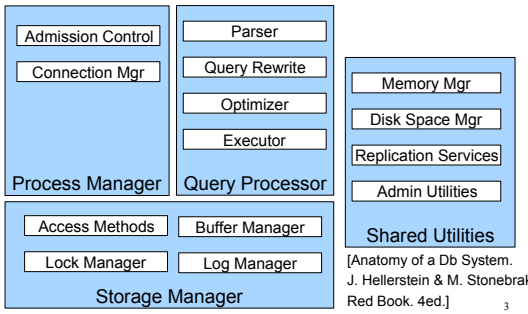
CSE 444: Database Internals

Lecture 7
Query Execution and
Operator Algorithms (part 1)

What We Have Learned So Far

- Overview of the architecture of a DBMS
  - Main components (summarized on next slide)
  - Steps involved in query evaluation
- Access methods
  - Storing data in heap files or sequential files
  - Indexes (hash or B+ trees)
- Role of buffer manager
- Interaction of components during query execution

DBMS Architecture

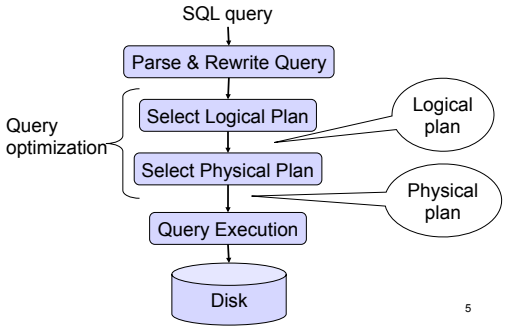


[Anatomy of a Db System. J. Hellerstein & M. Stonebraker. Red Book. 4ed.]

What We Will Learn Next

- How to answer queries efficiently!
  - Operator algorithms, especially for joins
  - How to leverage indexes for selections and joins
  - How to compute costs of individual operations
- How to automatically find good query plans
  - How to compute the cost of a complete plan?
  - How to pick a good query plan for a query?
- Start by reviewing physical plans

Query Evaluation Steps Review



Physical Query Plan

(On the fly) π_{sname}

(On the fly) $\sigma_{sscity='Seattle' \wedge sstate='WA' \wedge pno=2}$

(Nested loop) $\bowtie_{sno = sno}$

Suppliers (File scan)

Supplies (File scan)

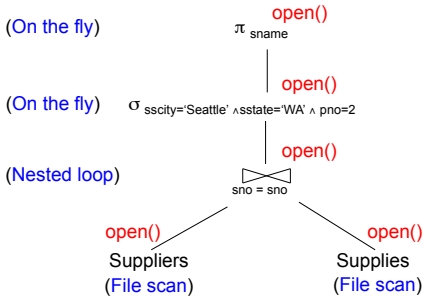
Physical Query Plan

- Logical query plan with extra annotations
- **Access path selection** for each relation
  - Use a file scan or use an index with a predicate
  - We learned various alternatives in past few lectures
- **Implementation choice** for each operator
  - We will learn about operator algorithms
- **Scheduling decisions** for operators
  - Pipelined execution
  - Or intermediate tuple materialization

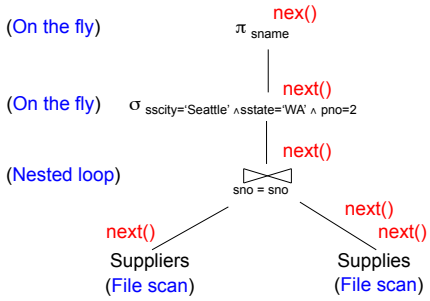
Iterator Interface

- Each **operator implements this interface**
- Interface has only three methods
- **open()**
  - Initializes operator state
  - Sets parameters such as selection condition
- **next()**
  - Operator invokes get_next() recursively on its inputs
  - Performs processing and produces an output tuple
- **close()**: clean-up state

Pipelined Query Execution



Pipelined Query Execution



Pipelined Execution

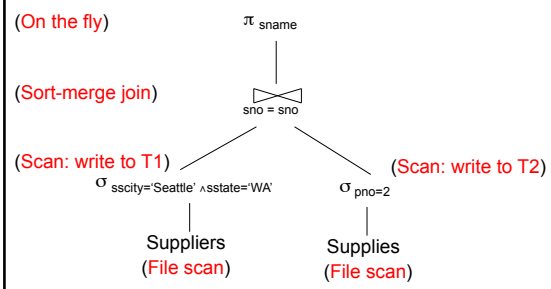
- Applies parent operator to tuples directly as they are produced by child operators
- Benefits
  - No operator synchronization issues
  - Saves cost of writing intermediate data to disk
  - Saves cost of reading intermediate data from disk
  - Good resource utilizations on single processor
- This approach is used whenever possible

