>>

Hash Join

^ >>

❖ Hash Join

Basic idea:

- use hashing as a technique to partition relations
- to avoid having to consider all pairs of tuples

Requires sufficent memory buffers

- to hold substantial portions of partitions
- (preferably) to hold largest partition of outer relation

Other issues:

- works only for equijoin **`R.i=S.j`** (but this is a common case)
- susceptible to data skew (or poor hash function)

Variations: *simple*, *grace*, *hybrid*.

❖ Simple Hash Join

Basic approach:

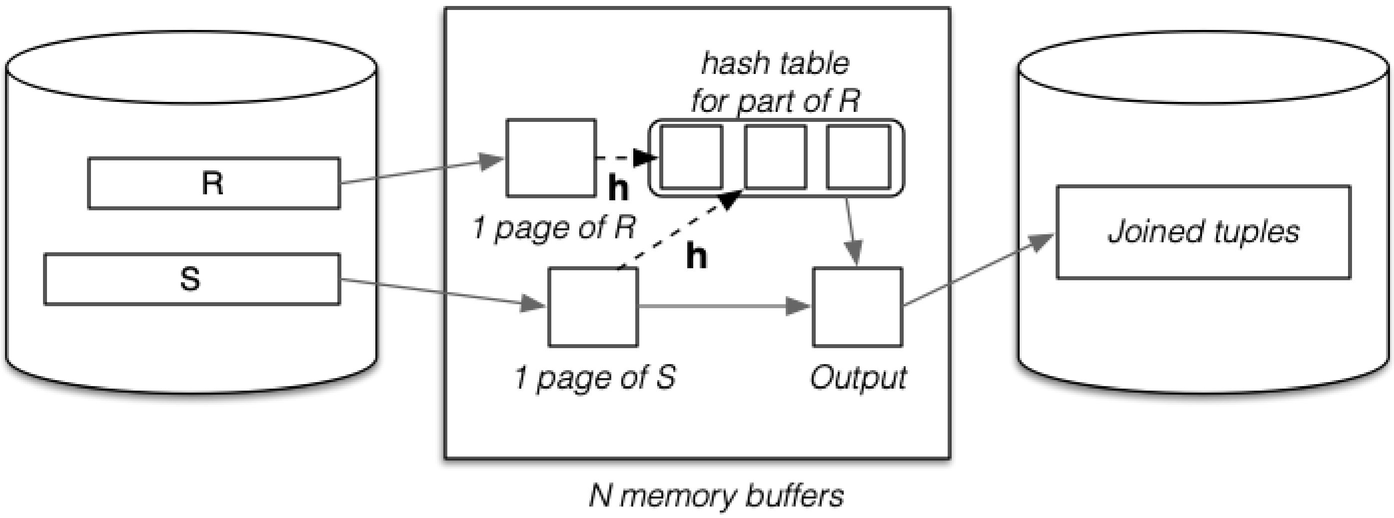
- hash part of outer relation *R* into memory buffers (build)
- scan inner relation *S*, using hash to search (probe)
    - if R.i=S.j, then h(R.i)=h(S.j) (hash to same buffer)
    - only need to check one memory buffer for each *S* tuple
- repeat until whole of *R* has been processed

No overflows allowed in in-memory hash table

- works best with uniform hash function
- can be adversely affected by data/hash skew

❖ Simple Hash Join (cont)

Data flow in hash join:



<< ∧ >>

❖ Simple Hash Join (cont)

Algorithm for simple hash join *Join[R.i=S.j](R,S)*:

```
for each tuple r in relation R {
   if (buffer[h(R.i)] is full) {
      for each tuple s in relation S {
         for each tuple rr in buffer[h(S.j)] {
            if ((rr,s) satisfies join condition) {
               add (rr,s) to result
      }  }  }
      clear all hash table buffers
   }
   insert r into buffer[h(R.i)]
}
```

Best case: # join tests ≤ $r_S.c_R$ (cf. nested-loop $r_S.r_R$)

❖ Simple Hash Join (cont)

Cost for simple hash join ...

