Iterators, Relational Operators and Joins

Hash Join Algorithms

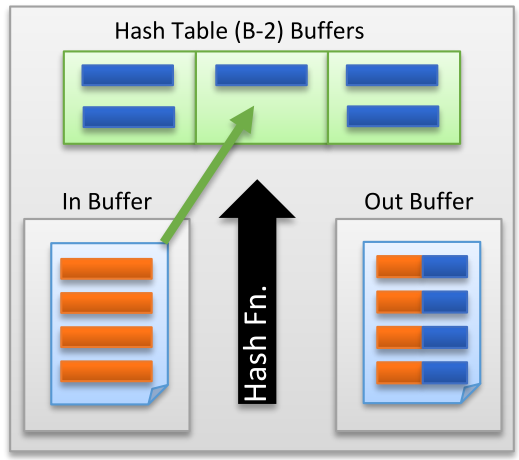
Alvin Cheung
Aditya Parameswaran

Reading: R & G Chapter 12 & 14

Naïve in Memory Hash Join: R ⋈ S

- Requires equality predicate:
  - Works for Equi-Joins & Natural Joins
- Assume R is smaller relation
  - Require R to fit in memory
- Simple algorithm:
  - Load all R into hash table
  - Scan S and probe R
- Memory requirements?
  - [R] < (B-2) * hash_fill_factor

What if R doesn't fit?



Properties that help

- $\sigma_{sid=4 \vee sid=6}(R \bowtie_{sid} S) = \sigma_{sid=4}(R \bowtie_{sid} S) \cup \sigma_{sid=6}(R \bowtie_{sid} S)$

- Can Decompose Into Smaller "Partial Joins"
- $R \bowtie_{sid} S = \cup (\sigma_{hash(sid)}(R) \bowtie_{sid} \sigma_{hash(sid)}(S))$
- Pick a hash function so that each $\sigma_{hash(sid)}(R)$ fits in memory!

Announcements

- Midterm next week!
  - Review session tomorrow
- Include descriptions in your OH tickets
  - Try out “party” mode and let us know how it goes

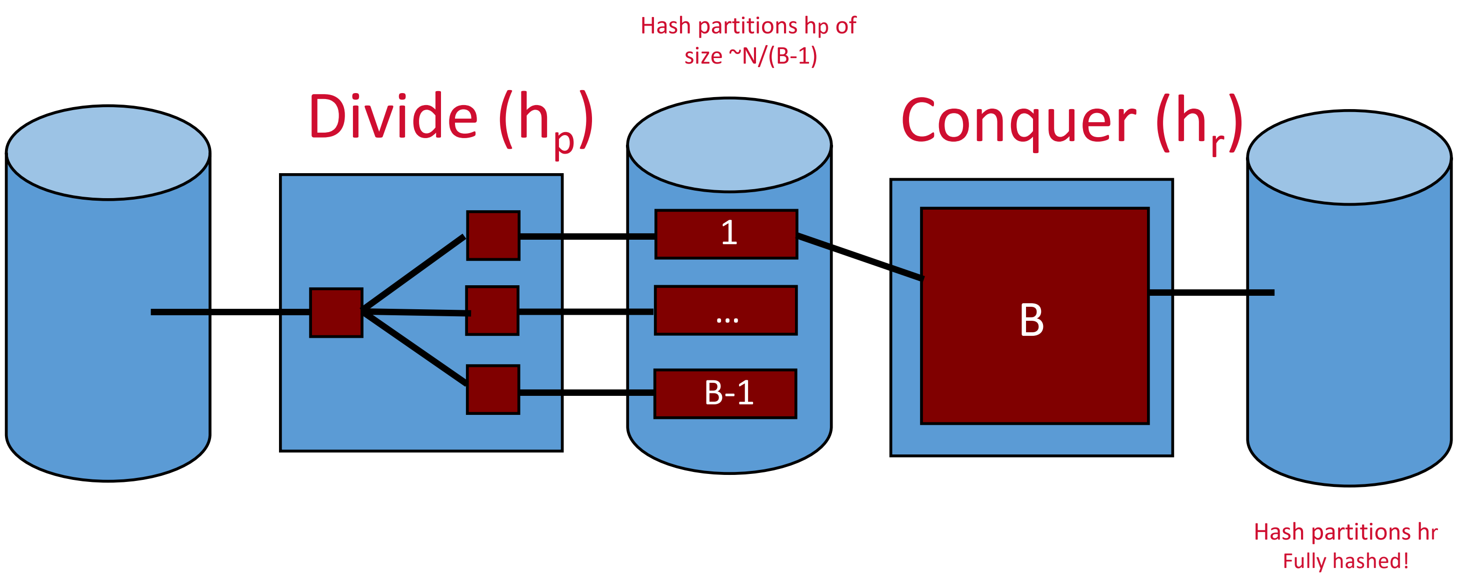
- Please turn on your video if you can

Grace Hash Join

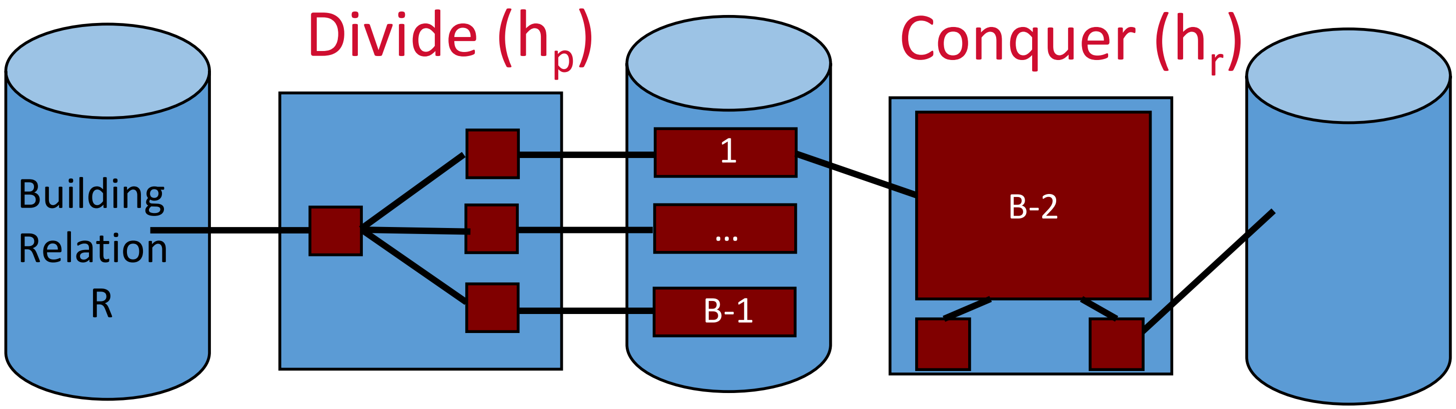
University of Tokyo's GRACE

- Requires equality predicate:
  - Equi-Joins & Natural Joins
- Two Stages:
  - **Partition** tuples from R and S by join key and store on scratch disk
    - all tuples for a given key now reside in same partition
    - same partition might have tuples with different keys but same hash value
  - **Build & Probe** a separate hash table for each partition (like in Naïve Hash)
    - Assume **partition** of smaller relation fits in memory
      - Recurse if necessary…

Remember External Hashing?

