Query Execution

Chapter 15 of GUW

Sections 15.1 to 15.8

Objectives

- Analyze several possible algorithms for each relational algebra operations.
  - The best algorithm depends on the particular relations involved, and on the internal memory available

- Lecture outline

- Query Processor
- 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
- Algorithms Using More Than Two Passes

- Query Processor

- Query processor is a group of DBMS components that turns user queries and data-modification commands into a sequence of database operation and executes those operations.

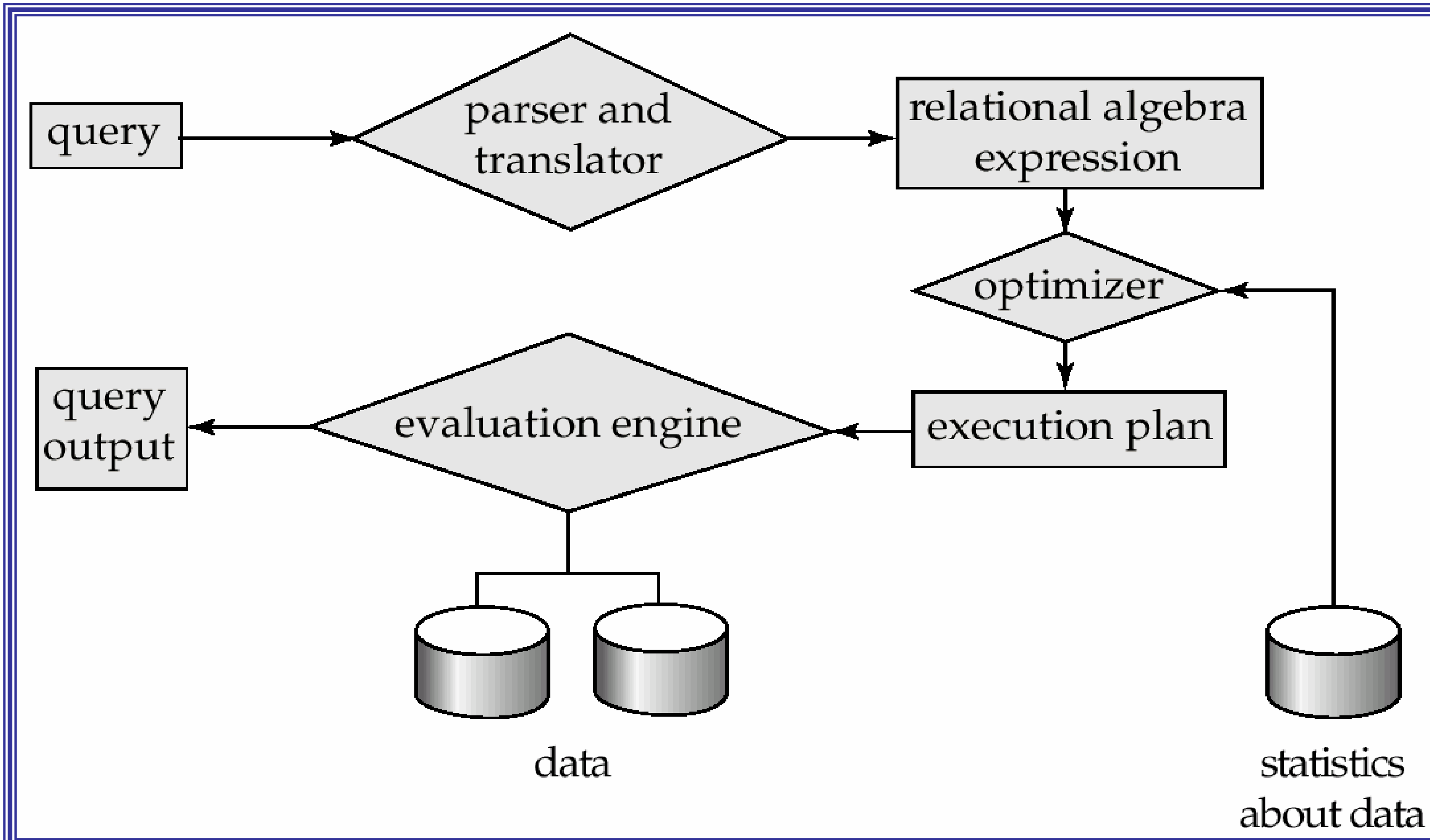
- Query Processor is divided into:
  - Query compilation (Ch 16)
  - Query execution (Ch. 15)

- Query compilation is divided into 2 main components:
  - Parsing
    - A parse tree, representing the query and its structure, is constructed.
  - Query optimization

-- Query Optimization

- **Query rewrite**
  - parse tree is converted into an initial query plan, which is usually an algebraic representation of the query.
  - The initial plan is then transformed into an equivalent plan that is expected to take less time to execute
  - The result of this step is logical query plan
- **Physical plan generation**
  - Selecting algorithms to implement each of the operators of the logical query plan.
  - Selects order of execution of the operators.
  - Includes details of how the queried relations are accessed and when and if a relation should be sorted.
  - Is also represented by an expression tree.

