Interleave Join

Thomas Stovall [thomas.stovall@walmart.com (mailto:thomas.stovall@walmart.com)]

Richard Ulrich [richard.ulrich@walmart.com (mailto:richard.ulrich@walmart.com)]

Abstract

A flexible algorithm for computing database-like joins in near linear time is described. The algorithm employs memory coalescing and can be trivially distributed accross many SIMD threads.

Introduction

A table, or dataframe, is a sequence of tuples, called records, where each record is associated with a key.

For a natural join, for each record A in one table B, the key K associated with A is used to search the other table R, and for each record B with associated key K in R a new record C is output as a union of A with B.

Outer join extends natural join by additionally outputting a record C for each record A in L with associated key K where no record B exists in R with associated key K. Instead, B is instantiated as a record with all empty values, and C is output as the union of A with B.

Related Work

Nested Loop Join

The simplest implementation of the natural join is a nested loop. For each record A in table L, the inner loop performs a linear scan over table R, then for each record B in table R, if the key associated with A matches the key associated with B, a record C is output as the union of A with B.

Sort-Merge Join

To optimize the nested loop approach, table R can be sorted by each record's associated key, then for each key K associated with record A in R the inner loop can simply start and end at key-run boundaries of records in R associated with key K. If table L is not sorted, then a mapping of key to key-run boundaries must be used, and a key-run boundaries lookup must be performed for each record of L. Otherwise, for each run of key K in table L, the key-run of K in R is simply the next key-run after the previously processed key-run, at the cost of committing to highly sequential computation.

Hash Join

A second optimization approach to the nested loop is the hash join. Rather than sorting records of table R by key and creating a mapping of keys to key-run boundaries, records of R are hashed by key into a mapping of key to record index sequences, where each unique key K maps to a sequence containing the indices of all records in R associated with K. Then, for each record A with key K in table L, K is used to look up the sequence S of indices of records in table R with associated key K, and for each index I in S, record C is output as the union of A with B = R[I] (the record in R at index I). In addition to avoiding a full linear scan of table R for each record in table L, this approach has the benefit of simplifying the task of partitioning tables L and R and the join task across multiple compute nodes, at the cost of requiring atomic random memory access.

Proposed Algorithm

We propose a modified version of the first optimization (the sort-merge join) described above. Given two tables L and R to join, concatenate L and R, insert the boolean column table ID T insert the boolean column table ID T, and sort the result by each key K associated with a record in L or R. Hence, a stable sort operation groups together the set of records of L with key K immediately followed by the set of records in R with key K. What remains is to identify the record group boundaries, to select the record groups that satisfy the join criteria, and to parallel gather the join result.

