

# Query Execution

## Chapter 6

# Outline

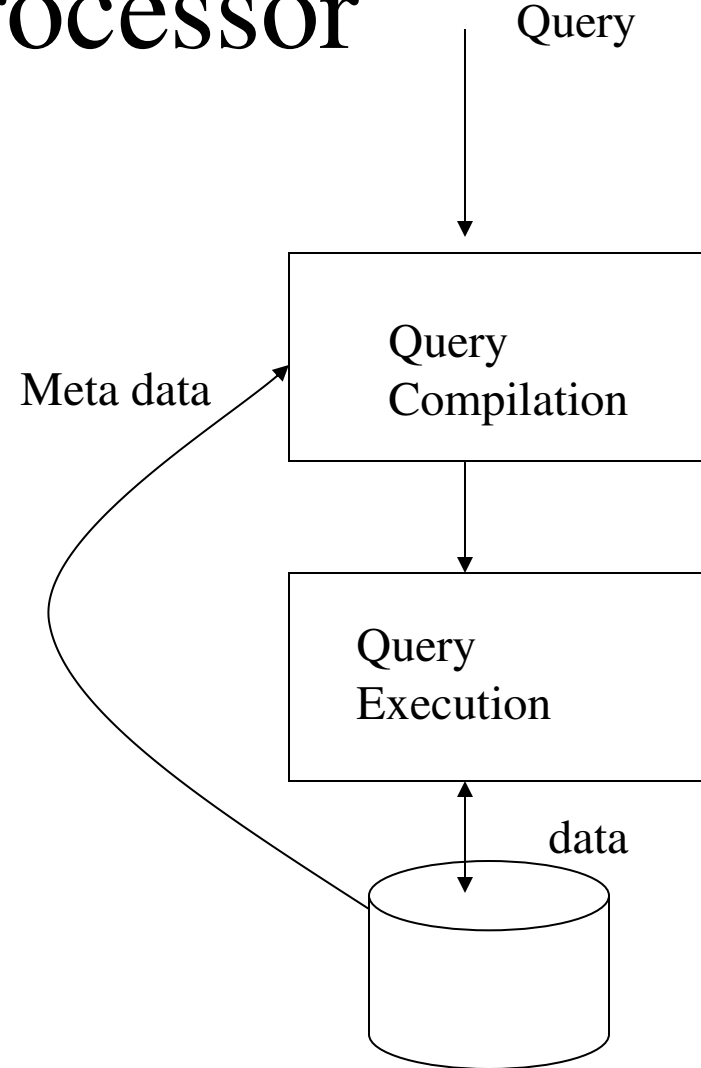
- Introduction
- An algebra for queries
- Introduction to physical query plan operators
- One pass algorithms for Database Operations
- Nested-loop joins
- Two-pass algorithms Based on Sorting
- Two-pass algorithms based on hashing
- Index-based algorithms
- Buffer Management
- Overview of parallel algorithms

# Introduction

- Previous chapters
  - Finding tuples given a search key
- How to answer queries ?
  - This chapter and next chapter
- Query processor
  - Converts queries into a sequence of operations on the database and executes the same.
  - Naïve conversion may require more time.

# Query Processor

- Query compilation
  - Chapter 7
- Query execution
  - Chapter 6



# Overview of Query Compilation

- Three major steps
  - Parsing
    - Query and its structure is constructed
  - Query rewrite
    - Parse tree is converted to initial query plan
    - Algebraic representation of a query
  - Physical plan generation
    - Selecting algorithms for each operations
    - Selecting the order of operations
- Query rewrite and physical plan generation are called query optimizer.
- Choice depends on meta data

# Query Execution

- Execution of relational algebra operations
  - Union
  - Intersection
  - Difference
  - Selection
  - Projection
  - Joins
  - sorting

# An algebra for queries

- SQL operators for Union, intersect and difference
  - UNION, INTERSECT and EXCEPT
- Selection:
  - Where clause
- Projection
  - SELECT clause
- Product
  - Relations are FROM clause, whose products forms the relation with condition of WHERE clause and projection of SELECT clause.
- JOIN
  - JOIN, NATURAL JOIN, OUTER JOIN
- Duplicate elimination
  - DISTINCT
- Grouping
  - GROUP BY clause
- Sorting
  - ORDER BY clause

# UNION, INTERSECTION and DIFFERENCE

- Differences between relations and sets
  - (a) Relations are bags
  - (b) Relations have schemas, a set of attributes that name their columns
- Due to (b) schemas of two argument relations must be the same.
- $R \cup S$ :  $t$  appears as many times it is in  $R$  plus it is in  $S$ .
- $R \cap S$ :  $t$  appears in the result the minimum number of times it is in  $R$  and  $S$
- For  $R - S$ : number of times in  $R$  minus number of times in  $S$ .
- By default, duplicates are eliminated
- We can use the key word `ALL` (for example `UNION ALL`) to take the bag versions
- The bag versions are followed by duplicate elimination.