-- Query Processing

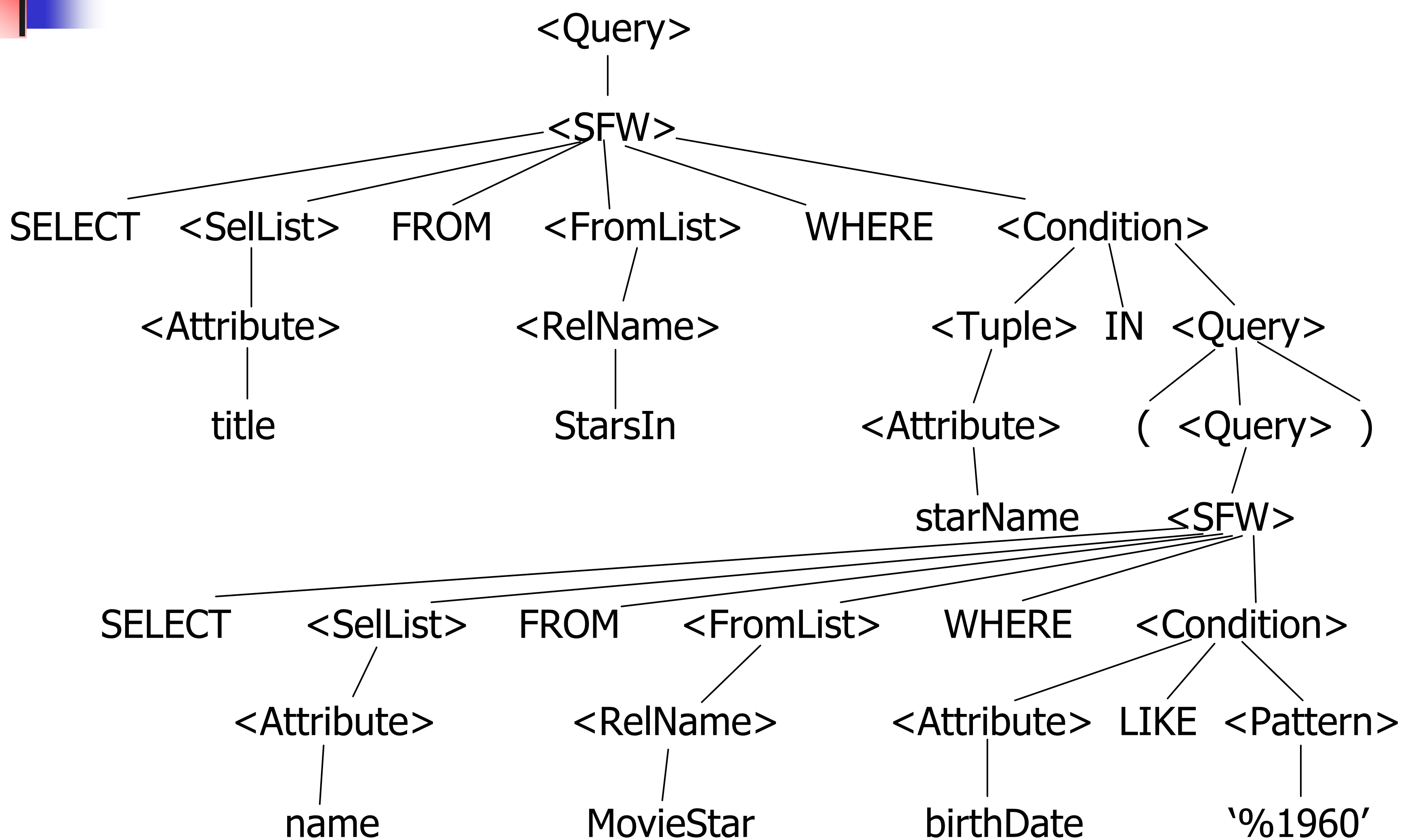


Example: SQL query

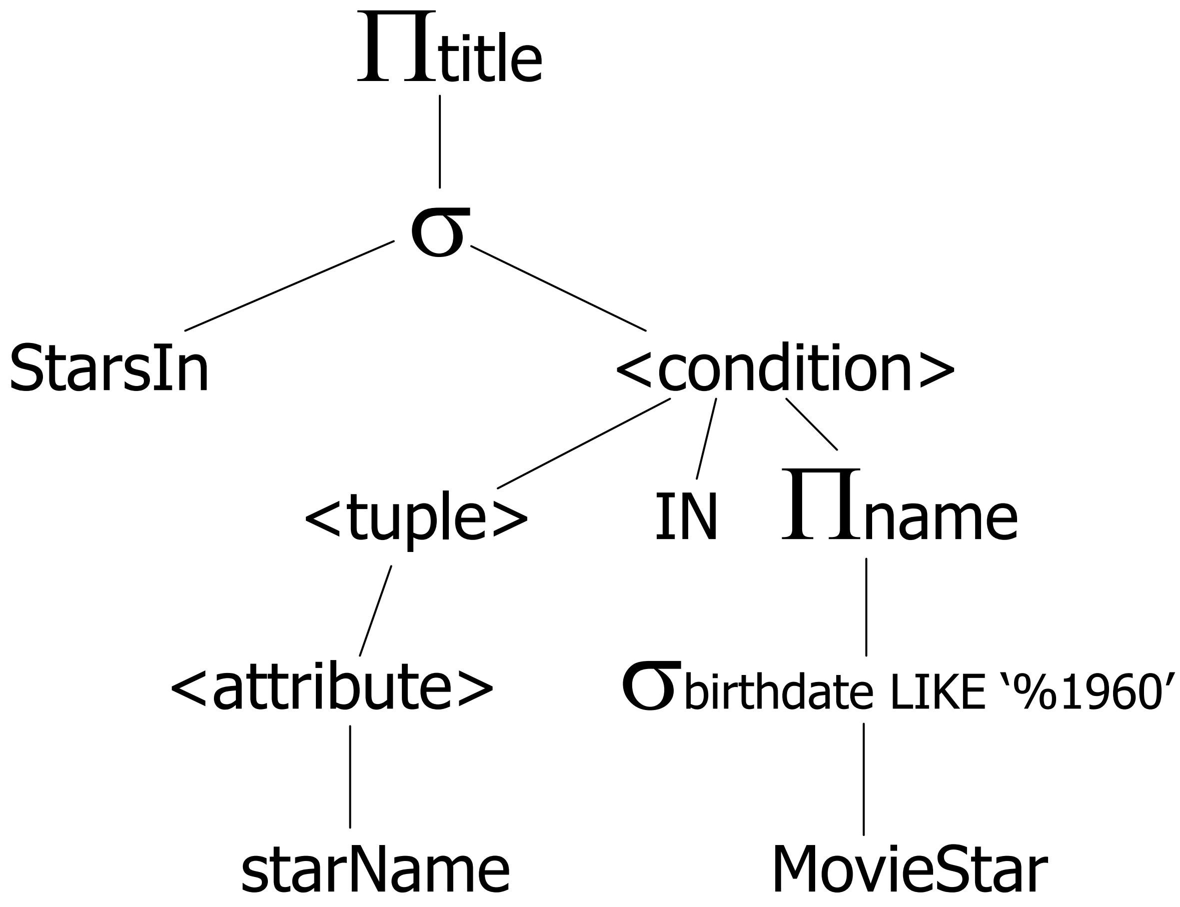
```
SELECT title
FROM StarsIn
WHERE starName IN (
          SELECT name
          FROM MovieStar
          WHERE birthdate LIKE '%1960'
);
```

