Hash Join Algorithms

Instructor: Sharma Chakravarthy
sharma@cse.uta.edu
The University of Texas @ Arlington

Hash Join Algorithms

- In-memory Hash join
  - When you can hold one of the 2 relations in memory
- Simple hash-based join
  - Efficient when memory is large
  - Too many I/O operations when memory is small
- GRACE hash-based join
  - Separate partitioning and join phases
  - Easy to parallelize
  - Avoids bucket overflow
- Hybrid hash-based join
  - Combines Basic and Grace hash-join
  - Better memory usage

In-memory hash-join Algorithm

This is used for the map-side join. We are using the reduce-side join for project 3

<table>
<thead>
<tr>
<th>Emp Dept</th>
<th>Smith</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Chang</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Miller</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proj Dept</th>
<th>P1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>P3</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>P5</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>P6</td>
<td>12</td>
</tr>
</tbody>
</table>

R join S (output)

Build phase
- Read R once and construct in-memory hash table
- I/Os: M (# of pages of R)

Probe phase
- Read all of S and search for matching tuples
- I/Os: N (# of pages of S)

Total Cost: O(M+N) if we have enough memory to hold one relation in memory

How do you choose the relation for Build?
How do you choose the relation for probe?
What if we do not have enough memory?
Simple Hash Join algorithm

- Use whatever memory is available as buckets of one in-memory hash table and write the rest to disk
- Repeat this process until the entire join is performed
- Disadvantages: introduces too many I/O operations when the memory is not too large!
- Cost: $O(b(M+N))$ where $b$ is the number of buckets (range of hash function!)

Complexity

- Let size of $R$ be $M$ pages; size of $S$ be $N$ pages
- Let the hash function divide them uniformly into $b$ buckets
- If you have $b$ hash buckets for the simple hash join algorithms, then you need $b^2 (M+N)$ I/O’s (Try to derive this expression!)
- You read and write each relation $b$ times!
- Typically, $b$ ranges from 10 to 1024 or even larger
- How can we reduce it further?
- How many buffer pages do we need

Simple Hash Join Algorithm

```c
/* $h$ is the hash function; $h[0...n]$ is the range of hash function */
/* $R[0...n]$ and $S[0...n]$ are buckets */

i=0; do
  for (each tuple $r$ in $R$)
    if ($h(r)$ in current_range)
      insert $r$ into the in-memory hash table;
    else write $r$ into $R_temp$;
  for (each tuple $s$ in $S$)
    if ($h(s)$ is in current_range)
      use $s$ to probe the in-memory hash table;
      If (any match is found) output the matching tuples;
      else write $s$ into $S_temp$;
    $R = R_temp$;
    $S = S_temp$;
    current_range = $h[i+1]$;
  While ($R_temp$ is not empty and $S_temp$ is not empty);
```

GRACE Hash Join Algorithm

Partitioning phase

- Apply a hash function $h(x)$ to the join attributes of the tuples in both $R$ and $S$. Assume $b$ buckets
- According to the hash value, each tuple is put into a corresponding bucket. Write these buckets to disk as separate files.

Joining phase:

- Use the basic hash-join algorithm
- Get one partition of $R$ and the corresponding partition of $S$ and apply the basic hash algorithm using a different hash function. Why?
**Hash-Join**

- Partition both relations using hash function $h$: R tuples in partition i will only match S tuples in partition i.

- Read in a partition of R, hash it using $h_2 (\not\equiv h_1)$. Scan matching partition of S, search for matches.

**Grace Hash Join**

- Range of $H(x)$ is $1, \ldots, N$
- $R_1, \ldots, R_n$ and $S_1, \ldots, S_n$ are disjoint subsets of R and S
- R is the Union ($R_1, \ldots, R_n$) and S is the union($S_1, \ldots, S_n$)
- We need to join only $R_i$ with $S_i$. Why?
- The efficiency comes from the reduction in work load which is illustrated below.

**Grace Hash Join Algorithm**

```c
/* h[1..n]: range of hash functions; R[1..n] and S[1..n] are buckets */
apply hash function to the join attributes of r;
put r into the appropriate bucket $R[i]$
for (each tuple s in $S[i]$
apply hash function to the join attributes of s;
put r into the appropriate bucket $S[i]$
for (i=1; i <= n; i++)
build the hash table for $R[i]$. /* using a different hash function $h_2$*/
for (each tuple s in $S[i]$)
apply the hash function $h_2$ to the join attributes of s;
use s to probe the hash table;
output any matches to the result relation;}
```

**Workload in hash join**

- Nested loop join
- Grace hash join
Observations on Hash-Join

- Given B buffer pages, the maximum # of partitions is B-1
- Assuming that partitions are of equal size, the size of each R partition is $M/(B-1)$
- The number of pages in the (in-memory) hash table built during the building phase is $fM/(B-1)$ where f is the fudge factor
- During the probing phase, in addition to the hash table for the R partition, we require a buffer page for scanning the S partition, and an output buffer.
- Therefore, we need $B > fM/(B-1) + 2$
- Approximately, we need $B > \sqrt{M}$ for the hash join algorithm to perform well.

