Performance evaluation and comparative study between sort-merge join and hash join

Sambit Kumar Mishra
Associate Professor
Department of Computer Sc.&Engg.

Prof.(Dr.) Srikanta Pattnaik
Professor , S.O.A. University , Bhubaneswar

Abstract: Join is the basic key operation in relational databases which facilitates the combination of two relations based on a common key. It is usually an expensive operation and efficient implementation to improve the performance of many database queries. Join is also a memory intensive operation and consequently is directly affected by the performance of memory subsystem. As compared to the join algorithms, e.g. sort-merge join and hash join, Sort merge join is highly dependent during implementation. Also it is understood that Quicksort is one of the fastest algorithms in practical application, but still it is very difficult to map efficiently to the single instruction stream multiple data stream architecture. With the increasing capacity of main memory, as a large number of database tables reside completely in main memory and the typical databases consist of tables with numerous columns, with each column having different width (in bytes), the user queries performing join on more than two such tables are decomposed into pair wise table join operations. For main-memory databases, the number of entries in a table is typically less than the range of 32-bit numbers e.g. 2^32. As the system performance improves at a much faster rate than memory subsystem performance, memory access latency cannot be improved.

In this paper the primary importance is given to create hash table with keys of the inner relation, and reorder its tuples. This partitioning phase is followed by the actual join phase, by iterating through the tuples in the relation, and for each key – searching for matching keys in the hash table and appending the matching tuples from the relation to the output table.

Keywords : Sort-merge join , hash join , memory access latency, matching keys, tuple.

Introduction

The basic idea behind a hash join implementation is to create a hash table with keys of the inner relation (S), and reorder its tuples. This partitioning phase is followed by the actual join phase, by iterating through the tuples in upper relation R, and for each key, searching for matching keys in the hash table, and appending the matching tuples from S to the output table. The expected search cost of hash tables makes a practical option for database implementations. In order to avoid wasteful comparisons during join, it is imperative to avoid collisions during hash lookups.

Theoretically, this requires a hash table with size around two times larger than the number of input elements, or the cardinality of the input keys. With large table sizes, it is possible to access regions of memory whose page entries are not cached in the buffer thereby incurring a large increase in latency. The general advice is to use sort-merge join in the presence of significant intrinsic skew, because bucket overflow in hash join is so expensive. However, we are aware of no paper either on the impact of intrinsic skew on the performance of (centralized) sort-merge join, nor on ways to deal with such skew. In fact, the classical sort-merge algorithm presented in many application shows incorrect results in the presence of intrinsic skew. While the sort-merge implementations in commercial systems yield correct results.

Cost associated with sort-merge join:

The actual data set from the University Information System is taken to the consideration. Specifically, the Incumbents table is used that includes information on job assignments for University employees.

Query 1:
Select il.SSN, il.pay_hourly_rate, i2.SSN
from incumbents il, incumbents i2
where il.SSN = i2.SSN
and il.PCN = i2.PCN
and il.start_date = i2.start_date

Query 2:
select il.SSN, il.pay_hourly_rate, i2.SSN
from incumbents il, incumbents i2
where il.pay_hourly_rate = i2.pay_hourly_rate
and ((il.start_date >= i2.start_date
and il.end_date <= i2.end_date)
or (il.start_date > i2.start_date
and il.end_date > i2.end_date
and il.start_date < i2.end_date)
or (il.start_date < i2.start_date
and il.end_date < i2.end_date
and i2.start_date < il.end_date)
or (il.start_date <= i2.start_date
and il.end_date >= i2.end_date))

Basic approaches for selection:

First approach: (1) Find the most selective access path, retrieve tuples using it.
(2) Apply any remaining terms that don’t match the index on the fly.
Most selective access path: An index or file scan that we estimate will require the fewest page I/Os. Terms that match this index reduce the number of tuples retrieved; other terms are used to discard some retrieved tuples, but do not affect number of tuples/pages fetched.

Considering day<20/03/12 AND bid=5 AND sid=3.

A B+ tree index on day can be used; then, bid=5 and sid=3 must be checked for each retrieved tuple.
A hash index on <bid, sid> could be used; day<20/03/12 must then be checked.

Second approach: If we have 2 or more matching indexes that use alternatives for data entries. Get sets of row identifiers of data records using each matching index.
Then intersect these sets of row identifiers.
Retrieve the records and apply any remaining terms.
Consider day<20/03/12 AND bid=5 AND sid=3. If we have a B+ tree index on day and an index on sid, both using Alternative (we can retrieve rids of records satisfying day<8/9/94 using the first, row identifiers of records satisfying sid=3 using the second, intersect these row identifiers, retrieve records and check bid=5.

