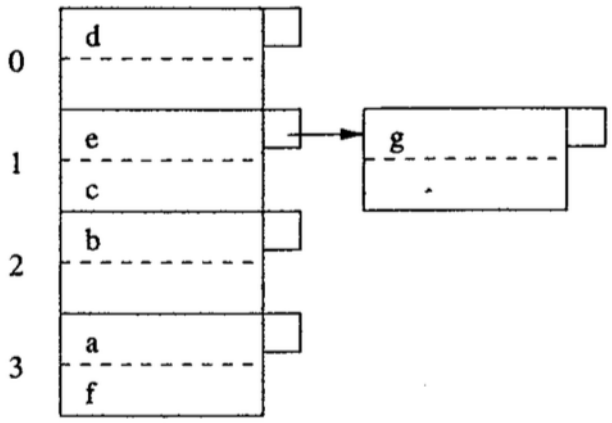
- $B(R)$ if R is clustered
- $T(R)$ if R is not clustered

Memory Requirement

- $M \geq 1$ for the input buffer, regardless of $B(R)$
- Since the output buffer may be an input buffer of some other operator, we do not count the output buffer as needed space

Hash-based Selection

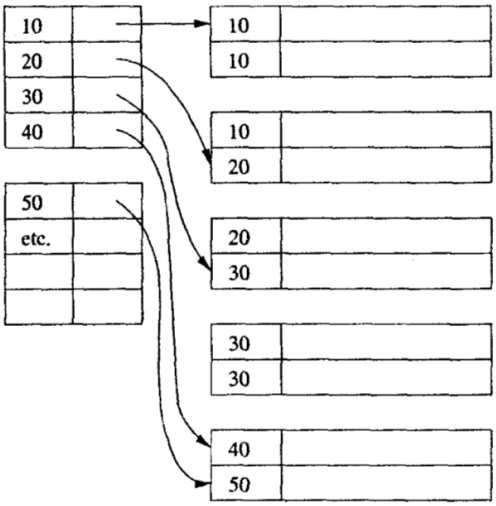
- Condition C is of the form $A = v$
- Relation R is stored in a hash table
- Attribute A is the hash key of R



- I/O cost: the number of blocks of bucket $h(v) \approx \lceil B(R)/V(R,A) \rceil$
- Memory requirement: $M \geq 1$ for the input buffer

Index-based Selection

- Condition C is of the form $A = v$ or $l \leq A \leq u$
- There is an index on R with search key A



- I/O cost $\approx \lceil B(R)/V(R,A) \rceil$ if the index is clustered
- I/O cost $\approx \lceil T(R)/V(R,A) \rceil$ if the index is non-clustered
- Memory requirement: $M \geq 1$ for the input buffer

8.3 Execution of Operations of Relational Algebra
8.3.2 Execution of Projection

Execution of Projection w/o Duplicate Elimination

Algorithm

1. Read the blocks of R one at a time into an input buffer
2. Project each tuple
3. Move the projected tuples to the output buffer

I/O cost

- $B(R)$ if R is clustered
- $T(R)$ if R is nonclustered

Memory requirement

- $M \geq 1$ for the input buffer, regardless of $B(R)$

8.3 Execution of Operations of Relational Algebra
8.3.3 Execution of Duplicate Elimination

Execution of Duplicate Elimination $\delta(R)$

- One-pass duplicate elimination
- Sort-based duplicate elimination
- Hash-based duplicate elimination

One-Pass Duplicate Elimination

Algorithm

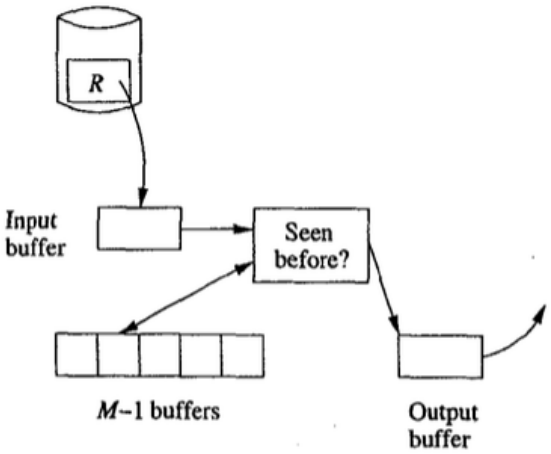
1. Read the blocks of R one at a time into an input buffer
2. For each tuple in the block, we copy it to the output buffer if it has not been seen before