# SELECTION

- $\sigma_C(R)$  takes a relation  $R$  and a condition  $C$ . The condition  $C$  may involve
  - Arithmetic
  - Comparison
  - Boolean

# PROJECTION

- $\Pi_L(R)$  projection R in list L.
- L can have the following elements
- An expression  $x \rightarrow y$ , where x and y are names for attributes.
- An expression  $E \rightarrow z$ , where E is an expression involving attributes of R, constants, arithmetic operators and string operators and z is a new names of the attribute.

# The Product of Relations

- $R \times S$ 
  - Schema consists of attributes of  $R$  and attributes of  $S$ .
  - If a tuple  $r$  appears  $m$  times in  $R$  and a tuple  $s$  appears  $n$  times in  $S$ , then the tuple  $rs$  appears  $mn$  times.

# Joins

- Natural join
  - $\Pi_L(\sigma_C(R \bowtie S))$
  - C is the condition that equates all the pairs of R and S that have the same name
  - L is a list of all the attributes of R and S, except one copy of each pair of attributes is omitted.
- Theta join
  - $\sigma_C(R \bowtie S)$ , where condition C is of the form like x any condition y.
- Equijoin
  - $\sigma_C(R \bowtie S)$ , where condition C is of equal form like  $x = y$ .

- Outer join
  - Adding dangling tuples from R and S
- Left outer join
  - Only dangling tuples of the left argument are added
- Right outer join
  - Only dangling tuples of right argument S are added.

# Outer Join – Example

- Relation *loan*

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

## Relation *borrower*

<i>customer-name</i>	<i>loan-number</i>
Jones	L-170
Smith	L-230
Hayes	L-155

## Full Outer Join

*loan*  $\bowtie$  *borrower*

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>	<i>customer-name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	<i>null</i>
L-155	<i>null</i>	<i>null</i>	Hayes

*loan*  $\bowtie$  *Borrower*

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>	<i>customer-name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

## Left Outer Join

*loan*  $\bowtie_{\leftarrow}$  *Borrower*

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>	<i>customer-name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	<i>null</i>

## Right Outer Join

*loan*  $\bowtie_{\rightarrow}$  *borrower*

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>	<i>customer-name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	<i>null</i>	<i>null</i>	Hayes

# Duplicate Elimination

- $\Delta(R)$  returns one copy of each tuple.

# Grouping and Aggregation

- Aggregate operators
  - AVG, SUM, COUNT, MIN, and MAX
- Group By
  - The relation is grouped according to the value of the attribute
- Having
  - Provides a condition
- Grouping and having should be implemented together



# Sorting operator

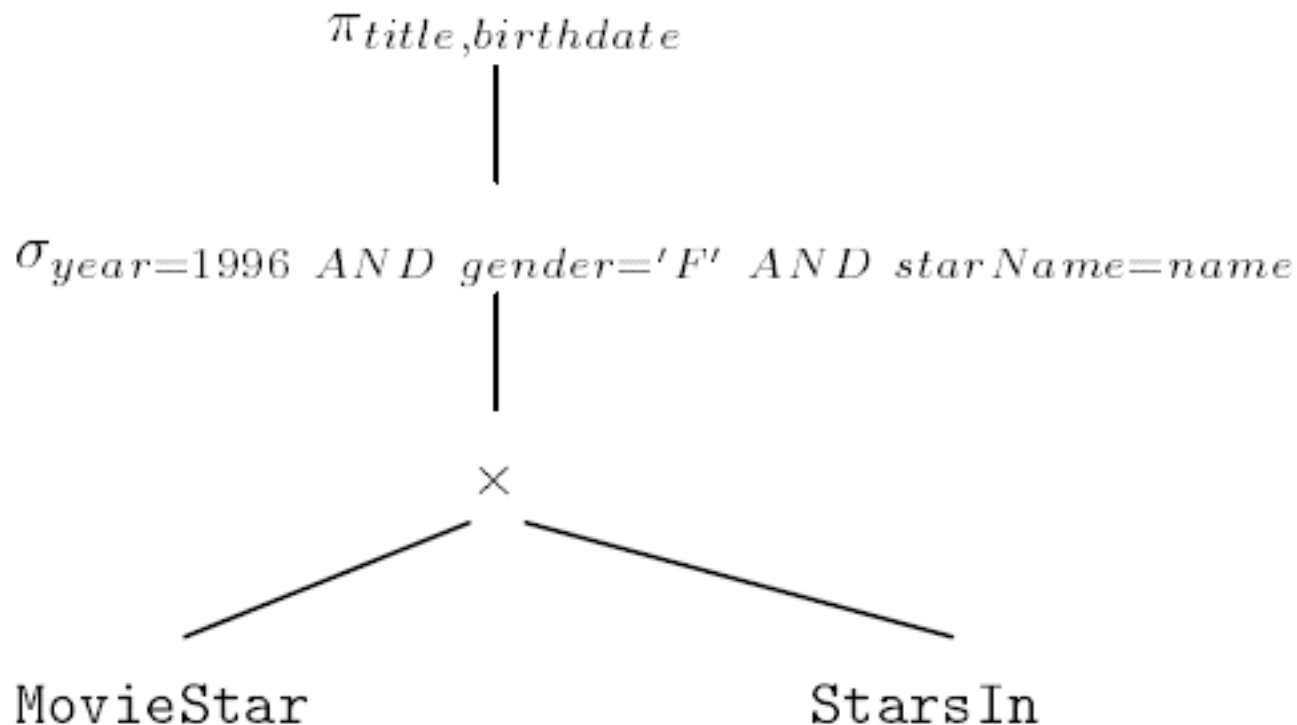
- We use the operator  $\tau$  to sort a relation
- SQL ORDER BY
- $\tau_L(R)$ , L is a list of attributes ( $a_1, a_2, \dots, a_n$ )
  - Ties are broken as per  $a_2 \dots$
  - They agree till “ $a_n$ ”, the tuples are sorted arbitrarily.
  - Default is ascending order, can be changed using DESC
  - The result is list of tuples. So it is the last operator.
    - If this operator is applied, the notion of set is disturbed.

