Query Evaluation and Parallelizing Queries

9 May 2010
Lecture 8

Topics for Today

- Index types
- Introduction to Query Evaluation
- Parallelizing Operations
  - Data Partitioning
  - Parallel Evaluation of Operators

Index Types

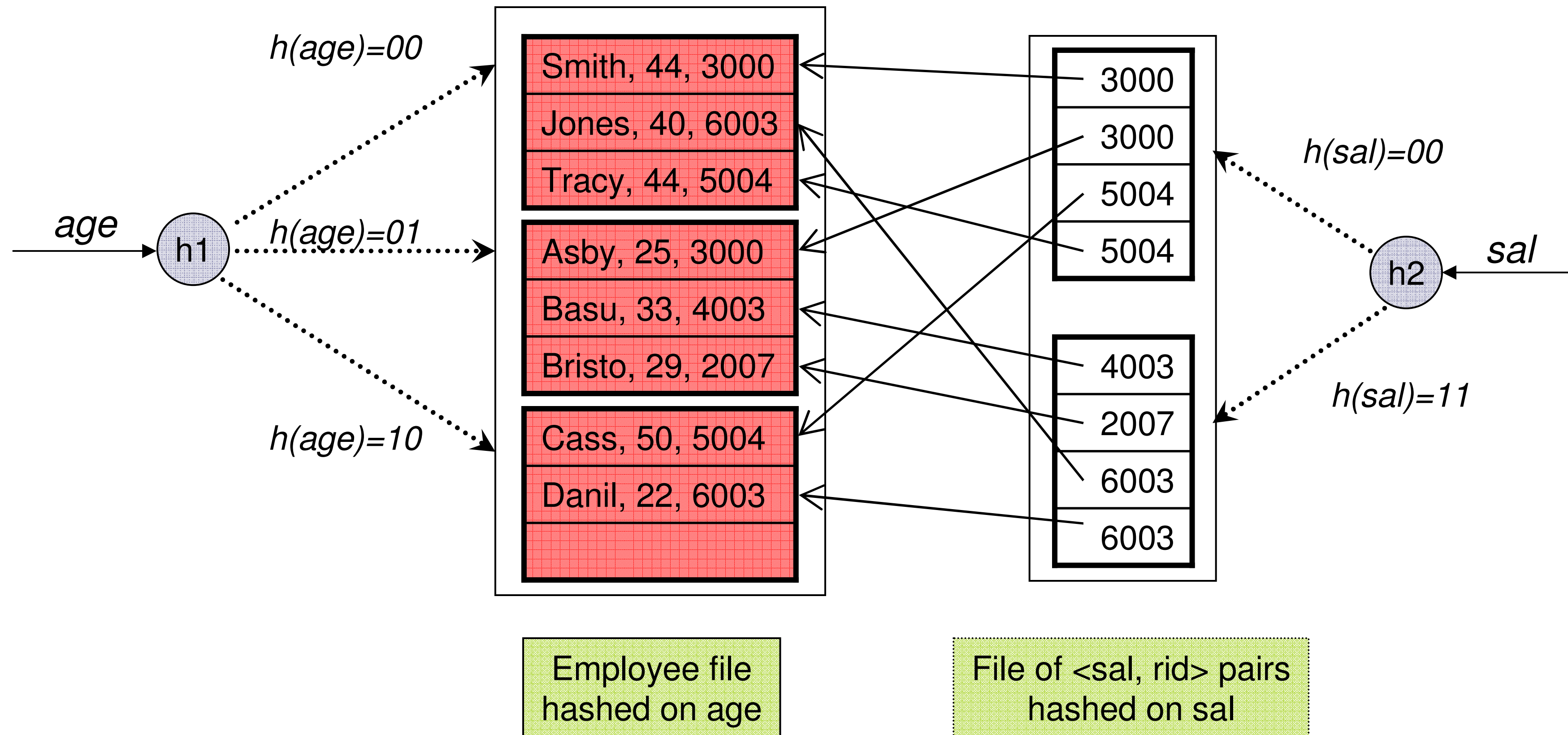
- Alternative (1) above is a *clustered index file*
  - Means the relation is sorted based on the index
  - Accessing records can be much faster this way (if selecting by the search key)