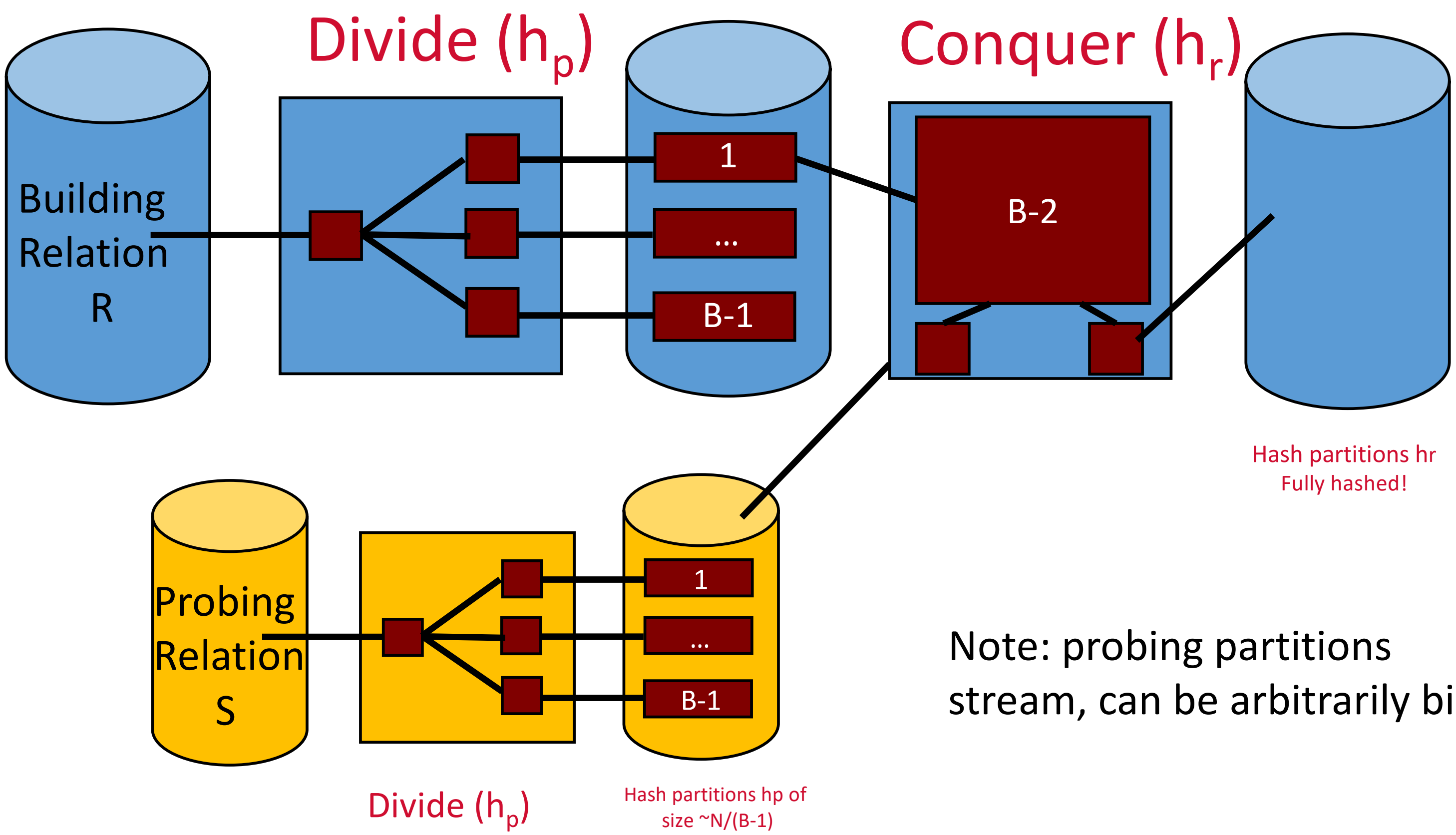


Sketch of Grace Hash Join



Hash partitions hr
Fully hashed!

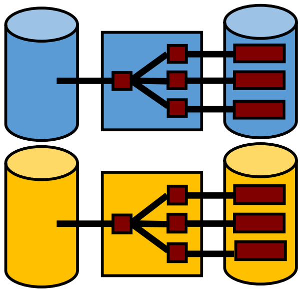
Sketch of Grace Hash Join, cont.



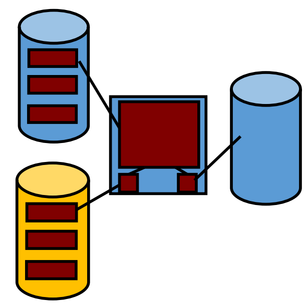
Note: probing partitions stream, can be arbitrarily big!

PsuedoCode, Grace Hash

```
For Cur in {R, S}
   For page in Cur
      Read page into input buffer
      For tup on page
         Place tup in output buf hash_p(tup.joinkey)
         If output buf full then flush to disk partition
   Flush output bufs to disk partitions
```



PsuedoCode, Grace Hash, cont.

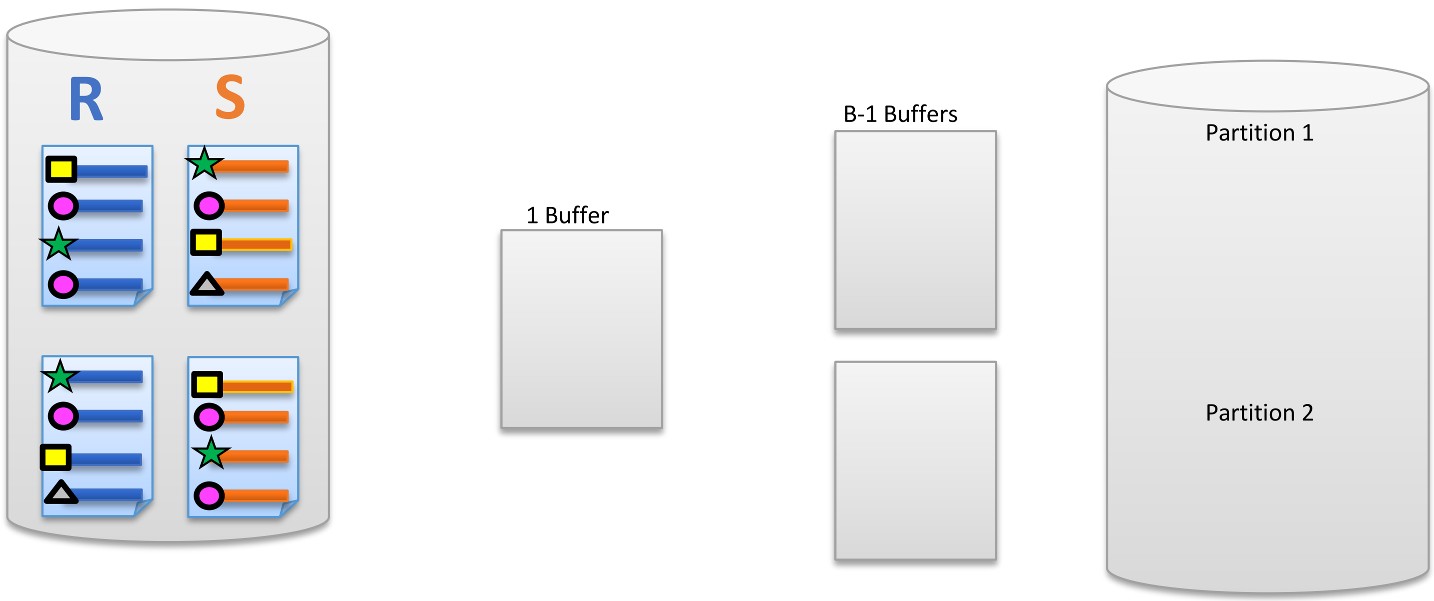


```
For Cur in {R, S}
   For page in Cur
      Read page into input buffer
      For tup on page
         Place tup in output buf hash_p(tup.joinkey)
         If output buf full then flush to disk partition
   Flush output bufs to disk partitions
For i in [0..(B-1))   // for each partition
   For page in R_i
      For tup on page
         Build tup into memory hash_r(tup.joinkey)
   For page in S_i
      Read page into input buffer
      For tup on page
         Probe memory hash_r(tup.joinkey) for matches
         Send all matches to output buffer
         Flush output buffer if full
```

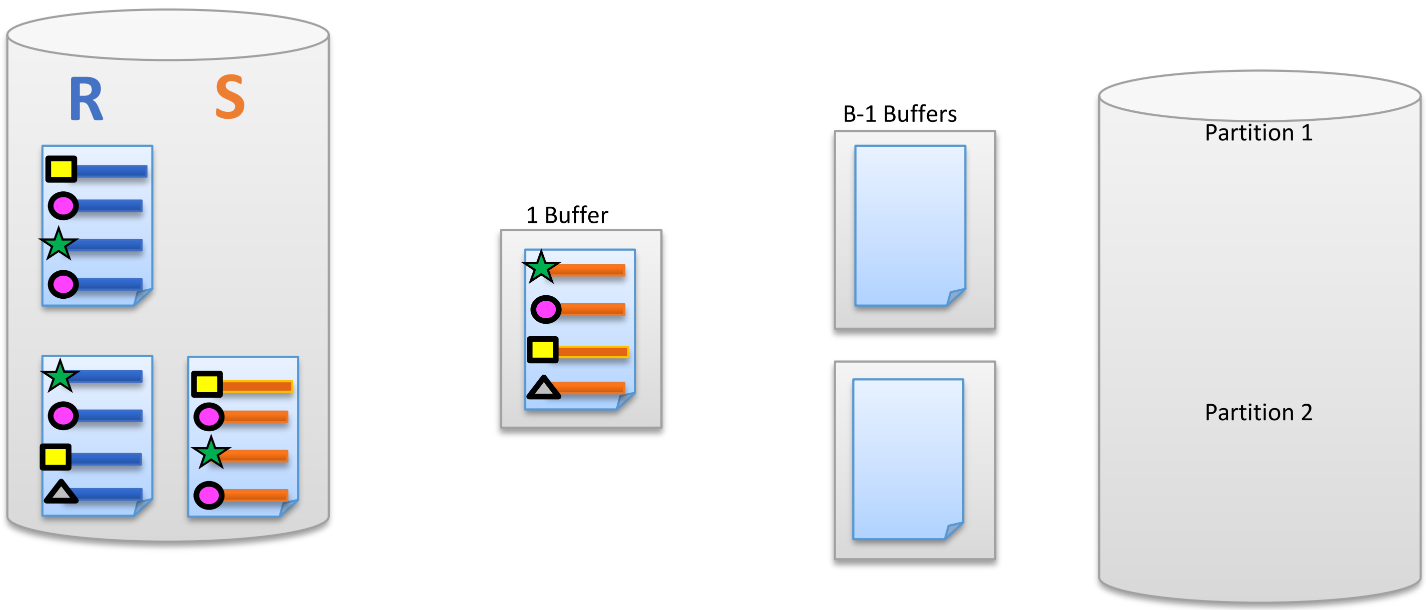
Grace Hash Join

- An animation
- Two phases:
  - Partition (divide)
  - Build & Probe hash tables (conquer)