# Expression Trees

- Schema
  - MovieStar(name, address, gender, birthdate)
  - StarsIn(title, year, starName)
- Query
  - Select title, birthdate
  - From MovieStar, StarsIn
  - Where Year = “1996”  $\wedge$  gender = F  $\wedge$  starName=name

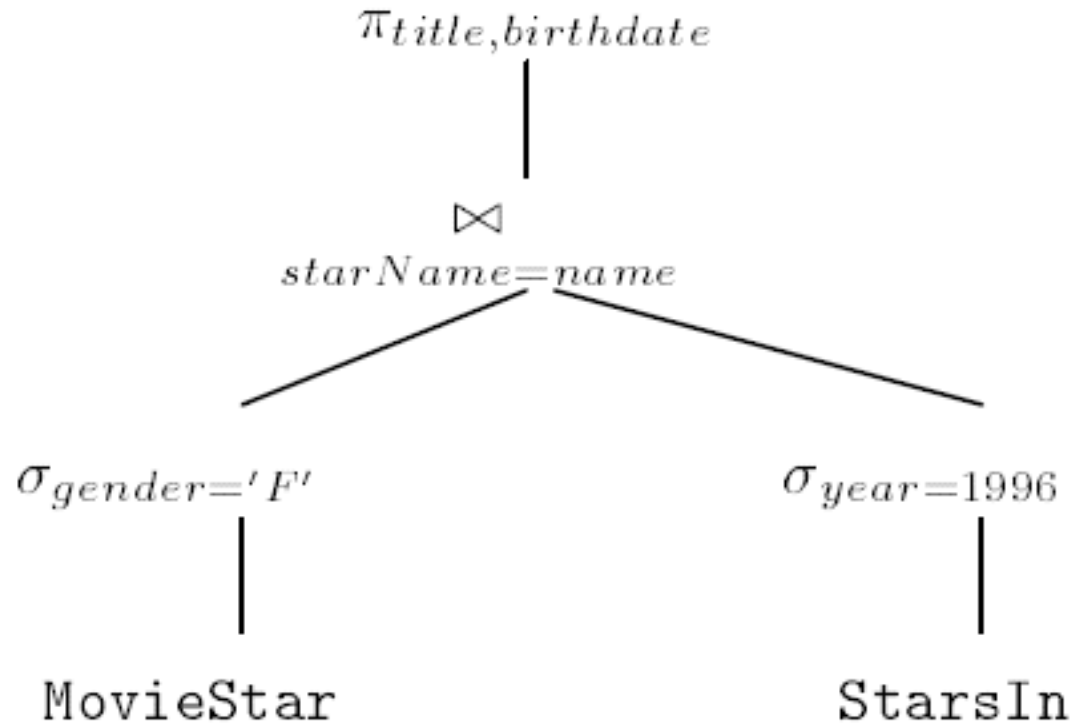
# Expression Tree 1

Plan I (from definition)



# Another logical query plan

## Plan II



# Introduction to Physical Query Plan Operators

- Scanning tables
  - Read the entire contents of R.
  - Two basic operators
    - Get the blocks one by one
      - Table scan
    - Use index to get all the tuples of R
      - Index scan

# Sorting While Scanning Tables

- The operator sort-scan takes a relation  $R$  and a specification of attributes on which the sort is to be made, and produces  $R$  in that sorted order.
- Ways to implement sort-scan
  - If we are to produce a relation  $R$  sorted by attribute “a”, and there is a Btree index on “a”, simple scan produces sorted file.
  - If  $R$  fits in main memory, we can retrieve tuples using index scan or table scan and use main memory sorting algorithms.
  - If  $R$  is too large to fit in main memory, follow multi-way merging approach.

# Model of Computation of Physical Operators

- Query consists of several operations of relational algebra and corresponding physical query plan is composed of several physical operators.
  - Operations like scanning are invisible in relational algebra.
- Physical plan operators should be chosen wisely.
- Number of disk I/O's is the measure.
- Assumption:
  - Arguments can be found on the disk and the result of the operation is in the main memory.

# Parameters to measure the cost

- $M$  denotes the number of main memory buffers.
  - Input and intermediate results
  - $M$  is much smaller than the main memory
- $B(R)$ - number of blocks that hold  $R$
- $T(R)$  - number of tuples in  $R$
- $V(R,a)$  – number of distinct values of the column for “ $a$ ” in  $R$ .