Example

```
Compute L LEFT JOIN R WHERE L.K == R.K .

  L           R
K   A       K   B
------      ------
0   0       9   0
0   1       9   1
0   2       8   2
2   3       7   3
2   4       6   4
4   5       4   5
5   6       3   6
6   7       3   7
7   8       0   8
7   9
```

```
Concatenate L and R .

I0 = range(len(L) + len(R))
0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18

C0 = I0 >= len(L)   // table ID
0   0   0   0   0   0   0   0   0   0   1   1   1   1   1   1   1   1   1

C1 = L.K : R.K      // key
0   0   0   2   2   4   5   6   7   7   9   9   8   7   6   4   3   3   0

C2 = L.A : R.B      // value
0   1   2   3   4   5   6   7   8   9   0   1   2   3   4   5   6   7   8
```

```
sort ((C0, C1, C2), key=C1) .

C3 = sort (C0, key=C1)  // table ID
0   0   0   1   0   0   1   1   0   1   0   0   1   0   0   1   1   1   1

C4 = sort C1            // key
0   0   0   0   2   2   3   3   4   4   5   6   6   7   7   7   8   9   9

C5 = sort (C2, key=C1)  // value
0   1   2   8   3   4   6   7   5   5   6   7   4   8   9   3   2   0   1
```

```
Compute run lengths and offsets .

C6 = not I0 or (C3[I0], C4[I0]) != (C3[I0-1], C4[I0-1])
1   0   0   1   1   0   1   0   1   1   1   1   1   1   0   1   1   1   0

C7 = exclusive sum (C6)
0   1   1   1   2   3   3   4   4   5   6   7   8   9   10  10  11  12  13

I1 = C7 if C6
0   1   2   3   4   5   6   7   8   9   10  11  12

C8 = C3 if C6           // record subset table id
0   1   0   1   0   1   0   0   1   0   1   1   1

C9 = C4 if C6           // record subset key
0   0   2   3   4   4   5   6   6   7   7   8   9

C10 = I0 if C6          // record subset offset
0   3   4   6   8   9   10  11  12  13  15  16  17

C11 = (len(I0) if I1+1 == len(I1) else C10[I1 + 1]) - C10[I1]
3   1   2   2   1   1   1   1   1   2   1   1   2
```

```
Select left join components .
// (inner join components would be C9[I1] == C9[I1 - C8 - C8 + 1])

C12 = not C8 or (I1 and C9[I1] == C9[I1 - 1])
1   1   1   0   1   1   1   1   1   1   1   0   0

C13 = exclusive sum (C12)
0   1   2   3   3   4   5   6   7   8   9   10  10

I2 = C13 if C12
0   1   2   3   4   5   6   7   8   9

C14 = C9 if C12          // selected key
0   0   2   4   4   5   6   6   7   7

C15 = C10 if C12         // selected offset
0   3   4   8   9   10  11  12  13  15

C16 = C11 if C12         // selected length
3   1   2   1   1   1   1   1   2   1
```

```
Partition inner from outer join components .

C17 = I2 and C14[I2] == C14[I2-1]
0   1   0   0   1   0   0   1   0   1

C18 = (I2 + 1) < len(I2) and C14[I2] == C14[I2+1]
1   0   0   1   0   0   1   0   1   0

C19 = C17 or C18
1   1   0   1   1   0   1   1   1   1

C20 = exclusive sum (C19)
0   1   2   2   3   4   4   5   6   7
C21 = not C19
0   0   1   0   0   1   0   0   0   0
C22 = exclusive scan (C21, y0=(len(I2)-1), op=(-))
9   9   9   8   8   8   7   7   7   7

C23 = C20 if C19 else C22       // write address
0   1   9   2   3   8   4   5   6   7

C24[C23] = C14       // key
0   0   4   4   6   6   7   7   5   2

C25[C23] = C15       // offset
0   3   8   9   11  12  13  15  10  4

C26[C23] = C16       // length
3   1   1   1   1   1   2   1   1   2
```

```
Compute join groups lengths and offsets .
//  N0 = # of inner join components = C19[-1] + C20[-1] = 8 .

I3 = range(len(I2) - N0/2)
0   1   2   3   4   5

C27 = C24[I3 + min(I3, N0/2)]
0   4   6   7   5   2

C28 = C26[I3 + I3] * C26[I3 + I3 + 1] if I3 < N0/2 else C26[I3 + N0/2]
3   1   1   2   1   2

C29 = exclusive sum (C28)
0   3   4   5   7   8
```

```
Build join index .

I4 = range(C28[-1] + C29[-1])
0   1   2   3   4   5   6   7   8   9

J0 = I4 and I4 in C29        // join boundaries
0   0   0   1   1   1   0   1   1   0

J1 = inclusive sum (J0)
0   0   0   1   2   3   3   4   5   5

J2 = I4 - C29[J1]
0   1   2   0   0   0   1   0   0   1

J3 = C26[J1 + min(J1, N0/2)]
3   3   3   1   1   2   2   1   2   2

J4 = J2 % J3
0   1   2   0   0   0   1   0   0   1

J5 = J2 / J3
0   0   0   0   0   0   0   0   0   0

J6 = C25[J1 + min(J1, N0/2)]
0   0   0   8   11  13  13  10  4   4

J7 = C25[J1 + J1 + 1] if J1 < N0/2 else -1
3   3   3   9   12  15  15  -1  -1  -1

J8 = J4 + J6
0   1   2   8   11  13  14  10  4   5

J9 = J5 + J7
3   3   3   9   12  15  15  -1  -1  -1
```

```
Gather the join result .

J1:     0   0   0   1   2   3   3   4   5   5
C27:    0   4   6   7   5   2

J8: 0   1   2   8   11  13  14  10  4   5
J9: 3   3   9   12  15  15  -1  -1  -1
C5: 0   1   2   8   3   4   6   7   5   5   6   7   4   8   9   3   2   0   1

J.K = C27[J1]
0   0   0   4   6   7   7   5   2   2

J.A = C5[J8]
0   1   2   5   7   8   9   6   3   4

J.B = C5[J9] if J9 != -1 else -1
8   8   8   5   4   3   3   -1  -1  -1
```

```
J = L LEFT JOIN R .

  L           R              J
K   A       K   B       K   A   B
------      ------      ---------
0   0       9   0       0   0   8
0   1       9   1       0   1   8
0   2       8   2       0   2   8
2   3       7   3       4   5   5
2   4       6   4       6   7   4
4   5       4   5       7   8   3
5   6       3   6       7   9   3
6   7       3   7       5   6  -1
7   8       0   8       2   3  -1
7   9                   2   4  -1
```