Indexes are commonly organized by either

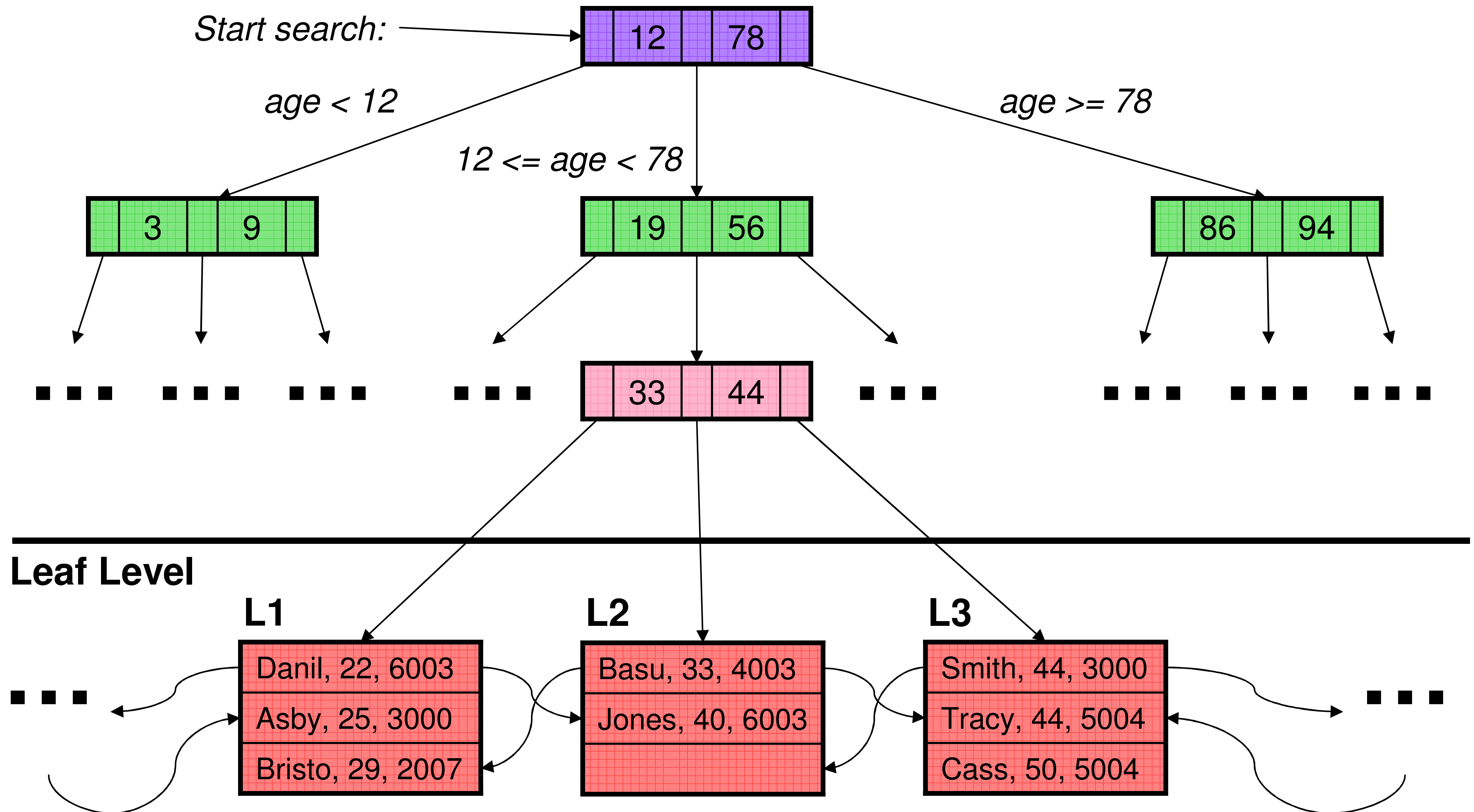
- Hashing
  - Hash the values for the search key to determine where the entry goes in the index
- Tree based indexing
  - Create a sorted search tree based on the values of the search key
    - Think of a binary search tree