# I/O cost for Scan Operators

- If R is clustered the number of disk I/O's = B.
- If R fits in main memory, B I/O's.
- If R is clustered but requires a two-phase multi-way merge sort we require  $3B$  disk I/O's
- If R is not clustered, the number of disk I/Os is much higher.
- If R is distributed among the tuples of other relations, the I/O cost is T.
- If R is not clustered and requires a two-way merge sort it requires:  $T+2B$ .
- Index scan
  - Fewer than  $B(R)$  blocks

# Iterators for Implementation for Physical Operators

- Iterator
  - Getting one tuple at a time.
- Three functions
  - Open:
    - Starts the process of getting tuples, but does not get a tuple. It initializes any data structures needed to perform the operation and calls Open for any arguments of the operation.
  - GetNext
    - The function returns the next tuple in the result and adjusts data structures as necessary to allow subsequent tuples to be obtained. It returns null, if there are no more tuples produced.
  - Close
    - The function ends iteration after all tuples or all the tuples the consumer wanted, have been obtained.
- (Refer Figure 6.8 for table scan operator)

# Iterator for Table Scan

- `Open(R){`  
    `b:= the first block of R;`  
    `t:= the first tuple of block b;`  
    `found := true;`  
    `}`
- `GetNext {`  
    `IF(t is past the last tuple on block b) { increment b to the next block;`  
    `IF(there is no next block) {Found:=FALSE; RETURN;}`  
    `ELSE /* b is a new block */`  
        `t:= first tuple on bock b;`  
        `Oldt:=t;`  
        `Increment t to the next tuple of b;`  
        `RETURN oldt;`  
    `}`
- `CLOSE(R) {}`

# Iterator for sort-scan

- The open function read all the tuples in main-memory sized chunks, sort them and stores them on the disk
- Initialize the data structure for the second merge phase, load the first block of each sublist into main-memory structure.

# Iterators

- Iterators can be combined by calling other iterators.
- `Open(R,S){ R.open(); CurRel:=R;}`
- `GetNext(R,S) { IF CurRel=R {t=R.GetNext()  
IF (Fund) RETURN t ELSE S.open();  
CurRel=S;}} Return S.GetNext();`
- `Close(R,S){ R.Close(); S.Close();}`

# Algorithms For Database Operations

- How to execute each individual steps
  - Join and selection
  - Selecting a choice of an algorithm
- Three categories
  - Sorting-based methods
  - Hash-based methods
  - Index-based methods
- One pass algorithms
  - Requires reading only once from the disk
  - At least one of the arguments fits in the memory
- Two pass algorithms
  - Too large to fit in the memory but not the largest.
  - Requires two readings
    - First read and process and write back.
    - Read again for further processing
- Multi pass algorithms
  - No limit on the size of data
  - Recursive generalizations of two pass algorithms

# Three types of operators

- Tuple at a time, unary operations
  - Selection and projection
- Full relation
  - Seeing all the tuples in memory once
- Full-relation, binary operations
  - Union, intersection, joins, products

# One pass algorithms for tuple at a time Operations

- $\sigma(R)$ ,  $\Pi(R)$ ,
  - Read the blocks of R one at a time into input buffer and move the tuples to output buffer.
  - Complexity is B, if R is clustered and T if R is not clustered.
  - If the condition variable has an Index, it affects the performance.



# One Pass algorithms for Unary and Full Relation Operations

- Duplicate elimination and grouping
- For each tupe
  - If we have seen it first time, write it to output buffer
  - If the tuple is seen before, do not output this tuple.
    - For this we have to keep one copy of every tuple we have seen.
    - One memory buffer holds the block of R's tuples, the remaining M-1 blocks can be used to hold a single copy of every tuple seen so far.

# One pass algorithms...

- Each new tuple takes processor time proportional to  $n$ .
  - Complete operation takes  $n^2$  processor time.
  - If  $n$  is large, it becomes significant.
- The main memory data structure to allow
  - Add a new tuple
  - Tell a whether a given tuple is already there.
- Use hash tables or binary search tree
- Assumption: no extra space is required
- If  $B(\delta(R)) \leq M$ , it is OK, otherwise thrashing might occur.

# Grouping

- The grouping operation  $\gamma_L$  gives zero or more grouping attributes and one or more aggregated attributes.
- For each of grouping attribute create one entry for each value and scan each block at a time
  - For a MIN(a) or MAX(a) aggregate, record min and max value, respectively and change min or maximum whenever each time a tuple of a group seen
  - For COUNT, add one to the corresponding group
  - For SUM(a), add the value of “a” to the accumulated sum.
  - AVG(a): maintain two accumulations: count and sum of values.
- Until all the tuples are seen, we can not output the result.
- Entire grouping is done in OPEN function.
- Hash tables and balanced trees are used to find the entry in each group.
- The number of I/Os is B.