I/O cost

- $B(R)$ if R is clustered
- $T(R)$ if R is nonclustered

Memory requirement

- $B(\delta(R)) \leq M - 1$



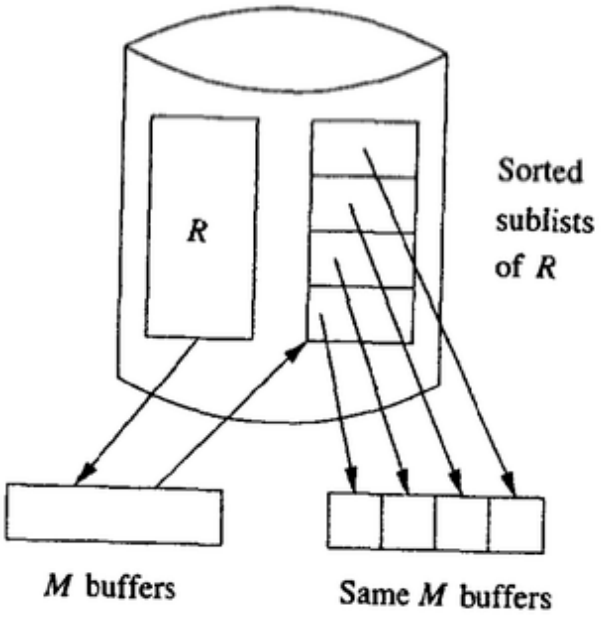
One-Pass Duplicate Elimination (Cont'd)

Example (One-Pass Duplicate Elimination)

- $R = \{2, 5, 2, 1, 2, 2, 4, 5, 4, 3, 4, 2, 1, 5, 2, 1, 3\}$
- $M = 4$
- Only two tuples fit on a block
- Let us check this example on the blackboard

Sort-based Duplicate Elimination

The algorithm is the same as the Two-Pass Multiway Merge Sort except that we do duplicate elimination instead of sorting in the second phase.



Sort-based Duplicate Elimination (Cont'd)

Example (Sort-based Duplicate Elimination)

- $R =$ |2 5|2 1|2 2|4 5|4 3|4 2|1 5|2 1|3|, so $N = 17$ (# of records, here each integer represents a record)
- $M = 3$ (# of pages available in the buffer pool)
- $B = 2$ (only B tuples fit on a block)
- Requirement: $N \leq BM^2$

Pass 0: Read in M pages at a time and sort internally to produce $\lceil N/M \rceil$ sublists (or runs) of M pages each (except for the last sublist, which may contain fewer pages)

- Read |2 5|2 1|2 2| and write sublist $R_1 =$ |1 2|2 2|2 5|
- Read |4 5|4 3|4 2| and write sublist $R_2 =$ |2 3|4 4|4 5|
- Read |1 5|2 1|3| and write sublist $R_3 =$ |1 1|2 3|5|

Sort-based Duplicate Elimination (Cont'd)

Pass 1: Read in one page for each of the $\lceil N/B \rceil$ sorted sublists and eliminate duplicate records in the loaded pages

Sublist	In memory	Waiting on disk
R_1	1 2	2 2 | 2 5
R_2	2 3	4 4 | 4 5
R_3	1 1	2 3 | 5

Output: |1| (Red integers indicate the duplicate records)

Sublist	In memory	Waiting on disk
R_1	2	2 2 | 2 5
R_2	2 3	4 4 | 4 5
R_3	2 3	5

Output: |1 2|

Sort-based Duplicate Elimination (Cont'd)

Pass 1:

Sublist	In memory	Waiting on disk
R_1	2 2	2 5
R_2	3	4 4 | 4 5
R_3	3	5

Output: 1 2

Sublist	In memory	Waiting on disk
R_1	2 5	
R_2	3	4 4 | 4 5
R_3	3	5

Output: 1 2

Sort-based Duplicate Elimination (Cont'd)

Pass 1:

Sublist	In memory	Waiting on disk
R_1	5	
R_2	3	4 4 | 4 5
R_3	3	5

Output: 1 2 \| 3

Sublist	In memory	Waiting on disk
R_1	5	
R_2	4 4	4 5
R_3	5	

Output: 1 2 \| 3 4

Sort-based Duplicate Elimination (Cont'd)

Pass 1:

Sublist	In memory	Waiting on disk
R_1	5	
R_2	4 5	
R_3	5	

Output: 1 2 | 3 4

Sublist	In memory	Waiting on disk
R_1	5	
R_2	5	
R_3	5	

Output: 1 2 | 3 4 | 5

Sort-based Duplicate Elimination (Cont'd)

- I/O cost: $3B(R)$
  - $B(R)$ I/O's to read each block of R when creating the sorted sublists
  - $B(R)$ I/O's to write each of the sorted sublists to disk
  - $B(R)$ I/O's to read each block from the sublists
- Memory requirement: $B(R) \leq M^2$
  - Each sublist consists of at most M blocks
  - There are no more than M sublists

Hash-based Duplicate Elimination

Algorithm (Hash-based Duplicate Elimination)

1. Read the blocks of R one at a time, and hash the tuples of R to $M-1$ buckets. Two copies of the same tuple will hash to the same bucket.
2. For each bucket R_i, perform the one-pass duplicate elimination algorithm on R_i, and write out the resulting unique tuples.

- I/O cost: $3B(R)$
- Memory requirement: $B(R) \leq (M-1)^2$

Hash-based Duplicate Elimination (Cont'd)

Example (Hash-based Duplicate Elimination)

- $R = \{2, 5, 2, 1, 2, 2, 4, 5, 4, 3, 4, 2, 1, 5, 2, 1, 3\}$
- $M = 4$
- Only 2 blocks fit on a block.
- $h(K) = K \mod 3$

Buckets:

- $R_0 = \{3, 3\}$
- $R_1 = \{1, 4, 4, 4, 1, 1\}$
- $R_2 = \{2, 5, 2, 2, 2, 5, 2, 5, 2\}$

For $0 \leq i \leq 2$, we use the one-pass algorithm on R_i

8.3 Execution of Operations of Relational Algebra
8.3.4 Execution of Aggregation Operations

Execution of Aggregation Operations

- One-pass aggregation
- Sort-based aggregation
- Hash-based aggregation

- The aggregation algorithms are similar to the duplicate elimination algorithms

8.3 Execution of Operations of Relational Algebra
8.3.5 Execution of Set Operations

Execution of Set Operations

- One-Pass Set Union/Intersection/Difference
- Hash-based Set Union/Intersection/Difference
- Sort-based Set Union/Intersection/Difference

One-Pass Set Union $R \cup S$

Assume S is smaller than R, and S fits in maim memory

Algorithm (One-Pass Set Union)

1. Read S into $M-1$ input buffers and build a search structure (hash table or balanced binary search tree) where the search key is the entire tuple
2. Copy all tuples of S to the output
3. Read the blocks of R one at a time into the Mth input buffer
4. For each tuple $t \in R$, we copy t to the output if $t \notin S$

- I/O cost: $B(R) + B(S)$
- Memory requirement: $\min(B(R), B(S)) \leq M - 1$

One-Pass Set Union $R \cup S$ (Cont'd)

Example (One-Pass Set Union)

- $R = \{1, 5, 8, 2, 3, 10, 4, 7, 6, 9\}$
- $S = \{4, 11, 9, 5, 7, 3, 6, 12, 8, 10\}$
- $M = 6$
- Only 2 records fit on a block

Order	The Mth block	Output
1	1, 5	1
2	8, 2	2
3	3, 10	
4	4, 7	
5	6, 9	

One-Pass Set Intersection $R \cap S$

Assume S is smaller than R and S fits in maim memory

Algorithm (One-Pass Set Intersection)

1. Read S into $M-1$ input buffers and build a search structure where the search key is the entire tuple
2. Read the blocks of R one at a time into the Mth input buffer
3. For each tuple $t \in R$, we copy t to the output if $t \in S$

- I/O cost: $B(R) + B(S)$
- Memory requirement: $\min(B(R), B(S)) \leq M - 1$

One-Pass Set Difference $R - S$

Algorithm (One-Pass Set Difference)

