CSE 444: Database Internals

Lecture 7

Query Execution and

Operator Algorithms (part 1)

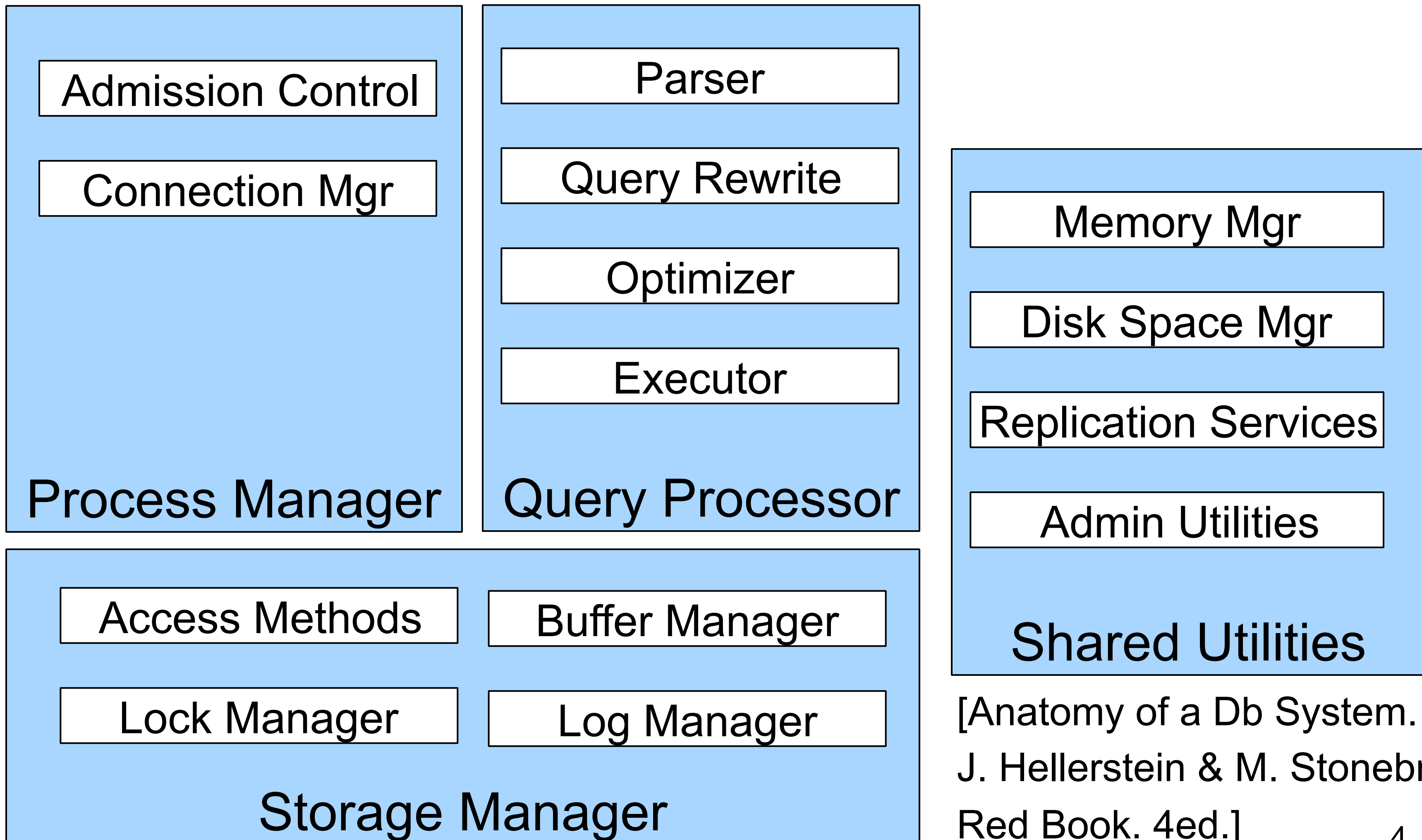
Announcements

- Lab 2 / part 1 due Friday, 11pm
- CSE544M: review 2 due today, 11pm

What We Have Learned So Far

- Overview of the architecture of a DBMS
- Access methods
  - Heap files, sequential files, Indexes (hash or B+ trees)
- Role of buffer manager

