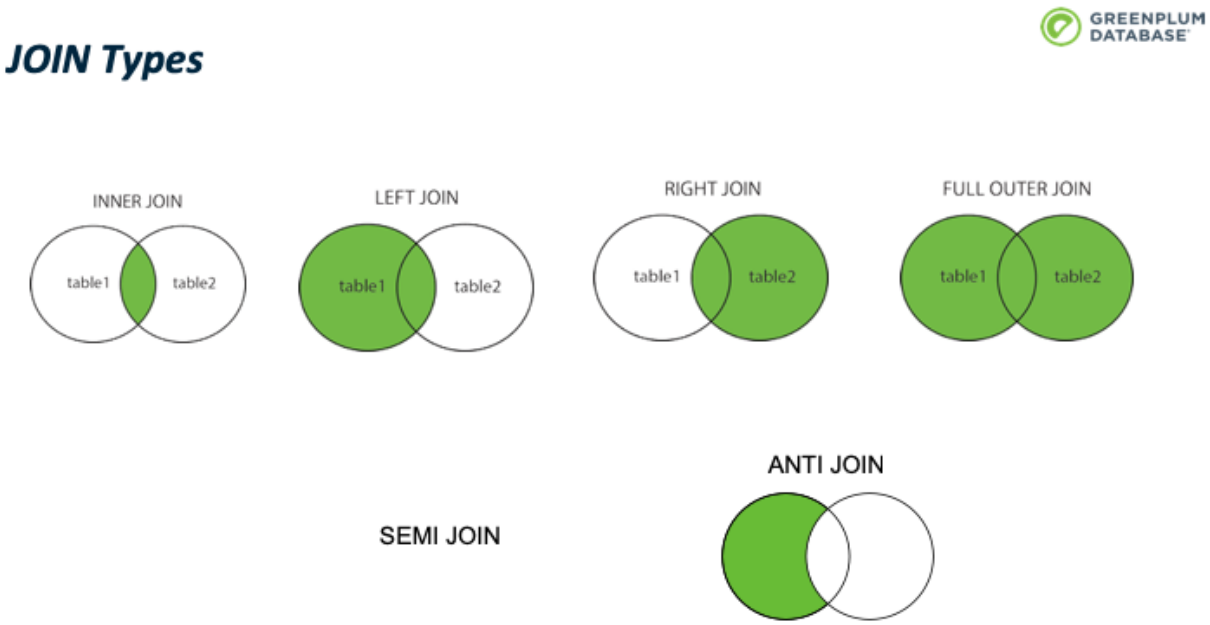
course 8 implementing Join

1. Join type



there are two tables as examples: students table, score table

Examples

GREENPLUM DATABASE

student table

id	name	age
10	Jack	25
12	Tom	26
13	Ariel	25

score table

id	stu_id	subject	score
1	10	math	95
2	10	history	98
3	12	math	97
4	15	history	92

```
CREATE TABLE student(id int, name text, age int);
CREATE TABLE score(id int, stu_id int, subject text, score int);
INSERT INTO student VALUES (10, 'Jack', 25), (12, 'Tom', 26), (13, 'Ariel', 25);
INSERT INTO score VALUES (1, 10, 'math', 95), (2, 10, 'history', 98), (3, 12, 'math', 97), (4, 15, 'history', 92);
```

and results of several types of joinning is shown below:

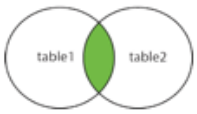
JOIN Examples

GREENPLUM DATABASE

```
# SELECT * FROM student;
 id | name  | age
----+-------+-----
 10 | Jack  |  25
 13 | Ariel |  25
 12 | Tom   |  26
```

```
# SELECT * FROM score;
 id | stu_id | subject | score
----+--------+---------+-------
  1 |     10 | math    |    95
  2 |     10 | history |    98
  3 |     12 | math    |    97
  4 |     15 | history |    92
```

INNER JOIN




```
 name | score
------+-------
 Tom  |    97
 Jack |    95
 Jack |    98
(3 rows)
```

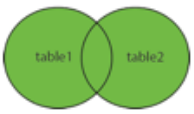
LEFT JOIN


```
 name  | score
-------+-------
 Tom   |    97
 Jack  |    95
 Jack  |    98
 Ariel |
(4 rows)
```

RIGHT JOIN


```
 name | score
------+-------
 Tom  |    97
      |    92
 Jack |    95
 Jack |    98
(4 rows)
```

FULL OUTER JOIN

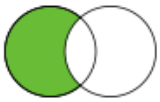



```
 name  | score
-------+-------
 Tom   |    97
       |    92
 Jack  |    95
 Jack  |    98
 Ariel |
(5 rows)
```

Semi JOIN

```
 id | name
----+------
 10 | Jack
 12 | Tom
(2 rows)
```

ANTI JOIN



```
 name
-------
 Ariel
(1 row)
```

2. implementing join

we consider the strategies to implement join:

- nested join: the most simple and unefficient without buffer
- sort-merge: work best if tables are sorted on join attributes
- hash-based: requires good hash function and sufficient buffering
    - usually has the best performance for large amount of data, but only deal with equal-join, not join condition such as C1 > C2

2.1 Nested join

the useful methods of practice of Nested join is Block Nested Loop Join

Block Nested Loop Join

Method (for *N* memory buffers):

- read *N–2*–page chunk of *R* into memory buffers
- for each *S* page
  check join condition on all (t_R, t_S) pairs in buffers
- repeat for all *N–2*–page chunks of *R*

so why this methods feasible?

