Steps in Query Processing

1. Translation
   - parsing, syntax checking
   - check existence of referenced relations and attributes
   - disambiguation of overloaded operators
   - check user authorization
   - replace views by their definitions
   - query rewrites
2. Optimization
   - consider alternative **plans** for processing the query
   - select an efficient plan
3. Processing
   - execute the plan
4. Data Delivery

Example

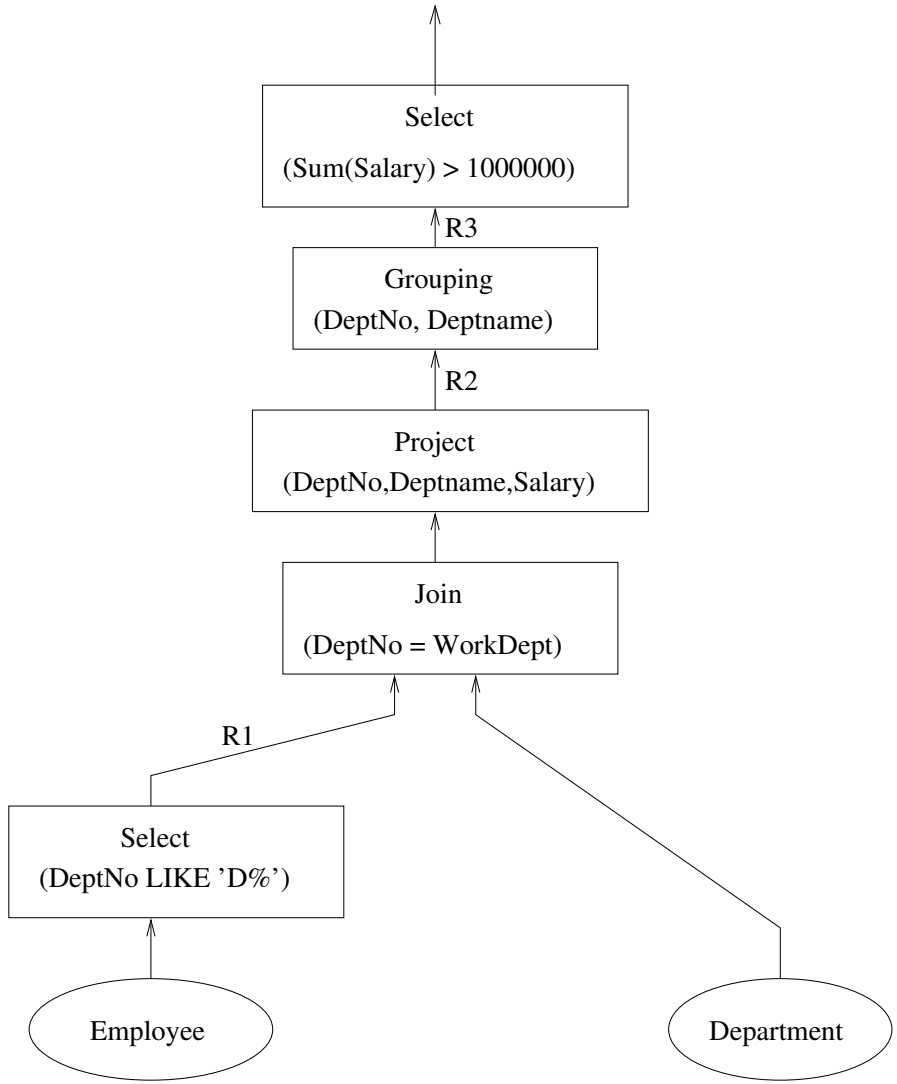
```
Department(DeptNo, Deptname)
Employee(EmpNo,Empname,Salary,WorkDept)
```

```
select DeptNo, Deptname, count(*), sum(Salary)
from Employee, Department
where WorkDept = DeptNo and DeptNo like 'D%'
group by DeptNo, Deptname
having sum(Salary) > 1000000
```

An Execution Plan

1. Scan the Employee table, select all tuples for which WorkDept starts with 'D', call the result R_1.
2. For each employee in R_1, find that employee's Department tuple. Eliminate attributes other than DeptNo, Deptname, and Salary. This is a join of R_1 with Department. Call the result R_2.
3. Group the tuples of R_2 by DeptNo. Call the result R_3.
4. For each group of tuples (employees) in R_3, count the number of employees in the group and calculate the sum of their salaries.
5. Eliminate groups for which the total employee salary is less than or equalto 1000000, and report the remaining groups.

Pictorial Access Plan



Pipelined Query Evaluation

- Many DBMS assume that the database (and even individual relations) may be substantially larger than primary memory.
- Therefore, the query evaluator will
  - use pipelining when possible, to avoid materializing intermediate results, and
  - when materialization is necessary, materialize in secondary storage
- At the same time, the query evaluator should be able to exploit primary memory during query evaluation
  - ideally, evaluation plans should be able to run faster when more memory is available

Implementing Plans: Iterators

- a plan is a tree of operators
- each operator is an iterator
- each call to GetNext returns a new tuple from the relation being computed by the iterator
- each iterator implements its interface using calls to the interface functions of its child (or children)

```
class iterator {
  void Open();
  tuple GetNext();
  void Close();
  (iterator *)input[];
  // additional
  // state here
}
```

Iterator-based plans

Iterator-based plans couple control flow and data flow, can be evaluated using a single worker thread, and implement pipelining naturally