DBMS Architecture

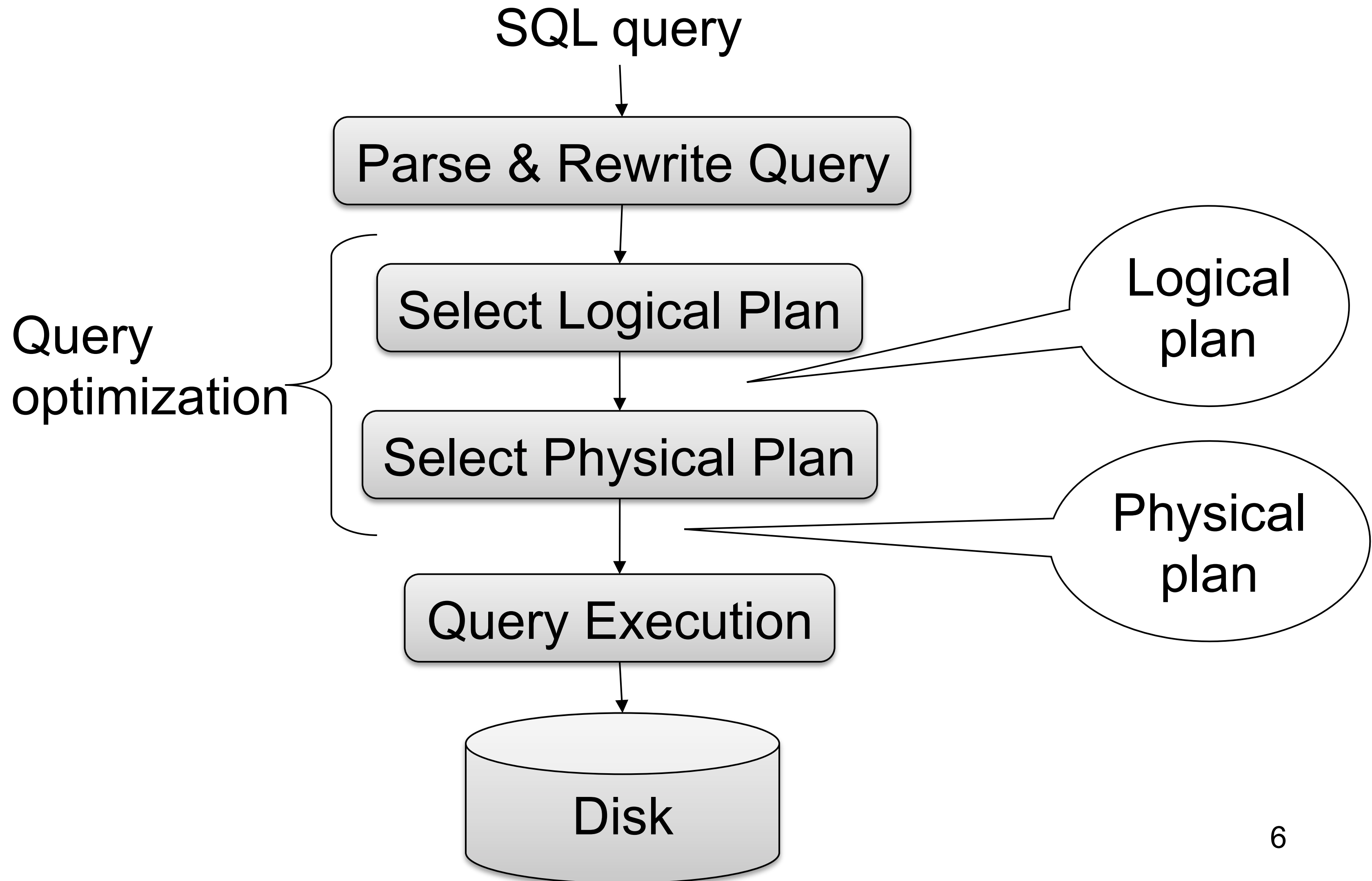


[Anatomy of a Db System. J. Hellerstein & M. Stonebraker. Red Book. 4ed.]

Next Lectures

- How to answer queries efficiently!
  - Operator algorithms, indexes
- 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

Query Evaluation Steps Review

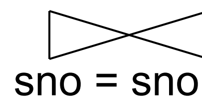


Physical Query Plan

(On the fly) π_{sname}

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

(Nested loop)



sno = sno

Suppliers (File scan)

Supplies (File scan)

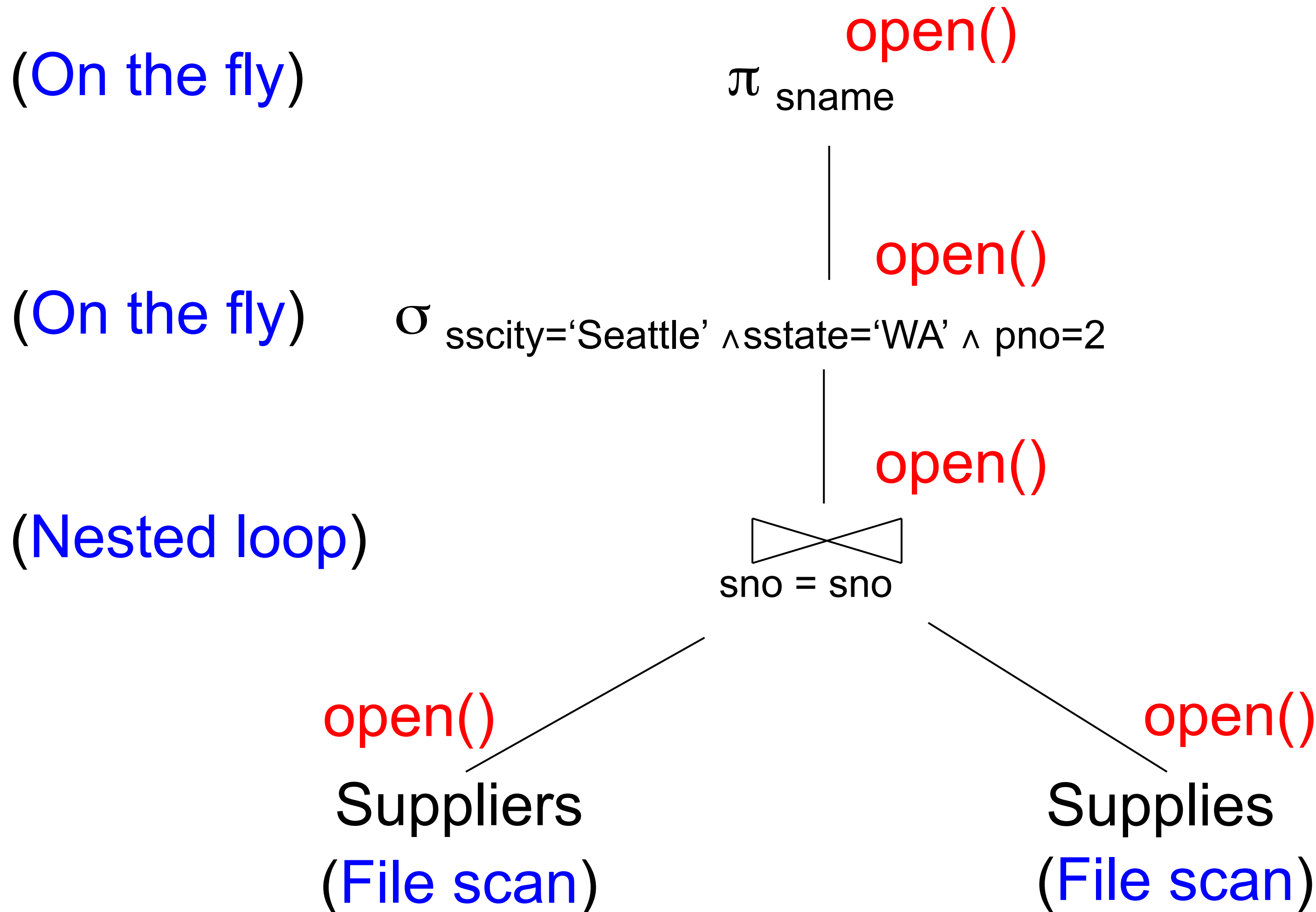
Physical Query Plan

- **Access path selection** for each relation:
  - File scan, or
  - Index lookup with a predicate
- **Implementation choice** for each operator
  - We will learn different algorithms
- **Scheduling decisions** for operators
  - Pipelined execution, or
  - Intermediate tuple materialization