These are virtual structures – in practice they are flat files with pointers to different offsets/pointers within the file

Index-Organized File Hashed on *age* with auxiliary index on *sal*



Tree-Structured Index



So Far

- Index types
- Introduction to Query Evaluation
- Parallelizing Operations
  - Data Partitioning
  - Parallel Evaluation of Operators

Overview of Query Evaluation

- We saw SQL and Relational Algebra last semester
  - The two are (intentionally) closely related
  - Algebra is the key to understanding how queries are evaluated

For our examples, consider the relations

Sailors(*sid* :integer, *string*:rating, *rating*:integer, *age*:real)

Reserves(*sid* :integer, *bid*:int, *day*:datetime, *rname*:string)

Relation	Bytes/Tuple	Tuples/Page	# Pages	# Tuples
Reserves	40	100	1,000	100,000
Sailors	50	80	500	40,000

The System Catalog

- The data in the database → Relations, Indexes
- Database also stores Metadata → data about the relations, indexes
  - Also stored in tables
  - Can use SQL to browse it!
- Metadata tables are called **catalog tables**
  - Or **data dictionary, system catalog, catalog**

The Attribute_Cat Relation

attr_name	***rel_name***	***type***	***position***
attr_name	Attribute_Cat	string	1
rel_name	Attribute_Cat	string	2
type	Attribute_Cat	string	3
position	Attribute_Cat	integer	4
sid	Sailors	integer	1
sname	Sailors	string	2
rating	Sailors	integer	3
age	Sailors	real	4
sid	Reserves	integer	1
bid	Reserves	integer	2
day	Reserves	datetime	3
rname	Reserves	string	4

The Catalog Stores…

System information

- Buffer pool size
- Page size

Per Table information

- Table name
- File name where it's stored
- Type of the file
- Attribute name and type for each field
- Index name of each index on the table
- Integrity constraints (keys, foreign keys)

Per Index information

- Index name and structure
- Search key attributes

Per View information

- View name
- View Definition

Other Table/Index Information (updated periodically)

- Cardinality of tables (*NTuples(R))*
- Cardinality of indexes (*INPages(I)*)
- Size in pages of tables (*NPages(R)*)
- Size in pages of indexes (*INPages(I)*)
  - For trees, the number of leaf nodes, range, height

Operator Evaluation

- Many ways to run operators – the best one for each query depends on many things
  - Table size, indexes, buffer pool

Three common techniques

- Indexing: For a WHERE or join, use an index to find the correct tuples
- Iteration: Scan the whole table (or index) to find the tuples that match
- Partitioning: Break down the table by a sort key, work on the parts
  - Can be done with hashing or sorting

Access Paths

An Access Path is a way (stratgey) of retrieving tuples

- Could be a file scan
- Could use an index which is matched with the selection condition

Say the selection condition is in Conjunctive Normal Form (CNF):

$$attr\ \mathbf{op}\ value \wedge attr\ \mathbf{op}\ value \wedge attr\ \mathbf{op}\ value \wedge \ldots$$

Each part is a **conjunct**

Using the indexes

attr **op** *value* $\wedge$ *attr* **op** *value* $\wedge$ *attr* **op** *value* $\wedge$ …

We can use an index if:

1. The index is a **hash index** and
   - The **op** are all = (why?)
   - The *attr*'s are identical (or a superset of) the attributes used in the hash index
2. The index is a **tree index** and
   - We can arrange the *attr*'s as a prefix of the index's search key
     - If the index is on (a,b,c) it can match (a), (a,b), (a,b,c)
     - It can't match (b), (b,c), (a,c)
   - The **op** can be anything

The index might not match all of the constraints

- Then we need some post-hoc filtering
- The ones that do match are the **primary conjuncts**

Example: $rname = \text{'Joe'} \wedge bid = 5 \wedge sid = 3$

Hash index on <rname, bid, sid>

- We can use the index (it matches the query)

If the query were $rname = \text{'Joe'} \wedge bid = 5$

- We'd be in trouble – which sid values do we use in the hash?

