>>

Sort-merge Join

❖ Sort-Merge Join

Basic approach:

- sort both relations on join attribute (reminder: *Join [i=j] (R,S)*)
- scan together using merge to form result **`(r,s)`** tuples

Advantages:

- no need to deal with "entire" *S* relation for each *r* tuple
- deal with runs of matching *R* and *S* tuples

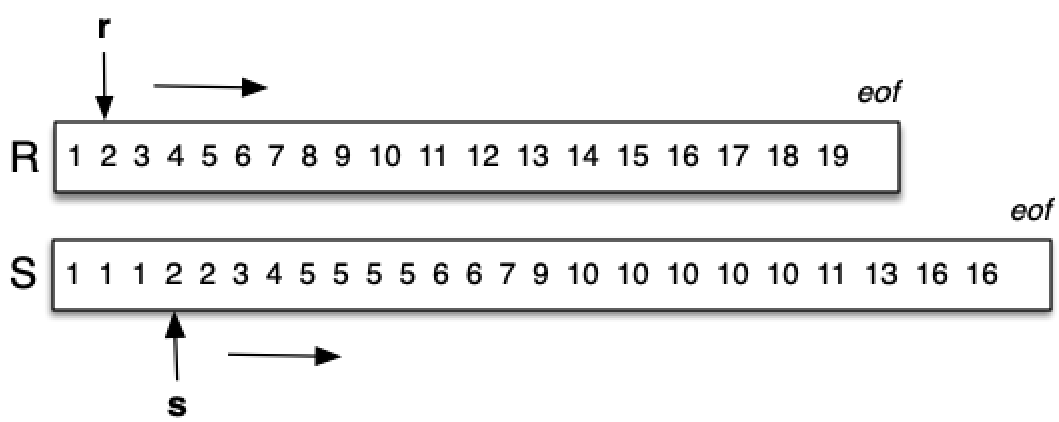
Disadvantages:

- cost of sorting both relations (already sorted on join key?)
- some rescanning required when long runs of *S* tuples

❖ Sort-Merge Join (cont)

Standard merging requires two cursors:

```
while (r != eof && s != eof) {
    if (r.val ≤ s.val) { output(r.val); next(r); }
    else { output(s.val); next(s); }
}
while (r != eof) { output(r.val); next(r); }
while (s != eof) { output(s.val); next(s); }
```

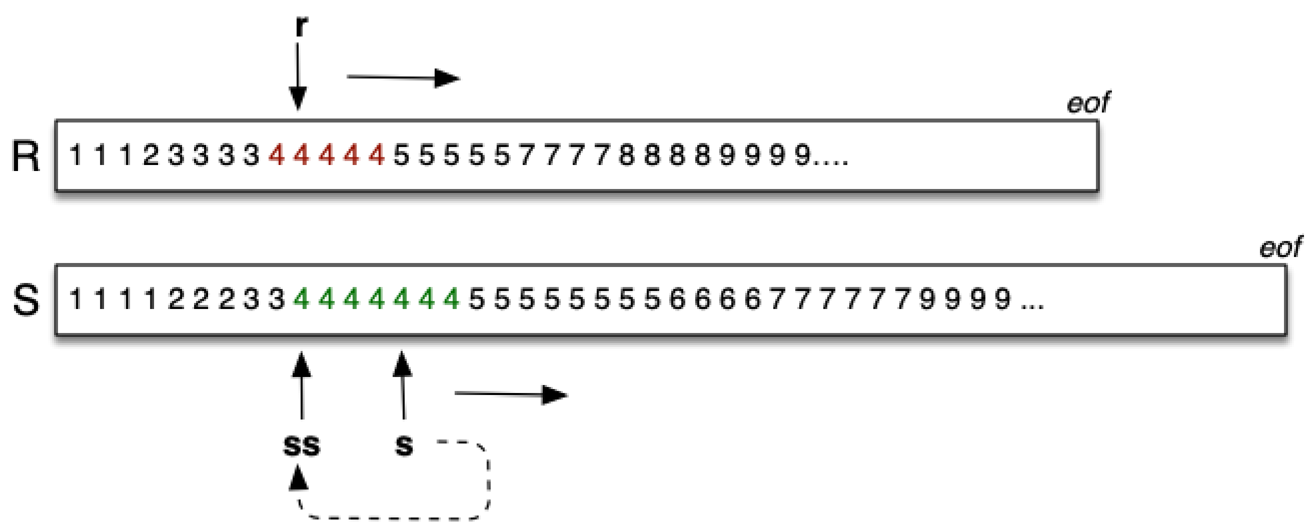


<< ^ >>

❖ Sort-Merge Join (cont)