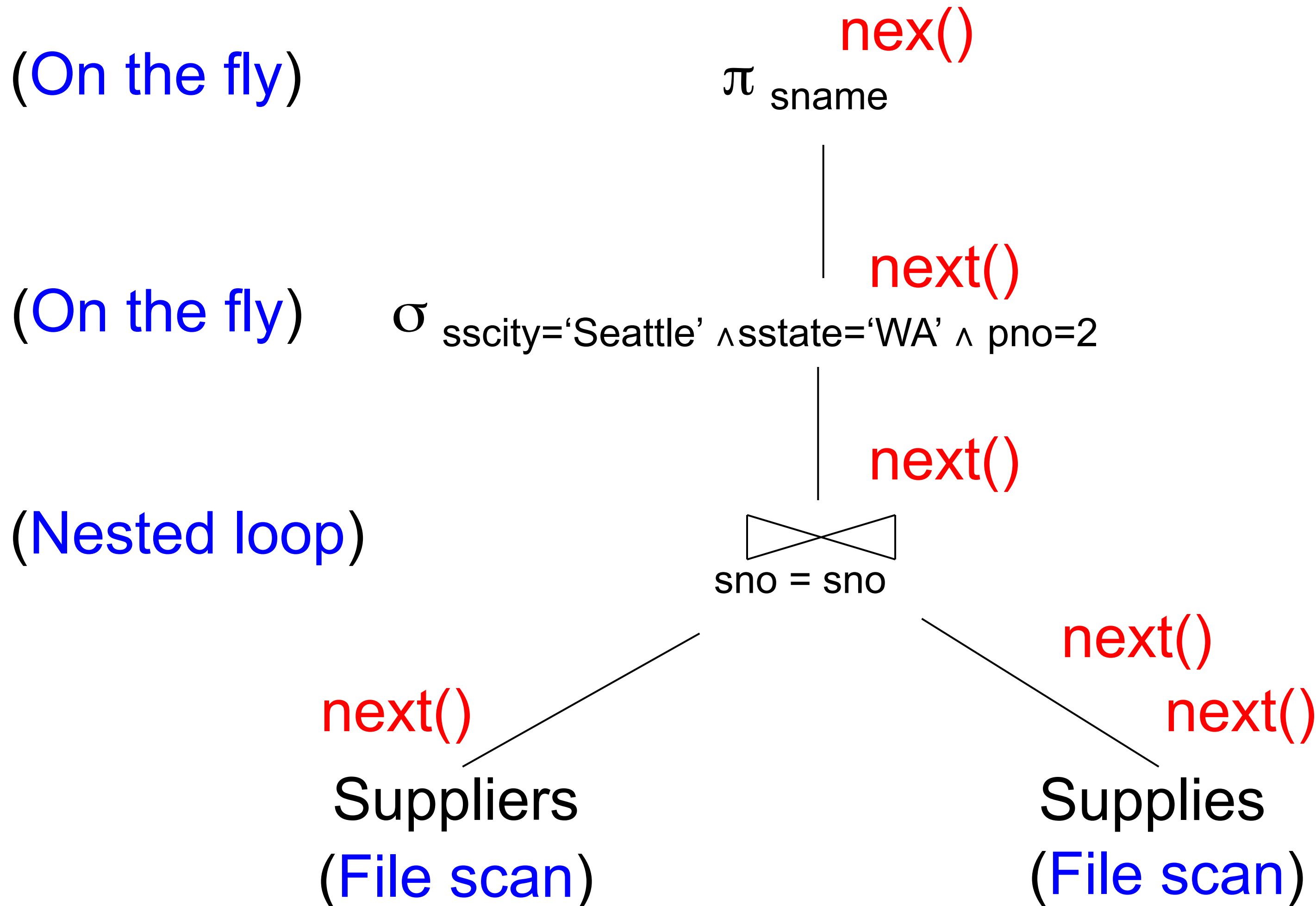
Iterator Interface

- **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

- Tuples generated by an operator are immediately sent to the parent

- Benefits:
  - No operator synchronization issues
  - Saves cost of writing intermediate data to disk
  - Saves cost of reading intermediate data from disk
- This approach is used whenever possible

Intermediate Tuple Materialization

- Tuples generated by an operator are written to disk an in intermediate table

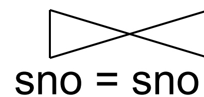
- No direct benefit
- Necessary:
  - For certain operator implementations
  - When we don't have enough memory

Intermediate Tuple Materialization

(On the fly)

π_{sname}

(Sort-merge join)



sno = sno

(Scan: write to T1)

$\sigma_{\text{sscity='Seattle'} \wedge \text{sstate='WA'}}$

(Scan: write to T2)

$\sigma_{\text{pno=2}}$

Suppliers

(File scan)

Supplies

(File scan)

Memory Management

Each operator:

- Pre-allocates heap space for tuples
  - Pointers to base data in buffer pool
  - Or new tuples on the heap
- Allocates memory for its internal state
  - Either on heap or buffer pool (depends on system)

DMBS may limit how much memory each operator, or each query can use

Query Execution Bottom Line

- SQL query transformed into physical plan
  - **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

Operator Algorithms

Design criteria

- Cost: IO, CPU, Network
- Memory utilization
- Load balance (for parallel operators)

Cost Parameters

- **Cost = total number of I/Os**
  - This is a simplification that ignores CPU, 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)**

Convention

- Cost = the cost of reading operands from disk
- Cost of writing the result to disk is *not included*; need to count it separately when applicable