Best case: all tuples of R fit in the hash table

- Cost = $b_R + b_S$
- Same page reads as block nested loop, but less join tests

Good case: refill hash table m times (where $m \geq ceil(b_R / (N\text{-}3))$)

- Cost = $b_R + m.b_S$
- More page reads than block nested loop, but less join tests

Worst case: everything hashes to same page

- Cost = $b_R + b_R.b_S$

❖ Grace Hash Join

Basic approach (for $R \bowtie S$):

- partition both relations on join attribute using hashing (*h1*)
- load each partition of R into N-3*buffer hash table (*h2*)
- scan through corresponding partition of S to form results
- repeat until all partitions exhausted

For best-case cost ($O(b_R + b_S)$):

- need $\geq \sqrt{b_R}$ buffers to hold largest partition of outer relation

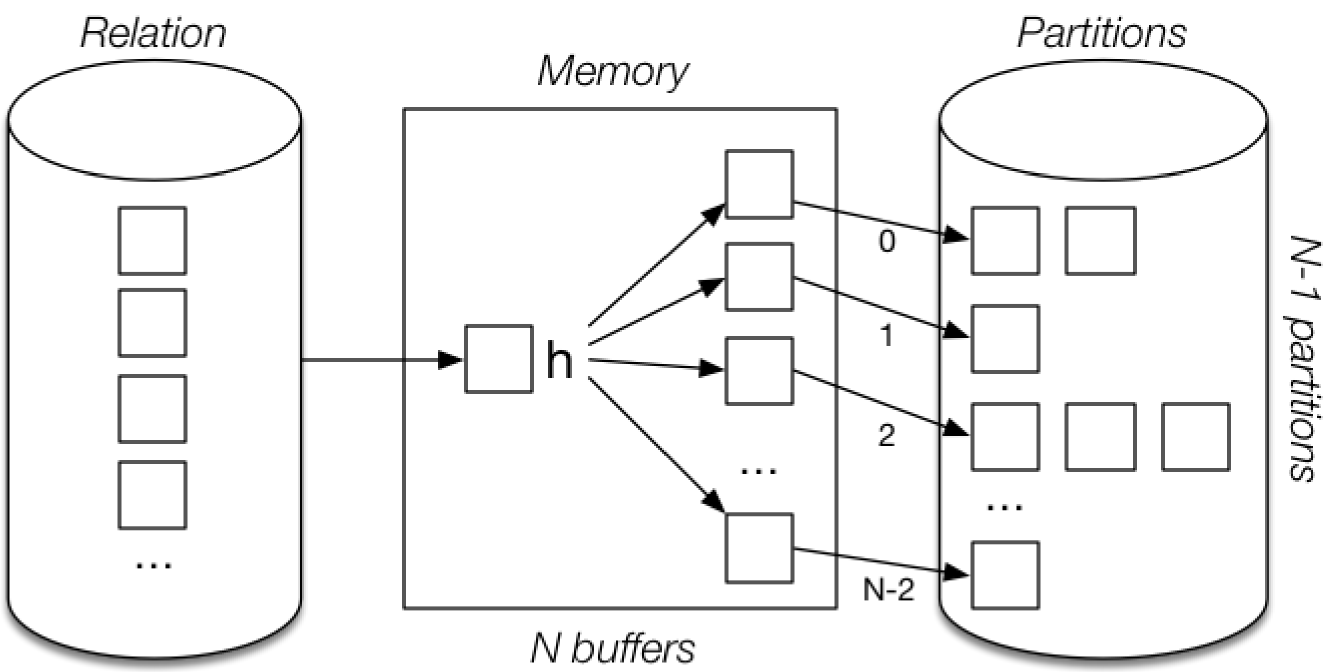
If $< \sqrt{b_R}$ buffers or poor hash distribution

- need to scan some partitions of S multiple times

<< ⋀ >>

❖ Grace Hash Join (cont)