Merging for join requires 3 cursors to scan sorted relations:

- **`r`** = current record in *R* relation
- **`s`** = current record in *S* relation
- **`ss`** = start of current run in *S* relation



<< ∧ >>

❖ Sort-Merge Join (cont)

Algorithm using query iterators/scanners:

```
Query ri, si;  Tuple r,s;

ri = startScan("SortedR");
si = startScan("SortedS");
while ((r = nextTuple(ri)) != NULL
       && (s = nextTuple(si)) != NULL) {
    // align cursors to start of next common run
    while (r != NULL && r.i < s.j)
           r = nextTuple(ri);
    if (r == NULL) break;
    while (s != NULL && r.i > s.j)
           s = nextTuple(si);
    if (s == NULL) break;
    // must have (r.i == s.j) here
...
```

<< ^ >>

❖ Sort-Merge Join (cont)

```
...
    // remember start of current run in S
    TupleID startRun = scanCurrent(si)
    // scan common run, generating result tuples
    while (r != NULL && r.i == s.j) {
        while (s != NULL and s.j == r.i) {
            addTuple(outbuf, combine(r,s));
            if (isFull(outbuf)) {
                writePage(outf, outp++, outbuf);
                clearBuf(outbuf);
            }
            s = nextTuple(si);
        }
        r = nextTuple(ri);
        setScan(si, startRun);
    }
}
```

COMP9315 21T1 ♢ Sort-merge Join ♢ [5/10]

<< ∧ >>

❖ Sort-Merge Join (cont)

Buffer requirements:

- for sort phase:
  - as many as possible (remembering that cost is $O(log_N)$)
  - if insufficient buffers, sorting cost can dominate
- for merge phase:
  - one output buffer for result
  - one input buffer for relation R
  - (preferably) enough buffers for longest run in S

❖ Sort-Merge Join (cont)

Cost of sort-merge join.

Step 1: sort each relation (if not already sorted):

- Cost = $2.b_R\,(1 + log_{N-1}(b_R\,/N)) + 2.b_S\,(1 + log_{N-1}(b_S\,/N))$
  (where N = number of memory buffers)

Step 2: merge sorted relations:

- if every run of values in S fits completely in buffers,
  merge requires single scan, Cost = $b_R + b_S$
- if some runs in of values in S are larger than buffers,
  need to re-scan run for each corresponding value from R

❖ Sort-Merge Join on Example

SQL query on student/enrolment database:

```
select E.subj, S.name
from   Student S join Enrolled E on (S.id = E.stude)
order  by E.subj
```

And its relational algebra equivalent:

Sort[subj] (Project[subj,name] (Join[id=stude](Student,Enrolled)))

Database: $r_S = 20000$, $c_S = 20$, $b_S = 1000$, $r_E = 80000$, $c_E = 40$, $b_E = 2000$

We are interested only in the cost of *Join*, with *N* buffers

<< ∧ >>

❖ Sort-Merge Join on Example (cont)

Case 1: $Join_{[id=stude]}(Student,Enrolled)$

- relations are not sorted on *id#*
- memory buffers $N=32$; all runs are of length < 30

$$
\begin{aligned}
\text{Cost} &= sort(S) + sort(E) + b_S + b_E \\
&= 2b_S(1+log_{31}(b_S/32)) + 2b_E(1+log_{31}(b_E/32)) + b_S + b_E \\
&= 2\times1000\times(1+2) + 2\times2000\times(1+2) + 1000 + 2000 \\
&= 6000 + 12000 + 1000 + 2000 \\
&= 21,000
\end{aligned}
$$

<< ^

❖ Sort-Merge Join on Example (cont)

Case 2: *Join[id=stude](Student,Enrolled)*

- *Student* and *Enrolled* already sorted on *id#*
- memory buffers *N=4* (*S* input, 2 × *E* input, output)
- 5% of the "runs" in *E* span two pages
- there are no "runs" in *S*, since *id#* is a primary key

For the above, no re-scans of *E* runs are ever needed

Cost = 2,000 + 1,000 = 3,000 (regardless of which relation is outer)

Produced: 31 Mar 2021