1. Read S into $M-1$ input buffers and build a search structure where the search key is the entire tuple
2. Read the blocks of R one at a time into the Mth input buffer
3. For each tuple $t \in R$, we copy t to the output if $t \notin S$

- I/O cost: $B(R) + B(S)$
- Memory requirement: $B(S) \leq M - 1$

Hash-based Set Union $R \cup S$

Algorithm (Hash-based Set Union)

1. Hash R into $M-1$ buckets $R_1, R_2, \ldots, R_{M-1}$
2. Hash S into $M-1$ buckets $S_1, S_2, \ldots, S_{M-1}$
3. For $1 \leq i \leq M-1$, use the one-pass algorithm to compute $R_i \cup S_i$, and write out the resulting tuples

- I/O cost: $3B(R) + 3B(S)$
- Memory requirement: $\min(B(R), B(S)) \leq (M-1)^2$

Hash-based Set Union $R \cup S$ (Cont'd)

Example (Hash-based Set Union)

- $R = \{1, 5, 8, 2, 3, 10, 4, 7, 6, 9\}$
- $S = \{4, 11, 9, 5, 7, 3, 6, 12, 8, 10\}$
- $M = 4$
- Only 2 blocks fit on a block
- $h(K) = K \mod 3$

Buckets:

- $R_0 = \{3, 6, 9\}$
- $R_1 = \{1, 10, 4, 7\}$
- $R_2 = \{5, 2, 8\}$
- $S_0 = \{9, 3, 6, 12\}$
- $S_1 = \{4, 7, 10\}$
- $S_2 = \{11, 5, 8\}$

For $0 \leq i \leq 2$, we use the one-pass algorithm to compute $R_i \cup S_i$

Hash-based Set Intersection $R \cap S$

Algorithm (Hash-based Set Intersection)

1. Hash R into $M-1$ buckets $R_1, R_2, \ldots, R_{M-1}$
2. Hash S into $M-1$ buckets $S_1, S_2, \ldots, S_{M-1}$
3. For $1 \leq i \leq M-1$, use the one-pass algorithm to compute $R_i \cap S_i$, and write out the resulting tuples

- I/O cost: $3B(R) + 3B(S)$
- Memory requirement: $\min(B(R), B(S)) \leq (M-1)^2$

Hash-based Set Difference $R - S$

Algorithm (Hash-based Set Difference)

1. Hash R into $M-1$ buckets $R_1, R_2, \ldots, R_{M-1}$
2. Hash S into $M-1$ buckets $S_1, S_2, \ldots, S_{M-1}$
3. For $1 \leq i \leq M-1$, use the one-pass algorithm to compute $R_i - S_i$, and write out the resulting tuples

- I/O cost: $3B(R) + 3B(S)$
- Memory requirement: $\min(B(R), B(S)) \leq (M-1)^2$

Sort-based Set Union $R \cup S$

Algorithm (Sort-based Set Union)

1. Create sorted sublists from both R and S
2. Use one main-memory buffer for each sublist of R and S. Initialize each with the first block from the corresponding sublist
3. Repeatedly find the least remaining tuple t among all the buffers. Copy t to the output and remove from the buffers all copies of t

- I/O cost: $3B(R) + 3B(S)$
- Memory requirement: $B(R) + B(S) \leq M^2$

Sort-based Set Union $R \cup S$ (Cont'd)

Example (Sort-based Set Union)

- $R = \{1, 5, 8, 2, 3, 10, 4, 7, 6, 9\}$
- $S = \{4, 11, 9, 5, 7, 3, 6, 12, 8, 10\}$
- $M = 4$, and only 2 blocks fit on a block

Sublists:

- $R_1 = \{1, 2, 3, 4, 5, 7, 8, 10\}$
- $R_2 = \{6, 9\}$
- $S_1 = \{3, 4, 5, 6, 7, 9, 11, 12\}$
- $S_2 = \{8, 10\}$

Sublist	In memory	Waiting on disk
R_1	1 2	3 4, 5 7, 8 10
R_2	6 9	
S_1	3 4	5 6, 7 9, 11 12
S_2	8 10	