Many queries have the form

```
select * from R,S where r.i=s.j and r.x=K
```

This would typically be evaluated as

```
Tmp = Sel[x=K](R)
Res = Join[i=j](Tmp, S)
```

If `Tmp` is small ⇒ may fit in memory (in small #buffers)

- Index Nested Loop join

and since every-time we read some Outer-realtion records into buffer, and then we need to iterate all tuples of inner-relation, and it really cost time, so we plan to use indexed for

inner-relation.

we know there are a problem with nested-loop that it needs repeated scans of entire inner relation S, and if there is an index on inner relation, we can avoid such repeated scanning

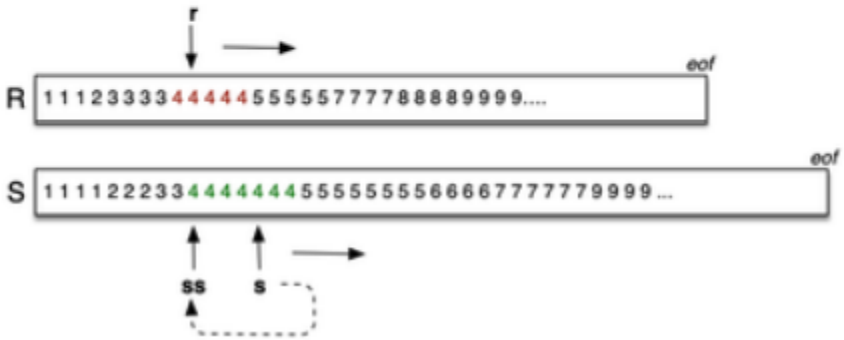
```
for each tuple r in relation R {
    use index to select tuples
        from S where s.j = r.i
    for each selected tuple s from S {
        add (r,s) to result
}   }
```

2.2 Sort-Merge join

we sort both relation on join key, and then compare each pair in (outer-relation, inner-relation), and then merge two relation into one.

Method requires several cursors to scan sorted relations:

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



2.3 Hash join

generally, Hash join has twp phases: build and probe

- build phase: build hash table for the small table (inner table)

- probe phase: scanning the content of another associated table and detecting whether there is a matching row/tuple through hash table (outer table)

there are several types of joining, such as:

- inner join
- full outer join
- semi join
- anti join

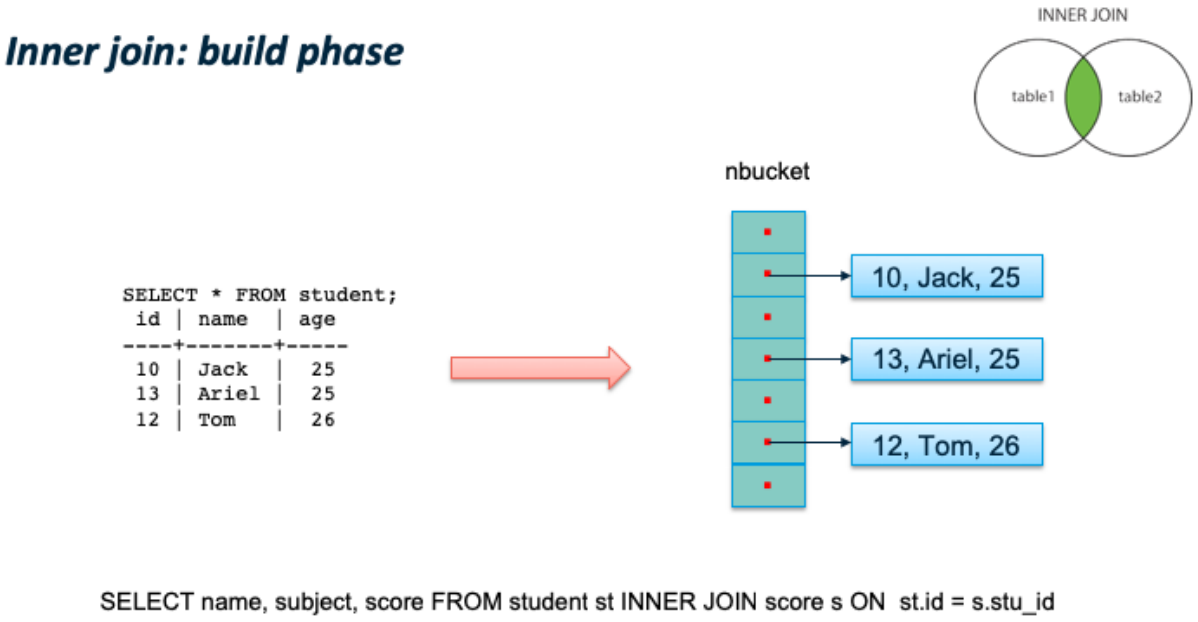
all of them implementing based on the hypothesis that the inner table is smaller enough to fit into memory

