Lecture 18: Query execution, optimization

Friday, May 14, 2010

Big Picture

Query processor:

- Query execution
- Query optimization

Review (1/2)

- Each **operator implements this interface**
- **open()**
  - Initializes operator state
  - Sets parameters such as selection condition
- **get_next()**
  - Operator invokes get_next() recursively on its inputs
  - Performs processing and produces an output tuple
- **close()**
  - Cleans-up state

Review (2/2)

- Three algorithms for main memory join:
  - Nested loop join
  - Hash join
  - Merge join

If $|R| = m$ and $|S| = n$, what is the asymptotic complexity for computing $R \bowtie S$?

Other Main Memory Algorithms

- Grouping: γ(R)
  - Nested loop
  - Hash table
  - Sorting
- Duplicate elimination
  - *Exactly* the same algorithms (why?)

How do these algorithms work, and what are their complexities ?

External Memory Algorithms

- Data is too large to fit in main memory
- Issue: disk access is 3-4 orders of magnitude slower than memory access
- Assumption: runtime dominated by # of disk I/O's; will ignore the main memory part of the runtime

Cost Parameters

The *cost* of an operation = total number of I/Os

result assumed to be delivered in main memory

Cost parameters:

- B(R) = number of blocks for relation R
- T(R) = number of tuples in relation R
- V(R, a) = number of distinct values of attribute a
- M = size of main memory buffer pool, in blocks

Facts: (1) B(R) << T(R):
(2) When a is a key, V(R,a) = T(R)
When a is not a key, V(R,a) << T(R)

Ad-hoc Convention

- We assume that the operator *reads* the data from disk
- We assume that the operator *does not write* the data back to disk (e.g.: pipelining)
- Thus:

Main memory join algorithms for R $\bowtie$ S: Cost = B(R)+B(S)

Main memory grouping γ(R): Cost = B(R)

Sequential Scan of a Table R

- When R is *clustered*
  - Blocks consists only of records from this table
  - $B(R) << T(R)$
  - Cost = $B(R)$

- When R is *unclustered*
  - Its records are placed on blocks with other tables
  - $B(R) \approx T(R)$
  - Cost = $T(R)$

Nested Loop Joins

- Tuple-based nested loop $R \bowtie S$

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

R=outer relation
S=inner relation

- Cost: T(R) B(S) when S is clustered
- Cost: T(R) T(S) when S is unclustered

Examples

M = 4;   R, S are clustered

- Example 1:
  - B(R) = 1000, T(R) = 10000
  - B(S) = 2, T(S) = 20
  - Cost = ?

Can you do better ?

- Example 2:
  - B(R) = 1000, T(R) = 10000
  - B(S) = 4, T(S) = 40
  - Cost = ?

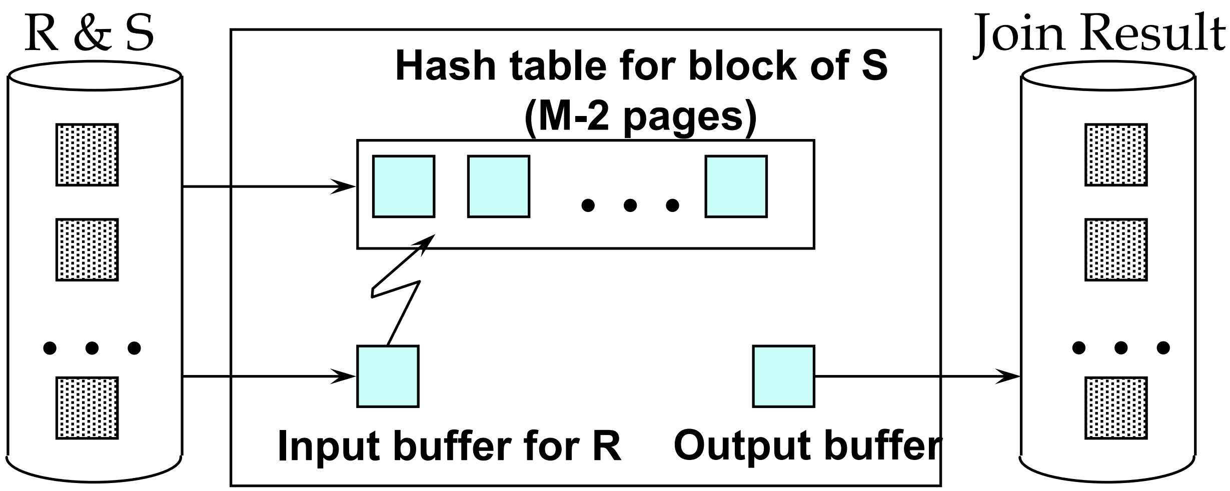
Block-Based Nested-loop Join

Why not M ?

```
for each (M-2) blocks bs of S do
   for each block br of R do
      for each tuple s in bs
         for each tuple r in br do
            if "r and s join" then output(r,s)
```