Let us continue the example on blackboard

Sort-based Set Intersection $R \cap S$

Algorithm (Sort-based Set Intersection)

1. Create sorted sublists from both R and S
2. Use one main-memory buffer for each sublist of R and S. Initialize each with the first block from the corresponding sublist
3. Repeatedly find the least remaining tuple t among all the buffers. Copy t to the output if t appears in both R and S and remove from the buffers all copies of t

- I/O cost: $3B(R) + 3B(S)$
- Memory requirement: $B(R) + B(S) \leq M^2$

Sort-based Set Difference $R - S$

Algorithm (Sort-based Set Difference)

1. Create sorted sublists from both R and S
2. Use one main-memory buffer for each sublist of R and S. Initialize each with the first block from the corresponding sublist
3. Repeatedly find the least remaining tuple t among all the buffers. Copy t to the output if t appears in R but not in S and remove from the buffers all copies of t

- I/O cost: $3B(R) + 3B(S)$
- Memory requirement: $B(R) + B(S) \leq M^2$

8.3 Execution of Operations of Relational Algebra
8.3.6 Execution of Joins

Execution of Joins

Consider natural joint $R(X, Y) \bowtie S(Y, Z)$

- One-Pass Joins
- Nested-Loop Joins
- Sort-based Joins
- Hash-Joins
- Index-based Joins

One-Pass Joins

Assume $S(Y, Z)$ is smaller than $R(X, Y)$

Algorithm (One-Pass Joins)

1. Read S into $M - 1$ input buffers and build a search structure where the search key is Y
2. Read the blocks of R one at a time into the Mth input buffer
3. For each tuple $t \in R$, find the tuples of S that agree with t on all attributes of Y, using the search structure. For each matching tuple of S, form a tuple by joining it with t, and move the resulting tuple to the output

- I/O cost: $B(R) + B(S)$
- Memory requirement: $\min(B(R), B(S)) \leq M - 1$

One-Pass Joins (Cont'd)

Example (One-Pass Joins)

- $R(X, Y) = \{(1, 1), (5, 5), (3, 2), (3, 1), (2, 1), (4, 2)\}$
- $S(Y, Z) = \{(2, 6), (1, 7), (1, 8), (2, 5), (2, 7)\}$
- $M = 4$
- Only two tuples fit on a block

Order	The Mth block	Output
1	$(1, 1), (5, 5)$	$(1, 1, 7), (1, 1, 8)$
2	$(3, 2), (3, 1)$	$(3, 2, 6), (3, 2, 5), (3, 2, 7), (3, 1, 7), (3, 1, 8)$
3	$(2, 1), (4, 2)$	$(2, 1, 7), (2, 1, 8), (4, 2, 6), (4, 2, 5), (4, 2, 7)$

Tuple-based Nested-Loop Joins

Algorithm (Tuple-based Nested-Loop Joins)

```
1: for each tuple s in S (outer relation) do
2:     for each tuple r in R (inner relation) do
3:         if r and s join to make a tuple t then
4:             output t
```

- I/O cost: $T(R)T(S)$
- Memory requirement: $M \geq 2$

Block-based Nested-Loop Joins

Assume $S(Y, Z)$ is smaller than $R(X, Y)$

Algorithm (Block-based Nested-Loop Joins)

```
1: for each chunk of M − 1 blocks of S (outer relation) do
2:     read these M − 1 blocks into main-memory buffer
3:     organize their tuples into a search structure whose search key is Y
4:     for each block b of R (inner relation) do
5:         read b into main-memory buffer
6:         for each tuple r in b do
7:             find the tuples of S in main memory that join with r
8:             output the join of t with each of these tuple
```

- I/O cost: $B(S) + \frac{B(R)B(S)}{M-1}$
- Memory requirement: $M \geq 2$

Block-based Nested-Loop Joins (Cont'd)

Example (Block-based Nested-Loop Joins)

- $R(X, Y) = \{(1,1), (5,5), (3,2), (3,1), (2,1), (4,2)\}$
- $S(Y, Z) = \{(2,6), (1,7), (1,8), (2,5), (2,7)\}$
- $M = 3$
- Only two tuples fit on a block