Example: 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 discuss only algorithms for joins
  - Other operators are easier: read the book

Join Algorithms

- Hash join
- Nested loop join
- Sort-merge join

Hash Join

Hash join: $R \bowtie S$

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

- One-pass algorithm when $B(R) \leq M$

Hash Join Example

Patient(pid, name, address)

Insurance(pid, provider, policy_nb)

Patient $\bowtie$ Insurance

Two tuples per page

Patient

1	'Bob'	'Seattle'
2	'Ela'	'Everett'

3	'Jill'	'Kent'
4	'Joe'	'Seattle'

Insurance

2	'Blue'	123
4	'Prem'	432

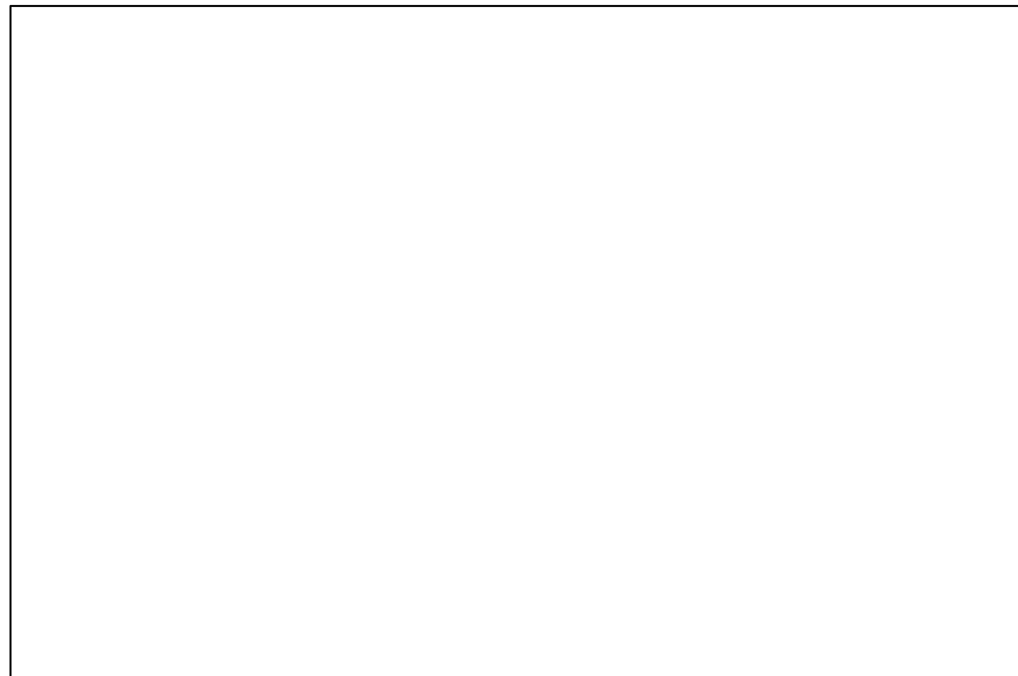
4	'Prem'	343
3	'GrpH'	554

Hash Join Example

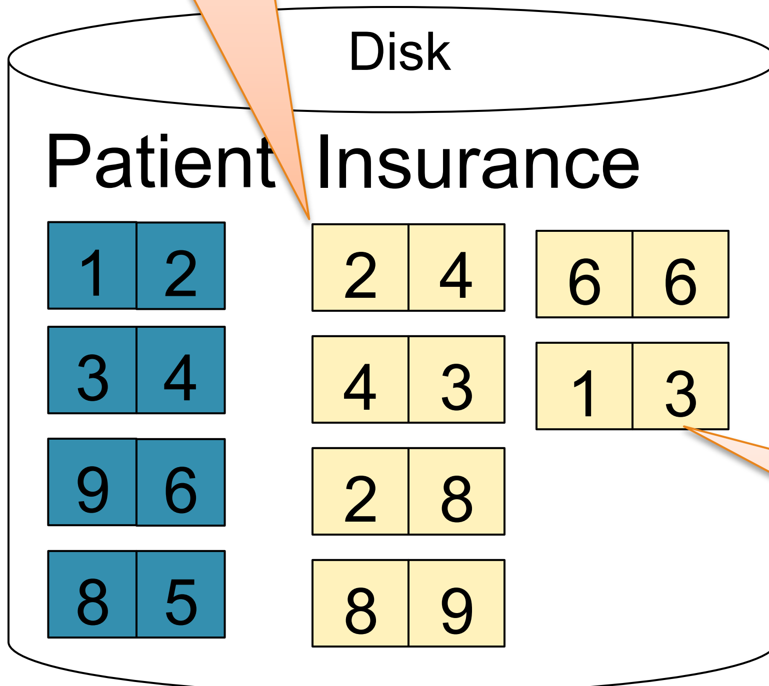
Patient ⋈ Insurance

Some large-enough nb

Memory M = 21 pages



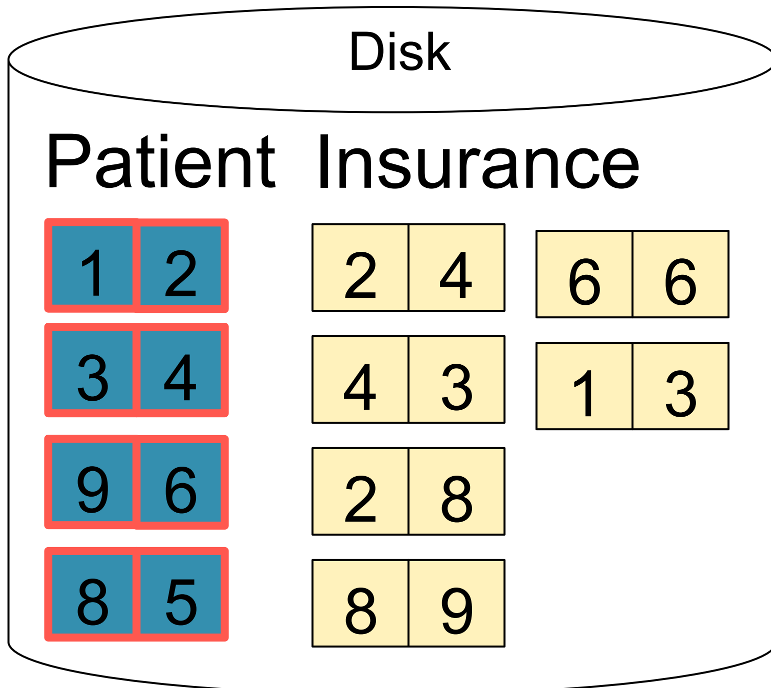
Showing pid only

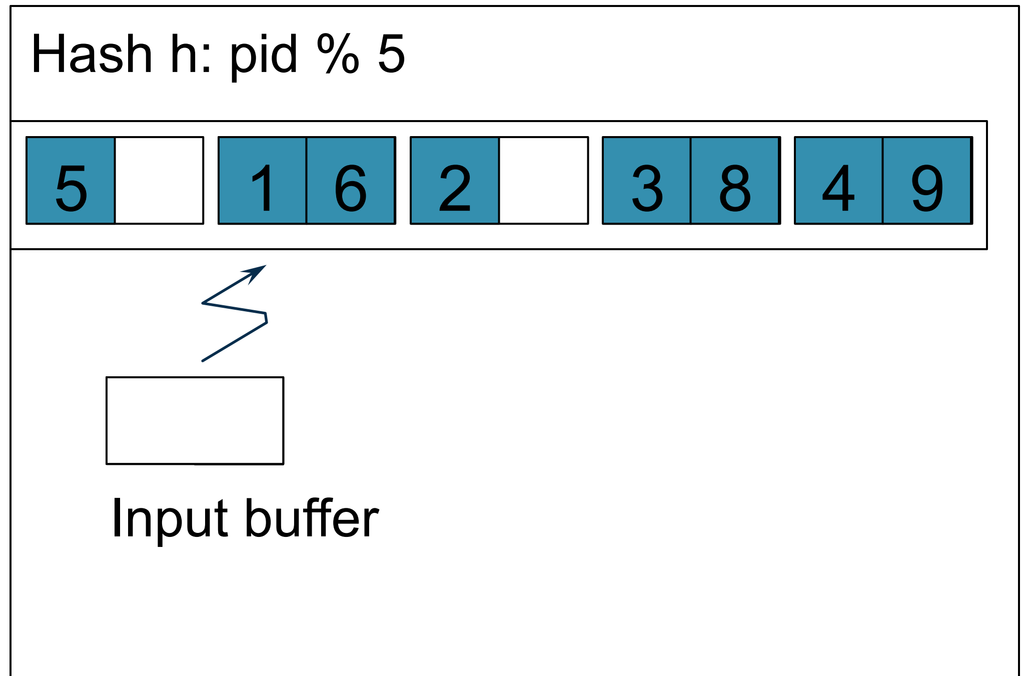


This is one page with two tuples

Hash Join Example

Step 1: Scan Patient and build hash table in memory





Hash Join Example

Step 2: Scan Insurance and probe into hash table

Memory M = 21 pages

Hash h: pid % 5

5 | | 1 6 | 2 | | 3 8 | 4 9

Disk

Patient: 1 2, 3 4, 9 6, 8 5

Insurance: 2 4, 4 3, 2 8, 8 9, 6 6, 1 3

Input buffer: 2 4

Output buffer: 2 2

Write to disk or pass to next operator

Hash Join Example

Step 2: Scan Insurance and probe into hash table

Memory M = 21 pages

Hash h: pid % 5

5		1	6	2		3	8	4	9

2	4

Input buffer