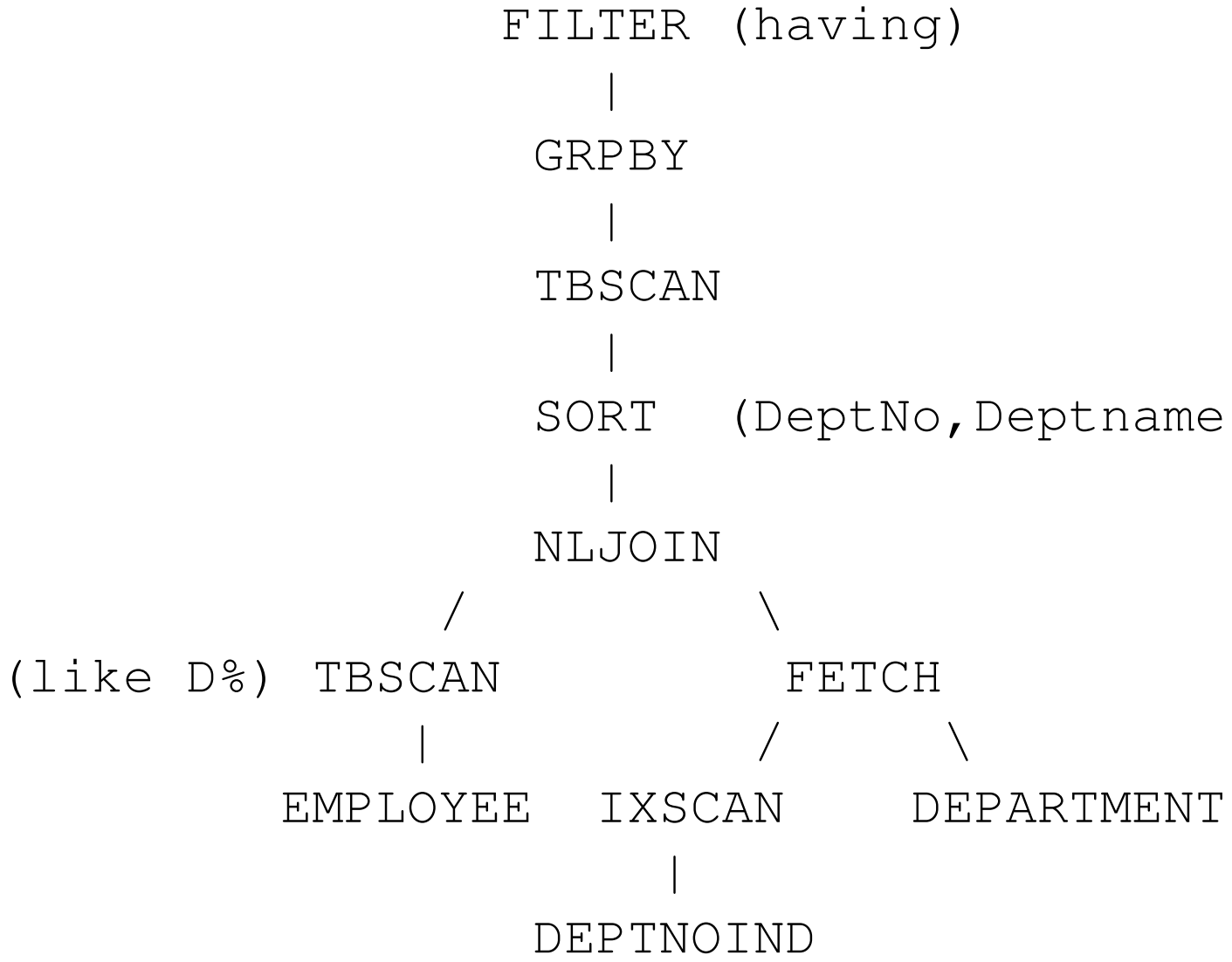
Evaluation of Iterator Plans

Query evaluator (plan interpreter) executes a pipelined plan by calling

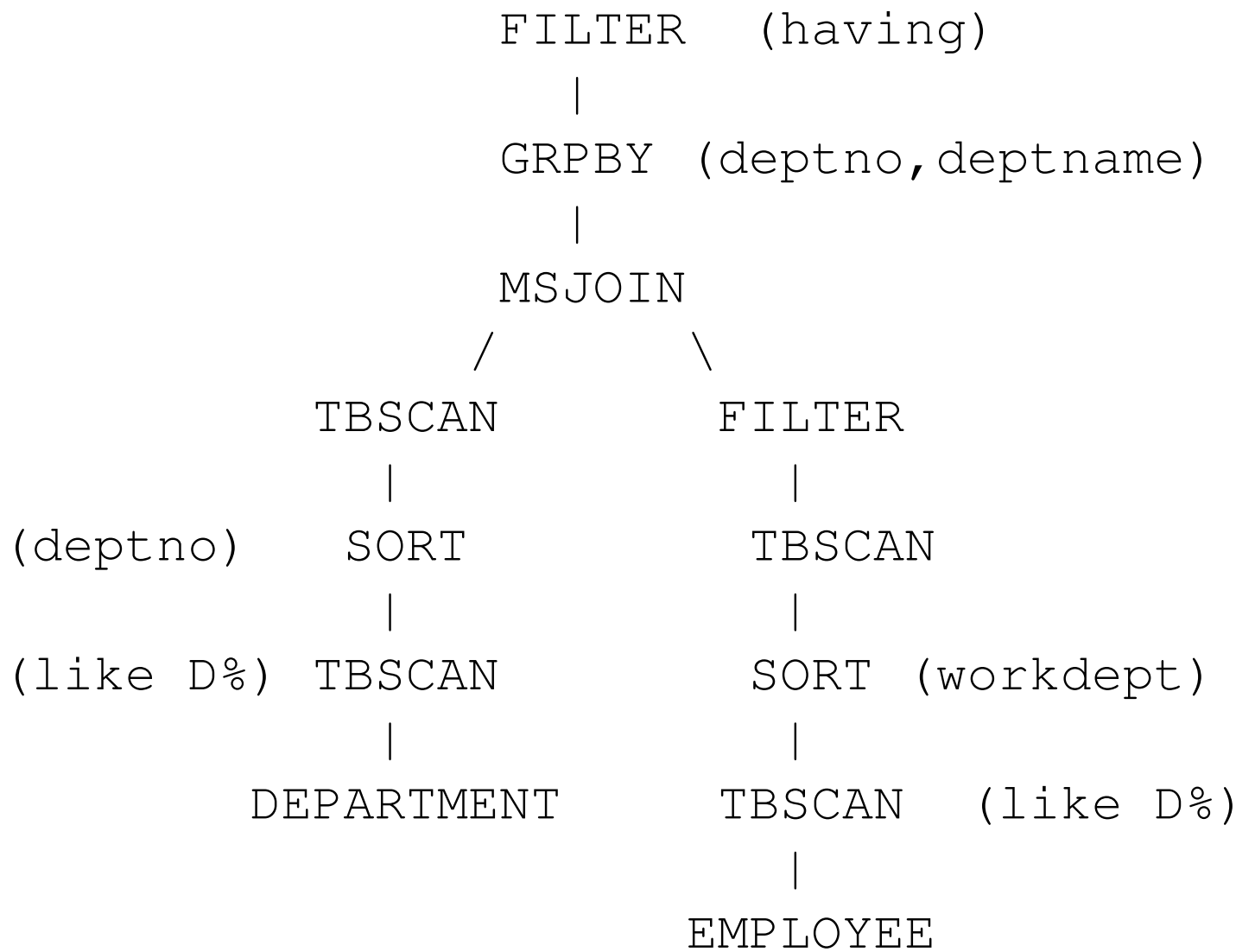
```
Open
GetNext
GetNext
GetNext
...
GetNext
Close
```

on the plan tree's top (root) operator.