Terminology alert: book calls S the *inner* relation

Block Nested-loop Join



Examples

M = 4; R, S are clustered

- Example 1:
  - B(R) = 1000, T(R) = 10000
  - B(S) = 2, T(S) = 20
  - Cost = B(S) + B(R) = 1002

- Example 2:
  - B(R) = 1000, T(R) = 10000
  - B(S) = 4, T(S) = 40
  - Cost = B(S) + 2B(R) = 2004

Note: T(R) and T(S) are irrelevant here.

Cost of Block Nested-loop Join

- Read S once: cost B(S)
- Outer loop runs B(S)/(M-2) times, and each time need to read R: costs B(S)B(R)/(M-2)

$$\boxed{\text{Cost} = B(S) + B(S)B(R)/(M-2)}$$

Index Based Selection

Recall IMDB; assume indexes on Movie.id, Movie.year

```
SELET *
FROM Movie
WHERE id = ‘12345’
```

```
SELET *
FROM Movie
WHERE year = ‘1995’
```

B(Movie) = 10k
T(Movie) = 1M

What is your estimate of the I/O cost ?

Index Based Selection

Selection on equality: $\sigma_{a=v}(R)$

- Clustered index on a: cost B(R)/V(R,a)
- Unclustered index : cost T(R)/V(R,a)

Index Based Selection

- Example:

$B(R) = 10k$
$T(R) = 1M$
$V(R, a) = 100$

cost of $\sigma_{a=v}(R) = ?$

- Table scan (assuming R is clustered):
  - B(R) = 10k I/Os
- Index based selection:
  - If index is clustered: B(R)/V(R,a) = 100 I/Os
  - If index is unclustered: T(R)/V(R,a) = 10000 I/Os

Rule of thumb:
don't build unclustered indexes when V(R,a) is small !

Index Based Join

- R $\bowtie$ S
- Assume S has an index on the join attribute

```
for each tuple r in R do
    lookup the tuple(s) s in S using the index
    output (r,s)
```

Index Based Join

Cost (Assuming R is clustered):

- If index is clustered: B(R) + T(R)B(S)/V(S,a)
- If unclustered: B(R) + T(R)T(S)/V(S,a)

Operations on Very Large Tables

- Compute R $\bowtie$ S when each is larger than main memory
- Two methods:
  - Partitioned hash join (many variants)
  - Merge-join
- Similar for grouping

Partitioned Hash-based Algorithms

Idea:

- If $B(R) > M$, then partition it into smaller files:
  $R1, R2, R3, \ldots, Rk$

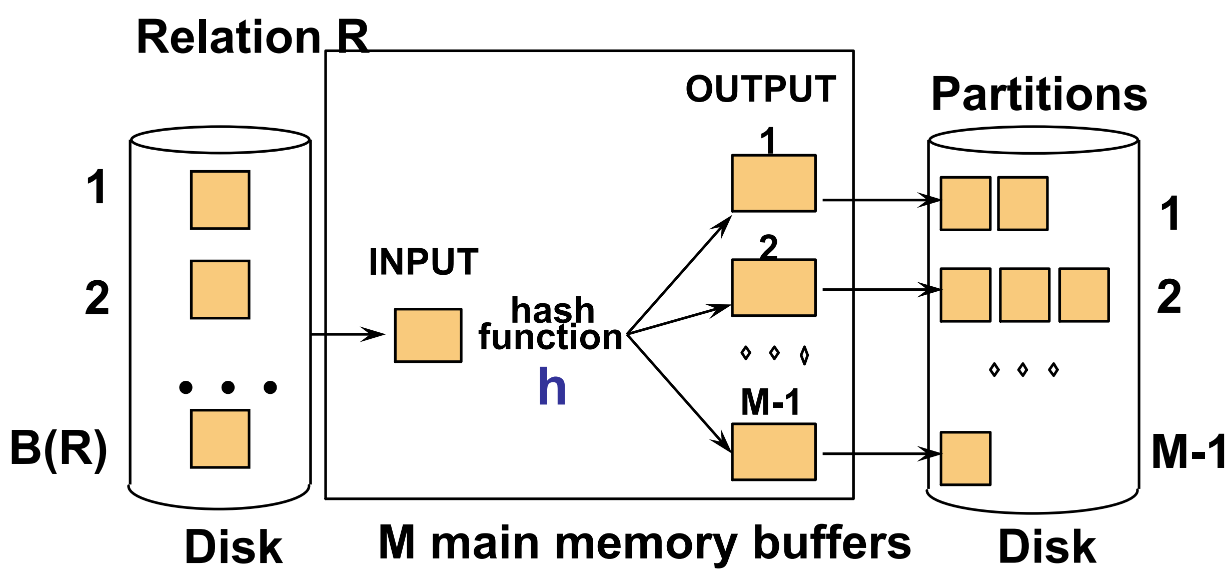
- Assuming $B(R1)=B(R2)=\ldots= B(Rk)$, we have
  $B(Ri) = B(R)/k$