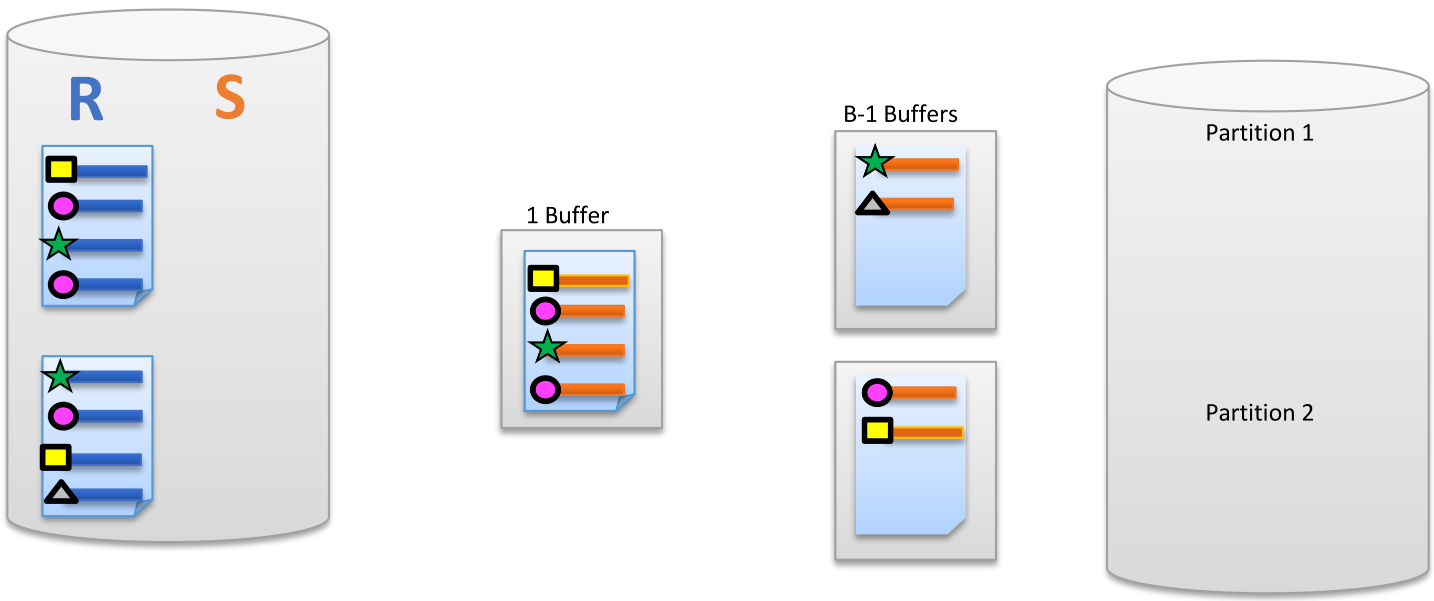
Grace Hash Join: ***Partition***



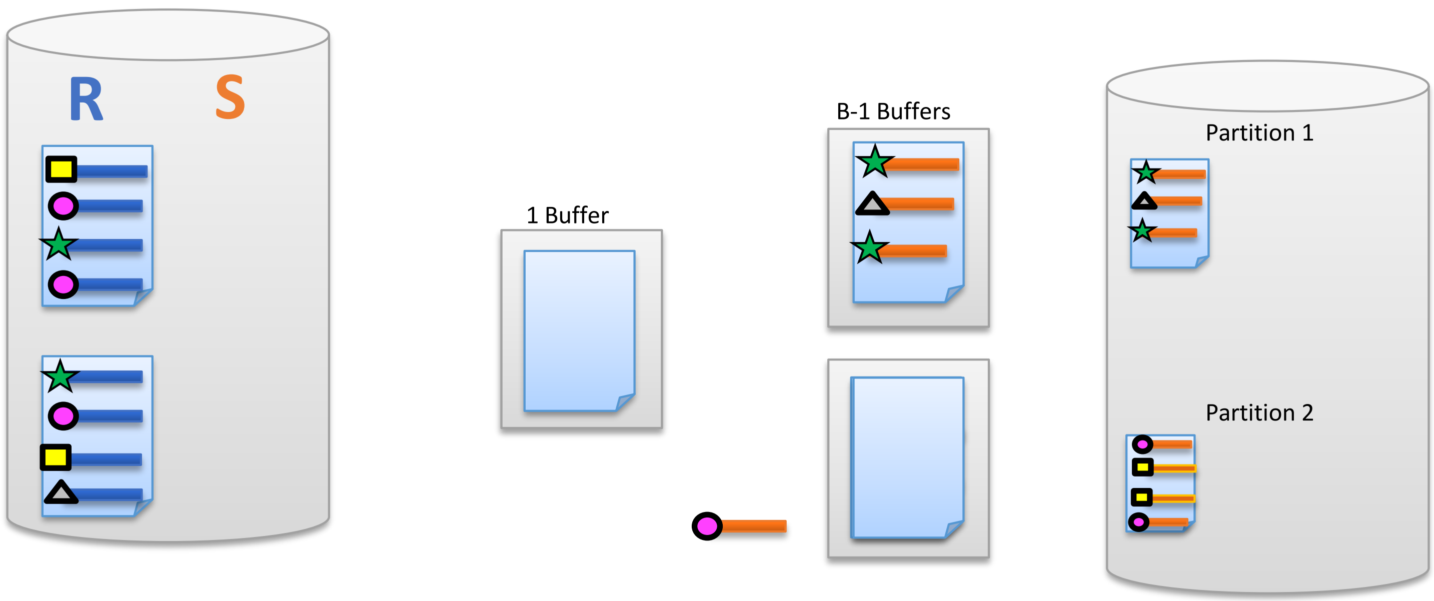
Grace Hash Join: ***Partition, Part 2***



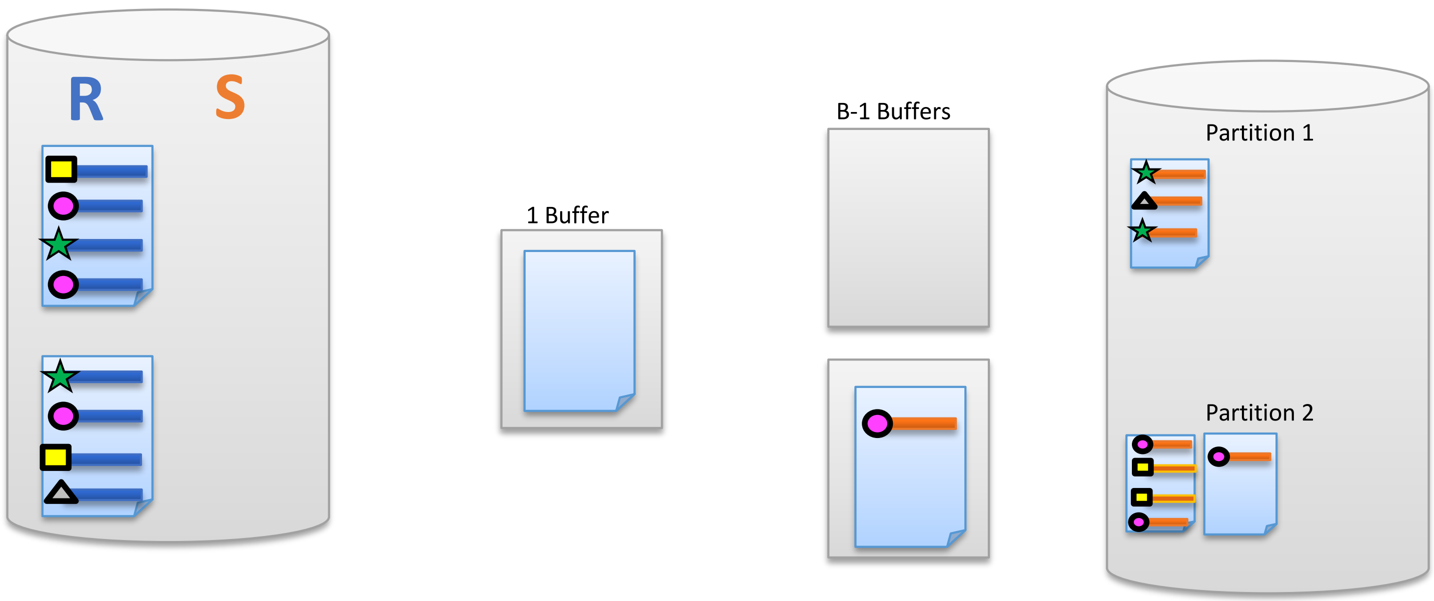
Grace Hash Join: ***Partition, Part 3***