(Find the movies with stars born in 1960)

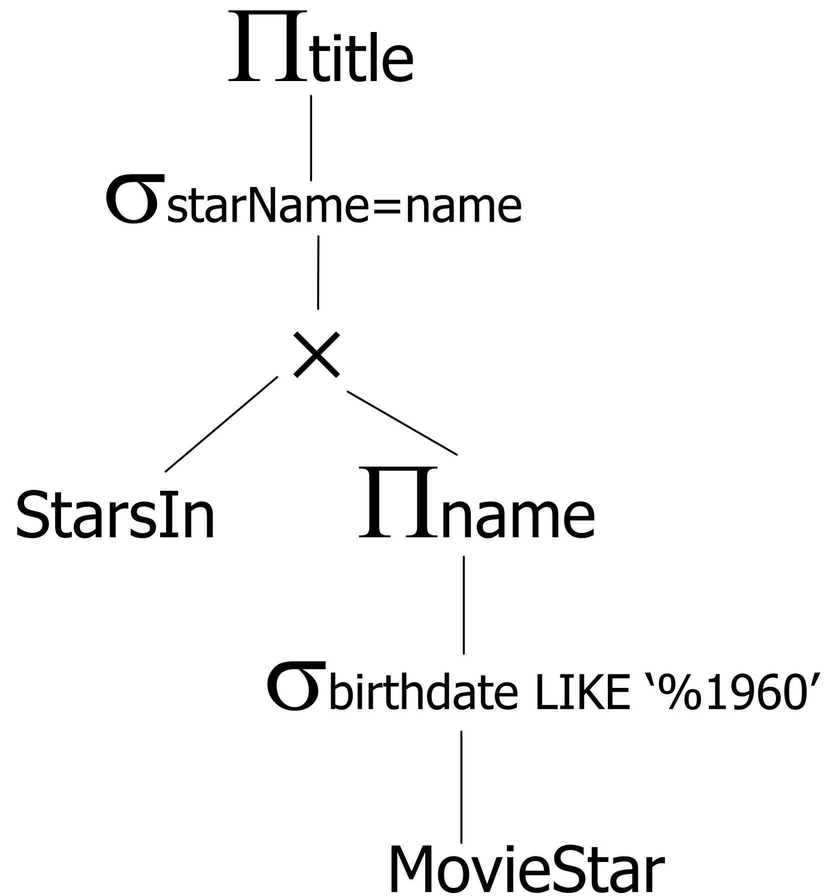
Example: Parse Tree



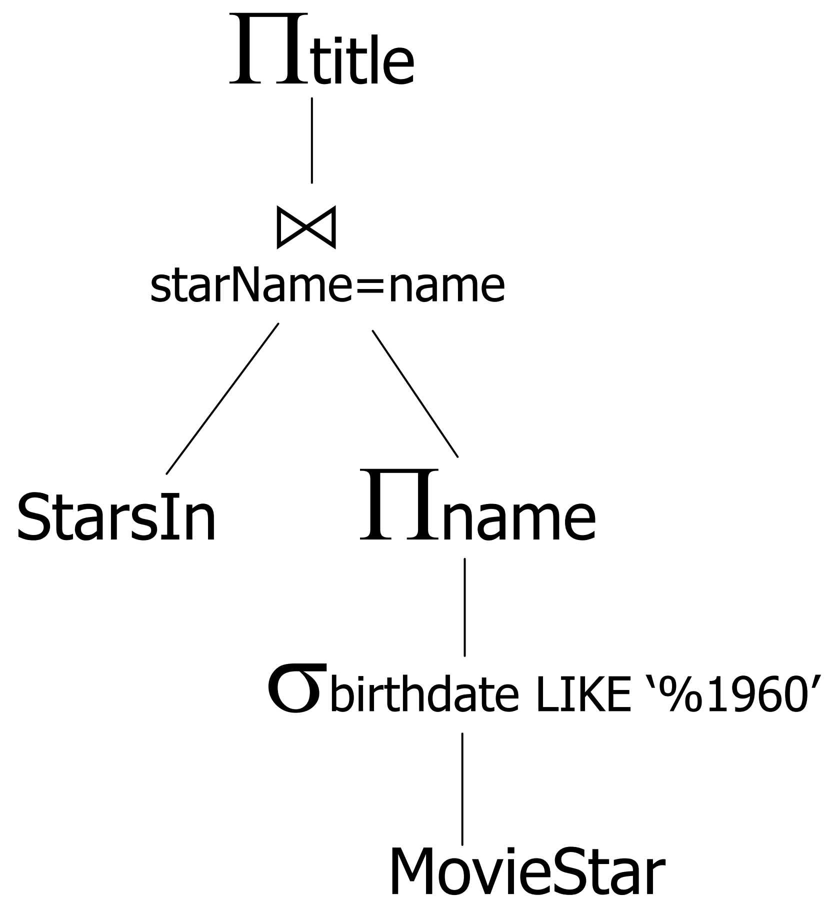
Example: Generating Relational Algebra



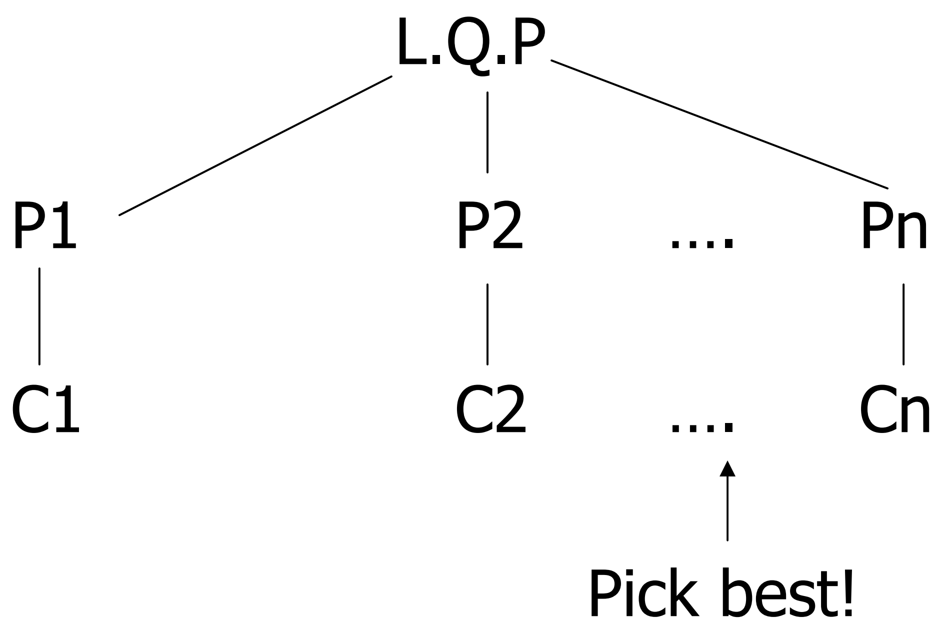
Example: Logical Query Plan



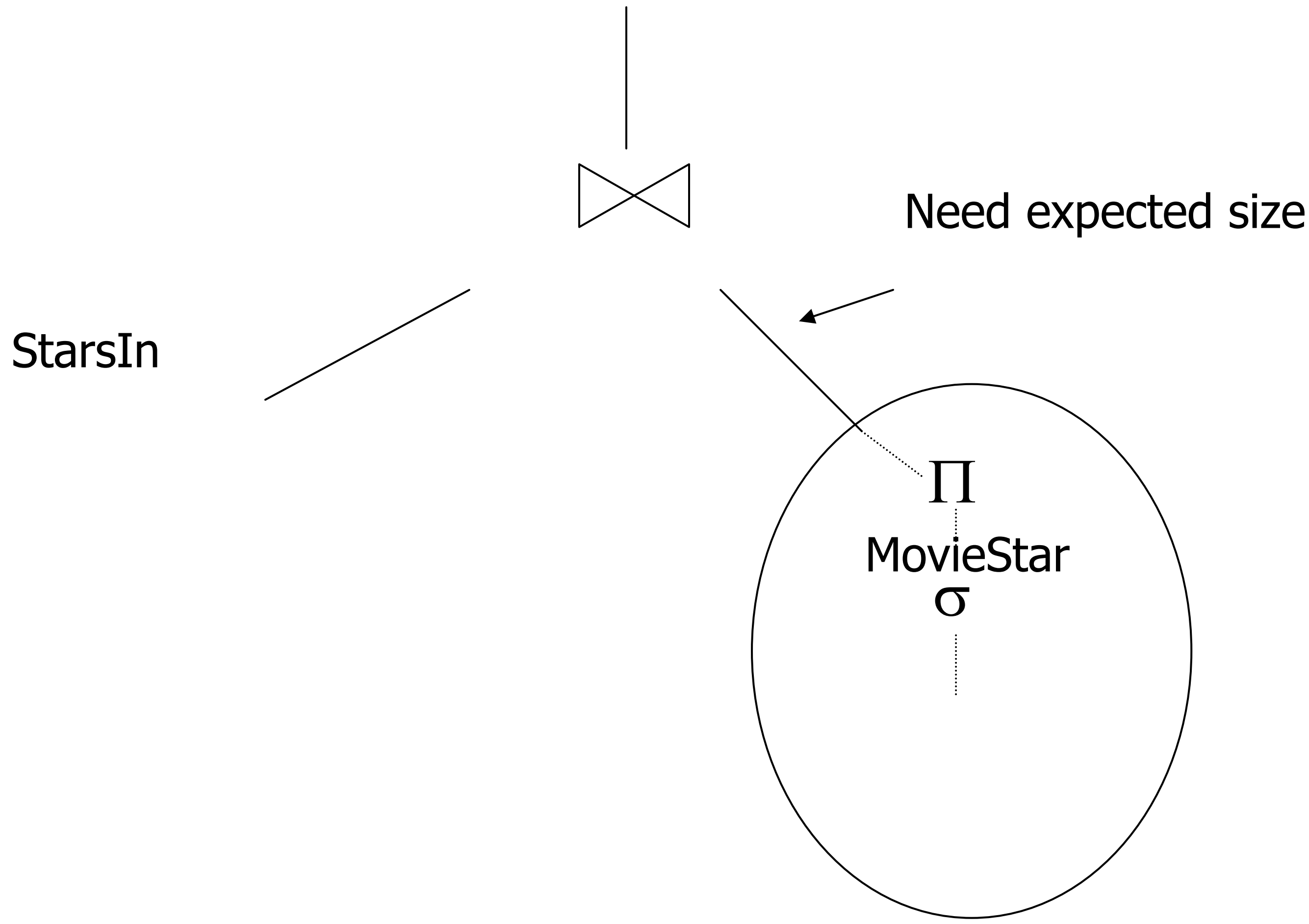
Example: Improved Logical Query Plan



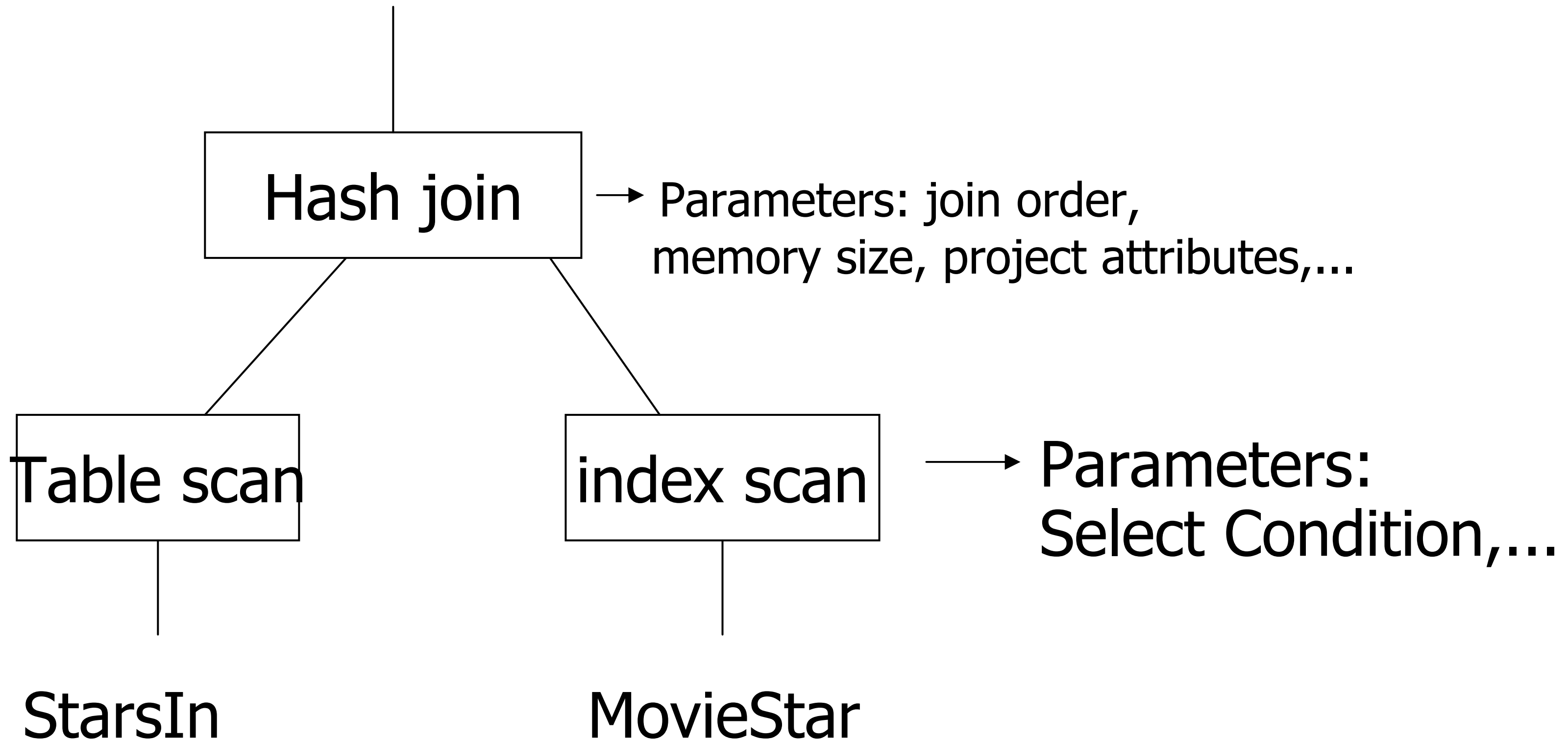
Example: Estimate costs



Example: Estimate Result Sizes



Example: One Physical Plan



- Introduction to Physical-Query-Plan Operators

- What are physical operators
- Scanning tables
- Model of computation
- Parameters for measuring cost
- I/0 Cost of Scan Operator

-- What are Physical Operators

- Are implementations for one of the operators of relational algebra.
- They are also implementation of non relational algebra operators like:
  - Bringing tuples from disk to memory

-- Scanning tables

- Table-scan
  - No index is used
- Index-scan
  - Index is used
- Sorting while scanning
  - Sorting relation R on attribute A while scanning it can be implemented:
    - By index-scan if there is an index on A.
    - By efficient main memory sorting algorithm if R is small and fits in the available memory
    - By Using multiway merge approach if R is too large to fit in main memory.