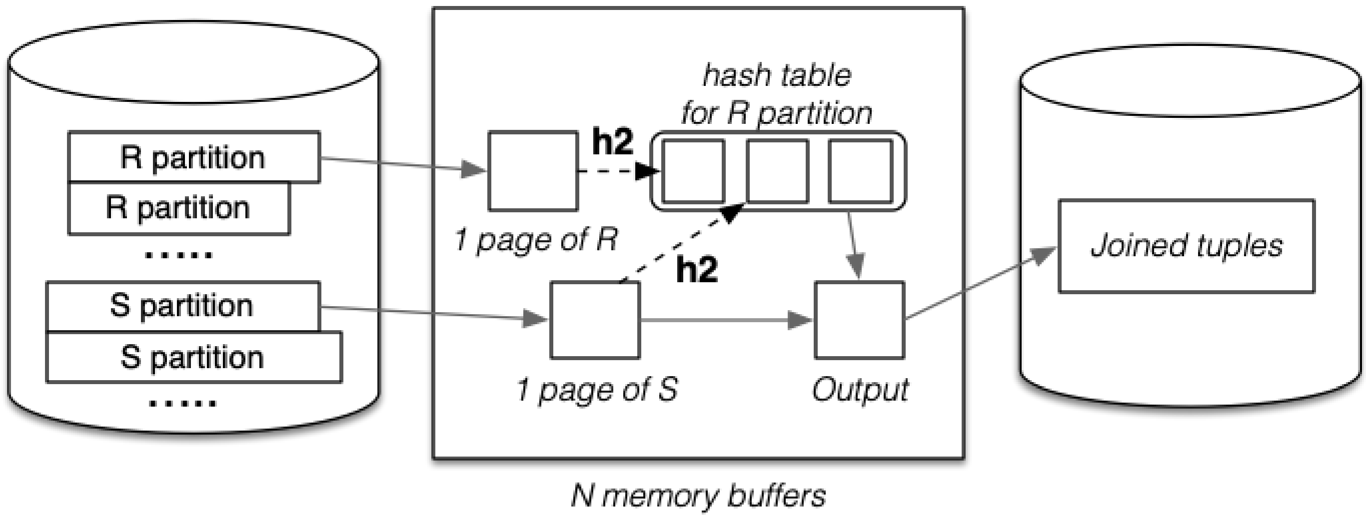
Partition phase (applied to both *R* and *S*):



<< ^ >>

❖ Grace Hash Join (cont)

Probe/join phase:



The second hash function (**`h2`**) simply speeds up the matching process.
Without it, would need to scan entire *R* partition for each record in *S* partition.

<< ^ >>

❖ Grace Hash Join (cont)

Cost of grace hash join:

- #pages in all partition files of $Rel \cong b_{Rel}$ (maybe slightly more)
- partition relation R ... Cost = $read(b_R) + write(\cong b_R)$ = $2b_R$
- partition relation S ... Cost = $read(b_S) + write(\cong b_S)$ = $2b_S$
- probe/join requires one scan of each (partitioned) relation
  Cost = $b_R + b_S$
- all hashing and comparison occurs in memory $\Rightarrow$ tiny cost

Total Cost = $2b_R + 2b_S + b_R + b_S$ = $3\,(b_R + b_S)$

<< ∧ >>

❖ Hybrid Hash Join

A variant of grace hash join if we have $\sqrt{b_R} < N < b_R+2$

- create $k \ll N$ partitions, 1 in memory, $k\text{-}1$ on disk
- buffers: 1 input, $k\text{-}1$ output, $p = N\text{-}k\text{-}2$ for in-memory partition

When we come to scan and partition S relation

- any tuple with hash 0 can be resolved (using in-memory partition)
- other tuples are written to one of k partition files for S

Final phase is same as grace join, but with only $k\text{-}1$ partitions.