2.3.1 Inner join

notes: the optimizer always choose the small table as inner table

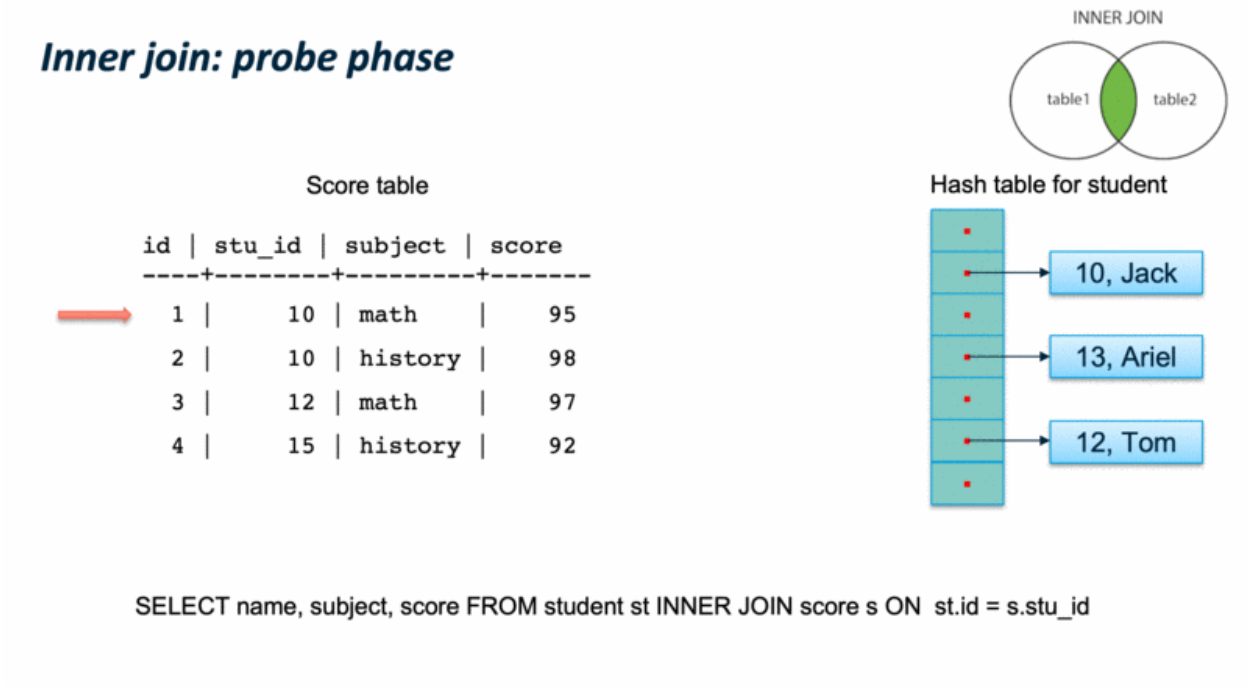
build phase

first, we need to build the hash table by scanning each tuple in the innear table, calculating it hash value according to the value of the join key, and put it into corresponding bucket of the hash table. After scanning the innear table, the hash table is built.



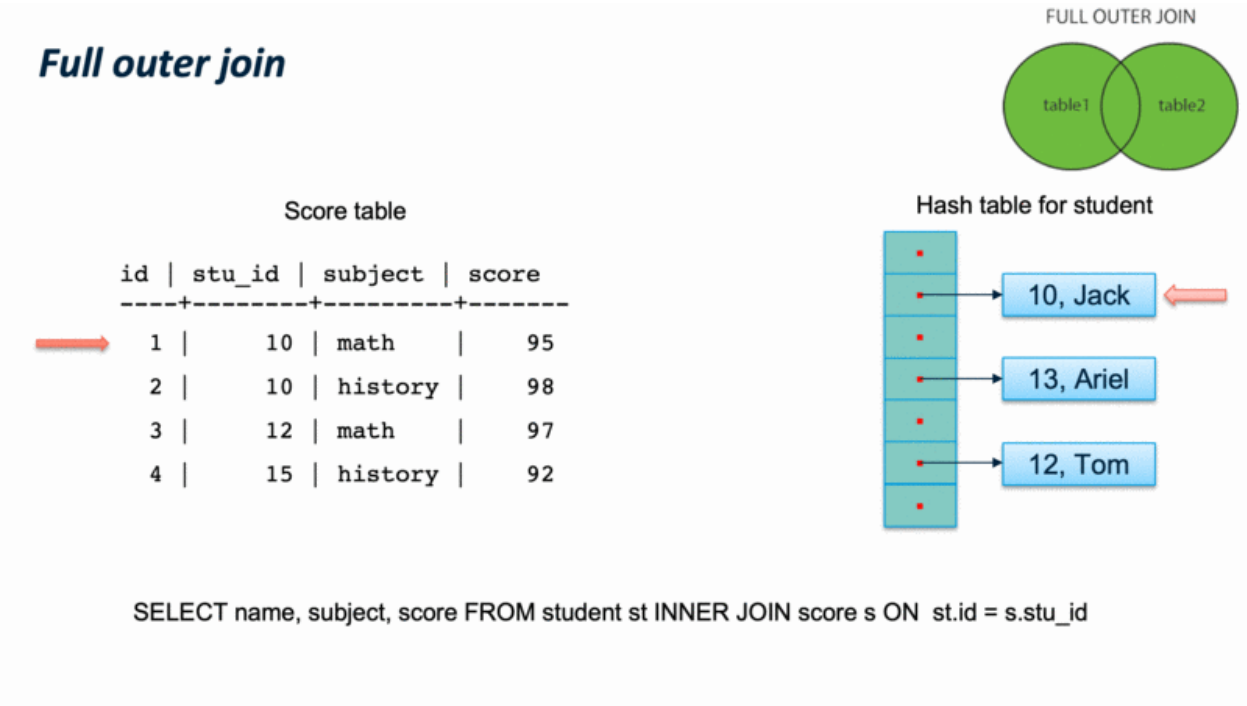
probe phase

in this stage, we need to scan the each tuple of the outer table, and then calculate their hash value using the same way as shown above, if there is a matching tuple in the hash table, and all the query conditions are met, the tuple will be ouput.