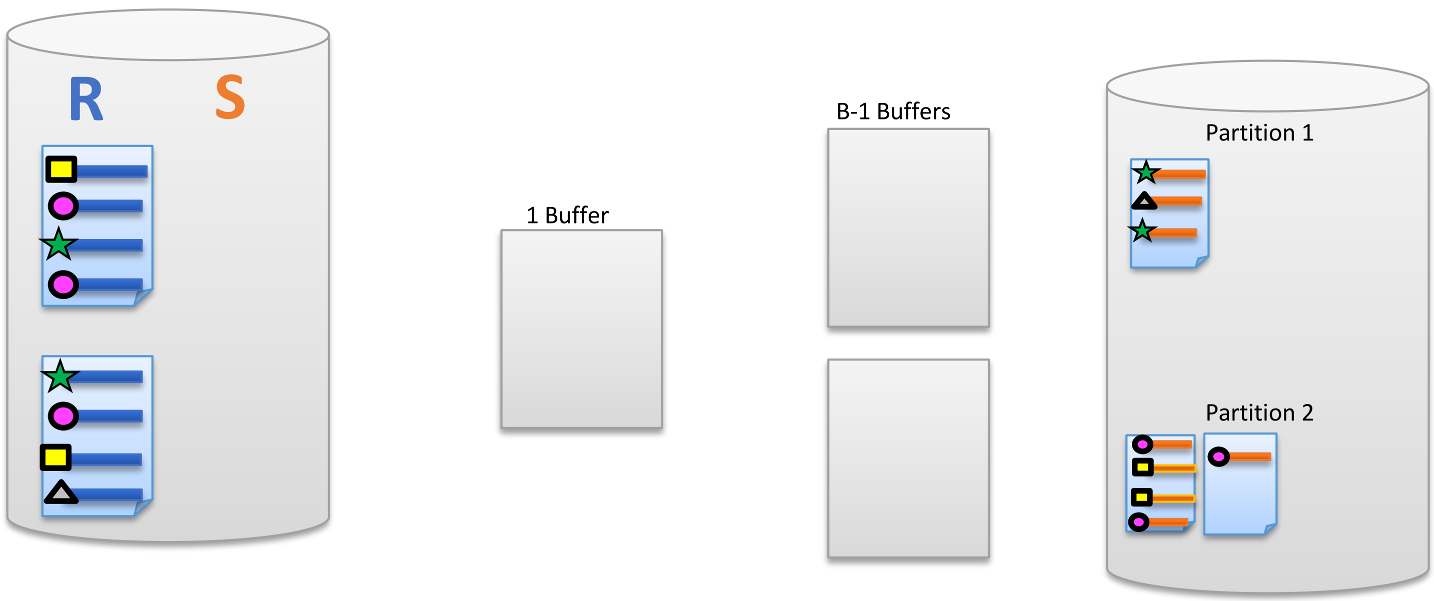
Grace Hash Join: ***Partition Part 4***



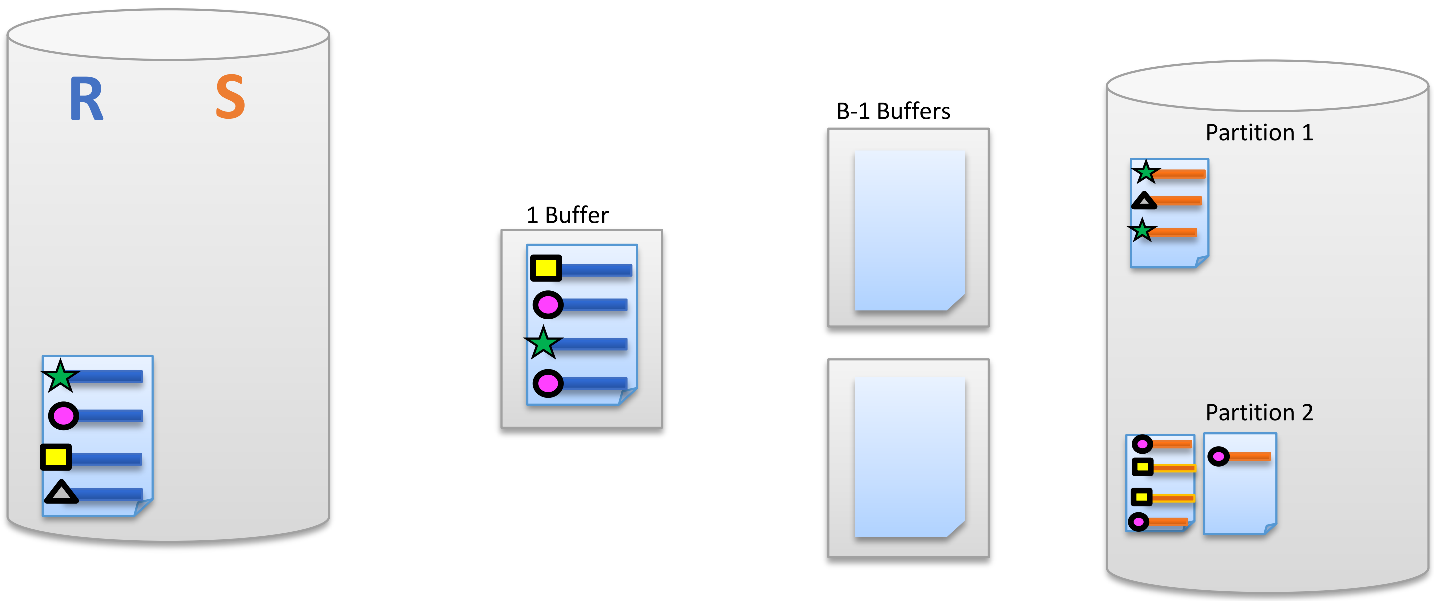
Grace Hash Join: ***Partition Part 5***



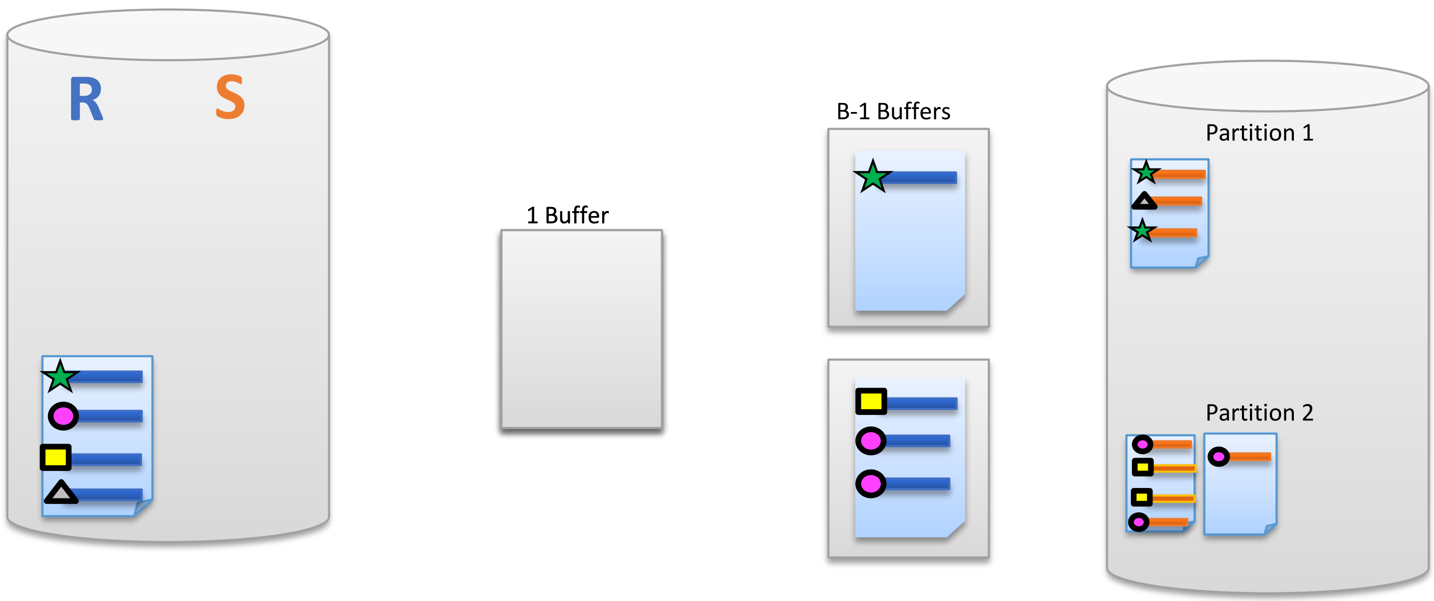
Grace Hash Join: ***Partition Part 6***



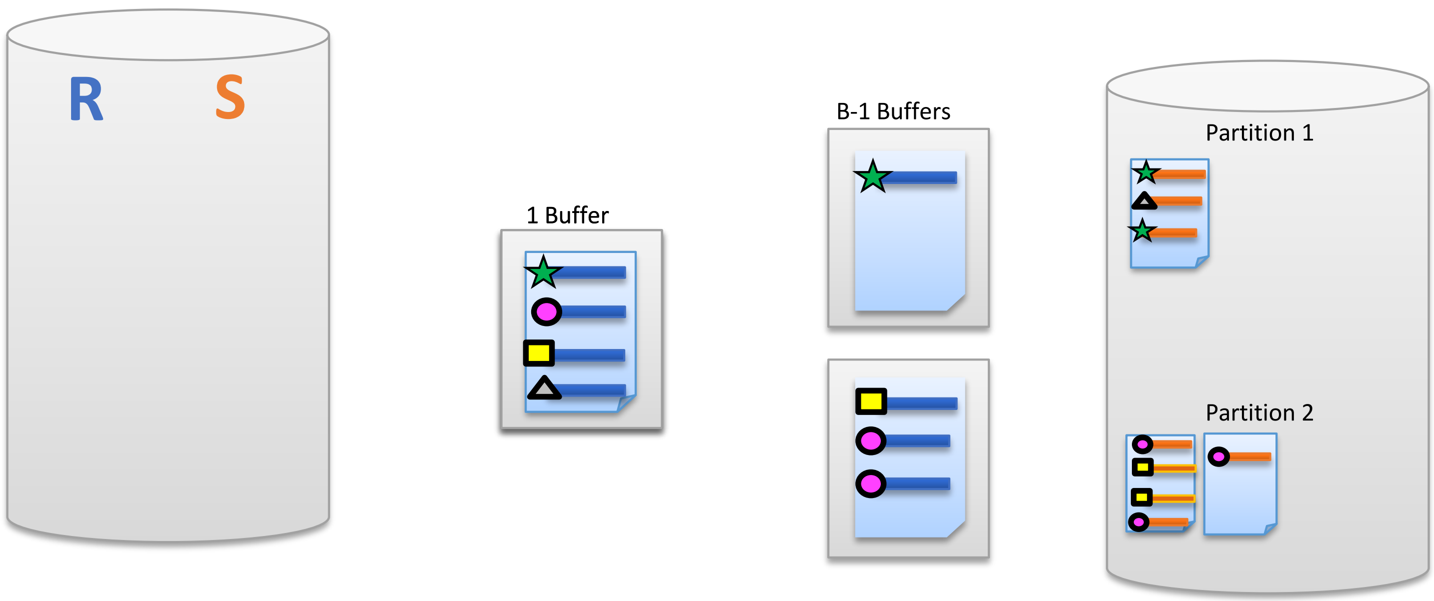
Grace Hash Join: ***Partition Part 7***



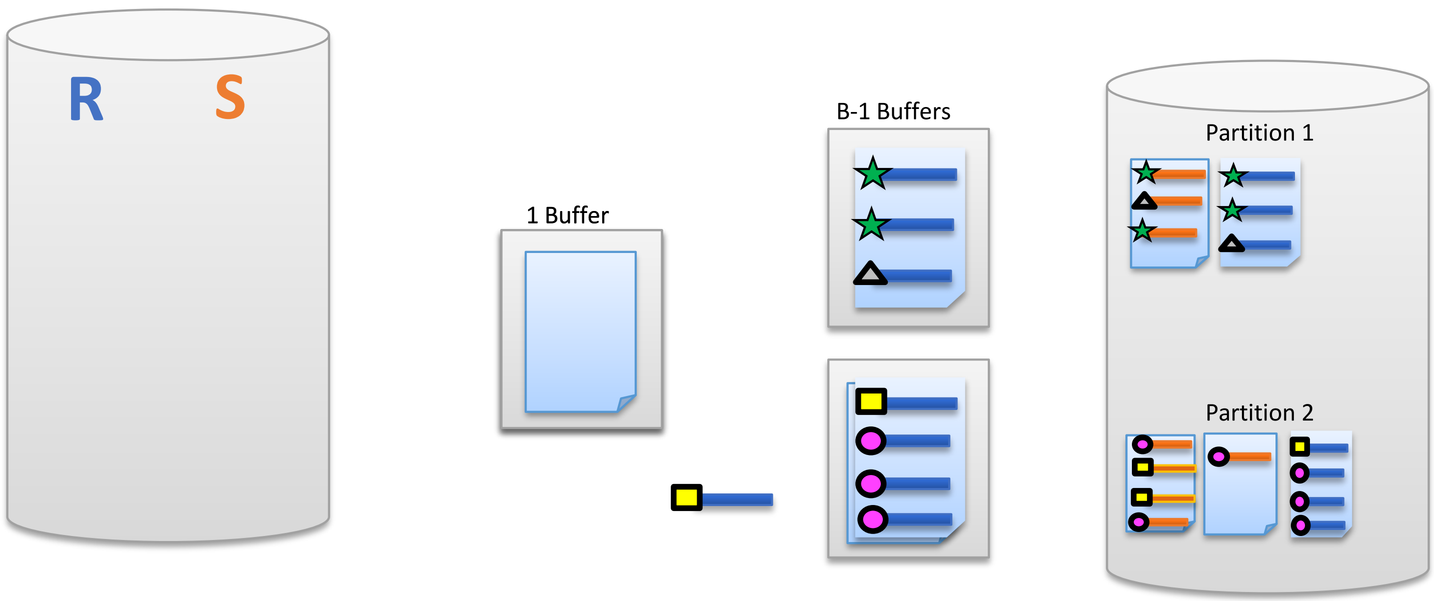
Grace Hash Join: ***Partition Part 8***



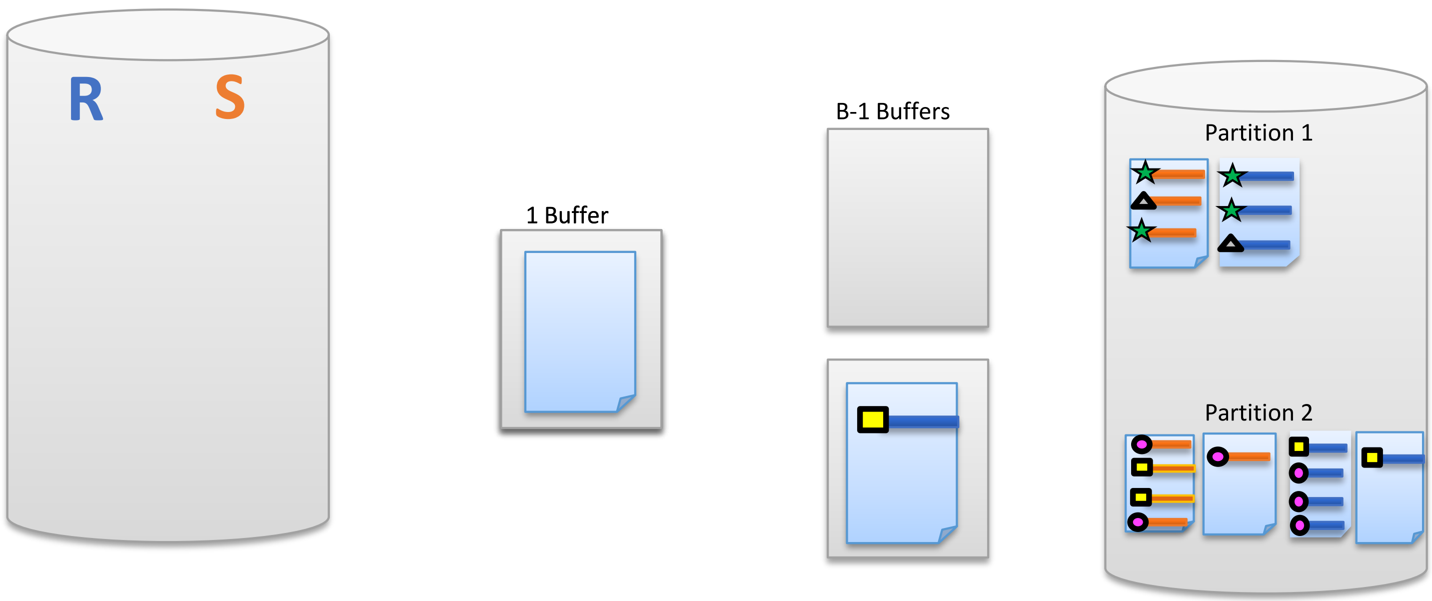