# One pass Algorithms for Binary Operations

- Set Union
  - Read S into M-1 buffers of main memory and build a search structure where the search key is the entire tuple. All the tuples are copied to the output
  - Read R into mth buffer, one at a time. If “t” is in S, skip “t”, otherwise write “t” to output.
- Set Intersection
  - Read S into M-1 buffers and build a search structure with full tuples as search key. Read each block of R, for each “t” of R, see if t is also in S. If so, copy t to the output. If not, ignore it.
- Set difference
  - Assume R is larger relation. Read S to M-1 buffers and build a search structure with full tuple as a search key.
  - R-S
    - Read each block of R. IF t is in S, ignore t; if it is not in S, copy t to the output
  - S-R
    - Read each block of R. IF t is in S, ignore t; if it is in S, delete “t” from the copy of S. If “t” is not in S we do nothing. After completing R, output the remaining S tuples.

# Bag intersection and Bag difference

- Bag intersection
  - We read  $S$  into  $M-1$  buffers, we associate each distinct tuple a count, which measures the number of times the tuple appears in  $S$ .
  - Read each block of  $R$  whether  $t$  occurs in  $S$ . If not, ignore “ $t$ ”. If it appears, if the count is positive, output  $t$  and decrement the count by one. If  $t$  appears in  $S$  and count is zero, we do not output  $t$ .
- Bag difference
  - Similar.. (Check the book)

# Product

- Product
  - Read S into M-1 buffers. Read each block of R. For each tuple of R, concatenate t with each tuple of S. Output concatenated file.

# Natural Join

- Natural join
  - $R(X,Y)$  being joined with  $S(Y,Z)$
  - Read all the tuples of  $S$  into  $M-1$  blocks.
  - Read each block of  $R$ . For each  $t$  of  $R$ , find the tuples that agree with all the attributes of  $Y$ . For each matching tuple of  $S$ , form a tuple by joining it with  $t$ , and move the resulting tuple to the output.
  - Takes  $B(R)+B(S)$  disk I/Os

# Nested Loop Joins

- Tuple-based Nested loop join
  - Complexity:  $T(R)T(S)$  disk I/Os.

## Nested-Loop Join

To compute  $R \bowtie S$ :

```
for each tuple r of R do
  for each tuple s of S do
    if r and s join then
      output the resulting tuple
```



# Iterator for Tuple-based Nested Loop Join

- Iterators can be easily built.

# A block-based Nested Loop Join

- Organize the access to both relations by blocks.
- Using as much memory, store the tuples of S.

# Nested Block Join

```
FOR each chunk of M-1 blocks of S DO BEGIN
    read these blocks into main memory buffers;
    Organize their tuples into a search tree structure whose search key
        is the common attributes of R and S;
    FOR each block b of R DO BEGIN
        read b into memory;
        FOR each tuple t of b DO BEGIN
            find the tuples of S in main memory that
                join with t;
            output the join of t with each of these tuples;
        END;
    END;
END;
```

# Example

- $B(R)=1000$  and  $B(S)=500$  and let  $M=101$ .
- We use 100 blocks of memory for S
  - Outer loop iterates five times
  - Each iteration we have 100 disk I/Os
  - Second loop we must read entire R.
    - 1000 I/Os
  - 5500 I/Os
- If we reverse the role of R and S
  - 6000 I/Os

# Analysis of Nested –Loop Join

- Iterations of outer nested loop  $B(S)/(M-1)$
- For each iteration we read  $M-1$  blocks of  $S$  and  $B(R)$  blocks of  $R$ .
- The number of disk I/Os
  - $(B(S)/(M-1)) (M-1+B(R))$
  - $B(S) + B(S)B(R)/(M-1)$
  - $B(S)B(R)/M$

# Summary

Operators	Approximate M Required	Disk I/O
$\sigma, \pi$	1	B
$\gamma, \delta$	B	B
U, $\cap$ , $-$ , $\times$ , Join	$\text{Min}(B(R), B(S))$	$B(R) + B(S)$
Join	Any $M \geq 2$	$B(R)B(S)/M$

# Two-pass Algorithms Based on Sorting

- Two pass algorithms are sufficient.
- It can be generalized into higher passes
- If we have large relation  $R$ 
  - Read  $M$  blocks into main memory
  - Sort these  $M$  blocks
  - Write sorted list into  $M$  blocks of disk
  - Execute the desired operator on the sorted lists

# Duplicate Elimination Using Sorting

- Hold the first block from each of the sorted sublist.
- Consider first unconsidered tuple from each block, make copy of  $t$  to the output. Remove from the fronts of various input blocks all copies of  $t$ .
- If the block is exhausted, bring new block of that sorted sub-list.