DB2 Access Plan with Index



DB2 Access Plan



Some Basic Query Processing Operations

- Data Access Methods
  - Index scans
  - Table scans
- Projection and Selection
- Joins
  - nested loop join
  - hash join
  - sort-merge join
- Sorting
- Grouping and Duplicate Elimination
  - by sorting
  - by hashing

Selection

```
select DeptNo, Deptname, count(*), sum(Salary)
from Employee, Department
where WorkDept = DeptNo and DeptNo like 'D%'
group by DeptNo, Deptname
having sum(Salary) > 1000000
```

1. process selections on-the-fly
2. use a relevant index (e.g., index on `Department.DeptNo`)
3. use multiple indexes (if multiple predicates on a relation)

Using Indexes for Selection

select `*` **from** `R` **where** `R.a = 10` **and** `R.b > 5`

- suppose there is
  - a clustered index on `R.a`
  - an unclustered index on `R.b`
- Option 1: use index on `R.a`, filter `R.b` on-the-fly
- Option 2: use index on `R.b`, filter `R.a` on-the-fly
  - optimization
    1. find RIDs of matching records using `R.b` index
    2. sort the RIDs
    3. retrieve the matching records in RID order (why?)
    4. apply the `R.a` predicate on-the-fly
- Option 3: use both indexes (index intersection)

Index Intersection

select `*` **from** `R` **where** `R.a = 10` **and** `R.b > 5`

- using both indexes:
  1. use `R.a` index to retrieve RIDs of records satisfying `R.a` condition
  2. use `R.b` index to retrieve RIDs of records satisfying `R.b` condition
  3. intersect the two sets of RIDs (e.g., by sorting and merging) to find RIDs that appear in both
  4. fetch the records corresponding to RIDS in the intersection
- this is called index intersection or index ANDing
- can also do index union (or index ORing) for disjunctive selection conditions

Joining Relations

```
select DeptName, LastName
from Department, Employee
where DeptNo = WorkDept
```

The simple, tuple-at-a-time nested-loop join works like this:

```
foreach tuple d in Department do
   foreach tuple e in Employee do
      if d.DeptNo = e.WorkDept then
         output d,e
   end
end
```

Inner and Outer Relations

Department is the outer relation, Employee is the inner. The outer relation is read once, inner is potentially read many times.

Tuple NLJoin as an Iterator

```
// NLJoin extends
// iterator
class NLJoin {
  void Open();
  tuple GetNext();
  void Close();
  (iterator *)input[];
  tuple CurOuter;
}
```

```
Open() {
  input[outer]->Open();
}

Close() {
  input[inner]->Close();
  input[outer]->Close();
}
```

Tuple NLJoin as an Iterator (cont'd)

```
GetNext() {
while (true) {
  if (CurOuter == NULL) {
    CurOuter = input[outer]->GetNext();
    if (CurOuter == NULL) return NULL;
    input[inner]->Open();
  }
  while (CurInner=input[inner]->GetNext()!=NULL){
     if (CurInner joins with CurOuter) {
        return(CurInner join CurOuter);
     }
  }
  input[inner]->Close(); CurOuter=NULL;
}}
```

Block Nested Loop Join

Process outer relation a chunk at a time

```
foreach chunk C of Department
   foreach tuple e in Employee do
      foreach tuple d in C
         if d.DeptNo = e.WorkDept then
            output d,e
      end
   end
end
```

Exploiting Memory

C is determined by the amount of memory available to the join operator. More memory allows larger chunks, which means fewer scans of the inner relation.

Index Nested Loop Join

If there is an index on the join attribute of the inner relation (e.g., on Department.DeptNo), an index nested loop join can be used:

```
foreach tuple e in Employee do
   probe the index to find Department tuples d
      for which d.DeptNo = e.WorkDept
   for each such tuple d
      output d,e
end
```

Hash Join

To hash join `Employee` and `Department`:

- Use a hash function h_p applied to `WorkDept` to hash `Employee` tuples into P partitions, and write the partitions to secondary storage.
- Use the same hash function h_p applied to `DeptNo` to hash `Department` tuples into P partitions and write the partitions to secondary storage.
- For each partition $1 \leq i \leq P$, look for matches between tuples in the ith `Employee` partition and tuples in the ith `Department` partition. To do this:
  - Load ith partition of `Employee` into an in-memory hash table using hash function h_2 ($h_2 \neq h_p$)
  - Read ith partition of `Department` and probe the in-memory hash table using h_2 for matches from `Employee`

In this example, `Employee` is called the build relation and `Department` is called the probe relation.