Cost of Grace Hash Join

- In partitioning phase,
  - read+write both relations; that is, $2(M+N)$.
  - In matching phase, read both relations; that is, $M+N$ I/Os.
  - Total: $3(M+N)$ linear instead of log or quadratic!
- In our running example, this is a total of 4500 I/Os.

Observations on Hash-Join

If we build an in-memory hash table to speed up the matching of tuples, a little more memory is needed.
- If the hash function does not partition uniformly, one or more R partitions may not fit in memory. Can apply hash-join technique recursively to do the join of this R-partition with corresponding S-partition.

Sort-merge join vs. Hash Join

- If partitions in hash join are not uniformly sized, hash join could cost more
- If the available number of buffers falls between $\sqrt{N}$ and $\sqrt{S}$, hash join costs less than sort-merge, since we need enough memory to hold partitions of the smaller relation. Sort-merge buffer needs are based on the larger relation.
- Hash Join is superior on this count if relation sizes differ greatly. Also, Hash Join shown to be highly parallelizable.
- Sort-Merge less sensitive to data skew; result is sorted.
General Join Conditions

- Equivalences over several attributes (e.g., \( R.sid = S.sid \) AND \( R.rname = S.sname \)):
  - For Index NL, build index on \(<sid, sname>\) (if S is inner); or use existing indexes on sid or sname.
  - For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.

- Inequality conditions (e.g., \( R.rname < S.sname \)):
  - For Index NL, need (clustered!) B+ tree index.
  - Range probes on inner; # matches likely to be much higher than for equality joins.
  - Hash Join, Sort Merge Join not applicable.
  - Block NL quite likely to be the best join method here.

Hybrid Hash Join Algorithm

```c
/* H[0..n] is the range of hash; R[0..n] and S[0..n] are buckets */
for (each tuple in R){
    if (hash value of r is in H[0])
        insert r into the in-memory hash table;
    else put r into the appropriate bucket R[i];
}
for (each tuple s in S){
    if (hash value of s is in H[0])
        use s to probe the hash table;
        put any matching tuples into the result relation;
    else put s into appropriate bucket S[i];
    for (i=1; i<=n; i++)
        build the hash table from R[i];
        for (each tuple s in S[i])
            apply hash function to the join attributes of s;
            use s to probe the hash table;
            output any matches to the result relation;
}
```

Pointer Based Joins

1. Links represent a limited form of pre-computed results (OO has rekindled this concept)
2. Modeled as TID joins in Ingres

   1. Tuples of R has a pointer to an embedded S tuple
      - Scan R and retrieve S
      - Sort R on the pointers (according to the disk address they point to) and then retrieve all S items in one elevator pass over the disk, reading S page at most once

   2. Pointer based Joins (contd)

   - Hybrid-hash join: Partitions relation R on the pointer values ensuring that R tuples with S pointers to the same page are bought together, and then retrieve S pages and tuples
   - Direction of pointers fix the role of relations! (usually, the smaller relation is used for the build phase)
   - Maintenance effort is to be taken into account as well.
Alternative Join methods

- S is 10 times R, Memory size 100Kb
- Cluster Size is 8Kb, Merge fan-in and partitioning fan-out are 10, # of R records/cluster is 20

Conclusions

- Nested Loop joins are unsuitable for medium size and large relations
- sort based join is not as fast as hash join (merge levels are determined individually for each file, but only the smaller relation determines partition depth)
- The step is because additional partitioning or merge levels become necessary at that point

Aggregation and Duplicate Removal

- Surprisingly, a lot in common
- In one, duplicates are discarded whereas in the other, some computation (e.g., COUNT, SUM, AVG) is performed before discarding the tuple
“Duality” of Sorting and Hashing

- Both do approx the same amount of I/O
- Mirror-images in terms of sequentiality of phase 2
- Sort-based algorithms
  - Large data sets are divided into subsets using physical rule (into chunks as large as memory)
- Hash-based algorithms
  - Large data sets are divided into subsets using a logical rule (hash values)
- Handling large inputs
  - Multi-pass sort vs. recursive partitioning hash
- It actually goes deeper than this