Intermediate Tuple Materialization

- Writes the results of an operator to an intermediate table on disk

- No direct benefit but
- Necessary for some operator implementations
- When operator needs to examine the same tuples multiple times

Intermediate Tuple Materialization



Memory Management

- Each operator has some pre-allocated space for input and output tuples
  - These are tuples in-flight between operators
  - These input/output queues either point to base data in buffer pool
  - Or point to new tuples on the heap
- Each operator can also allocate separate memory for its internal state
  - Either on heap or buffer pool (depends on system)

Memory Management (cont.)

- DBMS limits how much memory each operator can use
  - This limit can be configurable
- DBMS limits how much memory each query can use
  - This limit can be configurable

Query Execution Bottom Line

- SQL query transformed into physical plan
  - Logical query plan with extra annotations
  - **Access path selection** for each relation
  - **Implementation choice** for each operator
  - **Scheduling decisions** for operators
- Execution of the physical plan is pull-based
- Operators given a limited amount of memory

Operator Algorithms

Why Learn About Op Algos?

- Good algorithms can greatly improve performance
  - Need to know operator algorithms to understand query plans
  - Need to understand query plans to tune a DBMS
- Implemented in commercial DBMSs
  - DBMSs implement different subsets of known algorithms
- Operator costs are first step toward query optimization
- Basic ideas to achieve high performance in operators go beyond relational operators

Operator Algorithms

- How to compare implementations/algorithms?
  - Using a cost model: IO, CPU, (and network bw)
  - Later, will see how this plays a role in optimization
- Some key design criteria
  - Cost: Different algorithms have different costs
    - Cost depends on input data and other parameters
  - Memory utilization
    - Operators only have access to limited amount of memory
  - Load balance (for parallel operators)

Cost Parameters

- In database systems the data is on disk
- **Cost = total number of I/Os**
  - This is a simplification
  - Normally, need to consider IO, CPU, and network
- Parameters:
  - **B(R) = # of blocks (i.e., pages) for relation R**
  - **T(R) = # of tuples in relation R**
  - **V(R, a) = # of distinct values of attribute a**
    - When a is a key, V(R,a) = T(R)
    - When a is not a key, V(R,a) can be anything < T(R)

Cost

- Cost of an operation = number of disk I/Os to
  - Read the operands
  - Compute the result
- Cost of writing the result to disk is *not included*
  - Need to count it separately when applicable

Cost of Scanning a Table

- Result may be unsorted: B(R)
- Result needs to be sorted: 3B(R)
  - We will discuss sorting later

Outline

- **Join operator algorithms**
  - One-pass algorithms (Sec. 15.2 and 15.3)
  - Index-based algorithms (Sec 15.6)
  - Two-pass algorithms (Sec 15.4 and 15.5)
  - Note about readings:
    - In class, we will discuss only algorithms for join operator (because other operators are easier)
    - Read the book to get more details about these algos
    - Read the book to learn about algos for other operators

Basic Join Algorithms

- Logical operator:
  - Product(pname, cname) ⋈ Company(cname, city)
- Propose three physical operators for the join, assuming the tables are in main memory:
  - **Hash join**
  - **Nested loop join**
  - **Sort-merge join**

Hash Join

Hash join: R ⋈ S

- Scan R, build buckets in main memory
- Then scan S and join
- Cost: B(R) + B(S)

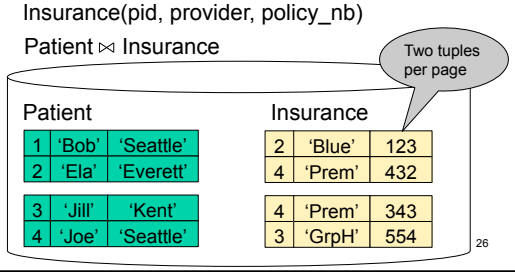
- One-pass algorithm when B(R) <= M
  - By "one pass", we mean that the operator reads its operands only once. It does not write intermediate results back to disk.