# Complexity

- Ignore the writing of output
  - $B(R)$  to read each block of  $R$  when creating sorted sublists
  - $B(R)$  to write each of sublists to disk
  - $B(R)$  to read each block from sublist.
  - Total  $= 3 * B(R)$
- Memory requirement
  - $B \leq M^2$  is required for the two pass algorithm to be feasible  
(for one pass  $B \leq M$ )
  - $\delta(R)$  requires  $\sqrt{B(R)}$  blocks of memory  
(for one pass  $B=M$ )

# Grouping and Aggregation

- The algorithm for  $\gamma(R)$  is similar to  $\delta(R)$ .
- Read  $R$ ,  $M$  blocks at a time. Sort each blocks grouping attributes of  $R$  as the sorted key. Write each sorted sublist to disk.
- Use the one memory buffer for each sublist, load the first block of each sublist into its buffer
- Repeatedly find the least value in the available sub-buffers. This value,  $v$ , becomes the next value
  - Prepare to compute all the aggregates on list  $L$  for this group
  - Examine each of the tuples with sort key  $v$ , and accumulate the needed aggregates.
  - If a buffer becomes empty, replace it with the next block from the same list.
- Complexity is similar to  $\delta$ 
  - $B \leq M^2$  is required for the two pass algorithm to be feasible
  - $\delta(R)$  requires  $\sqrt{B(R)}$  blocks of memory

# Sort-based Union Algorithm

- One-pass algorithm works, if at least one relation is in main memory.
- Two pass algorithm
  - Create sorted sublists of R of size M.
  - Create sorted sublists of S of size M.
  - Use one main-memory buffer for each sublist of R and S. Initialize with first block of each R and S.
  - Repeatedly find the first remaining tuple t among all the buffers. Copy t to the output and remove from all the copies of t.
  - If the buffer becomes empty read the next block in the sublist.
- Total cost=  $3(B(R) + B(S))$ 
  - Each tuple is read twice in main memory: Once when sublists are being created and the second time as a part of one of the sublists.
  - Each tuple is written once
  - The size of two relations should not exceed  $M^2$ .
    - That is,  $B(R) + B(S) \leq M^2$

# Sort-based Algorithms for Intersection and Difference

- Same as union
- Set intersection:
  - Output  $t$ , if  $t$  appears in both  $R$  and  $S$ .
- Bag intersection
  - Output  $t$  the minimum number of times  $t$  appears in  $R$  and in  $S$ .
- $R-S$  (set)
  - Output  $t$ , if only it appears in  $R$  but not in  $S$ .
- $R-S$  (bag)
  - Output  $t$  the number of times it appears in  $R$  minus the number of times it appears in  $S$ . If both occur at equal number of times, we do not output any tuples.
- Analysis
  - $3(B(R)+B(S))$  disk I/Os.
  - $B(R)+B(S) \leq M^2$  for the algorithm to work.

# Simple Sort-based Joining Algorithm

- Sort R, with multi-way merge sort, with Y as the sort key.
- Sort S similarly.
- Merge the sorted R and S. Use only two buffers contains current blocks of R and S respectively.
  - Find least value y that is currently in front of the blocks for R and S.
  - If y does not appear in front of other relation, remove tuples with sort key “y”.
  - Otherwise, identify all the tuples from both relations having sort key y. read blocks from R and S, until no more keys. Use M buffers for this purpose.
  - Output all such tuples.
  - If either relation has no more unconsidered tuples in main memory, reload the buffer for that relation.

# Simple Sort join: modification for the worst case

- If Y-is the key for R or S no problem arises.
- If the tuples from one of the relations, say R, that have Y-value  $y$  fit in to  $M-1$  buffers, do one pass join
  - load these blocks of R into buffers and read the blocks of S that hold tuples with  $y$ , one at a time into remaining buffer.
- If neither relation has sufficiently few tuples with Y-value  $y$  that they all fit into  $M-1$  buffers, do nested-loop join
  - Use  $M$  buffers to perform nested loop join on the tuples with Y-value  $y$  from both relations.

# Analysis of Simple Sort-based Join

- Analysis
  - We shall not run out of buffers, if the tuples in common y value can not fit in buffers.
  - $5(B(R)+B(S))$  disk I/Os.
  - $B(R) \leq M^2$  and  $B(S) \leq M^2$  to work.

# A More Efficient Sort-based Join

- Second phase of join and sorting can be combined. It is called sort-join, sort-merge-join, merge join.
- Create sorted sublists on R and S using Y.
- Bring the first block of each sublist into buffer (No more than M sublists)
- 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  $y$ , using some of the available M buffers. Output the join of all tuples from R with all tuples from S that share this common Y-value. If the buffer of one sublist exhausted, replenish it from disk.
- Analysis
  - $3(B(R)+B(S))$  disk I/Os.
  - $(B(R)+B(S)) \leq M^2$  is sufficient.



# Problem of many tuples with Y-value

- Problem may not arise
  - Y is the key of R
- If  $B(R) + B(S)$  is much less than  $M * M$ , use many unused buffers.
- In the worst case follow nested loop join.

# Summary of Sort-based algorithms

Operators	Approximate M Required	Disk I/O
$\gamma, \delta$	$\sqrt{B}$	$3B$
$U, \cap, -$	$\sqrt{(B(R)+B(S))}$	$3(B(R) + B(S))$
Join	$\sqrt{\max(B(R), B(S))}$	$5(B(R) + B(S))$
Join	$\sqrt{(B(R)+B(S))}$	$3(B(R) + B(S))$

# Two-Pass Algorithms Based on Hashing

- Basic idea
  - If the data is too big to store in main-memory buffers, hash all the tuples of the argument or arguments using an appropriate hash key.
  - If  $M$  buffers are available, we can pick  $M$  as number of buckets.
  - We can gain a factor of  $M$  similar to sort-based algorithms, but different means.