First $M-1$ blocks	Mth block	Output
$(2,6), (1,7), (1,8), (2,5)$	$(1,1), (5,5)$	$(1,1,7), (1,1,8)$
$(2,6), (1,7), (1,8), (2,5)$	$(3,2), (3,1)$	$(3,2,6), (3,2,5), (3,1,7), (3,1,8)$
$(2,6), (1,7), (1,8), (2,5)$	$(2,1), (4,2)$	$(2,1,7), (2,1,8), (4,2,6), (4,2,5)$
$(2,7)$	$(1,1), (5,5)$	
$(2,7)$	$(3,2), (3,1)$	$(3,2,7)$
$(2,7)$	$(2,1), (4,2)$	$(4,2,7)$

Sort-based Joins (a.k.a. Sort-Merge-Join or Merge-Join)

Algorithm (Sort-based Joins)

1. Create sorted sublists of size M, using Y as the sort key, for both R and S
2. Bring the first block of each sublist into a buffer. We assume there are no more than M sublists in all.
3. Repeatedly find the least Y-value y among the first available tuples of all the sublists. Identify all the tuples of both relations that have Y-value y, perhaps using some of the M available buffers to hold them, if there are fewer than M sublists. Output the join of all tuples from R with all tuples from S that share this common Y-value. If the buffer for one of the sublists is exhausted, then replenish it from disk.

- I/O cost: $3B(R) + 3B(S)$
- Memory requirement: $B(R) + B(S) \leq M^2$

Sort-based Joins (Cont'd)

Example (Sort-based Joins)

- $R(X, Y) =$ $\{(1,1),(5,4),(3,2),(3,1),(6,3),(2,1),(4,2),(8,5),(4,1),(3,4)\}$
- $S(Y, Z) =$ $\{(2,6),(1,7),(1,8),(5,9),(5,3),(2,5),(3,1),(2,7),(3,7),(4,9)\}$
- $M = 5$
- Only two tuples fit on a block

Sublists:

- $R_1 = \{(1,1),(2,1),(3,1),(3,2),(6,3),(5,4)\}$
- $R_2 = \{(4,1),(4,2),(3,4),(8,5)\}$
- $S_1 = \{(1,7),(1,8),(2,5),(2,6),(5,3),(5,9)\}$
- $S_2 = \{(2,7),(3,1),(3,7),(4,9)\}$

Let us show the following steps on the blackboard

Hash-Join

Algorithm (Hash-Join)

1. Hash R into $M-1$ buckets $R_1, R_2, \ldots, R_{M-1}$
2. Hash S into $M-1$ buckets $S_1, S_2, \ldots, S_{M-1}$
3. For $1 \leq i \leq M-1$, use the one-pass join algorithm to $R_i \bowtie S_i$, and write out the resulting tuples

- I/O cost: $3B(R) + 3B(S)$
- Memory requirement: $\min(B(R), B(S)) \leq (M-1)^2$

Index-based Joins

Suppose S has an index on the attribute(s) Y

Algorithm (Index-based Joins)

1. Read each block of R, one at a time, into main-memory buffers
2. For each tuple t within the block, use the index to find all those tuples of S that have $t[Y]$ in their Y-component(s). These are exactly the tuples of S that join with tuple t, so we output the join of each of these tuples with t.

- I/O cost: $\frac{T(R)T(S)}{V(S,Y)}$ if the index for S is non-clustered
- I/O cost: $T(R)\max(1, \frac{B(S)}{V(S,Y)})$ if the index for S is clustered
- Memory requirement: $M \geq 2$

Index-based Joins (Cont'd)

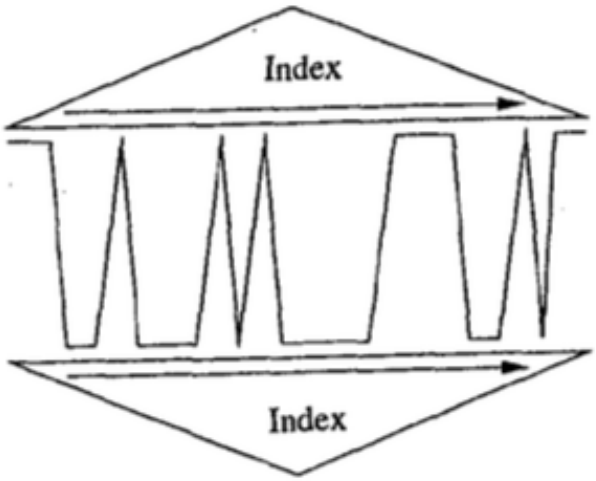
Example (Index-based Joins)