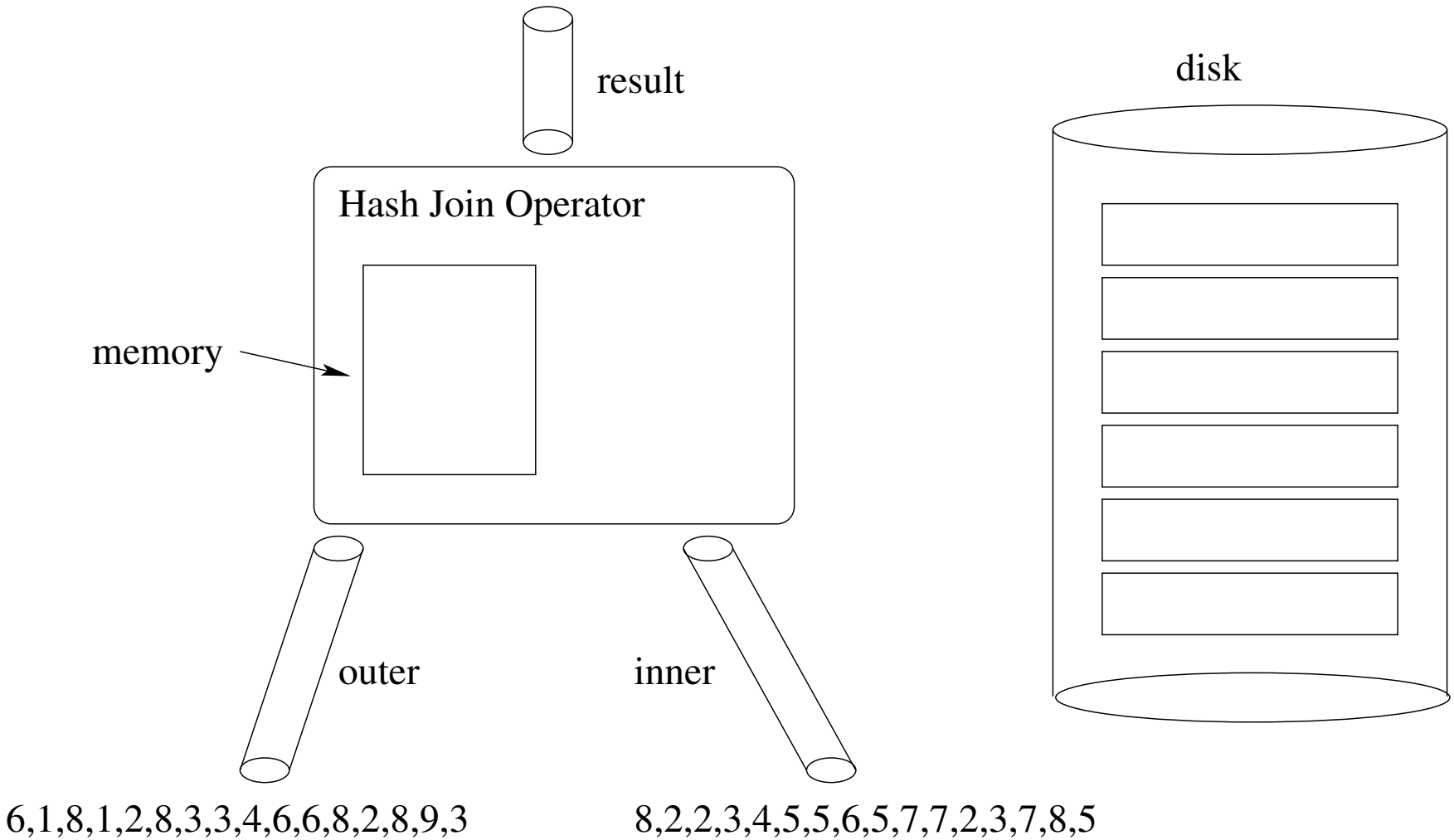
Hash Join Memory Consumption

- During partitioning, we need one block of memory for each partition, to stage that partition's tuples to secondary storage.
- During probing, we need to be able to fit each of the build relation's partitions in memory.
- If we have M blocks of memory available, we can
  - produce at most M partitions
  - have a maximum partition size of M (ignoring the space overhead of hashing)
- Assuming that h_p produces partitions of approximately the same size, this means that the size of the build relation can be at most M^2 blocks.
- Q: what to do if both relations are larger than M^2?

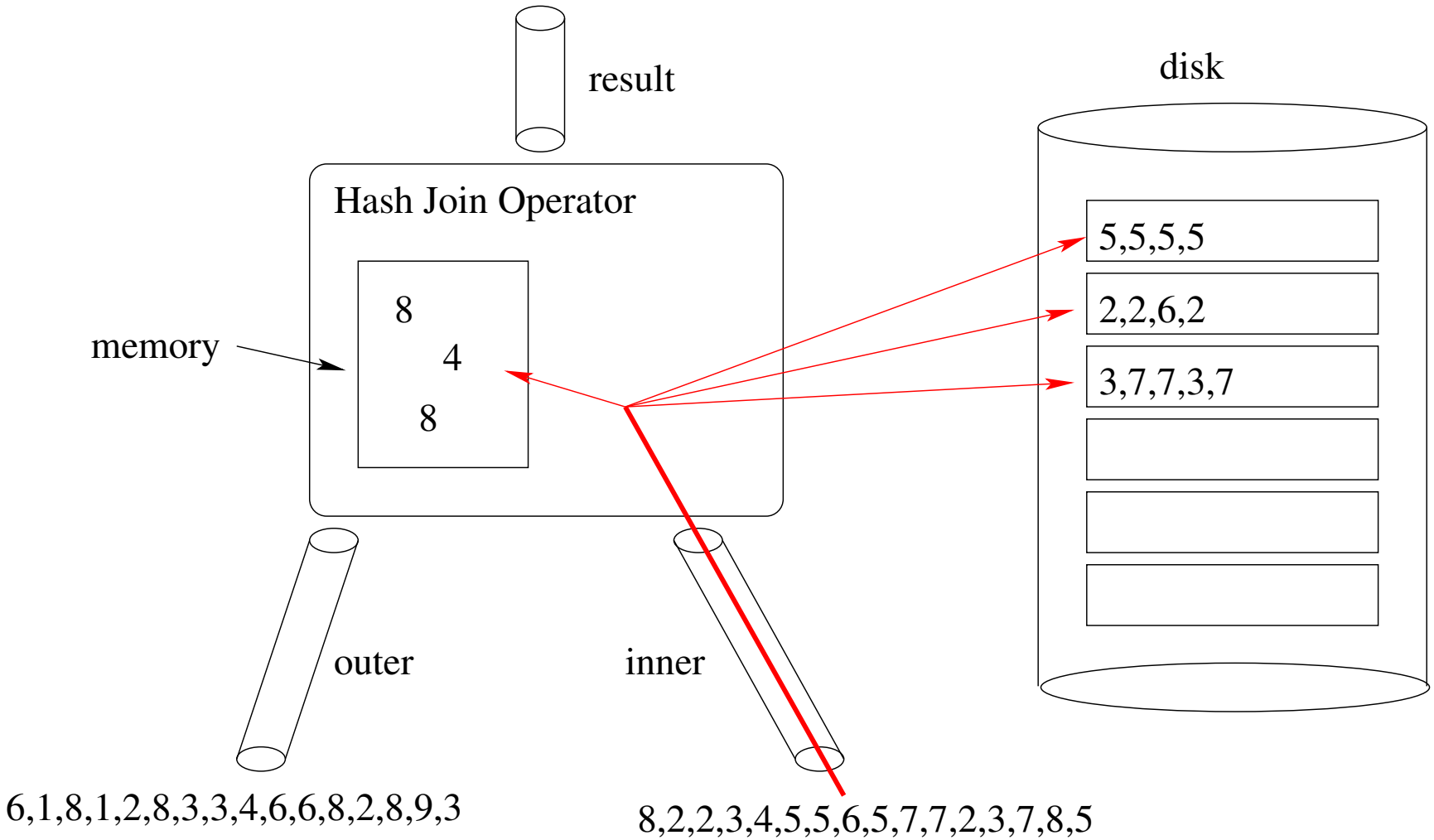
Hybrid Hash Join

- Hybrid Hash Join is a hash join variant that is useful when the hash operator has extra memory available.
- Basic Idea:
  - create the in-memory hash table for the first partition of the build relation during the partitioning phase (remaining partitions get written to secondary storage as before)
  - probe the in-memory hash table with the probe relation's first partition during the partitioning phase (remaining partitions get written to secondary storage as before)
  - after the partitioning phase, build and probe the remaining partitions as in the original hash join algorithm