- Goal: each Ri should fit in main memory:
  $B(Ri) \leq M$

How big can k be ?

Partitioned Hash Algorithms

- Idea: partition a relation R into M-1 buckets, on disk
- Each bucket has size approx. B(R)/(M-1) ≈ B(R)/M



Assumption: $B(R)/M \le M$, i.e. $B(R) \le M^2$

Grouping

- $\gamma(R)$ = grouping and aggregation
- Step 1. Partition R into buckets
- Step 2. Apply γ to each bucket (may read in main memory)

- Cost: 3B(R)
- Assumption: $B(R) <= M^2$

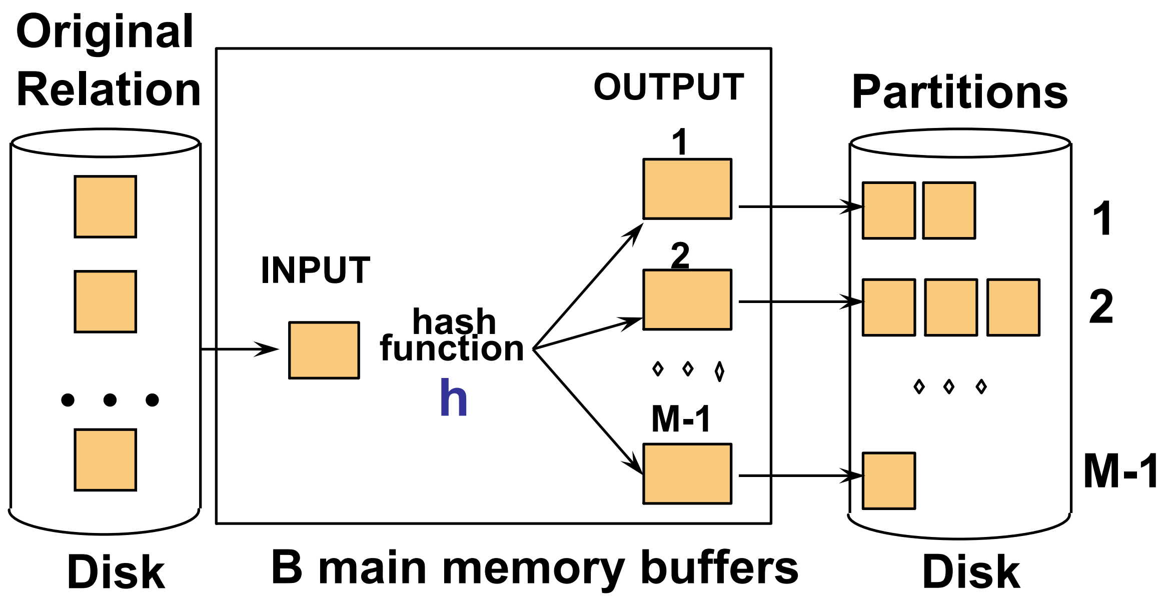
Partitioned Hash Join

R ⋈ S