- $R(X, Y) = \{(1, 1), (5, 5), (3, 2), (3, 1), (2, 1), (4, 2)\}$
- $S(Y, Z) = \{(2, 6), (1, 7), (1, 8), (2, 5), (2, 7)\}$
- $M = 2$
- Only two tuples fit on a block
- There is an index on Y for S

Order	Buffer	Output
1	$(1,1),(5,5)$	$(1,1,7),(1,1,8)$
2	$(3,2),(3,1)$	$(3,2,6),(3,2,5),(3,2,7),(3,1,7),(3,1,8)$
3	$(2,1),(4,2)$	$(2,1,7),(2,1,8),(4,2,6),(4,2,5),(4,2,7)$

Sorted-Index-based Joins

- An index is sorted if we can easily extract tuples in sorted order, e.g., B+ trees
- There is a sorted index on each of $R(X, Y)$ and $S(Y, Z)$ with search key Y
- This algorithm is as same as an ordinary sort-join, but we do not have to perform the first step of sorting R and S to sorted sublists



Sorted-Index-based Joins (Cont'd)

Example (Sorted-Index-based Joins)

- $R(X, Y) = \{(1, 1), (2, 1), (3, 1), (3, 2), (4, 2), (5, 5)\}$
- $S(Y, Z) = \{(1, 7), (1, 8), (2, 5), (2, 6), (2, 7)\}$
- There is a sorted index on $R(Y)$
- There is a sorted index on $S(Y)$

8.4 Execution of Expressions

Execution of Expressions

Task: Evaluate an expression containing multiple operations

- The materialization approach (物化法)
- The piplining approach (流水线法)

8.4 Execution of Expressions
8.4.1 Materialization

Materialization

- Evaluate one operation at a time, in an appropriate order
- The result of each evaluation is materialized in a temporary relation for subsequent use

Disadvantages

- Temporary relations must be written to disk and later read back (unless they are small), which increases the cost of query evaluation
- There may be a long delay before a user sees any query results

Example (Materialization)

$\Pi_{ID,salary}$ - - - - - - - - - - - - ▸ (output relation)

|

$\sigma_{salary<75000}$; use index ◂ - - - - ▸ (temporary relation)

|

instructor ◂ - - - - - - - - - - - - (input relation)

8.4 Execution of Expressions
8.4.2 Piplining

Piplining

- Goal: reduce the number of temporary relations that are produced
- Idea: combine several relational operations into a pipeline of operations, in which the results of one operation are passed along to the next operation in the pipeline
- Advantages
  - It eliminates the cost of reading and writing temporary relations
  - It can start generating query results quickly
- Implementation
  - Demand-driven pipelines (需求驱动的流水线)
  - Producer-driven pipelines (生产者驱动的流水线)

Example (Pipelining)

$\Pi_{ID,salary}$ - - - - - - - - - - - - ➤ (output relation)

| pipelined

$\sigma_{salary<75000}$; use index (~~temporary relation~~)

|

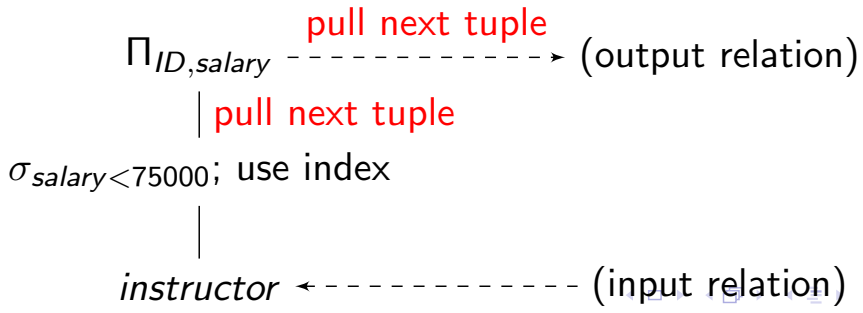
instructor ➤ - - - - - - - - - - - - (input relation)