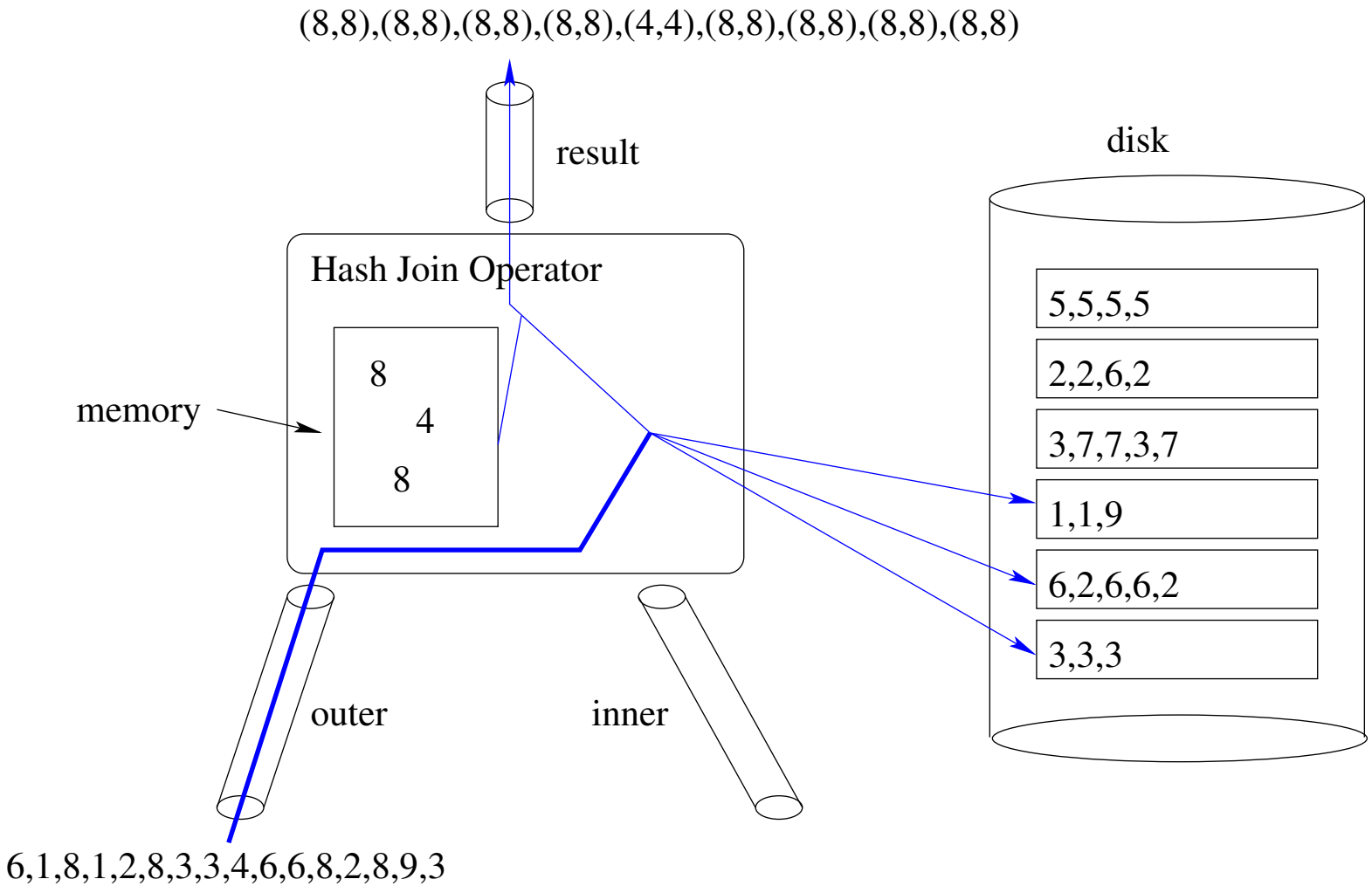
Hybrid Hash Join Example



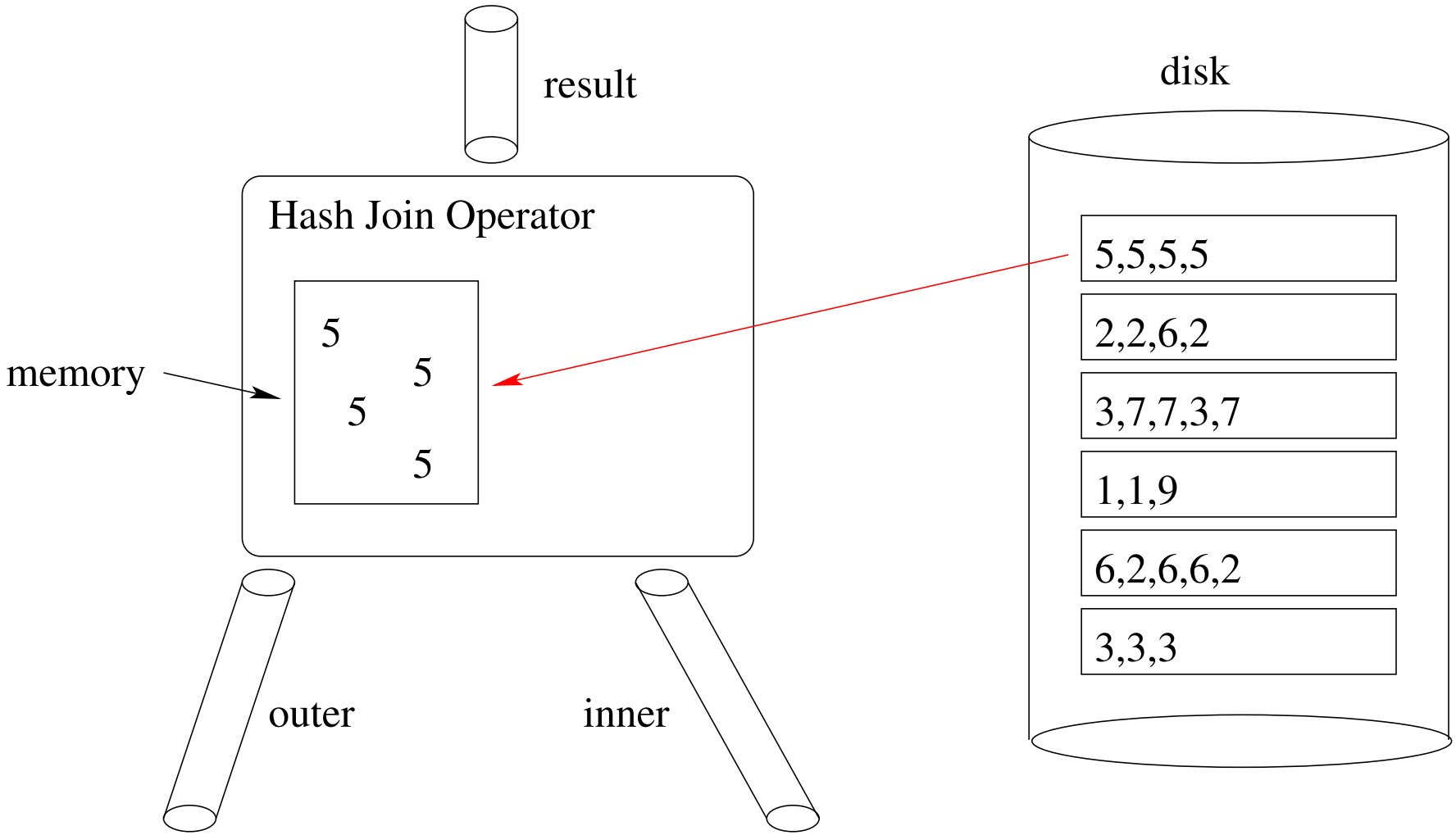
Hybrid Hash Join Example (cont'd)