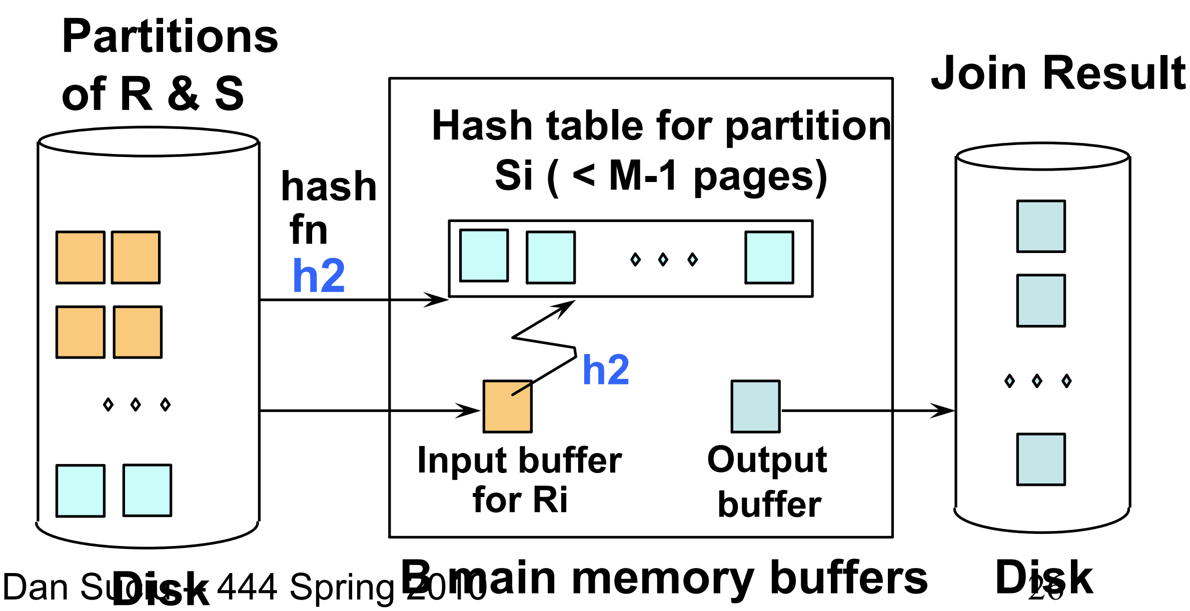
- Step 1:
  - Hash S into M buckets
  - send all buckets to disk
- Step 2
  - Hash R into M buckets
  - Send all buckets to disk
- Step 3
  - Join every pair of buckets

Hash-Join

- Partition both relations using hash fn **h**: R tuples in partition i will only match S tuples in partition i.



- Read in a partition of R, hash it using h2 (<> h!). Scan matching partition of S, search for matches.



Partitioned Hash Join

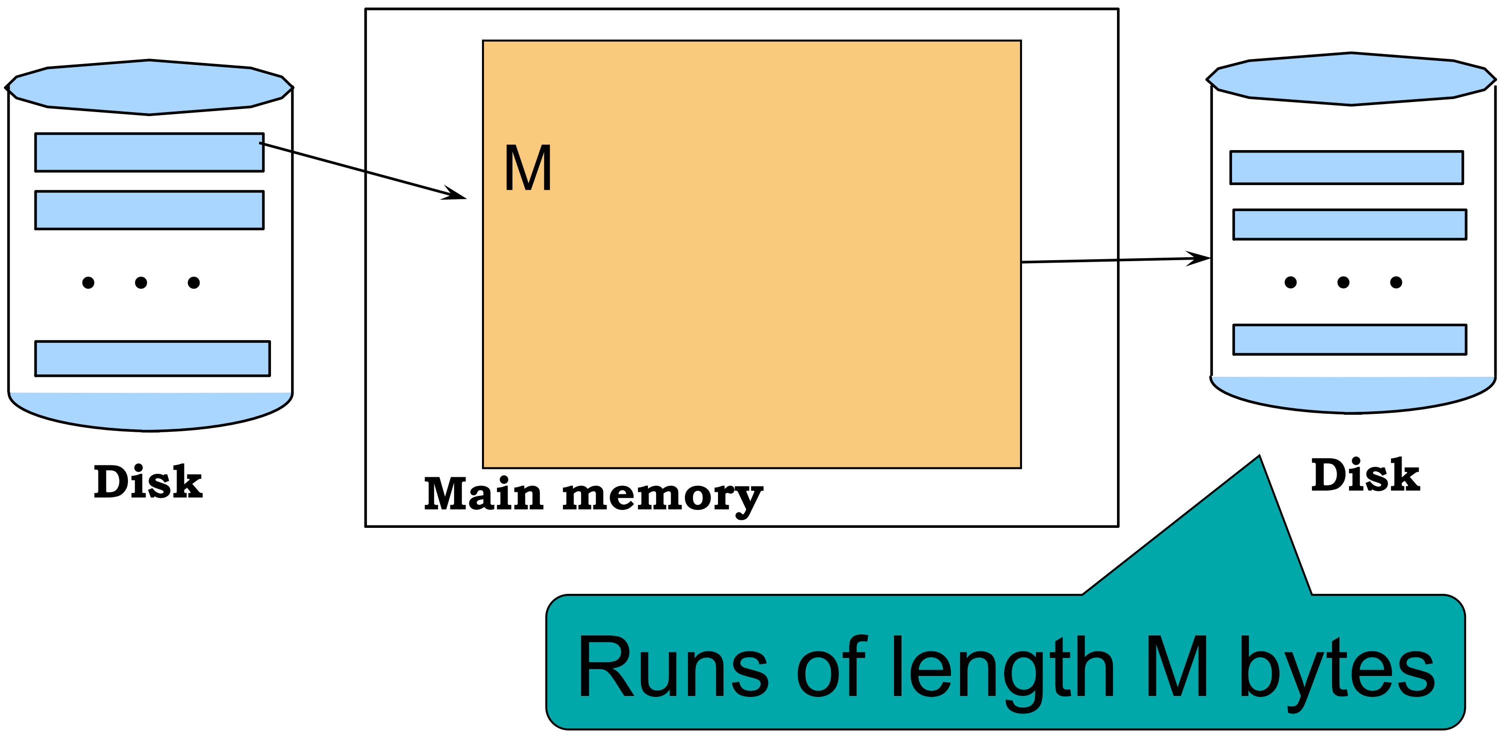
- Cost: 3B(R) + 3B(S)
- Assumption: min(B(R), B(S)) <= M^2