-- Parameters for measuring Cost

- Assume:
  - Data is accessed one block at a time
  - Memory buffer size = disk block size
  - Arguments of any operator are read from disk
  - Result are not written back to disk
  - Cost of a query is approximated by the number of disk blocks accessed.
- Parameters:
  - **M**: Estimate of memory buffers that can be used by operator
    - Wrong estimation of M can fool the optimizer.
  - **B**: Number of blocks
    - B(R): Number of block needed to hold tuples of R.
  - **T**: Number of tuples
    - T(R): Cardinality of R
  - **V**: Number of distinct values in a column
    - V(R,a): Number of distinct values in column a of relation R.

-- I/0 Cost of Scan Operator

- Table-scan of R
  - Clustered R: B(R)
  - Unclustered R: T(R)
- Index-scan of R:
  - Must be much less than Table-scan
  - To be discussed later
- <u>Note:</u>
  - All our subsequent calculations will assume clustered tables, unless specified.
  - Incase of binary operations S and R will be used, and we will assume B(R) ≥ B(S).

-- Model of Computation

- Assume:
  - Data is accessed one block at a time
  - Memory buffer size = disk block size
  - Arguments of any operator are read from disk
  - Result are not written back to disk
  - Cost of a query is approximated by the number of disk blocks accessed.
  - With binary operations involving Relations R and S, Assume is is smaller unless specified.

- One-Pass Algorithms for DB Operations

- Assumption: $B(S) < B(R)$ and $B(S) < M$
- Unary
  - Selection σ
  - Projection π
  - Duplicate elimination δ
  - Grouping ¥
- Binary
  - Bag Union $\cup_B$
  - Bag Intersection $\cap_B$
  - Bag Difference $-_B$
  - Set union $\cup_S$
  - Set Intersection $\cap_S$
  - Set Difference $-_S$
  - Product X
  - Join ⋈

-- Selection: $\sigma_c(R)$

- Algorithm
  - Read blocks of R one at a time into an input buffer
  - Perform the operation on each tuple
  - Move selected tuples to output buffer
- Memory Structures:
  - None
- Memory size
  - M = 1 suffices
- Cost
  - B(R)

-- Projection: $\pi(R)$

- Algorithm
  - Read blocks of R one at a time into an input buffer
  - Perform the operation on each tuple
  - Move projected tuples to output buffer
- Memory Structures
  - None
- Memory size
  - M = 1 suffices
- Cost
  - B(R)

-- Duplicate Elimination: δ(R)

- Algorithm
  - Read R one block at a time
  - For each tuple:
    - New tuple: add to structure
    - duplicate tuple: ignore
- Memory structures
  - Balanced tree or Hash
- Memory requirement
  - $B(\delta(R)) < M$
- Cost
  - $B(R)$

-- Grouping: ¥L(R) …

- <u>Algorithm</u>
  - Scan the tuples of R one block at a time
  - Compute the aggregate value for the corresponding group.
- <u>Memory Structure</u>
  - Balanced tree or Hash
- <u>Memory requirement</u>
  - M > B (¥L(R))
  - M not directly related to B(R).
- <u>Cost</u>
  - B(R)

-- Bag Union: R U_B S

- Algorithm
  - Read each Block of R one at a time
  - Copy each tuple of R to the output
  - Read each block of S one at a time
  - Copy each tuple of S to the output
- Memory Structures
  - None
- Memory requirement
  - M = 1 suffices
- Cost
  - B(R) + B(S)

-- Bag Intersection: $R \cap_B S$

- Algorithm
  - Read each tuple of S and associate a count which is equal to the number of times it is duplicated.
  - Read each tuple of R, and check whether it is also in S
    - If it is and its count is higher than zero, send the tuple to output and subtract the count.
    - If it isn't in S or its count is zero ignore it
- Memory Structures
  - Balanced tree or Hash
- Memory requirement
  - M > min(B(S), B(R))
- Cost
  - B(R) + B(S)

-- Bag Difference: $S -_B R$

- Algorithm
  - Read each tuple of S and associate a count which is equal to the number of times it is duplicated.
  - Read each tuple of R, and check whether it is also in S
    - If it is, subtract its count.
    - If it isn’t, ignore it
  - The output is those tuples of S with positive count copied as many times as their count.
- <u>Memory Structures</u>
  - Balanced tree or Hash
- <u>Memory requirement</u>
  - M > min(B(S), B(R))
- <u>Cost</u>
  - B(R) + B(S)

-- Set Union: $R \cup_S S$

- <u>Algorithm</u>
  - Read S into M-1 buffers and build a search structure where the search key is the hole tuple
  - Also copy all the S tuples to the output
  - Read each block of R to the Mth buffer one at a time
  - If a tuple t of R is not in S, then t is copied to the output, otherwise t is skipped.
- <u>Memory Structures</u>
  - Btree or Hash
- <u>Memory requirement</u>
  - M > min(B(S), B(R))
- <u>Cost</u>
  - B(R) + B(S)

