Overview Plan Operators

- different operators for different data and query characteristics

	Selection	Projection	Grouping	Join
Logical Plan Operators	$\sigma_p(R)$	$\pi_A(R)$	$\gamma_{G:agg(A)}(R)$	$R \bowtie_{R.a=S.b} S$
Physical Plan Operators	TableScan IndexScan **ALL**	**ALL**	SortGB HashGB	NestedLoopJN SortMergeJN HashJN
	Lecture 07		This Lecture	

-

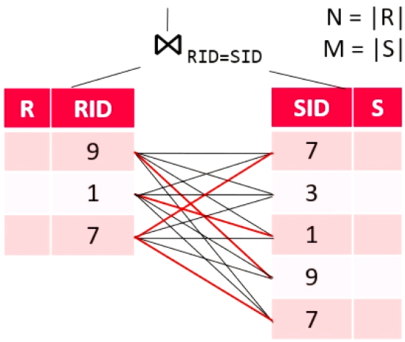
Nested Loop Join

- most general join operator
- no ordering/indexing
- slow

  - **Algorithm** (pseudo code)

```
for each s in S
  for each r in R
    if( r.RID θ s.SID )
      emit concat(r, s)
```

How to implement **next()**?



  - **Complexity**
    - Complexity: Time: **O(N * M)**, Space: **O(1)**
    - Pick smaller table as inner if it fits entirely in memory (buffer pool)
-

Block/Index Nested Loop Join

- **Block Nested Loop Join**
  - Avoid I/O by blocked data access
  - Read blocks of b_R and b_S R and S pages
  - Complexity unchanged but potentially much fewer scans

```
for each block b_R in R
  for each block b_S in S
    for each r in b_R
      for each s in b_S
        if( r.RID θ s.SID )
          emit concat(r, s)
```

- **Index Nested Loop Join**
  - Use index to locate qualifying tuples (==, >=, >, <=, <)
  - Complexity (for equivalence predicates): Time: **O(N * log M)**, Space: **O(1)**

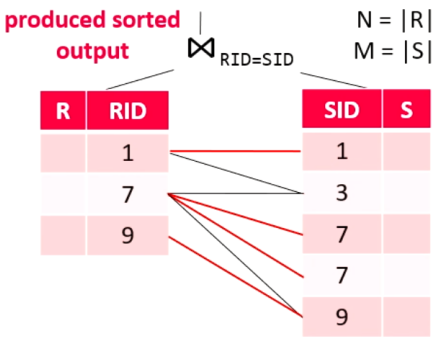
```
for each r in R
  for each s in S.IX(θ,r.RID)
    emit concat(r,s)
```

-

Sort Merge Join

- **Overview**
  - **Sort Phase:** sort the input tables R and S (w/ external sort algorithm)
  - **Merge Phase:** step-wise merge with lineage scan
- **Algorithm** (Merge, PK-FK)

```
Record next() {
  while( curR!=EOF && curS!=EOF ) {
    if( curR.RID < curS.SID )
      curR = R.next();
    else if( curR.RID > curS.SID )
      curS = S.next();
    else if( curR.RID == curS.SID ) {
      t = concat(curR, curS);
      curS = S.next(); //FK side
      return t;
  } }
  return EOF;
}
```



- **Complexity**
  - Time (unsorted vs sorted): **O(N log N + M log M)** vs **O(N + M)**
  - Space (unsorted vs sorted): **O(N + M)** vs **O(1)**

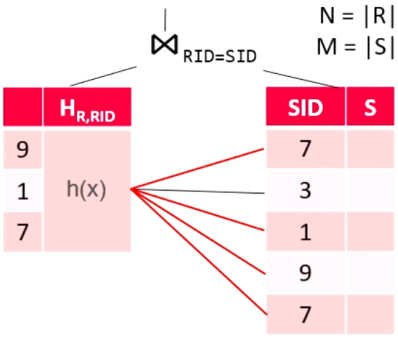
Hash Join

- **Overview**
  - **Build Phase:** read table S and build a hash table H_S over join key
  - **Probe Phase:** read table R and probe H_S with the join key
- **Algorithm** (Build+Probe, PK-FK)

```
Record next() {
  // build phase (first call only)
  while( (r = R.next()) != EOF )
    Hr.put(r.RID, r);

  // probe phase
  while( (s = S.next()) != EOF )
    if( Hr.containsKey(s.SID) )
      return concat(Hr.get(s.SID), s);

  return EOF;
}
```

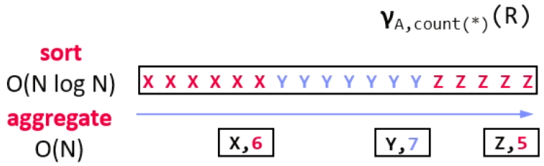


- **Complexity**
  - Time: **O(N + M)**, Space: **O(N)**
  - Classic hashing: p in-memory partitions of Hr w/ p scans of R and S

Group By Types

- **Recap: Classification of Aggregates (04 Relational Algebra)**
  - Additive, semi-additive, additively-computable, others

- **Sort Group-By**
  - Similar to sort-merge join (`Sort, GroupAggregate`)
  - Sorted group output



- **Hash Group-By**
  - Similar to hash join (`HashAggregate`)
  - Higher temporary memory consumption
  - Unsorted group output
  - **#1** w/ tuple grouping
  - **#2** w/ direct aggregation (e.g., count)
  - **Beware:** cache-unfriendly if many groups (size(H) > L2/L3 cache)