Grace Hash Join: ***Partition 9***



Grace Hash Join: ***Partition Part 10***

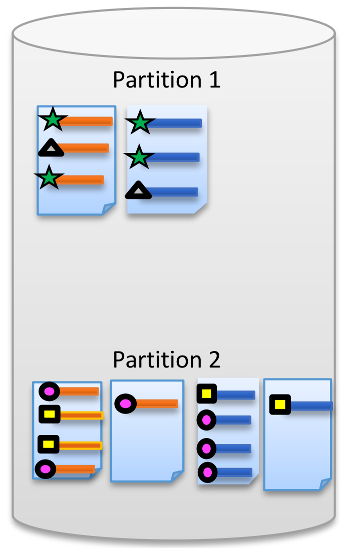


Grace Hash Join: ***Partition Part 11***

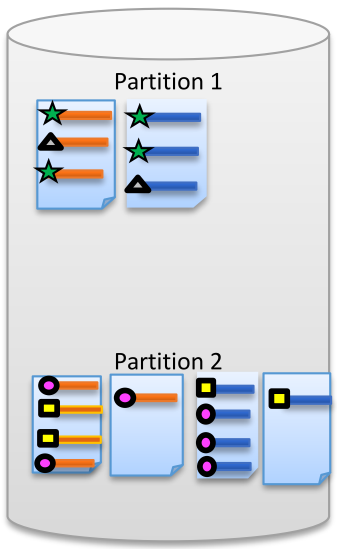


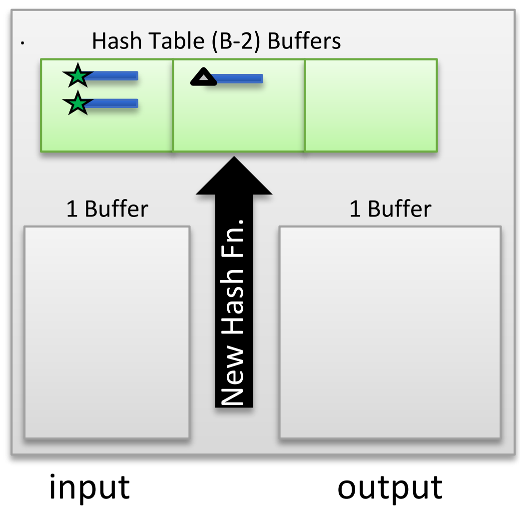
Grace Hash Join: ***Partition Part 12***

- Each key is assigned to one partition
  - e.g., green star keys only in Partition 1
- Sensitive to key Skew
  - Purple circle key
- Each partition could be on a different disk or even different machine