# Partitioning Relations Using Hashing

- Let hash function be  $h$
- Associate one buffer with each bucket.
- Partition  $R$  is using  $M$  buffers
- Partition  $R$  into  $M-1$  buckets.
- Read  $R$  in to  $M$ th buffer
- Each tuple is hashed to  $M-1$  buckets and copied to appropriate buffer.
  - If the buffer is full, write it to disk and initialize another block.
- Bucket size is bigger than buffers.

# A hash-based Algorithm for Duplicate Elimination

- $\delta(R)$ 
  - Hash  $R$  to  $M-1$  buckets.
  - Two copies of the same tuple are hashed to the same bucket.
  - Eliminate one bucket at a time.
  - One pass algorithm can be used.
- It works if individual  $R_i$ 's fits into main memory.
  - Each  $R_i$  is equal to  $B(R)/(M-1)$  blocks in size.
  - If the number of blocks no longer than  $M$ , i.e.,  $B(R) \leq M(M-1)$ , two pass hash based algorithm works.
  - $B(R) \leq M^2$  (same as sort-based algorithm for duplicate elimination)

# Hash-based algorithm for grouping and aggregation

- $\gamma_L(R)$ 
  - Chose the hash function on the grouping attributes of list L.
  - Hash all the tuples of R into M-1 buckets
  - Use one pass algorithm
  - We can process provided that  $B(R) \leq M^2$
  - Number of disk I/Os =  $3B(R)$

# Hash-based algorithms for $U, \cap$ , and $-$

- Binary operation
- Use the same hash function to hash tuples of both arguments.
- $R \cup S$ 
  - Hash both  $R$  and  $S$  to  $R_1, R_2, \dots, R_{M-1}$  and  $S_1, S_2, \dots, S_{M-1}$ .
  - Take set union of  $R_i$  and  $S_i$ , for all  $i$  and output the result.
- Intersection or difference
  - Apply appropriate one pass algorithm
  - $(B(R) + B(S))$  I/Os.
- Summary
  - Total  $3(B(R)+B(S))$  I/Os
  - The algorithms work iff  $\text{Min}(B(R), B(S)) \leq M^2$

# Hash Join Algorithm

- Compute  $R(X,Y)$  join  $S(Y,Z)$
- Use the hash key for the join attributes.
  - If tuples join they will wind up in corresponding  $R_i$  and  $S_i$  for some  $i$ .
  - One pass join of all pairs of corresponding buckets.
- Summary
  - Total  $3(B(R)+B(S))$  I/Os
  - The algorithms work iff  $\text{Min}(B(R), B(S)) \leq M^2$



# Hybrid Hash Join:

## Saving some disk I/Os

- Use several blocks for each bucket and write them out in a group , in consecutive blocks of disk.
- Hybrid hash join
  - Use several tricks to avoid writing of some of the blocks.
  - Suppose  $S$  is a smaller relation
  - Keep all  $m$  buckets in main memory and keep one block for each remaining bucket.
  - $(m * ((B(S)/k)) + (k-m)) \leq M$ 
    - $B(S)/k$  is the expected size of bucket.
- $M$  buckets of  $S$  are not written into memory
- One block for each  $(k-m)$  buckets of  $R$  whose corresponding buckets of  $S$  were written to disk.
- In the second pass,  $R$  and  $S$  can be joined as usual, Actually, there is no need to join.

# Saving some disk I/Os

- Saving disk I/Os equal to two for every blocks of  $S$  and corresponding  $R$  buckets. Note that  $m/k$  buckets are in main memory
  - Savings =  $2(m/k)(B(R)+B(S))$
  - Intuition:  $k$  should be as small as possible to maximize savings.
  - Buckets are equal to  $M$ , so  $k=B(S)/M$  and  $m=1$
  - Savings =  $(2M/B(S))(B(R)+B(S))$
  - Total cost  $(3-(2M/B(S)))(B(R)+B(S))$

# Summary of Hash-based Join algorithms

Operators	Approximate M Required	Disk I/O
$\gamma, \delta$	$\sqrt{B}$	$3B$
$U, \cap, -$	$\sqrt{B(S)}$	$3(B(R) + B(S))$
Join	$\sqrt{B(S)}$	$5(B(R) + B(S))$
Join	$\sqrt{B(S)}$	$(3 - 2M/B(S))(B(R) + B(S))$

Assume  $B(S) \leq B(R)$

# Sort-based and Hash-based algorithms

- Size
  - Hash depends on smaller relation size
  - Sort depends on sum of two relation sizes.
- Sort order
  - Sort algorithms produce sorted order
  - Hash algorithms not
- Size of buckets
  - Hash depends on buckets of equal size. It is not possible to use buckets that occupy  $M$  blocks
- In sort-based algorithms, sorted sub-lists can be written in consecutive blocks of disk. Rotational and seek latency can be reduced
- In hash-based algorithms also, we can write several blocks of buckets at once.

# Index-based Algorithms

- Index is available on one or more attributes of a relation

# Clustering and non-clustering indexes

- All the relations are clustered
  - If the tuples are packed roughly as few blocks as can possibly hold those tuples.
- Clustering indexes
  - Indexes on an attribute or attributes such that all the tuples with a fixed value for the search key of this index appear roughly as few blocks that hold them
  - Only clustered relations have clustered index.