External Sorting

- Problem:
- Sort a file of size B with memory M
- Where we need this:
  - ORDER BY in SQL queries
  - Several physical operators
  - Bulk loading of B+-tree indexes.
- Will discuss only 2-pass sorting, when $B < M^2$

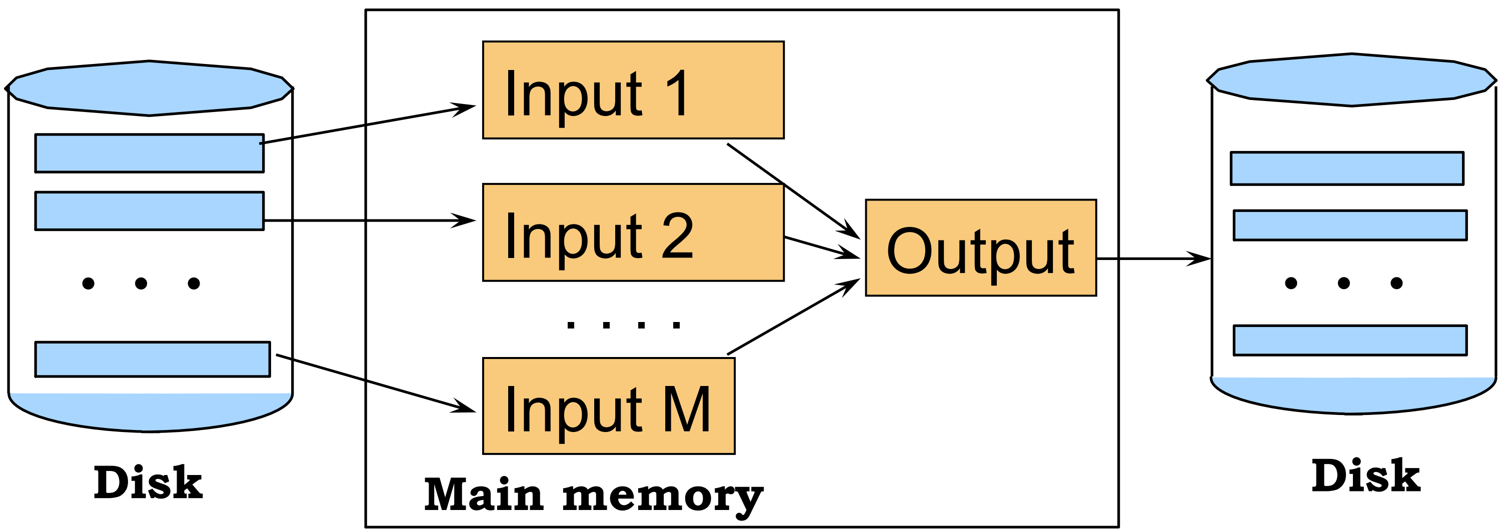
External Merge-Sort: Step 1

- Phase one: load M bytes in memory, sort



External Merge-Sort: Step 2

- Merge M – 1 runs into a new run
- Result: runs of length M (M – 1)≈ M^2



If B <= M^2 then we are done

Cost of External Merge Sort

- Read+write+read = 3B(R)
- Assumption: B(R) <= M^2

Grouping

Grouping: $\gamma_{a,\ sum(b)}$ (R)

- Idea: do a two step merge sort, but change one of the steps

- Question in class: which step needs to be changed and how ?