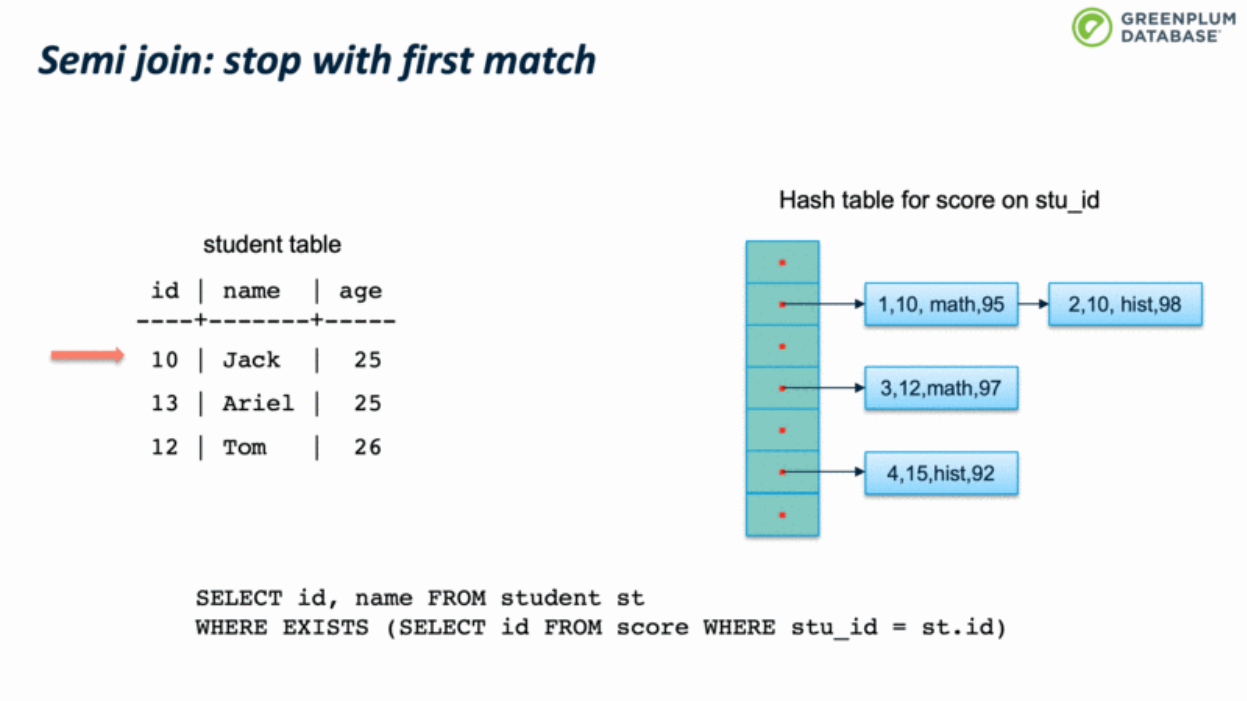
2.3.2 Full outer join

the full outer join obey the same pattern as inner join, the only difference is that during the scanning of outer table, if there is no matching tuple in hash table, the tuple is ouput and **null** is used to fill the column corresponding to inner table in the association result.

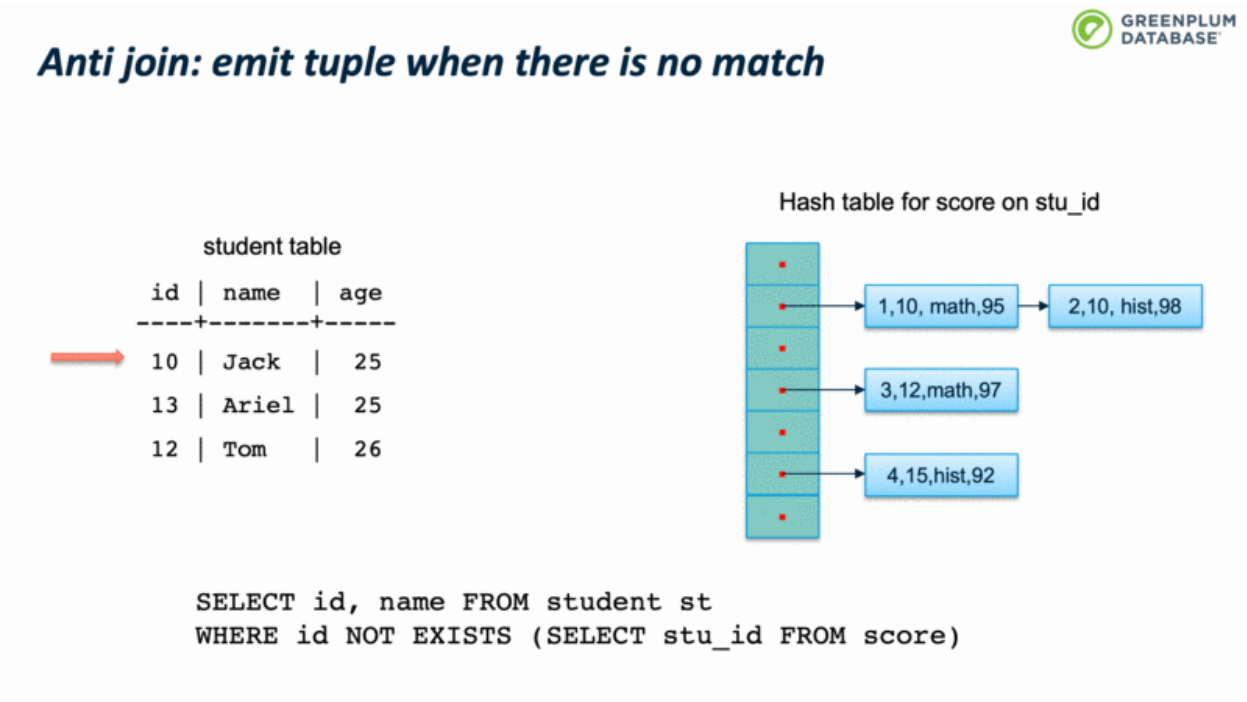


2.3.3 Semi join: stop with first match

semi join is usually used to implement exists



2.3.4 Anti join: emit tuple when there is no matc



---

so what about if inner table is too big to fit into memory?