Demand-Driven Pipelines

- The system makes repeated requests for tuples from the operation at the top of the pipeline
- Each time that an operation receives a request for tuples, it computes the next tuple (or tuples) to be returned, and then returns that tuple

  Case 1 (The inputs of the operation are not pipelined)
  - The next tuple(s) to be returned can be computed from the inputs
  - The system keeps track of what has been returned so far

  Case 2 (Some inputs of the operation are pipelined)
  - The operation makes requests for tuples from its pipelined inputs
  - Using the tuples received from its pipelined inputs, the operation computes tuples for its output, and returns them



Demand-Driven Pipelines (Cont'd)

- Each operation in a demand-driven pipeline can be implemented as an iterator (迭代器)
- An iterator provides the following functions:
  - `open()`: Start the iterator
  - `next()`: Returns the next output tuple of the operation
  - `close()`: Close the iterator
- The iterator maintains the state of its execution in between calls, so that successive `next()` requests receive successive result tuples

Example (Iterator for Sort-Merge Joins)

- `open()`: Open its inputs and sort them if they are not already sorted
- `next()`: Find the next pair of matching tuples in the inputs and return the joined tuple
- The state information would consist of up to where each input had been scanned

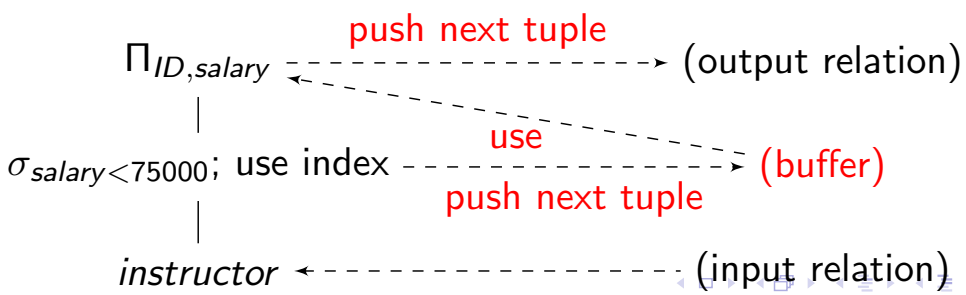
Producer-Driven Pipelines

- Operations do not wait for requests to produce tuples, but instead generate the tuples eagerly
- Each operation is modeled as a separate process or thread within the system

Demand-driven pipelines	Producer-driven pipelines
Top-down	Bottom-up
Pulling tuples	Pushing tuples
Lazily generating tuples	Eagerly generating tuples
Easier to implement	More complicated to implement

Producer-Driven Pipelines (Cont'd)

- For each pair of adjacent operations, the system creates a buffer to hold tuples being passed from one operation to the next
- Operations at the bottom of a pipeline
  - The operation continually generates output tuples and puts them in its output buffer until the buffer is full
  - Once the output buffer is full, the operation waits until its parent operation removes tuples from the buffer
- Operations not at the bottom of a pipeline
  - The operation generates output tuples when it gets input tuples from lower down in the pipeline until its output buffer is full
  - Once the operation uses a tuple from a pipelined input, it removes the tuple from its input buffer



Summary

1. 8.1 Overview of Query Processing
2. 8.2 External Sort
   - External Merge Sort
3. 8.3 Execution of Operations of Relational Algebra
   - 8.3.1 Execution of Selection
   - 8.3.2 Execution of Projection
   - 8.3.3 Execution of Duplicate Elimination
   - 8.3.4 Execution of Aggregation Operations
   - 8.3.5 Execution of Set Operations
   - 8.3.6 Execution of Joins
4. 8.4 Execution of Expressions
   - 8.4.1 Materialization
   - 8.4.2 Piplining

Excecise

1. Describe the *one-pass aggregation algorithm* and analyze its I/O cost and memory requirement
2. Describe the *hash-based aggregation algorithm* and analyze its I/O cost and memory requirement
3. Describe the *sort-based aggregation algorithm* and analyze its I/O cost and memory requirement
4. Write pseudocode for an iterator that implements a join algorithm (*one-pass join*, *block-based nested loop join*, *sort-merge join*, *hash join*, *index-based join*)