Grace Hash Join: ***Build*** **&** *Probe*

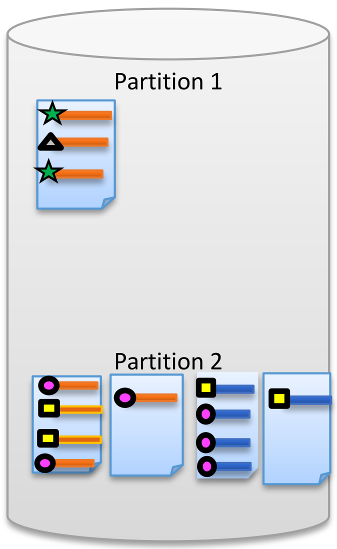


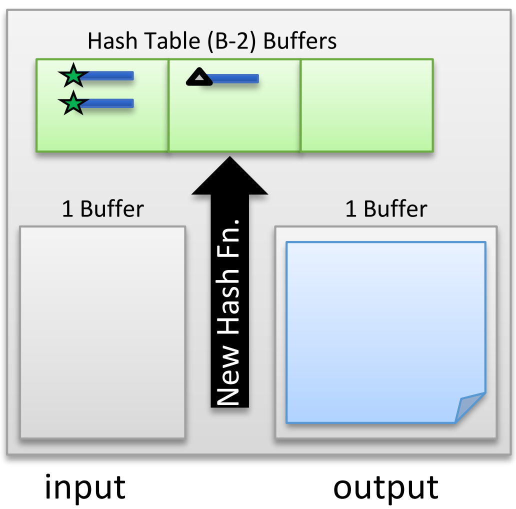


Blue tuples are from R

Orange tuples are from S

Grace Hash Join: *Build* & ***Probe Part 2***

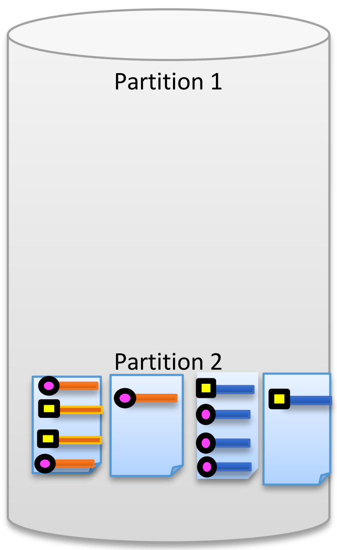


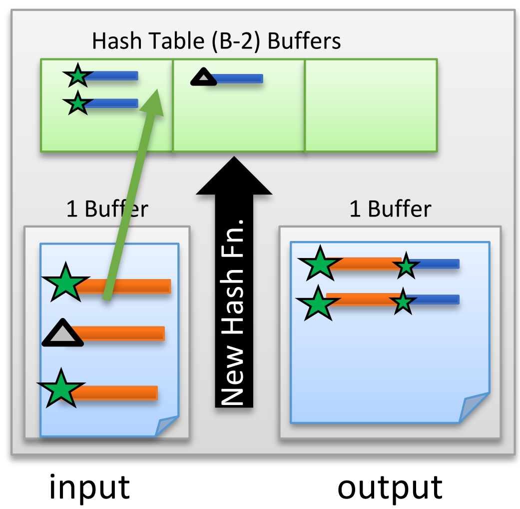


Blue tuples are from R

Orange tuples are from S

Grace Hash Join: *Build* & ***Probe Part 3***

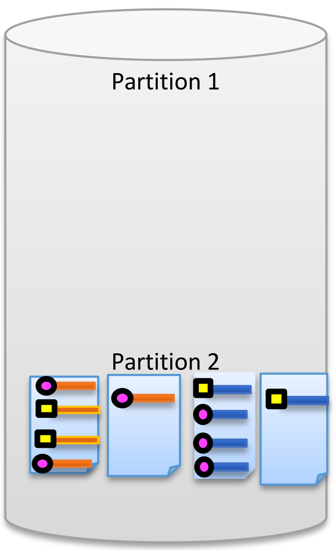


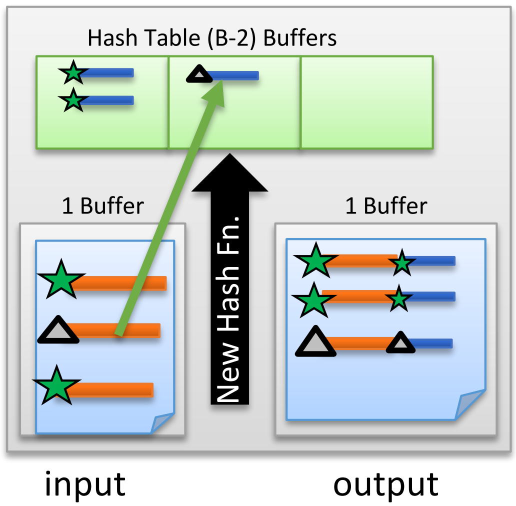


Blue tuples are from R

Orange tuples are from S

Grace Hash Join: *Build* & ***Probe Part 4***

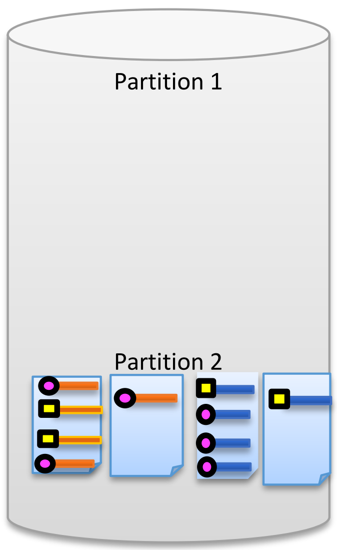


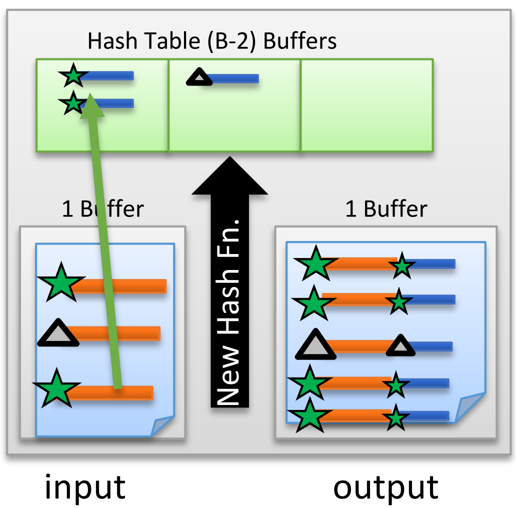


Blue tuples are from R

Orange tuples are from S

Grace Hash Join: *Build* & ***Probe Part 5***

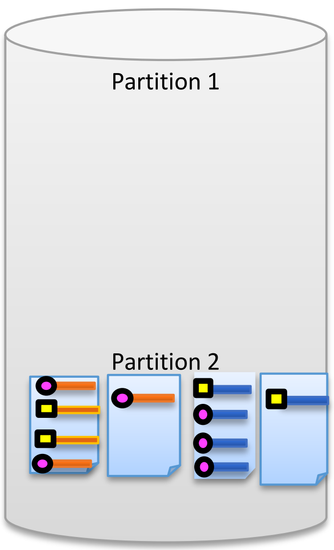