-- Set Intersection: $R \cap_S S$

- <u>Algorithm</u>
  - Read S into M-1 buffers and build a search structure where the search key is the hole tuple.
  - Read each block of R to the Mth buffer one at a time
  - If a tuple t of R is in S, then copy t to the output, otherwise skip it.
- <u>Memory Structures</u>
  - Balanced tree or Hash
- <u>Memory requirement</u>
  - $M > \min(B(S), B(R))$
- <u>Cost</u>
  - $B(R) + B(S)$

-- Set Difference: S $-_S$ R

- Algorithm
  - Read S into M-1 buffers and build a search structure where the search key is the hole tuple.
  - Read each block of R to the Mth buffer one at a time
  - If a tuple t of R is in S, delete t (in memory) from S
  - Then copy the undeleted tuples of S to the output.
- Memory Structures
  - Balanced tree or Hash
- Memory requirement
  - M > min(B(S), B(R))
- Cost
  - B(R) + B(S)

-- Product: S X R

- Algorithm
  - Read S into M-1 buffers
  - Read each block of R to the Mth buffer one at a time
  - Concatenate each tuple of R with each tuple of S and copy to output
- Memory Structures
  - None
- Memory requirement
  - M > min(B(S), B(R))
- Cost
  - B(R) + B(S)

-- Natural Join: $R(X,Y) \bowtie S(Y,Z)$

- Algorithm
  - Read S into M-1 buffers and build a search structure where the search key is Y.
  - Read each block of R to the Mth buffer one at a time
  - For each tuple t of R, join it with matching tuples of S and copy the result tuples to the output.
- Memory Structures
  - Hash or balanced tree
- Memory requirement
  - $M > \min(B(S), B(R))$
- Cost
  - $B(R) + B(S)$

- Nested-Loop Join: $S \bowtie R$ …

- Assumption B(S) and B(R) > M
- Algorithm

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

… - Nested-Loop Join: $S \bowtie R$

- Memory Structures
  - Hash or balanced tree
- Memory requirement
  - $M \geq 2$
- Cost
  - $B(S) + (B(S) * B(R))/(M-1)$

- Two-Pass Algorithms Based on Sorting

- The basic idea is:
  - Read M blocks of R Sort the M blocks
  - Write the sorted sublist into M disk blocks
  - In some way use the sorted sublists to execute one of the following operators.
    - Duplicate elimination δ
    - Grouping ¥
    - Bag Intersection $\cap_B$
    - Bag Difference $-_B$
    - Set union $\cup_S$
    - Set Intersection $\cap_S$
    - Set Difference $-_S$
    - Join $\bowtie$

-- Duplicate Elimination: δ(R)

- Algorithm
  1. Read the tuples of R into memory, M blocks at a time
  2. Sort each M block
  3. Write each sorted sublist to disk
  4. Load the first block of each sublist into a main memory buffer.
  5. Copy each tuple to the output and ignore its duplicates
  6. If a buffer becomes empty, replace it with the next block from the same sublist.
  7. Repeat steps 5 and 6 until all the blocks of R are processed.

- Memory structures
  - None

- Memory requirement
  - $B(R) < M*M$

- Cost
  - $3 * B(R)$

-- Grouping: ¥L(R)

- Algorithm
  1. Read the tuples of R into memory, M blocks at a time
  2. Sort each M block using the grouping attributes of L
  3. Write each sorted sublist to disk
  4. Load the first block of each sublist into a main memory buffer.
  5. Repeatedly find all the tuples with the least value of the sort key, accumulate its aggregates and copy the result tuple to output.
  6. If a buffer becomes empty, replace it with the next block from the same sublist.
- Memory Structure
  - None
- Memory requirement
  - M > SQRT(B(R))
- Cost
  - 3 * B(R)

-- Set Union: $R \cup_s S$

- Algorithm
  1. Repeatedly bring M blocks of R into memory
  2. Sort their tuples and write the sorted sublists back to disk.
  3. Do the same steps 1 and 2 for S.
  4. Use one main-memory buffer for each sublist of R and S. Initialize each with the first block from the corresponding sublist.
  5. Repeatedly find the first remaining tuple t, among all the buffers.
  6. Copy t to the output and remove its duplicates from the buffers.
  7. If a buffer becomes empty, reload it with the next block from its sublist.
- Memory Structures
  - None
- Memory requirement
  - M > SQRT(B(S) + B(R))
- Cost
  - 3 * (B(R) + B(S))

-- Intersection and Difference

- Algorithm
  - The same as that of U_S except:
    - For $\cap_S$, output t if it appears in R and S
    - **For** $\cap_B$, output t the minimum of the number of times it appears in R and S.
    - For $R -_S S$, output t if and only if it appears in R but not in S.
    - For $R -_B S$, output t, the number of times it appears in R minus the number of times it appears in S.
- Memory Structures
  - None
- Memory requirement
  - M > SQRT(B(S) + B(R))
- Cost
  - 3 * (B(R) + B(S))