4	4

Output buffer

Disk

Patient

1	2
3	4
9	6
8	5

Insurance

2	4
4	3
2	8
8	9
6	6
1	3

Hash Join Example

Step 2: Scan Insurance and probe into hash table

Memory M = 21 pages

Hash h: pid % 5

5 | 1 6 | 2 | 3 8 | 4 9

4 3

Input buffer

4 4

Output buffer

Keep going until read all of Insurance

Disk

Patient: 1 2, 3 4, 9 6, 8 5

Insurance: 2 4, 4 3, 2 8, 8 9, 6 6, 1 3

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.removeFirst( );
  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 t1 in R do
   for each tuple t2 in S do
      if t1 and t2 join then output (t1,t2)
```

What is the Cost?

Nested Loop Joins

- Tuple-based nested loop $R \bowtie S$
- R is the outer relation, S is the inner relation

```
for each tuple t1 in R do
   for each tuple t2 in S do
      if t1 and t2 join then output (t1,t2)
```

What is the Cost?

- Cost: B(R) + T(R) B(S)
- Multiple-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 t1 in r, t2 in s
              if t1 and t2 join then output (t1,t2)
```

What is the Cost?

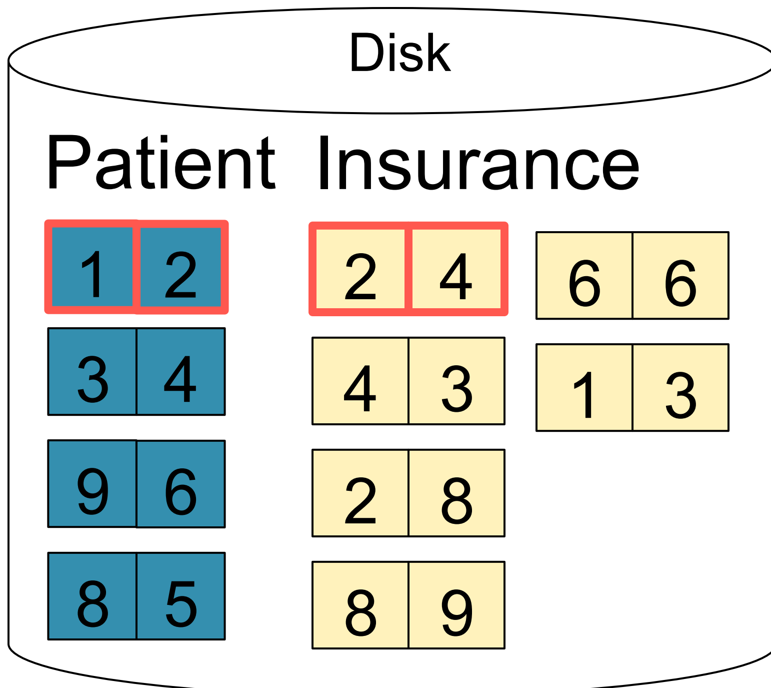
Page-at-a-time Refinement

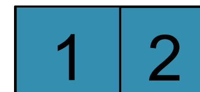
```
for each page of tuples r in R do
   for each page of tuples s in S do
         for all pairs of tuples t1 in r, t2 in s
               if t1 and t2 join then output (t1,t2)
```

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

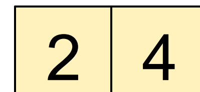
What is the Cost?