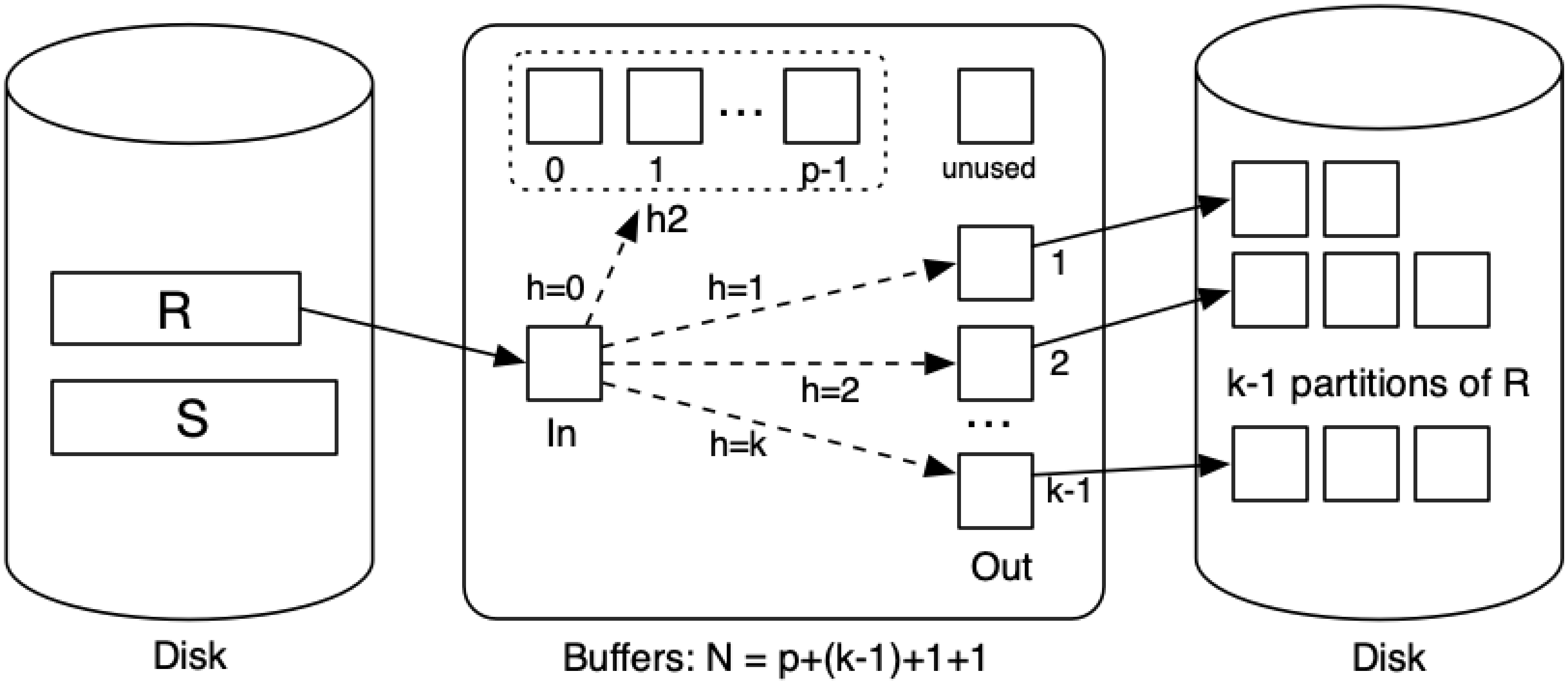
Comparison:

- grace hash join creates $N\text{-}1$ partitions on disk
- hybrid hash join creates 1 (memory) + $k\text{-}1$ (disk) partitions