If the query were $rname = \text{'Joe'} \wedge bid = 5 \wedge sid = 3 \wedge day = 2010\text{-}01\text{-}01$

- We could use the index, but would need to filter the day afterwards

Tree index on <rname, bid, sid>

- We can match $rname = \text{'Joe'} \wedge bid = 5 \wedge sid = 3$

If the query were $rname = \text{'Joe'} \wedge bid = 5$

- We can match it too (it's a prefix)

If the query were $bid = 5 \wedge sid = 3$

- We can't since the first level check is for rname (it's not a prefix)

Example: $day<1990\text{-}01\text{-}01 \land bid=5 \land sid=3$

Hash index on <bid, sid>, tree on <day>

- We can use the hash index (except for the day)
- We can use the tree index (getting only the day)

In either case we must do follow-up processing to remove extra tuples

Selectivity of Access Paths

Normally there are a few access paths:

- Scan the whole relation
- Search the index, open the relevant table pages
- For a clustered index file, just scan the index (why?)

The less pages an access path uses, the better

We calculate the selectivity of an access path based on the **reduction factor** of each conjunct

– How many tuples does the conjuct remove?

Example: Hash index on <rname, bid, sid>, query is *rname='Joe'* $\wedge$ *bid=5* $\wedge$ *sid=3*

- Look in the catalog to see how big *NPages(Reserves)* is and how many distinct hash keys there are *NKeys(H)*
- Approximately $NPages(Reserves) \times \frac{1}{NKeys(H)}$ pages are matched

Algorithms for Relational Ops (1/3)

Selection – $\sigma_{attr\ \mathbf{op}\ value}$ *(R)*

- No indexes – scan the whole relation
- With indexes we get some help

Example: rname < 'C%' (all *rnames* that begin with 'A' or 'B')

- With a tree index on *rname*, about 10% will match
  - Means about 10,000 will match the condition (100 pages)
  - If the index is clustered – 100 page I/Os
  - If it's not clustered, worst case: 10,000 page I/Os (one per row)

Rule of thumb: if we need more than 5% of the tuples, just scan the whole file

Algorithms for Relational Ops (2/3)

Projection: $\pi_{field1,field2}(R)$

- Just scan the whole relation and write out the fields we want
- If there is a clustered index on <field1, field2>, just scan it

- With DISTINCT, it's much more complicated
  - We first might sort the values, then remove (the adjacent) duplicates
  - Requires two passes:
    - First pass to scan the table and write out just the <field1, field2> pairs
    - Second pass to just output one copy of each pair (maybe by sorting
  - There may be a disk write/read pass in the middle if it won't all fit in memory
- With a good index, we can do better
  - Do duplicate elimination in the index (just take one copy per <field1,field2> entry)

Algorithms for Relational Ops (3/3)

Joins are very common (and very expensive)

- Index Nested Loop Join
- Sort-Merge Join
- Hash Join

Index Nested Loop Join

Example: Join of Reserves and Sailors on *sid*

- We have an index on Sailors on *sid*

- For each entry in Reserves, find the appropriate row in Sailors by *sid*
  - It's a foreign key to Sailors, so there must be 1 matching row
- With a hash index, it takes about 1.2 page I/Os to get the right hash index page
- One more page I/O to get the right page of Sailors

Cost:

- 1,000 pages of Reserves (1,000 I/Os)
- For each of the 100,000 rows in Reserves, (1.2 + 1) page I/Os
- 1,000 + 100,000 x (2.2) = 221,000 I/Os

Sort-Merge Join

- More complicated, but cheaper in general
  - Sort both tables on the search key
  - Merge them together to find matches

Example: Join of Reserves and Sailors on *sid*

- We don't need indexes here.

Assume we can sort each table in two passes

- Perhaps one disk read/write in the middle if there isn't enough memory. So two read/writes.

- We sort Reserves in: 2 x 2 x 1,000 = 4,000 I/Os
- We sort Sailors in: 2 x 2 x 500 = 2,000 I/Os
- Merge the two: 1,000 + 500 = 1,500 I/Os
- Total: 7,500 I/Os
  - **Much cheaper** than index nested loop join.