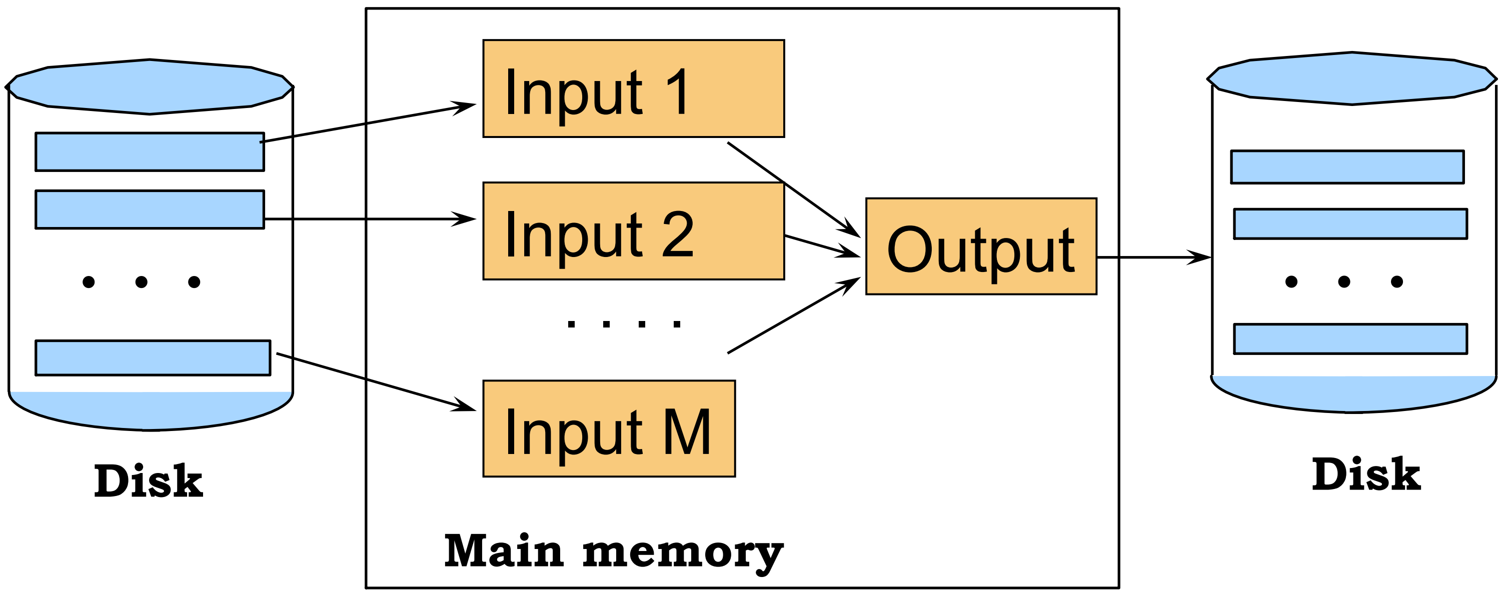
Cost = 3B(R)
Assumption: $B(\delta(R)) <= M^2$

Merge-Join

Join R $\bowtie$ S

- Step 1a: initial runs for R
- Step 1b: initial runs for S
- Step 2: merge and join

Merge-Join



$M_1 = B(R)/M$ runs for R

$M_2 = B(S)/M$ runs for S

Merge-join $M_1 + M_2$ runs;

need $M_1 + M_2 <= M$

Two-Pass Algorithms Based on Sorting

Join R ⋈ S

- If the number of tuples in R matching those in S is small (or vice versa) we can compute the join during the merge phase
- Total cost: $3B(R)+3B(S)$
- Assumption: $B(R) + B(S) <= M^2$

Summary of External Join Algorithms

- Block Nested Loop: B(S) + B(R)*B(S)/M
- Index Join: B(R) + T(R)B(S)/V(S,a)
- Partitioned Hash: 3B(R)+3B(S);
  - min(B(R),B(S)) <= M^2
- Merge Join: 3B(R)+3B(S)
  - B(R)+B(S) <= M^2