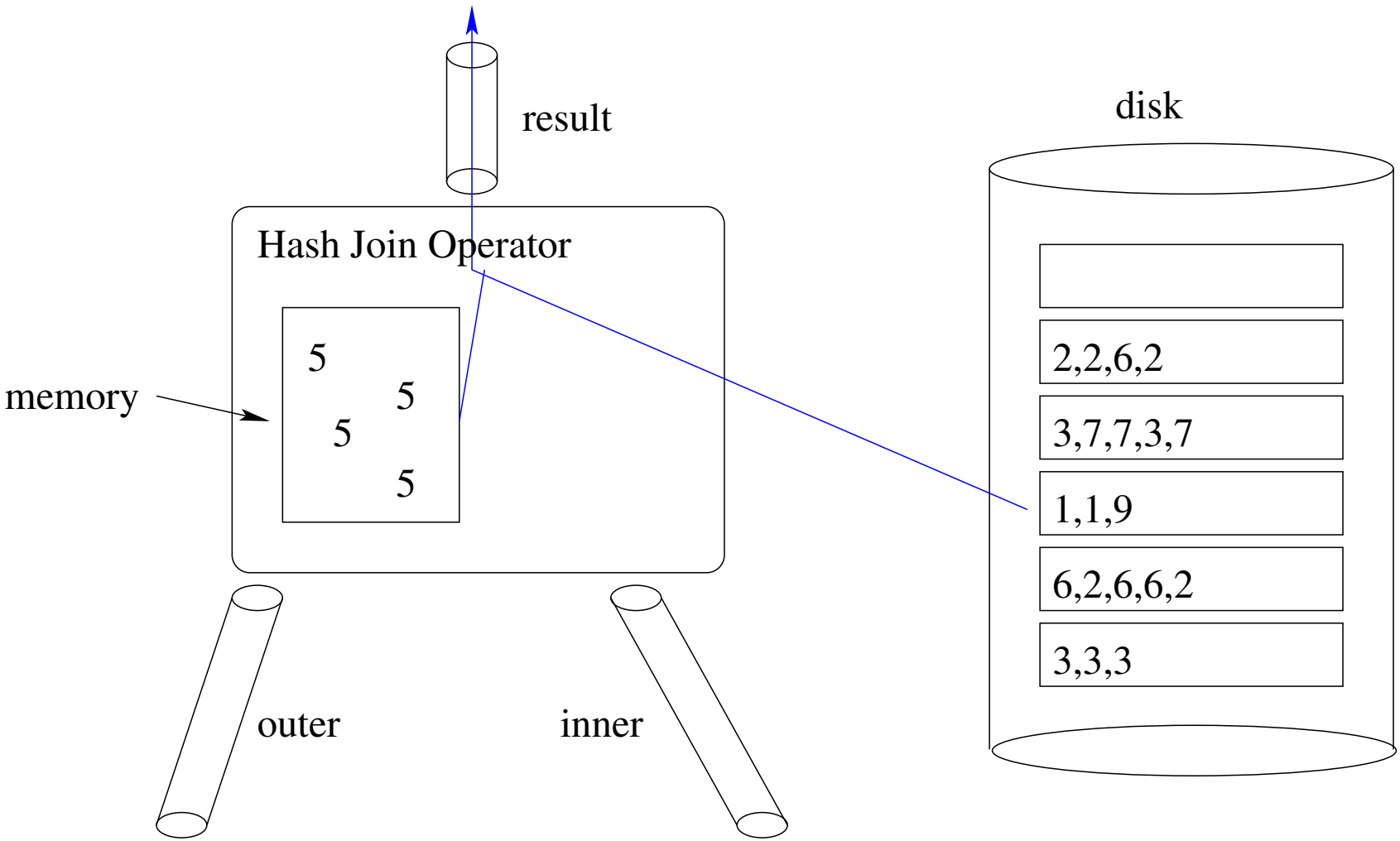
Hybrid Hash Join Example (cont'd)