Sort-Merge Join Code

```
proc smjoin(R, S, 'R_i = S_j')

if R not sorted on attribute i, sort it;
if S not sorted on attribute j, sort it;

T_r = first tuple in R;                  // ranges over R
T_s = first tuple in S;                  // ranges over S
G_s = first tuple in S;                  // start of current S-partition
while T_r ≠ eof and G_s ≠ eof do {

    while T_r_i < G_s_j do
        T_r = next tuple in R after T_r;                  // continue scan of R

    while T_r_i > G_s_j do
        G_s = next tuple in S after G_s                   // continue scan of S

    T_s = G_s;                     // Needed in case T_r_i ≠ G_s_j

    while T_r_i == G_s_j do {                // process current R partition
        T_s = G_s;                                          // reset S partition scan
        while T_s_j == T_r_i do {                          // process current R tuple
            add <T_r, T_s> to result;                       // output joined tuples
            T_s = next tuple in S after T_s;}               // advance S partition scan
        T_r = next tuple in R after T_r;                    // advance scan of R
        }                                                   // done with current R partition
    G_s = T_s;                                   // initialize search for next S partition
}
```

Hash Join: Main Idea

- We divide up R and S into buckets based on the hash and compare the members of the buckets
  - Reduces the number of tuples to search through (probe).

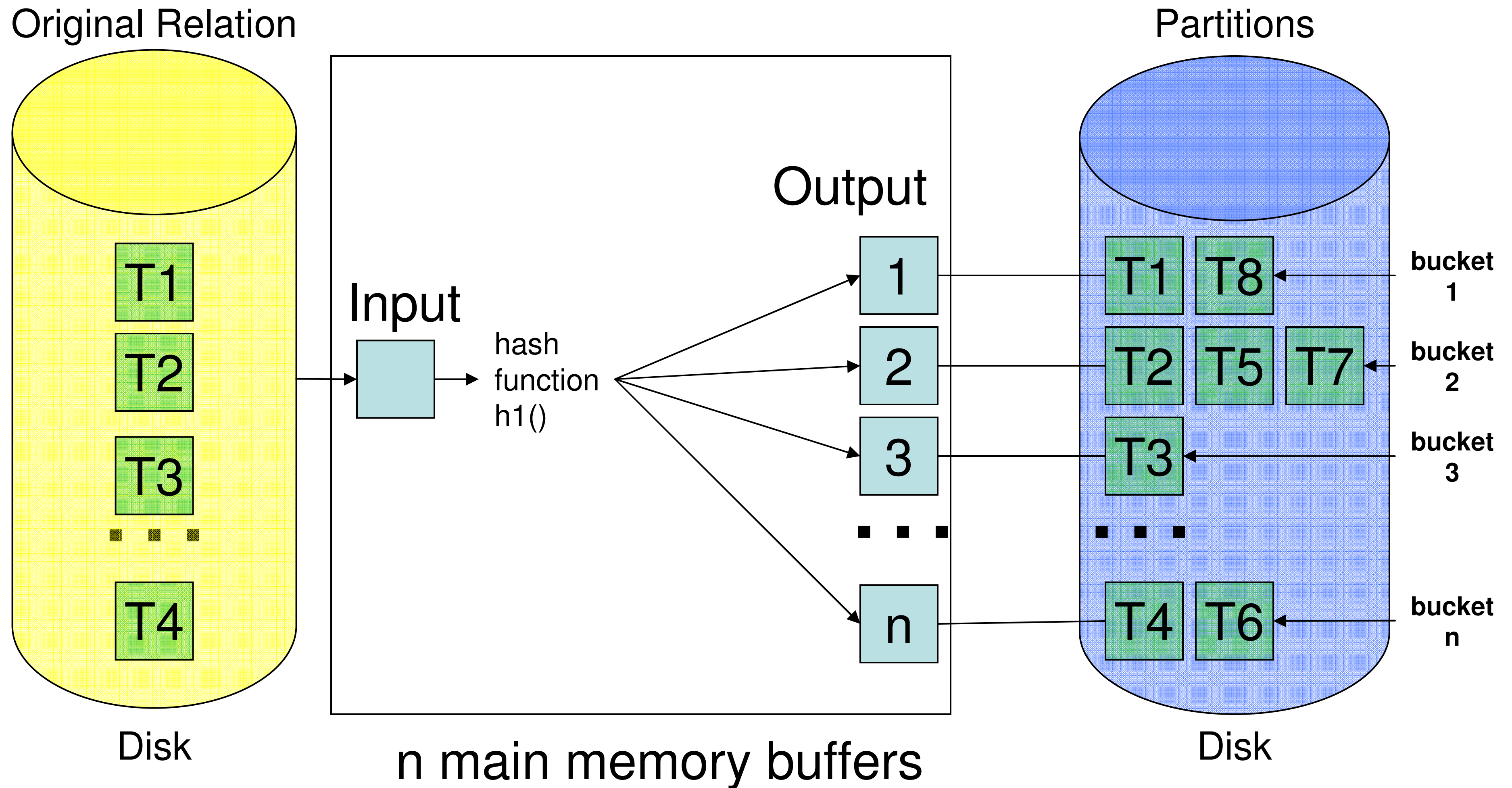
- We can improve the probe by using a second hash on each bucket
  - Saves comparisons wihtin the bucket