Review of Literature

Wei Li et.al.[1] have focused on the partition skew which is of concern in hash-based join. In the first step of hash join, some buckets may contain more tuples than other buckets due to an interaction between the distribution of attribute values and the hashing function itself. When this disparity becomes large, the bucket no longer fits in main memory and hash-based join degrades into nested-loop join. Partition skew originates in the hash function chosen by the optimizer.
Christopher B et.al[2] have discussed in their paper that the distribution of the input data values can have a dramatic impact on the performance of both sort- and hash-based algorithms. The term skew involves several related but different effects.
Margaret et.al[3] have focused on join algebraic operator that takes two input relations, of arity m and n, and produces a single resulting relation. A wide variety of joins have been defined, including equijoins, natural joins, semi-leoins, outer joins and composition.
Christopher B et.al[4] have discussed in their paper that the intrinsic skew occurs when attributes are not distributed uniformly. Skew impacts the performance of both hash- and sort based joins. Sort-merge join works best when the join attributes are the primary key of both relations.

Problem formulation

Initially the focus is given to Equi-join queries on two tables with each tuple consisting of two fields (key, rid), each being a 32-bit number. rid indicates the record identifier. The key may be a single attribute or composite attribute.

For example

Q: SELECT ... FROM R, S WHERE R.key = S.key
The relevant tuples may be retrieved after proper execution of the query.
In addition, the tuples completely reside in main memory.
We may use the following notation:
NR : Number of tuples in outer relation (R).
NS : Number of tuples in inner relation (S).

The basic idea behind a hash join implementation is to create a hash table with keys of the inner relation (S), and reorder its tuples. This partitioning phase is followed by the actual join phase, by iterating through the tuples in R, and for each key, searching for matching keys in the hash table, and appending the matching tuples from S to the output table. The expected search cost of hash tables makes the option for database implementations. In order to avoid wasteful comparisons during join, it is essential to avoid collisions during hash lookups. Theoretically, this requires a hash table with size around two times larger than the number of input elements, or the cardinality of the input keys. To overcome the dependence on memory latency and memory bandwidth, it is required to partition the input table into smaller disjoint tables, such that each sub table can reside in cache. This is followed by the actual join between the corresponding sub tables in the two relations.

A join dependency is a constraint on the set of legal relations over a database scheme. A table T is subject to a join dependency T can always be recreated by joining multiple tables each having a subset of the attributes of T. If one of the tables in the join has all the attributes of the table T, the join dependency is called trivial.

Usually a hash table a data structure that uses a hash function to map identifying values, known as keys to their associated values . Thus, a hash table implements an associative array. The hash function is used to transform the key into the index of an array element where the corresponding value is to be sought. Usually, the hash function maps each possible key to a unique slot index, but the new entries will not be added to the table after it is created. Generally the hash table algorithms calculate an index from the data item's key and use this index to place the data into the array. We may calculate the index with the formula as index = f(key, arrayLength) where f is the hash function. A good hash function algorithm is essential for good hash table performance. The basic requirement is that the function should provide a uniform distribution of hash values. A non-uniform distribution increases the number of collisions, and the cost of resolving them. Uniformity is sometimes difficult to ensure by design, but may be evaluated empirically using statistical tests. The distribution needs to be uniform only for table sizes that occur in the application. In particular, if one uses dynamic resizing with exact doubling and halving of the table sizes, the hash function needs to be uniform. The hash function avoids clustering, where the mapping of two or more keys to consecutive slots is achieved. Such clustering may cause the lookup cost to skyrocket, even if the load factor is low and collisions are infrequent. If all keys are known ahead of time, a perfect hash function is used to create a perfect hash table that has no collisions. If minimal perfect hashing is used, every location in the hash table can be used as well.

**Algorithm**

Step1 : for (each tuple r in R relation) apply hash function to the join attributes of r;
Step2 : put r into the appropriate bucket R;

Step 3 : for (each tuple s in S relation) apply hash function to the join attributes of s;
step 4 : put r into the appropriate bucket S;

Step 5 : 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;

The algorithm may also be viewed as

do
    {for (each tuple r in R) 
    {if (h(r) in current_range) 
        insert r into hash-table; 
        else 
        write r into R_temp; 
    }
    for (each tuple s in S) 
    {if (h(s) in current_range) 
    {use s to probe the 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);

Experimental Result and Analysis:

Total relation = 100
Relation_R = 20
Relation_S = 20
Size of query = 50
Crossover point (CP) = 7
Probability of crossover (Pc) = 0.05
Probability of mutation (Pm) = 0.02

<table>
<thead>
<tr>
<th>Sl.No.</th>
<th>H(r)</th>
<th>H(s)</th>
<th>Weight</th>
<th>Tuple_join cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>0.218</td>
<td>0.029</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>5</td>
<td>0.372</td>
<td>0.034</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>0.227</td>
<td>0.024</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>7</td>
<td>0.454</td>
<td>0.047</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>7</td>
<td>0.237</td>
<td>0.0297</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>9</td>
<td>0.393</td>
<td>0.037</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>9</td>
<td>0.496</td>
<td>0.049</td>
</tr>
</tbody>
</table>

The hash function is applied to both relation R and S. The value of H(r) and the value of H(s) are well proportionate. While the value of H(r) and H(s) are in the same range it reflects to the weight assigned to the tuples of relation R as well as relation S. The cost of join predicates is also proportional to the sizes of queries in the relation R as well as in the relation S.

Conclusion

As the hash function avoids clustering, if all keys are known ahead of time, a perfect hash function may be used to create a perfect hash table that has no collisions. If minimal perfect hashing is used, every location in the hash table may be reused.

Reference