Hash Join Example

Patient(pid, name, address)

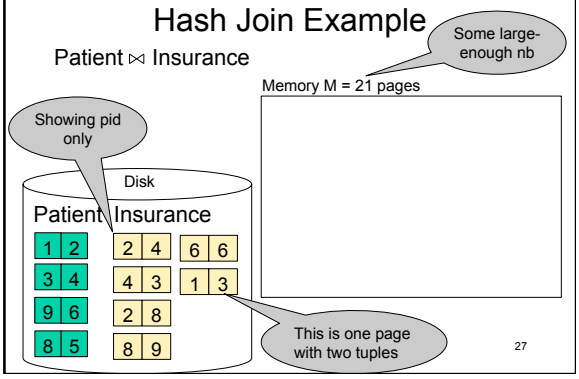
Insurance(pid, provider, policy_nb)

Patient ⋈ Insurance



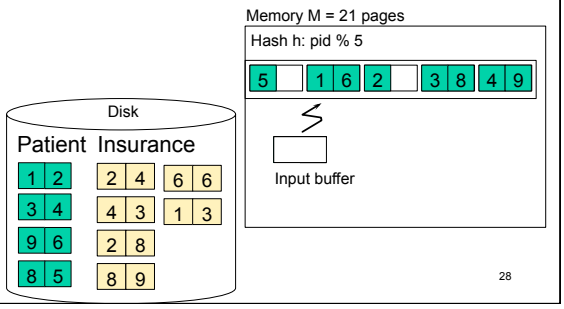
Hash Join Example

Patient ⋈ Insurance



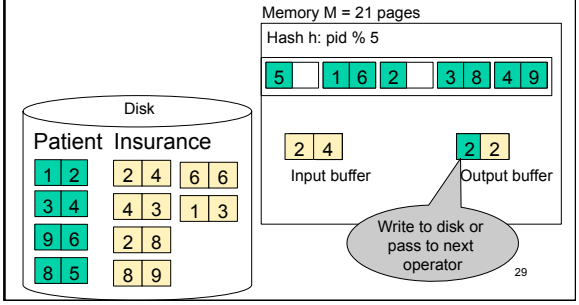
Hash Join Example

Step 1: Scan Patient and create hash table in memory



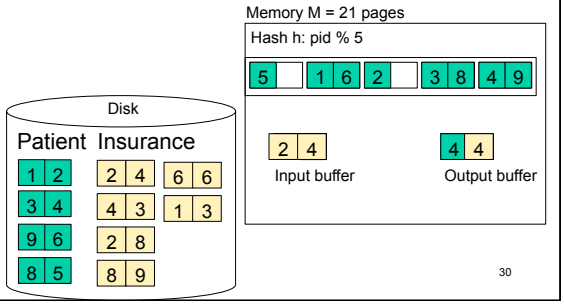
Hash Join Example

Step 2: Scan Insurance and probe into hash table



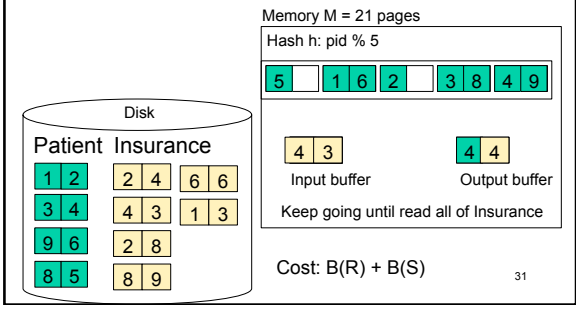
Hash Join Example

Step 2: Scan Insurance and probe into hash table



Hash Join Example

Step 2: Scan Insurance and probe into hash table



Cost: B(R) + B(S)

Hash Join Details

```
Open( ) {
  H = newHashTable( );
  R.Open( );
  x = R.GetNext( );
  while (x != null) {
        H.insert(x); x = R.GetNext( );
  }
  R.Close( );
  S.Open( );
  buffer = [ ];
}
```


Hash Join Details

```
GetNext( ) {
  while (buffer == [ ]) {
        x = S.GetNext( );
        if (x==Null) return NULL;
        buffer = H.find(x);
  }
  z = buffer.first( );
  buffer = buffer.reset( );
  return z;
}
```


Hash Join Details

```
Close( ) {
    release memory (H, buffer, etc.);
    S.Close( )
}
```


Nested Loop Joins

- Tuple-based nested loop R ⋈ S
- R is the outer relation, S is the inner relation

```
for each tuple r in R do
   for each tuple s in S do
      if r and s join then output (r,s)
```

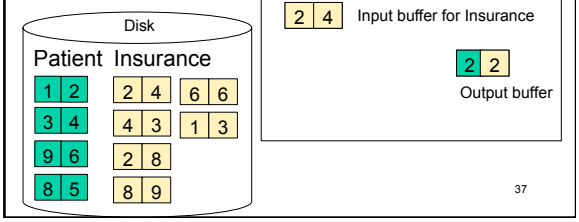
- Cost: B(R) + T(R) B(S)
- Not quite one-pass since S is read many times