Page-at-a-time Refinement

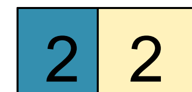




Input buffer for Patient

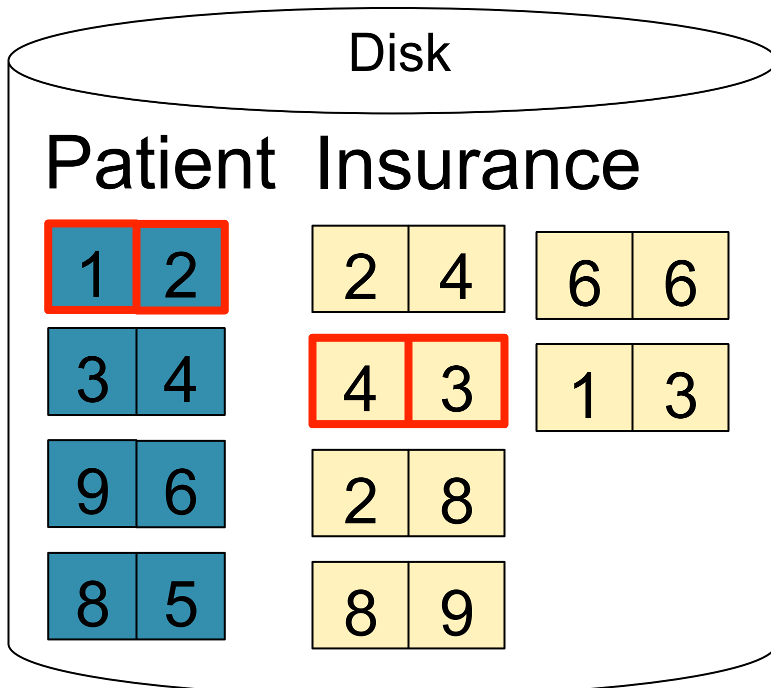


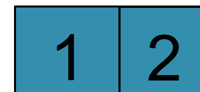
Input buffer for Insurance



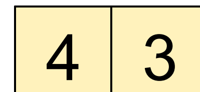
Output buffer

Page-at-a-time Refinement





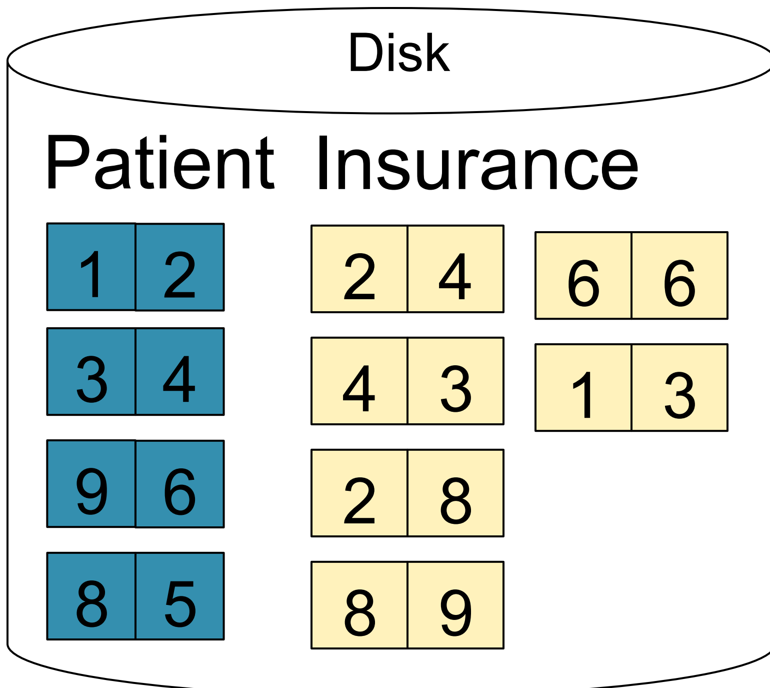
Input buffer for Patient



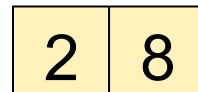
Input buffer for Insurance

Output buffer

Page-at-a-time Refinement



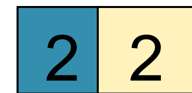
Input buffer for Patient



Input buffer for Insurance

Keep going until read all of Insurance

Then repeat for next page of Patient… until end of Patient



Output buffer

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

Block-Nested-Loop Refinement

```
for each group of M-1 pages r in R do
   for each page of tuples s in S do
          for all pairs of tuples t1 in r, t2 in s
                 if t1 and t2 join then output (t1,t2)
```

What is the Cost?

Block-Nested-Loop Refinement

```
for each group of M-1 pages r in R do
  for each page of tuples s in S do
      for all pairs of tuples t1 in r, t2 in s
          if t1 and t2 join then output (t1,t2)
```

- Cost: B(R) + B(R)B(S)/(M-1)

What is the Cost?