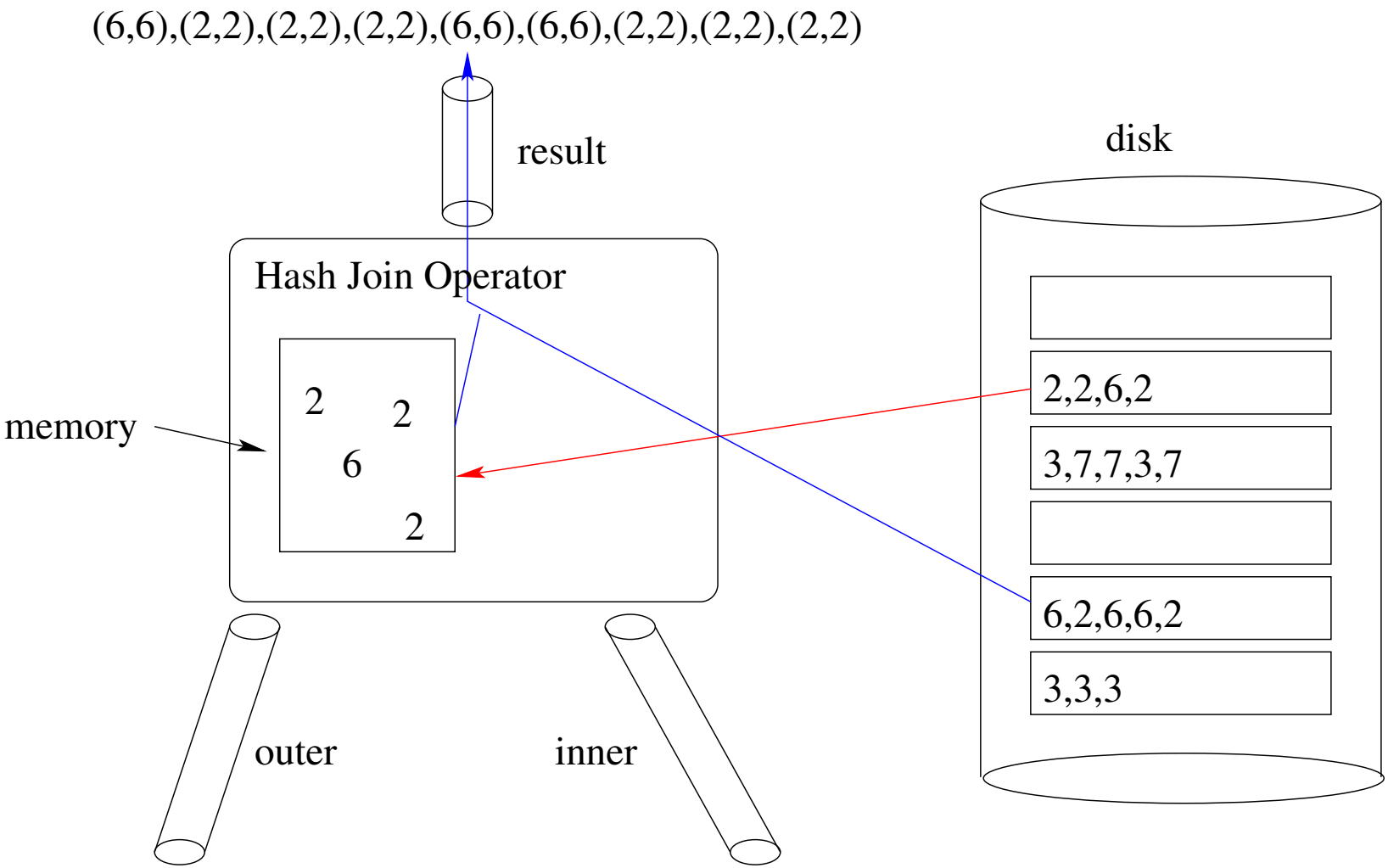
Hybrid Hash Join Example (cont'd)



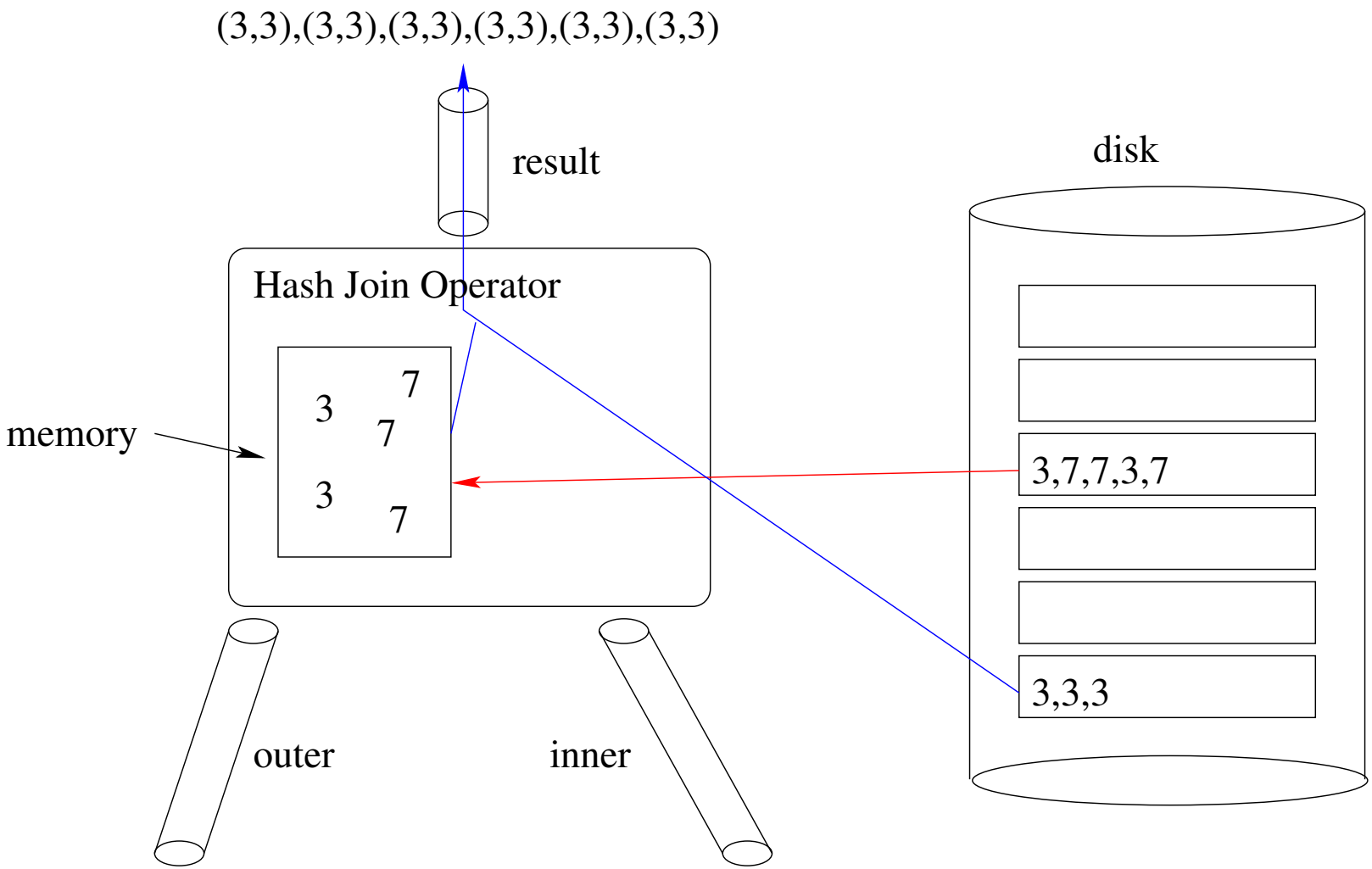
Hybrid Hash Join Example (cont'd)



Hybrid Hash Join Example (cont'd)