- we design two methods to conduct joining
  - **Grace Hash join**
  - **Hybrid Hash join**

disadvangtages:

use hashing as a technique to partition relations, and it require sufficient buffers

- works only for equal-join like R.i = S.j
- susceptible to data skew

2.3.5 simple hash join:

https://en.wikipedia.org/wiki/Hash_join#Grace_hash_join

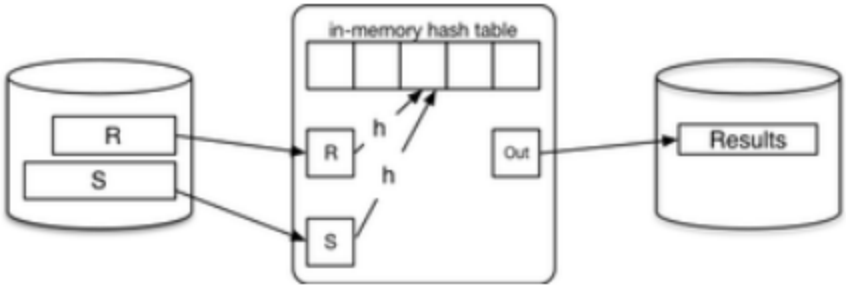
https://developpaper.com/a-comprehensive-interpretation-of-postgresql-and-greenplums-hash-join/

The first phase is usually called the **"build" phase**, while the second is called the **"probe" phase**. Similarly, the join relation on which the hash table is built is called the "build" input, whereas the other input is called the "probe" input.

the procedure of simple hash join:

1. For each tuple r in the build input R
    1. Add r to the in-memory hash table
    2. If the size of the hash table equals the maximum in-memory size:
        1. Scan the probe input S, and add matching join tuples to the output relation
        2. Reset the hash table, and continue scanning the build input R
2. Do a final scan of the probe input S and add the resulting join tuples to the output relation

Data flow:



2.3.6 Grace Hash join

basic methods:

1. Grace Hash join divides the inner table and the outer table into multiple paritions according to the association key
2. and then saves each partition to the disk
3. and then applies the hash join algorithm to each partition
4. each partition becomes a batch

and finnaly calculate bucket_no and batch_no:

advantages:

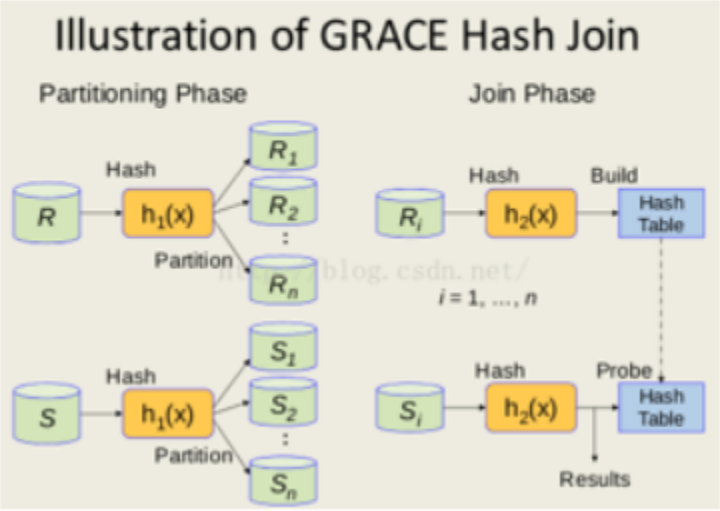
avoid rescanning the entire **S** relation by first partitioning both **R** and **S** via a hash function **H1,** and writing these partitions out to disk.

procesure:

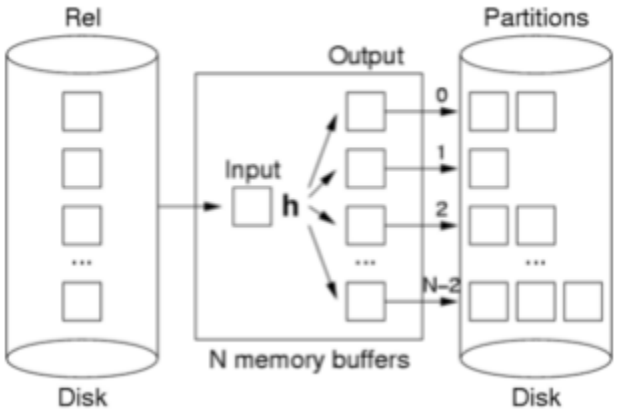
- after partition by hash function H1, the algorithm then loads pairs of partitions into memory, **builds a hash table** for the smaller partitioned relation
- and probes the other relation for matches with current hash table

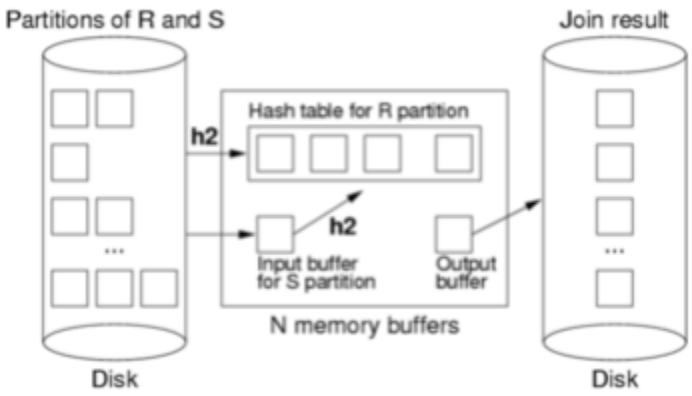
- then add the matched tuple into result table