# Index-based Selection

- $\sigma$ 
  - If there are no indexes on R, one has to read all the tuples of R
    - Number of disk I/Os =  $B(R)$  or  $t(R)$
  - If the search key is equal  $a=v$ , and  $a$  is an attribute for which an index exists and  $v$  is a value.
  - We can use index and get the pointers.
  - If index is clustered the number of disk I/Os =  $B(R)/V(R,a)$ .
    - Actual number might be higher
      - Disk I/Os are needed to support index lookup
      - The tuples can be spread over several blocks
      - The tuples can spread several blocks.
  - If the index is non-clustered, we must access  $T(R)/V(R,a)$  tuples.
    - The number might be higher because, we might read some index blocks. (sometimes lower also)

# Notions of clustering

- Clustered file organization
  - Tuples of two relations are placed together in the same block based on certain value.
- Clustered relation
  - One block contains the tuples of one relation.
- Clustered index
  - Tuples having given search key value are stored together.



# Joining by Using an Index

- Suppose S has an index on attributes Y
- Examine each block of R, identify tuples that match y, use index to find all tuples of S that have ty in their Y-components.
- Disk I/Os
  - Reading R
    - If R is clustered, we have to read  $B(R)$  blocks.
    - If R is not clustered, upto  $T(R)$  blocks
  - Reading S
    - For each tuple t of R, we must read an average of  $T(S)/V(S,Y)$  tuples of S.
    - S has non clustered index on Y, the number of disk I/Os required is  $T(R) T(S)/V(S,Y)$  disk I/O's
    - If the index is clustered we only need  $T(R) B(S)/V(S,Y)$  disk I/Os office.
  - Regardless of R is clustered or not, the cost of accessing tuples of S dominates
    - So we take  $T(R) T(S)/V(S,Y)$  or  $T(R) (\max(1, B(S)/V(S,Y)))$  for clusted or non-clustered indexes, respectively.

# Advantages of Index Join

- $R$  is very small compared with  $S$  and  $V(S, Y)$  is large.
- If we select before join, most of the  $S$  tuples will not be examined.
- Both sort/hash based methods examine every tuple of  $S$  at least once.

# Joins using Sorted Index

- If we have a sorting indexes on Y for both R and S we have to perform only the final step of the simple sorted-based join.

# Buffer Management

- Value of  $M$  vary depending on system conditions
- Buffer manager allocates buffers to processes such that delay is minimized.

# Buffer Management Architecture

- It controls main memory directly, or
- Buffer manager allocates buffers in virtual memory, allowing OS to decide which buffers are actually in main memory at any time.
- Should avoid thrashing.

# Buffer Management Strategies

- If buffers are full, it has to decide which block has to throw out to create space for new block.
- LRU
  - Throw out the block which has not been used for a longer period of time.
- FIFO
  - Throw out the block which has stayed for longer time.
- CLOCK algorithm
  - Approximation of LRU
  - Buffers are arranged in a circle.
  - When the block is read it is set to 1. When the contents are accessed it is set to 1.
  - Buffer manager replaces a block with 0. If it finds 1, it sets to 0.
- System control
  - Take advice from query processor about the algorithm and pinned blocks.

# Pinned blocks

- The blocks can not be moved without modifying other blocks that point to it.
- Buffer manager avoid expelling pinned blocks.
- If the block is pinned, it will remain in memory
  - Root block of B-tree is pinned
  - Pin the blocks of smaller relation in the join

# Relation between Physical Operator Selection and Buffer Management

- Query optimizer selects some operators
- Each operator assumes availability of some buffers
- Buffer manager may not guarantee
- So,
  - Can the algorithm of physical operator adapt changes to  $M$  ?
  -



# Observations

- If we use sort-based algorithm, it is possible to adapt to changes to  $M$ .
  - If  $M$  shrinks, we can change the  $h$ =size of sublist.
- If the algorithm is hash-based, we can reduce the number of buckets if  $M$  shrinks as long as buckets do not become so large they do not fit in allotted main memory.

# Algorithms with more than two passes

- Generalization of two pass algorithm.
  - Sorting-based
  - Hash-based

# Parallel Algorithms

- Database operations can profit from parallel processing

# Models of parallelism

- Shared memory
  - Each processor can access all the memory of all the processors.
  - Single physical address space
- Shared disk
  - Every processor has its own memory and not accessible by other processors
  - Disks are accessible by every processor through network
- Shared nothing
  - Each processor has its own memory and disk
  - Communication is through network

# Shared Nothing

## Tuple-at-a time operations in parallel

- If there are  $P$  processors, divide  $R$ 's tuples evenly among the  $P$  processors.
- Send the selection and projection to the processors.

# Full Relation Operations

- Use hash to distribute the tuples
- Apply grouping operation in parallel.
- For union, intersection and subtraction use the same hash function to distribute R.

# Performance

- Number of disk I/Os do not change
- Time =  $1/p$  of time + cost of shipping
- Shipment over network is cheaper than disk utilization.