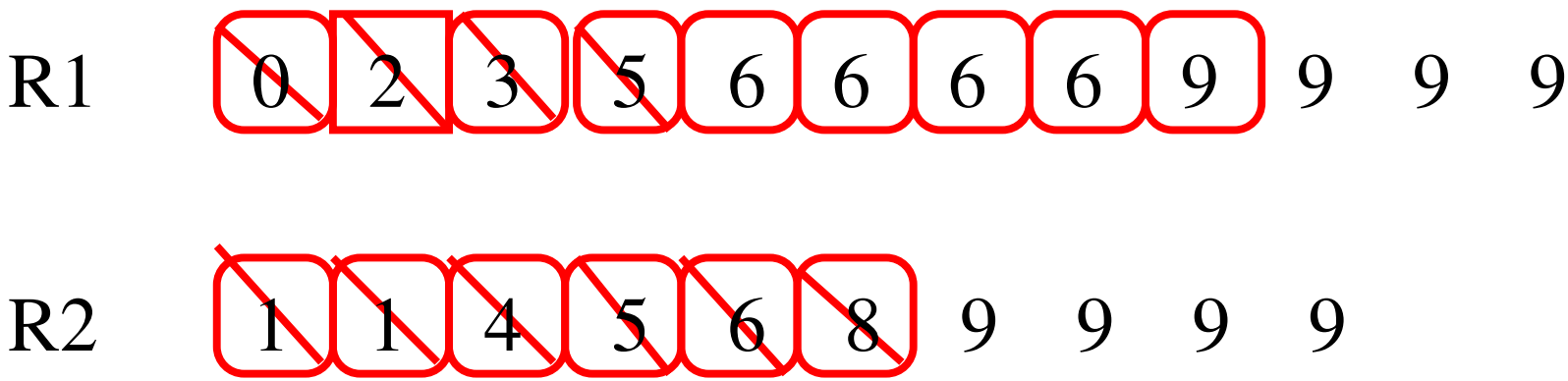


Hybrid Hash Join Example (cont'd)



Merge Join

- this join assumes each input is sorted on the join attribute
  - if input is not already sorted (e.g., because of retrieval through an index), must be sorted prior to the merge join
- sometimes called a sort-merge join
- an example:



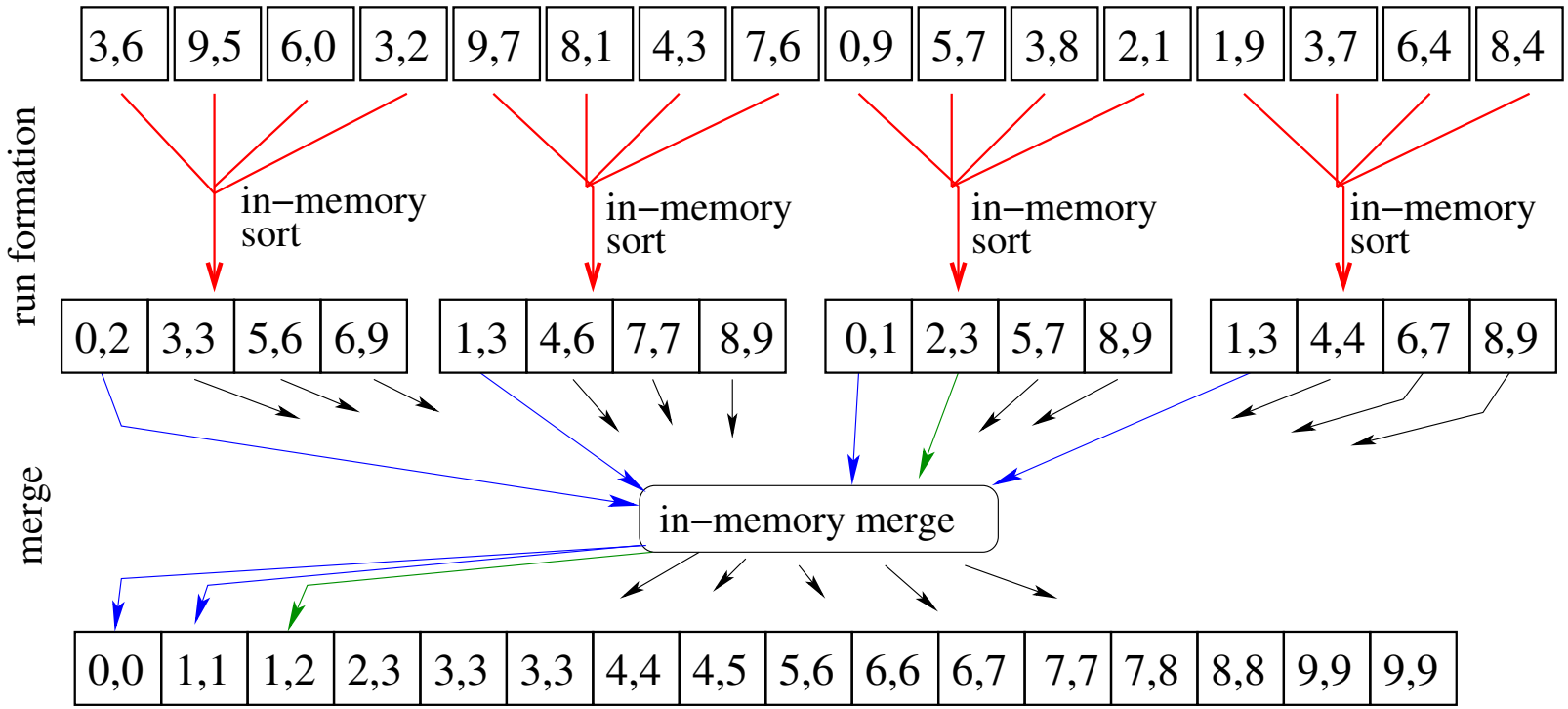
External Merge Sort

- objective is to sort a set of tuples according to the value of a sort key (values of one or more tuple attributes)
- not enough memory to hold all of the tuples
- general idea:

  run formation phase: load tuples in batches and sort each batch to produce a sorted run

  merge phase: merge sorted runs to produce longer runs, until there is a single run
- may require one or more merge phases depending on the amount of data and the amount of memory
- requires secondary storage to hold the runs

External Merge Sort Example



Grouping and Aggregation

- sort-based
  - grouping is easy if input is sorted on the grouping attributes
  - operator sees one group at a time
- hash-based
  - suitable if input is not sorted
  - if the number of groups is small enough, maintain running aggregates for all groups in an in-memory hash table
  - otherwise:
    - partition the groups by hashing
    - maintain running aggregates for groups in one partition, while spilling other partitions to disk
    - once input is exhausted, process spilled partitions one a time.