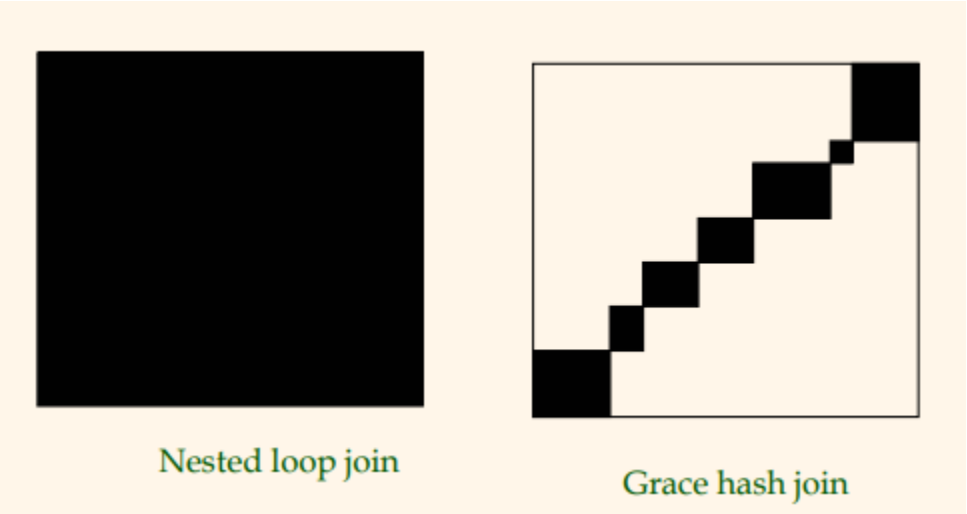
Grace Hash Join



Partition phase:







2.3.7 Hybrid Hash join

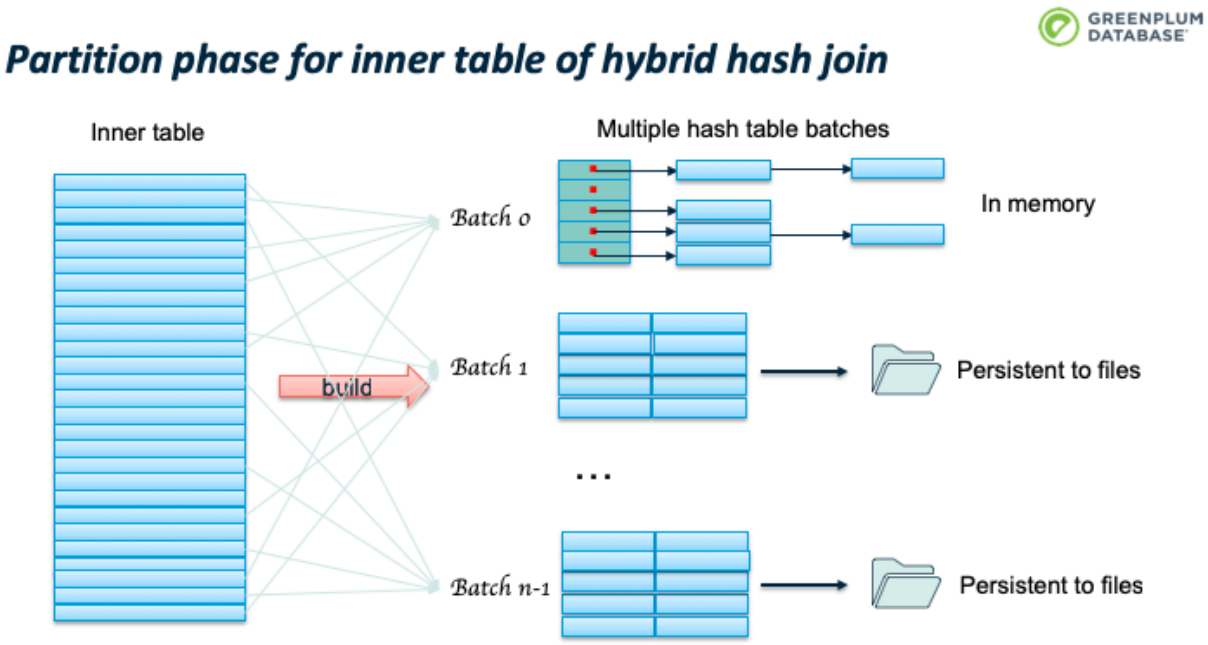
basic methods:

Hybrid Hash join imporve Grace hash join: **the first batch (batch 0) does not need to be written to disk,** so the disk I/O of the first batch can be avoided.

for inner table:

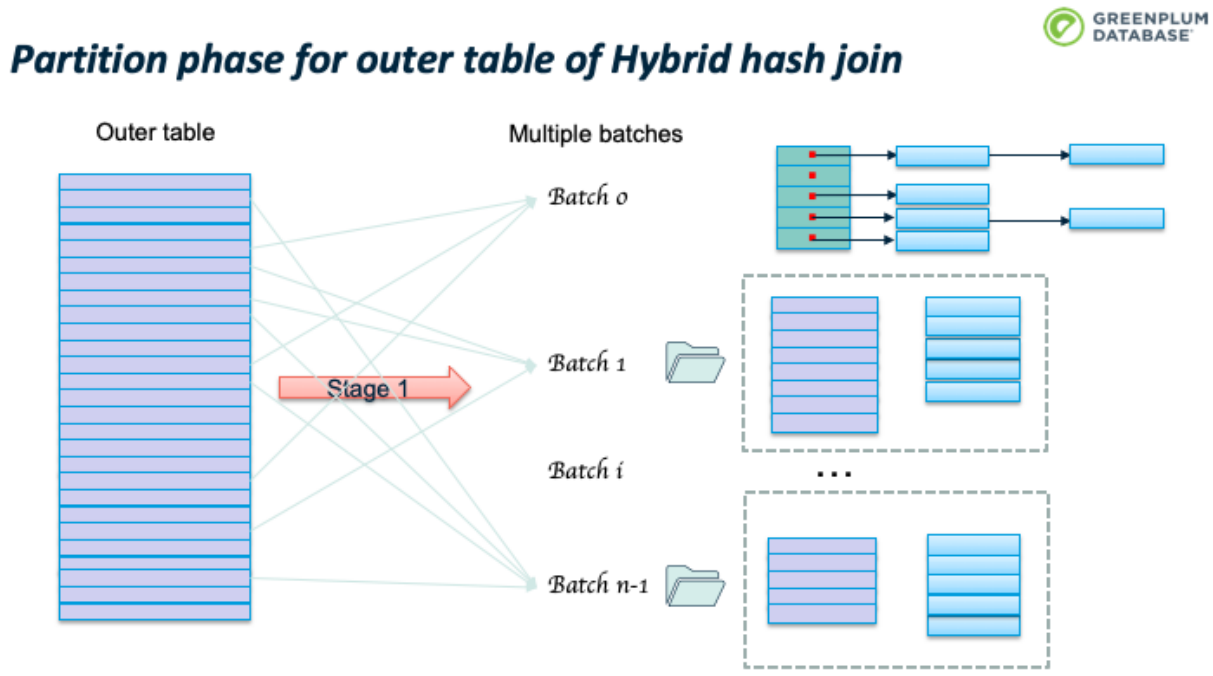
- Hybrid Hash join will parition inner table first and then calculate batch_no according to previous algorithm.

- if the tuple belongs to batch 0: it will be added to the hashtable in memory
- otherwise it will be written to the corresonding disk file of the batch

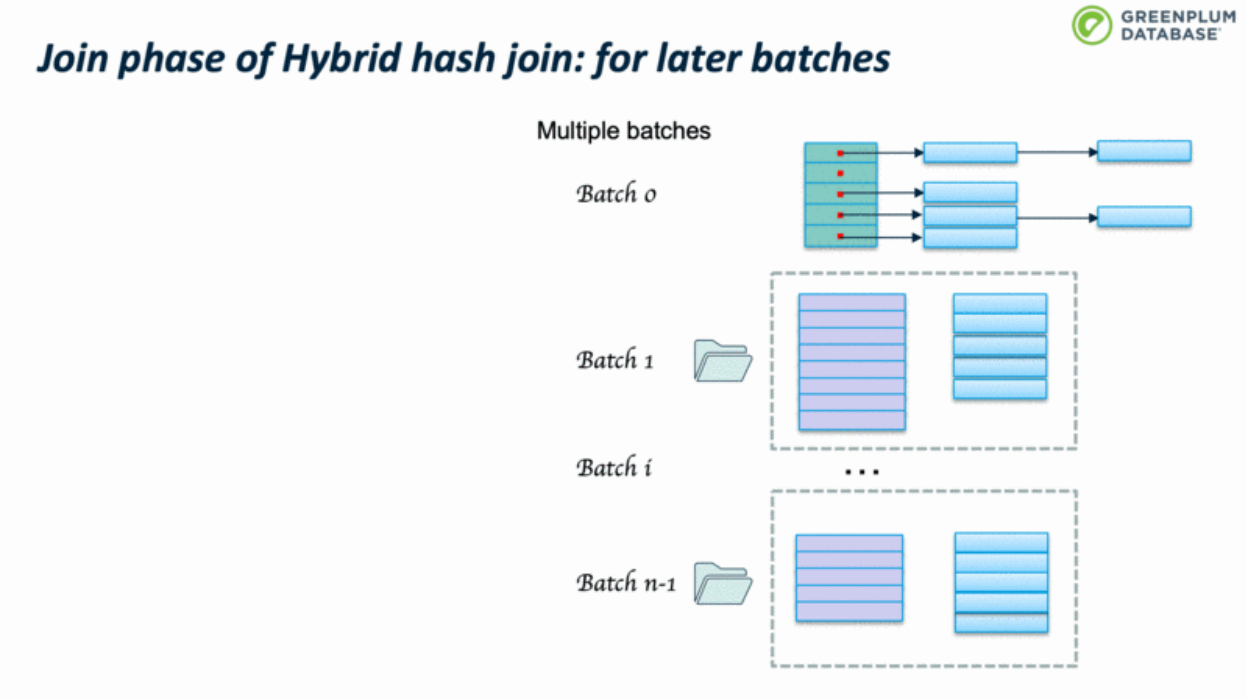


for outer table:

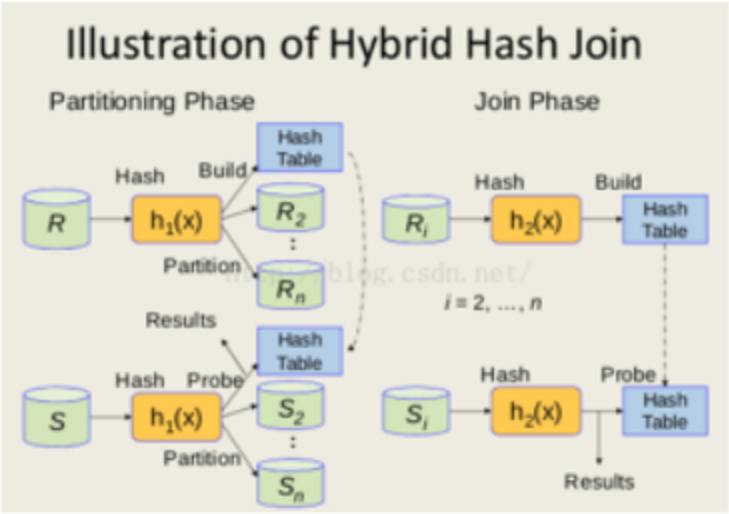
- If the tuple of the outer table belongs to batch 0, execute the hashjoin algorithm mentioned earlier:
  - judge whether there is an **inner tuple matching the outer tuple in the hashtable**.
    - If there is and all the where conditions are met, a matching is found and the result is output.
    - Otherwise, continue to the next tuple. If the outer tuple does not belong to batch 0, it is written to the disk file corresponding to the batch.



- At the end of the outer table scan, batch 0 is also processed.
  - Continue to process batch 1:
    - load the temporary data of inner table of batch 1 into memory, **build hashtable**, scan the temporary data of outer table of batch 1, and execute the probe operation skipped to.
    - After completing batch 1, continue to process batch 2 until all batches are completed.



Hybrid Hash Join



if you want to know more about Hybrid Hash join, please click the link below:

https://developpaper.com/a-comprehensive-interpretation-of-postgresql-and-greenplums-hash-join/

2.4 Pointer-based join

Conventional join algorithms set up R ↔ S connections via attribute values.

Join could be performed faster if direct connections already existed.

- *in OODBMSs, they generally already exist in the form of object references (oids)*
- *in RDBMSs, they could be introduced via extra rid attributes*

Such a modification to conventional RDBMS structure would be worthwhile:

- *if we know in advance what kind of joins will be required*
- *adding the extra rid attributes into tuples is feasible*

the advantage over value-based joins:

- rather than find S tuples via value-based lookup (e.g. hashing. index)
- we find S tuples by direct fetch with rid (much faster than per tuple)
- requires no assumption about sorted-ness of relations
- does not require large numbers of buffers

disadvantages:

- every fetch goes to a different page of S
- the join only works in “one direction” (from R to S)
- requires additional data for each different join type
- requires tuples to be larger → b_R is larger

The basic idea for pointer–based join is:

```
for each tuple r in relation R {
    for each rid associated with r {
        fetch tuple s from S via rid
        add (r,s) to result relation
    }
}
```

Often, each R tuple is associated with only one rid, so the inner loop is not needed.

2. General join conditions

above examples all used simple equaljoin like **Join[i=j](R,S)**

*For theta-join like **Join[i<j](R,S)***

- for index nested loop join, it need B+ trees index on inner relation
- for sort-merge join, it can be adapted, but is not very effective
- hash join is inapplicable

*for multi-equality(pmr) join like **Join[i=j ^ k=l](R,S)***

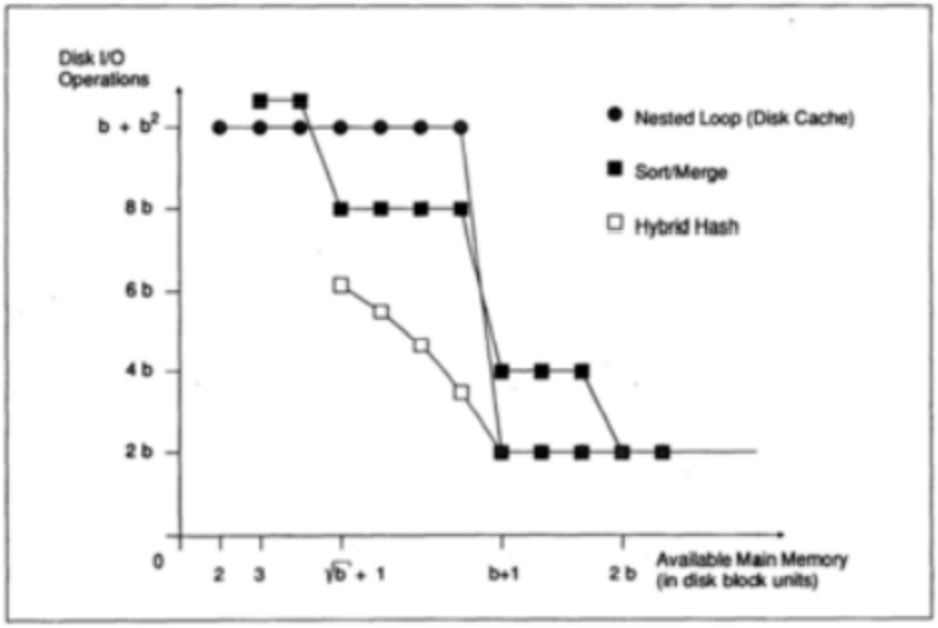
- index nested loop join: build index on all join attributes of inner relation
  - e.g. if S is inner, build index on (S.i, S.l)
- sort-merge join: sort both relations on combined join fields
  - e.g. sort R on (R.i, R.k), sort S on (S.j, S.l)
- hash-join
  - use multi-attribute hashing on combied join fields

3. Join Summary

No single join algorithm is superior in some overall sense.
Which algorithm is best for a given query depends on:

- sizes of relations being joined, size of buffer pool
- any indexing on relations, whether relations are sorted
- which attributes and operations are used in the query
- number of tuples in S matching each tuple in R

the comparison of join cost is shown below:


Disk I/O Operations
b + b²
8 b
6 b
4 b
2 b
0
2
3
√b + 1
b+1
2 b
Available Main Memory (in disk block units)
Nested Loop (Disk Cache)
Sort/Merge
Hybrid Hash