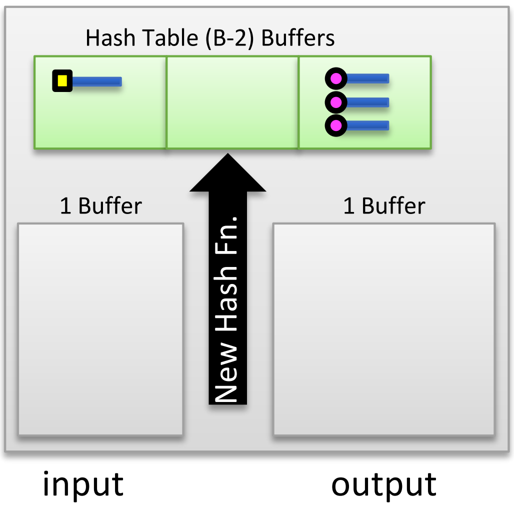


Blue tuples are from R

Orange tuples are from S

Grace Hash Join: ***Build*** *& Probe Part 6*

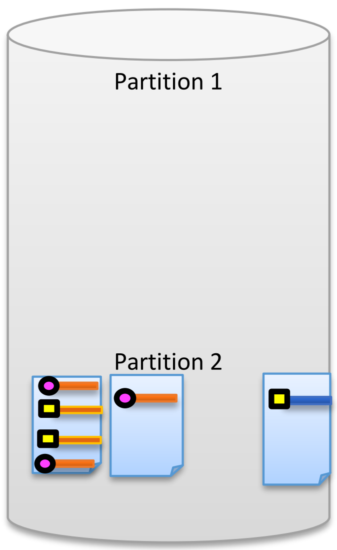


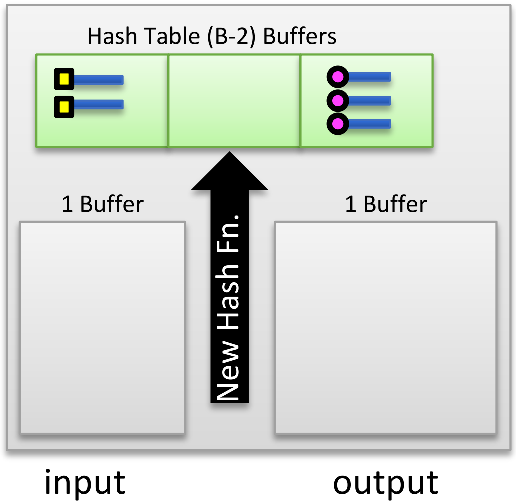


Blue tuples are from R

Orange tuples are from S

Grace Hash Join: ***Build*** *& Probe Part 7*

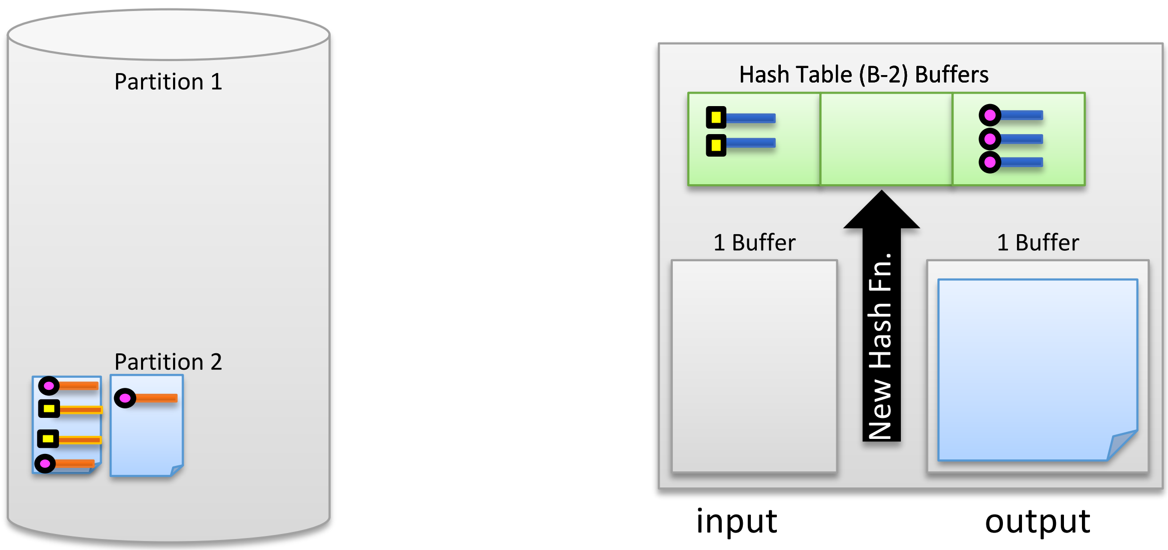




Blue tuples are from R

Orange tuples are from S

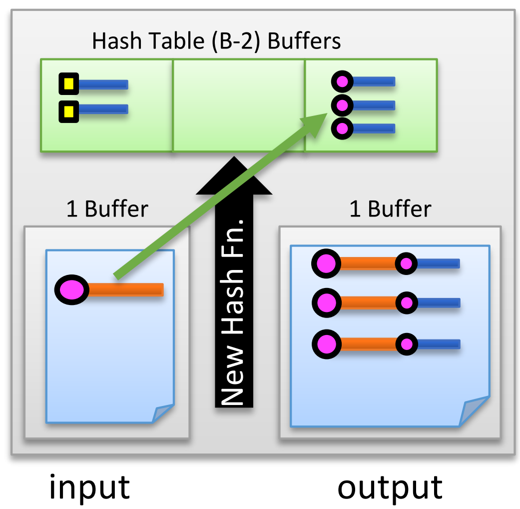
Grace Hash Join: *Build* ***& <u>Probe Part 8</u>***



Blue tuples are from R

Orange tuples are from S

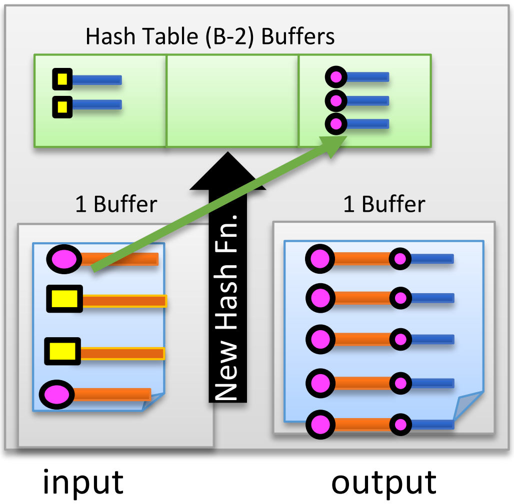
Grace Hash Join: *Build* **&** ***Probe Part 9***



Blue tuples are from R

Orange tuples are from S

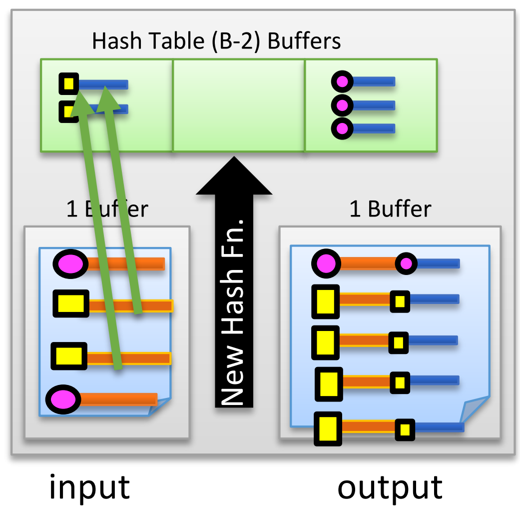
Grace Hash Join: *Build* & ***<u>Probe Part 10</u>***