Sort-Merge Join

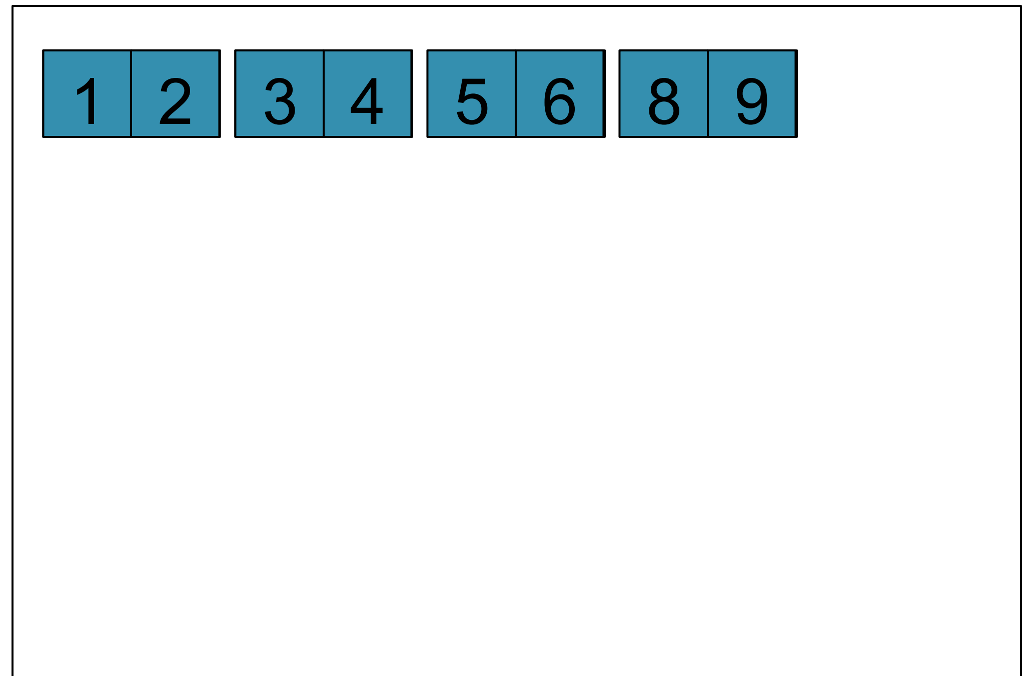
Sort-merge join: $R \bowtie S$

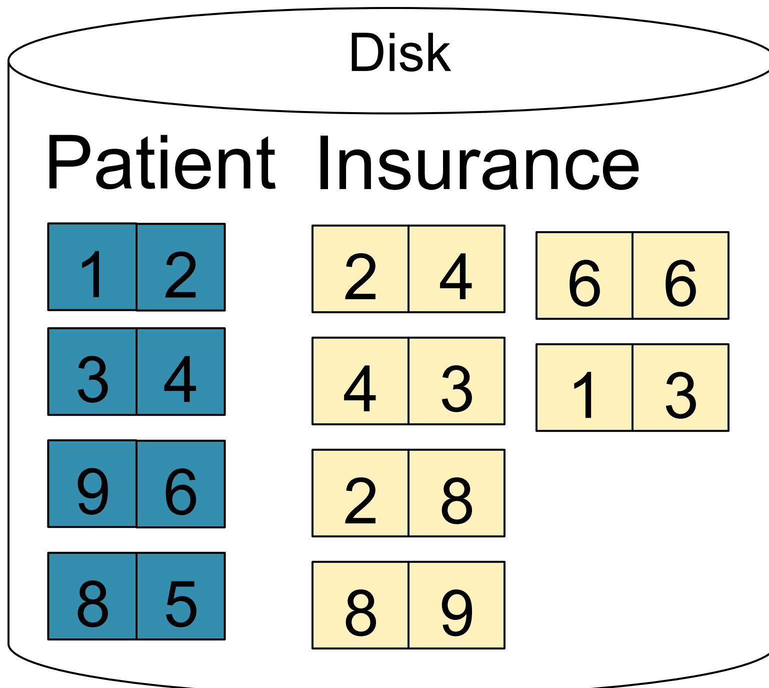
- Scan R and sort in main memory
- Scan S and sort in main memory
- Merge R and S