<< ^ >>

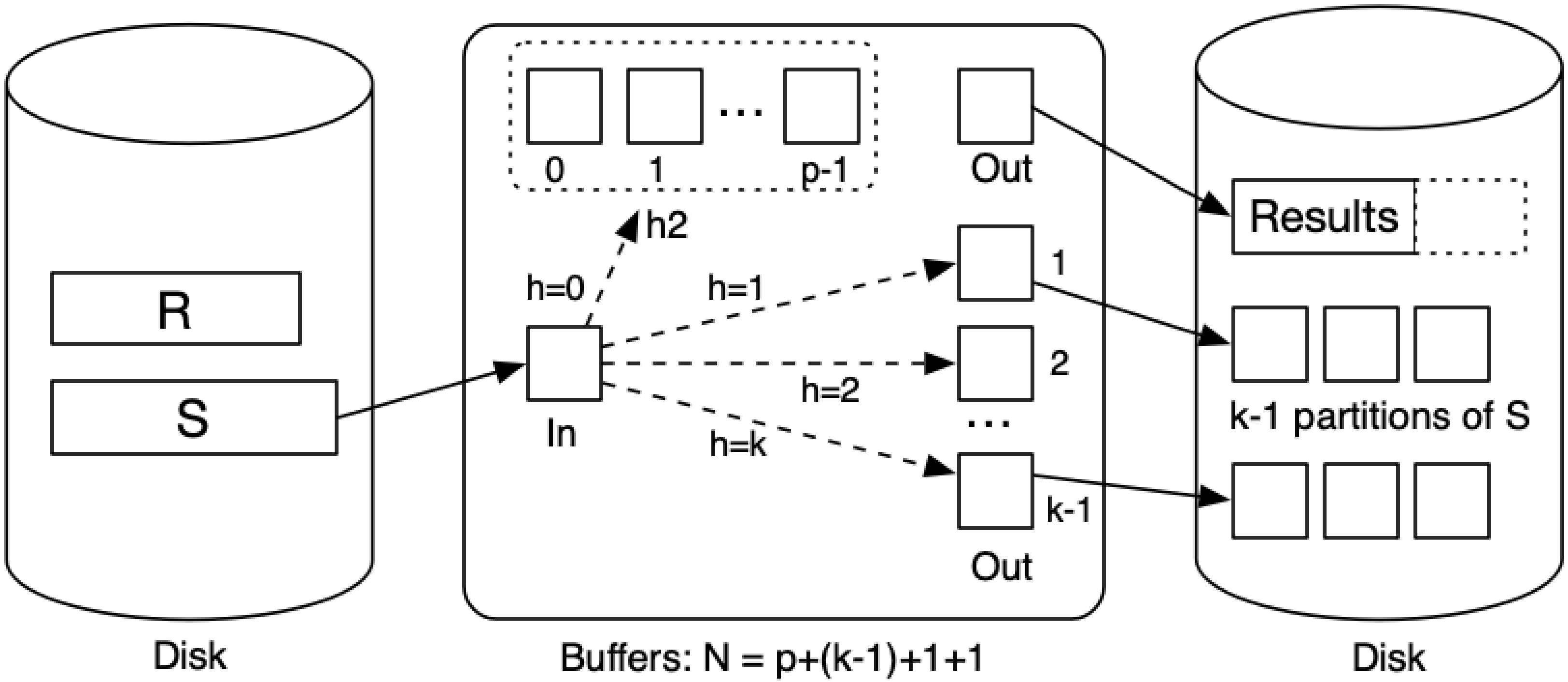
❖ Hybrid Hash Join (cont)

First phase of hybrid hash join (partitioning *R*):