Blue tuples are from R

Orange tuples are from S

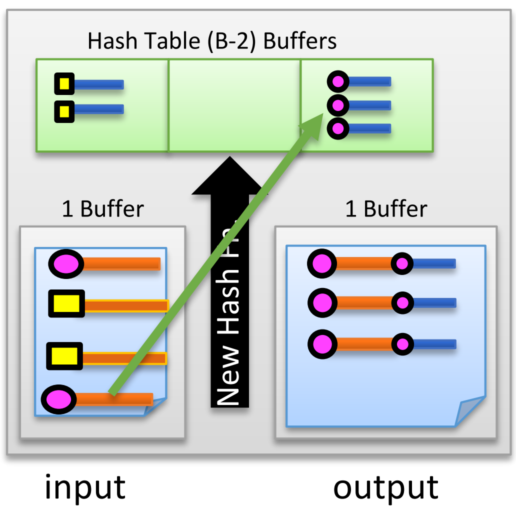
Grace Hash Join: *Build* ***&*** ***Probe Part 11***



Blue tuples are from R

Orange tuples are from S

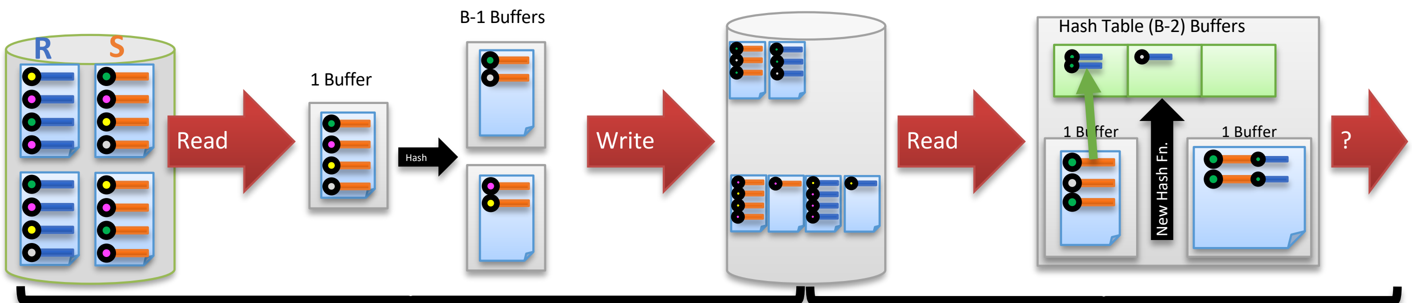
Grace Hash Join: *Build* & ***Probe Part 12***



Blue tuples are from R

Orange tuples are from S

Summary of Grace Hash Join



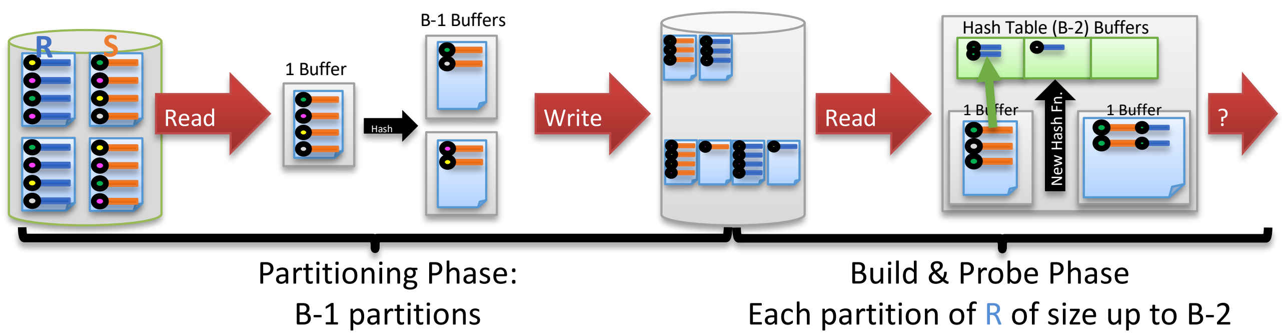
Partitioning Phase:
B-1 partitions

Build & Probe Phase
Each partition of R of size up to B-2

What is the Cost?

Cost of Hash Join

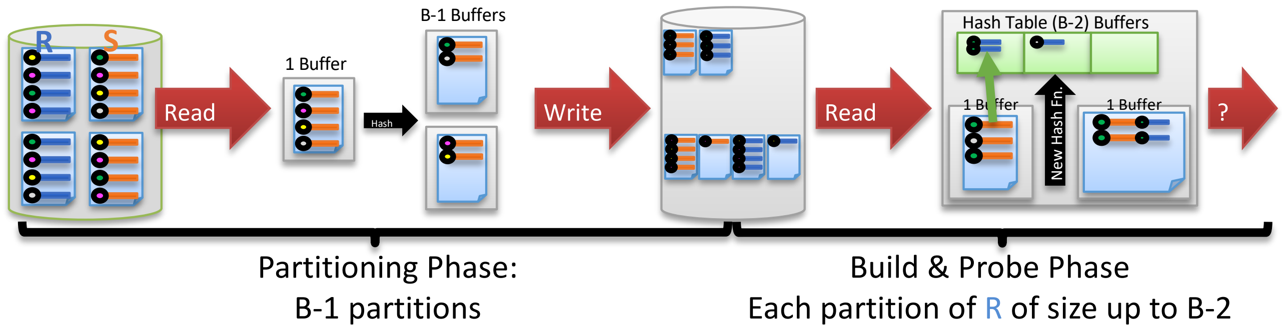
[R]=1000, p_R=100, |R| = 100,000
[S]=500, p_S=80, |S| = 40,000



- Partitioning phase: read+write both relations
  ⇒ 2([R]+[S]) I/Os
- Matching phase: read both relations, forward output
  ⇒ [R]+[S]
- Total cost of 2-pass hash join = 3([R]+[S])
  - 3 * (1000 + 500) = 4500

Cost of Hash Join Part 2

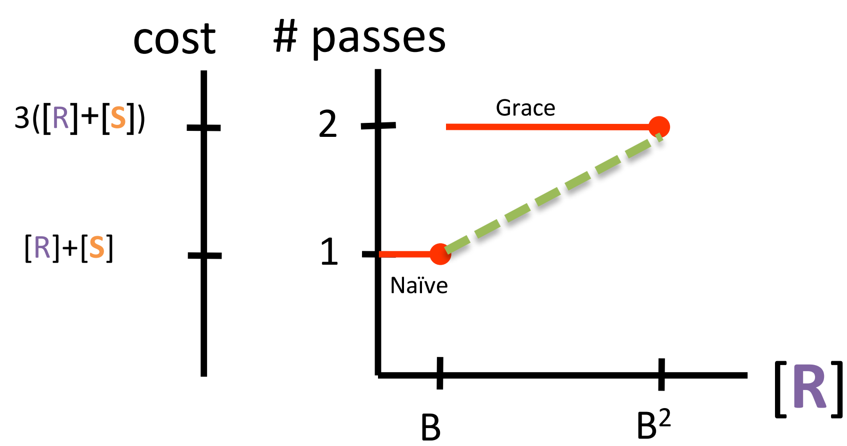
[R]=1000, p_R=100, |R| = 100,000
[S]=500, p_S=80, |S| = 40,000



- **What's the max size of R that can be processed in 1 pass of build & probe?**
- Build hash table on **R** with uniform partitioning
  - **Partitioning Phase divides R into (B-1) runs of size [R] / (B-1)**
  - **Matching Phase requires each ([R] / (B-1)) < (B-2)**
  - **Solving backwards gives R < (B-1) (B-2) ≈ B^2**
- Note: no constraint on size of S (probing relation)!

Cost of Hash Join Part 3

[R]=1000, p_R=100, |R| = 100,000
[S]=500, p_S=80, |S| = 40,000



- **Naïve Hash Join: *requires* [R] < B**
  - Put all of R in hash table
  - 1/3 the I/O cost of Grace!
- **Grace** Hash Join: 2-passes for [R] **< B²**
- **Hybrid** Hash Join: an algorithm that adapts between the two
  - Tricky to tune

TINSTAFL!!

Hash Join vs. Sort-Merge Join

- Sorting pros:
  - Good if input already sorted, or need output sorted
  - Not sensitive to data skew or bad hash functions
- Hashing pros:
  - For join: # passes depends on size of smaller relation
    - E.g. if smaller relation is <B, naïve/hybrid hashing is great
  - Good if input already hashed, or need output hashed

Recap

- Nested Loops Join
  - Works for arbitrary Θ
  - Make sure to utilize memory in blocks

- Index Nested Loops
  - For equi-joins
  - When you already have an index on one side

- Sort/Hash
  - For equi-joins
  - No index required
  - Hash better if one relation is much smaller than other

- No clear winners – may want to implement them all
- Be sure you know the cost model for each
  - You will need it for query optimization!