- 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

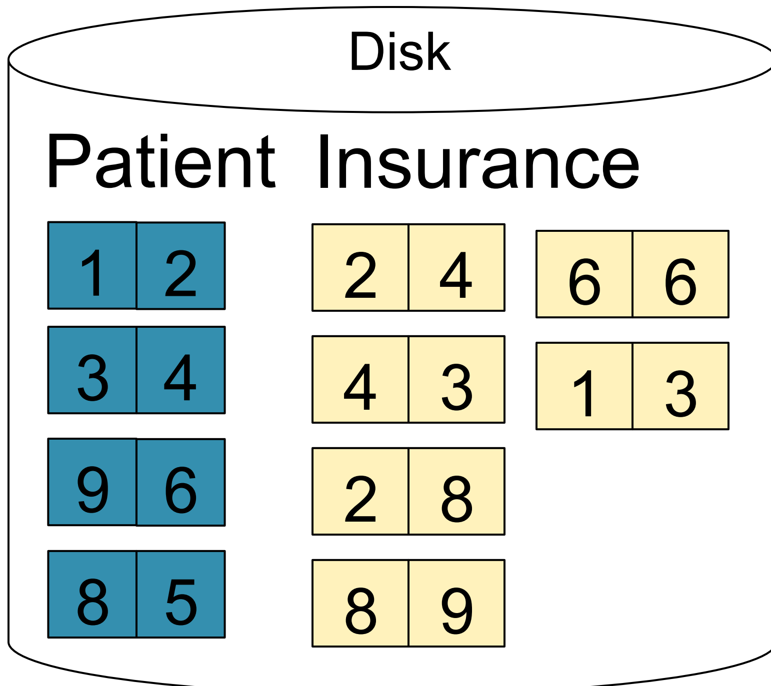
Step 1: Scan Patient and sort in memory

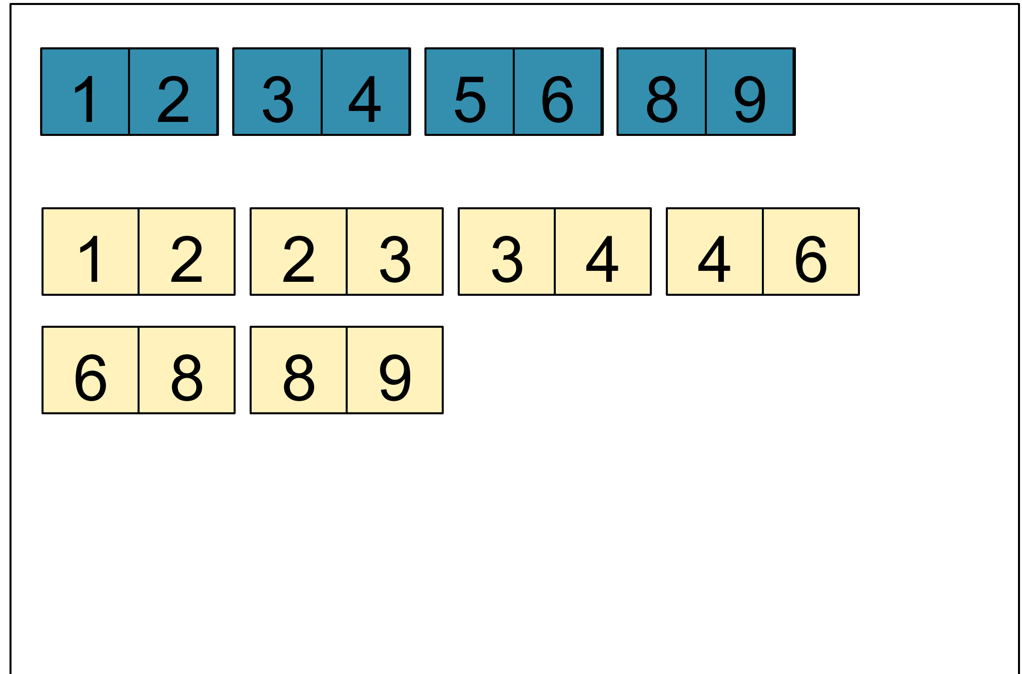




Sort-Merge Join Example

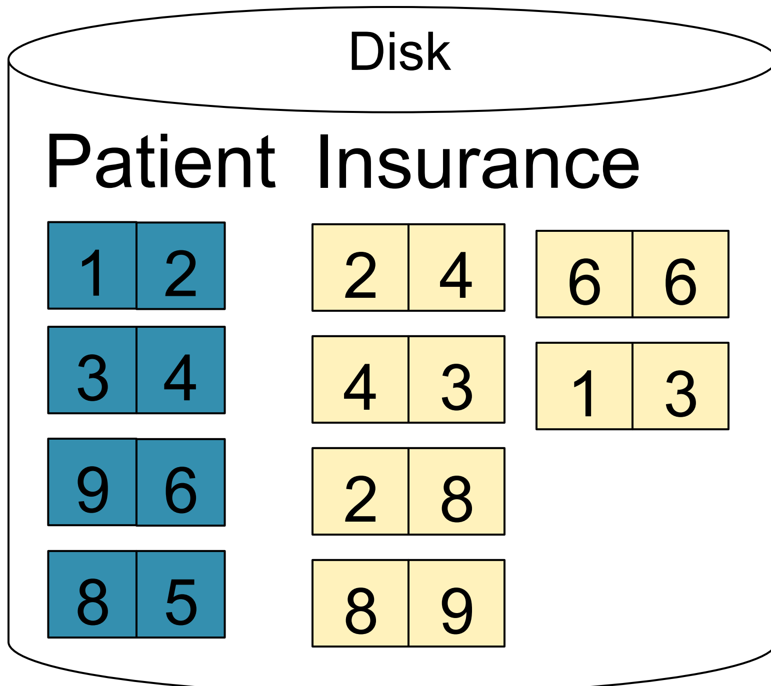
Step 2: Scan Insurance and sort in memory

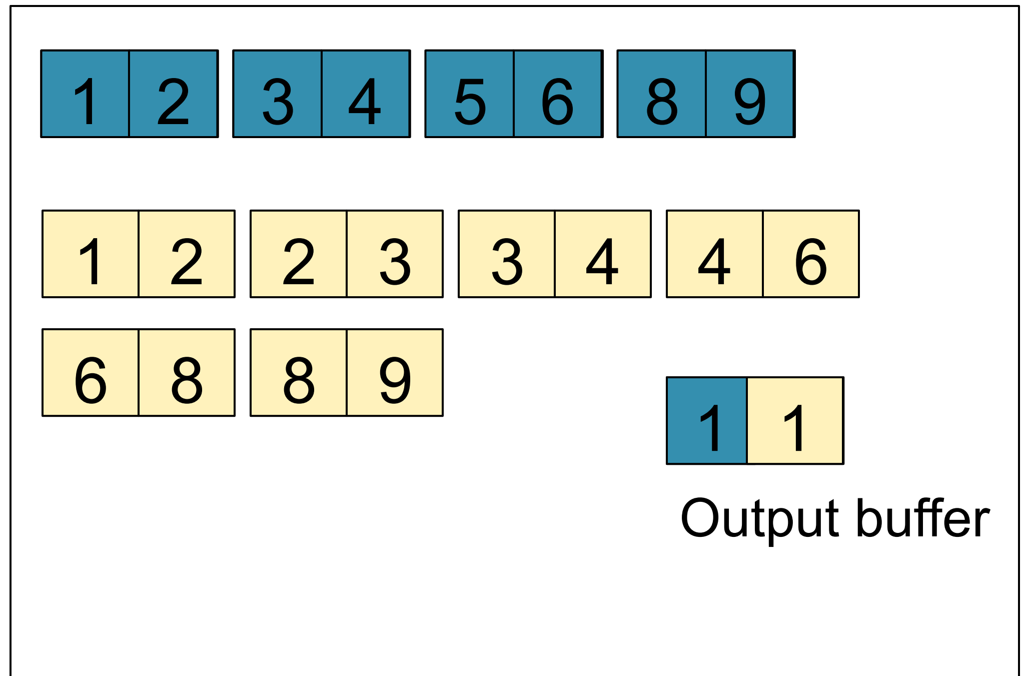




Sort-Merge Join Example

Step 3: Merge Patient and Insurance





Sort-Merge Join Example

Step 3: Merge Patient and Insurance