- The result is an algorithm in about 3(M+N) steps

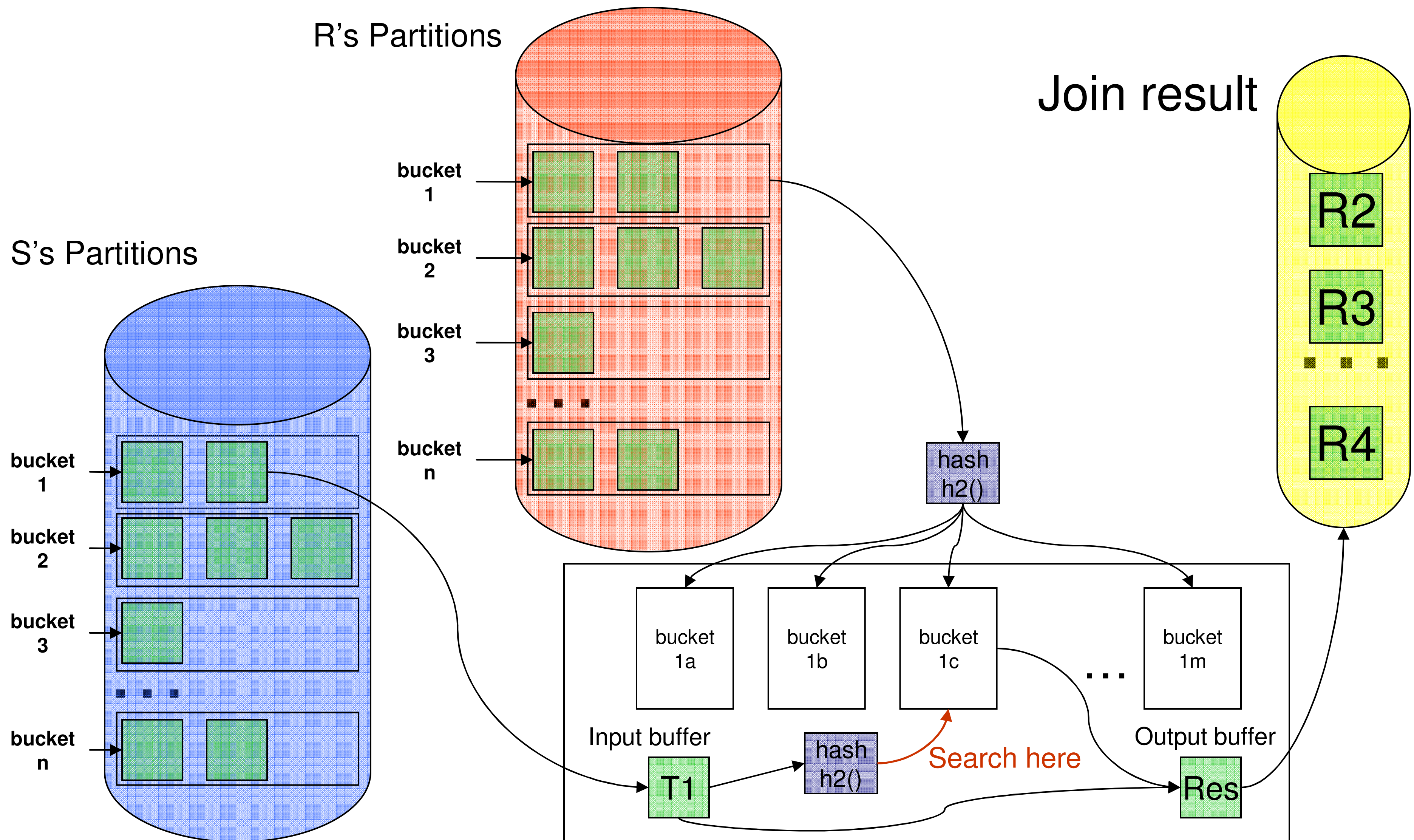
Two main steps:

- Partition Reserves and Sailors by a hash function h1()
- For each tuple in each bucket b_i of Reserves, find the right Sailor by checking bucket b_i of Sailors

Hash Join: Partition



Hash Join: Probe



Hash Join Code

```
// Partition R into k partitions
foreach tuple r1 in R do
    read r1 and add it to buffer page h(r1);   // flushed to disk as page fills

// Partition S into k partitions
foreach tuple s1 in S do
    read s1 and add it to buffer page h(s1);  // flushed to disk as page fills

// Probing phase
for l = 1, …, k do
{
    // Build in-memory hash table for Rl, using h2
    foreach tuple r1 in partition Rl do
            read r1 and insert into hash table using h2(r1);

    // Scan Sl and probe for matching Rl tuples
    foreach tuple s1 in partition Sl do
     {
            read s1 and probe table using h2(s1);
            for each matching R tuple r2, output <r1, s1> };

    clear hash table to prepare for next partition;
}
```

So Far

- Index types
- Introduction to Query Evaluation
- Parallelizing Operations
  - Data Partitioning
  - Parallel Evaluation of Operators

Parallelizing Individual Operations

- Instead of parallelizing a query, we could parallelize individual operations of a query plan
  - This would make more sense if the operations are very expensive by themselves
- The operations we consider:
  - Scanning
  - Sorting
  - Joins
- We need to:
  - Divide the data up fairly a priori
  - Limit how much data is shipped around
- We need good *data partitioning*

Data Partitioning

- How do we partition?
- A few options:
  - Round Robin
    - I % n
  - Hashing
    - H(i)
  - Range
    - X1 <= I <= X2

Question: What are the plusses and minuses of each?

- Skew?
- Adding (many) new tuples?
- Removing (many) tuples?

Scanning

- Scanning a relation reads it from disk and selects records from it
  - Corresponds to a simple WHERE
  - Can be naturally divided up and done in parallel
- In shared nothing: each processor reads the pages it has locally
  - If shared disk or shared memory, division is done differently
- Matching tuples should be assembled and shipped to the later operations (split/merge)

Sorting

- How do we do distributed sorting?
- One idea:
  - Each processor sorts its local tuples
  - They are sent to a “main” processor which assembles the sorted parts
- Drawbacks:
  - Merge adds another O(n)
  - Skew

Sorting: Range Partitioning

- A better option: Range Partitioning
  - Each processor sorts the tuples on some range [n, m]
  - Someone decides who gets what range
  - The outcome of the ranges selected is called a **splitting vector**

- In Shared Nothing: Each processor must scan the local tuples to find which match
  - A version of “Bucket Sort” O(n)
- The tuples are then split and shipped to the processor which gets the range
- When each processor is done, the main one visits them in order to collect the results

Sorting: Range Partitioning

- Problem: Skew

- How can we reduce skew?
  - Take a sample of the tuples
  - Sort the sample
  - Derive the ranges from the sample

Conclusion

- Index types
- Introduction to Query Evaluation
- Parallelizing Operations
  - Data Partitioning
  - Parallel Evaluation of Operators