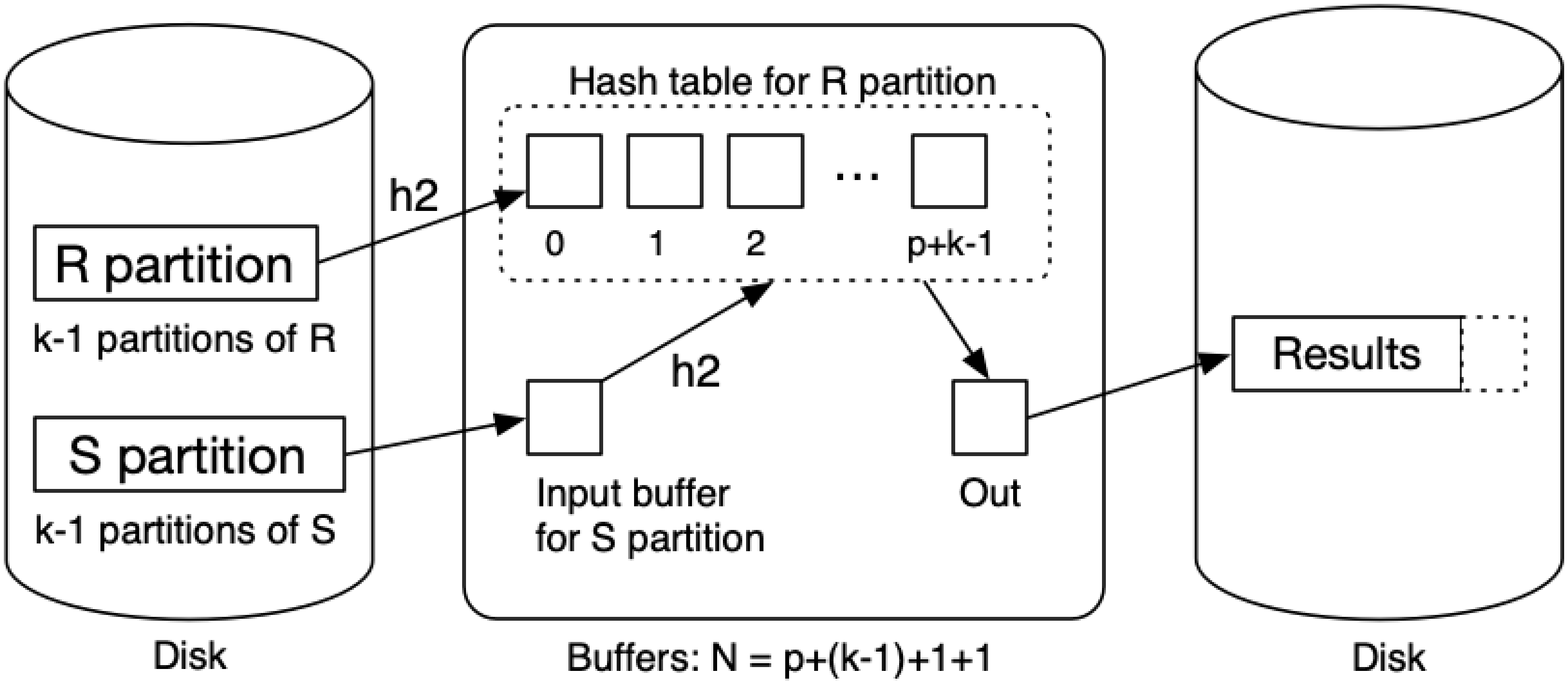
❖ Hybrid Hash Join (cont)

Next phase of hybrid hash join (partitioning *S*):



❖ Hybrid Hash Join (cont)

Final phase of hybrid hash join (finishing join):



<< ^ >>

❖ Hybrid Hash Join (cont)

Some observations:

- with *k* partitions, each partition has expected size $ceil(b_R/k)$
- holding 1 partition in memory needs $ceil(b_R/k)$ buffers
- trade-off between in-memory partition space and #partitions

Other notes:

- if $N = b_R+2$, using block nested loop join is simpler
- cost depends on *N* (but less than grace hash join)

For *k* partitions, Cost = $(3\text{-}2/k).(b_R+b_S)$

❖ Costs for Join 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

<< ∧ >>

❖ Costs for Join Example (cont)

Costs for hash join variants on example (*N*=103):

Hash Join Method	Cost Analysis	Cost
Hybrid Hash Join	$(3-2/k).(b_S+b_E)$ = 2.8((1000+2000) assuming k = 10 ... and one partition fits in 91 pages	8700
Grace Hash Join	$3(b_S+b_E)$ = 3(1000+2000)	9000
Simple Hash Join	$b_S + b_E.ceil(b_R/(N-3))$ = 1000 + ceil(1000/100).2000 = 1000 + 10.2000	21000
Sort-merge Join	$sort(S) + sort(E) + b_S + b_E$ = 2.1000.2 + 2.2000.2 + 1000 + 2000	11000
Nested-loop Join	$b_S + b_E.ceil(b_S/(N-2))$ = 1000 + 2000.ceil(1000/101) = 1000 + 10.2000	21000

COMP9315 21T1 ♢ Hash Join ♢ [16/17]

<< ⋀

❖ Join in PostgreSQL

Join implementations are under: `src/backend/executor`

PostgreSQL suports three kinds of join:

- nested loop join (`nodeNestloop.c`)
- sort-merge join (`nodeMergejoin.c`)
- hash join (`nodeHashjoin.c`) (hybrid hash join)

Query optimiser chooses appropriate join, by considering

- physical characteristics of tables being joined
- estimated selectivity (likely number of result tuples)

Produced: 25 Apr 2021