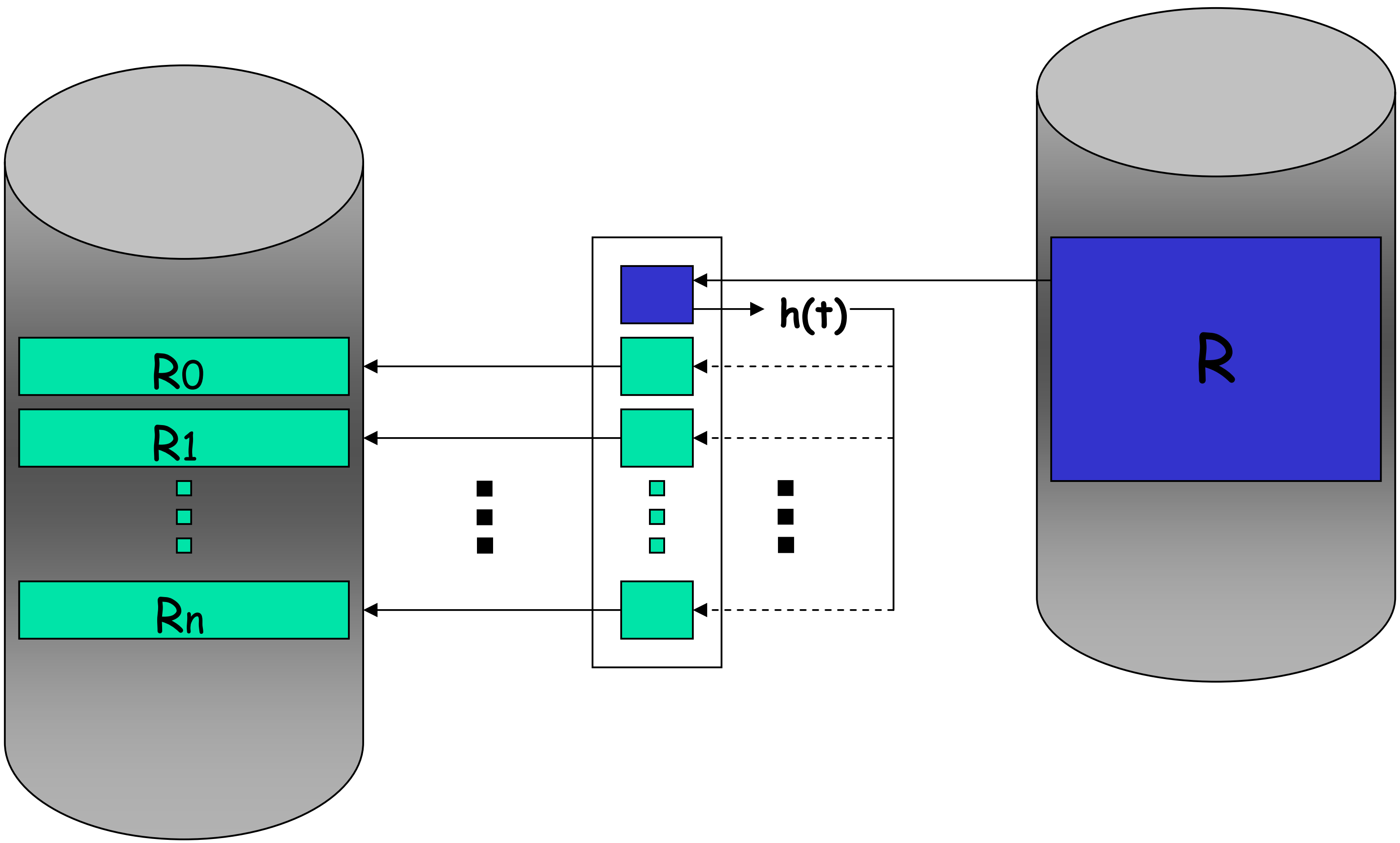
-- Join: R(X,Y) ⋈ S(Y,Z)

- Algorithm
  1. Create a sorted sublist of size M, using Y as the sort key, for both R and S.
  2. Bring the first block of each sublist into buffer. (Assume there are no more than M sublists in all).
  3. Repeatedly find tuples with the next minimum Y value in R, and join them with the corresponding tuples in S.
  4. If the buffer for one of the sublists is exhausted, then replenish it from disk.
- Memory Structures
  - None
- Memory requirement
  - M > SQRT(B(S) + B(R))
- Cost
  - 3 * (B(R) + B(S))

- Two-Pass Algorithms Based on Hash

- Partitioning
- operators.
  - Duplicate elimination δ
  - Grouping ¥
  - Bag Intersection $\cap_B$
  - Bag Difference $-_B$
  - Set union $\cup_S$
  - Set Intersection $\cap_S$
  - Set Difference $-_S$
  - Join $\bowtie$

-- Partition Relations By Hashing



-- Duplicate Elimination: δ(R)

- Algorithm
  - Has R into M-1 partitions
  - Read each partition and out put distinct copies. (duplicates will has to the same bucket.)
- Memory Structures
  - None
- Memory requirement
  - M < SQRT(B(R))
- Cost
  - 3 * (B(R))

-- Grouping and Aggregation: ¥L(R)

- <u>Algorithm</u>
  - Hash R into M-1 partitions using the attributes in *L*
  - Use the one pass algorithm to process each bucket in turn
- <u>Memory Structures</u>
  - Balanced tree or hash
- <u>Memory requirement</u>
  - $M < SQRT(B(R))$
- <u>Cost</u>
  - 3 * (B(R))

-- The Rest of the Relational operators

- Algorithm
  - Partition R and S into M partitions
  - Consider each partition as a mini table
  - Use the one-pass algorithm on this mini-tables to implement the rest of the relational operators.

- Summary

Operation	Memory	Cost
δ, ¥	SQRT(B(R))	3B(R)
∪, ∩, -	SQRT(B(S))	3(B(R) + B(S))
⋈	SQRT(BS)	(3-2M/B(S))(B(R)+B(S))

- Summary

- Query Processor
- 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
- Algorithms Using More Than Two Passes

- Reference

- Sections 15.1 to 15.8 of GUW

END