Page-at-a-time Refinement

```
for each page of tuples r in R do
   for each page of tuples s in S do
         for all pairs of tuples
               if r and s join then output (r,s)
```

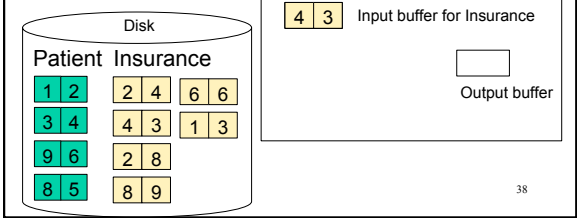
- Cost: B(R) + B(R)B(S)

Nested Loop Example



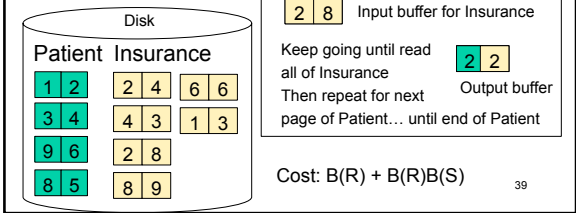
37

Nested Loop Example



38

Nested Loop Example



Cost: B(R) + B(R)B(S)

39

Sort-Merge Join

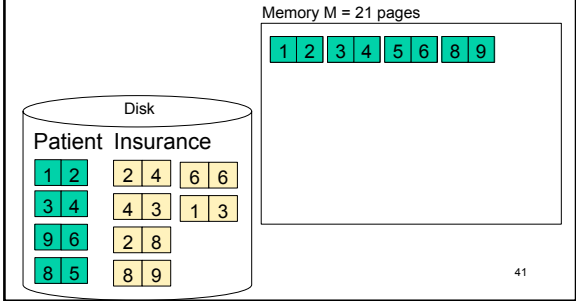
Sort-merge join: R ⋈ S

- Scan R and sort in main memory
- Scan S and sort in main memory
- Merge R and S

- Cost: B(R) + B(S)
- One pass algorithm when B(S) + B(R) <= M
- Typically, this is NOT a one pass algorithm

Sort-Merge Join Example

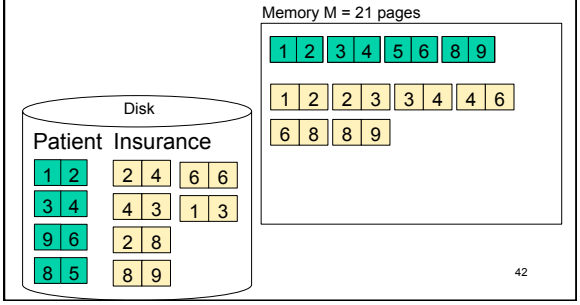
Step 1: Scan Patient and sort in memory



41

Sort-Merge Join Example

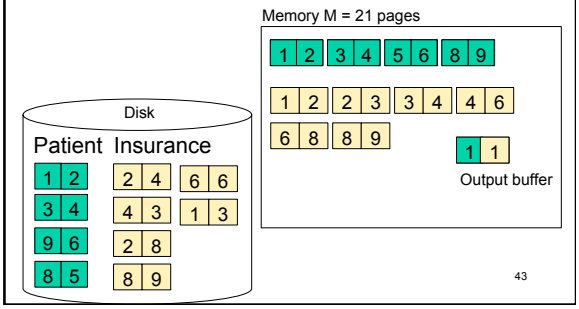
Step 2: Scan Insurance and sort in memory



42

Sort-Merge Join Example

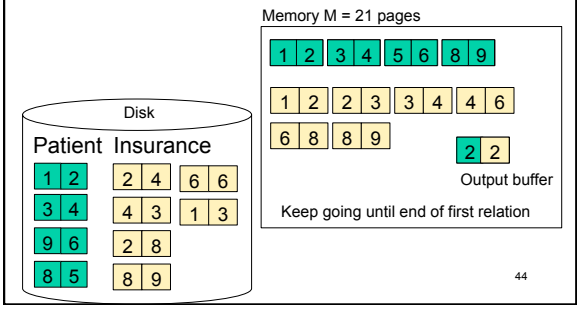
Step 3: Merge Patient and Insurance



43

Sort-Merge Join Example

Step 3: Merge Patient and Insurance



44