Concatenate and Sort Tables by Key

Sorting by key is a memory access efficient means for globally drawing together all records with a particular key into the same neighborhood. While building a hash table requires a random access write to global memory for each record hashed, the radix sort used is implemented as a sequence of coalesced memory accesses.

Compared to sort-merge join, having a sequence of neighborhoods each containing the records from both tables with a particular key allows the join task to be distributed simply among many compute nodes. Meanwhile, the need to lookup key-run boundaries for each record in table L or perform a key search of table R is avoided, as each set of records in L with key K is immediately followed by the set of records in R with key K.

The tables L and R are treated as abstract sequences of key-value pairs and concatenated into a single sequence. The "SortPairs" stable radix sort function from the CUB library is then used to sort the sequence.

Scan Record Group Boundaries

The representative (shared) keys and run lengths are computed using the "RunLengthEncoding" function from the CUB library applied to the table ID T. The offsets of the record set boundaries are recovered using "ExclusiveSum" over the run lengths, the partial prefix sum function from the CUB library.

Apply Join Criteria

Hence, each record in the summary table with key K will either be preceded by or followed by a record with the key K, if and only if there exists both a non-empty subset of L with key K and a non-empty subset of R with key K.

We first populate a flag array F, where each value in F is set if the key K for the corresponding record in the summary table is either preceded by or followed by a record with a matching key, or otherwise unset. If a left join is desired, we also flag records representing record groups of L, but have no matching record group of R. If a full join is desired, then all flags are set. The summary table is then filtered on F.

If an outer join is desired, then the records of the summary table satisfying inner join criteria are partitioned from the rest of the records so that inner join records precede the rest. Outer join records are reversed by the partition operation.

At this point all records have been grouped by key into neighborhoods. Additionally, the stable sort preserved the ordering of the table ID T, so that all records with key K from table L are followed by all records with key K from table R. The values of T change value at exactly the boundaries between neighborhoods and between record sets within neighborhoods. Using that as a flag, we summarize each record set by the shared key, the offset, and the set size of that record set.

Gather Join Result

To gather the result, we must first compute how many records are in the join result. The resulting record set for each key K in the natural join is computed as the cartesian product over the record set with key K in table L and the record set with key K in table R. Given the run lengths described previously, the length of each cartesian product with key K is the product of the lengths of the two record sets with key K. For outer joins, the run lengths of the outer join components have already been computed; we simply extend the cartesian products lengths sequence with those run lengths. This sequence of lengths is C28 in the example. We then run an exclusive prefix sum over the resulting join record sets lengths to get the record offset into the join for each record subset, C29. The length of the join result is given as the sum of the last value of C28 and the last value of C29.

With the join record subsets offsets, we then scatter a flag into the flag array `J0` marking the beginning of each record subset of the join (excluding a flag for the first offset). Running an inclusive prefix sum over `J0` produces J1, a join record subset index broadcasted to each join record subset. Then, for each value x in J1 at index `i`, the value `J2[i]` is computed as the difference `J2[i] = i - C29[x]`, producing a rank enumeration for each key in the join result. The rank is then used to compute the primary and secondary offsets of a sequence of gather index tuples `(J8, J9)` representing the offsets into the sorted keys C4 and sorted values C5 of which to gather the join result key columns, and the join result value columns, respectively. J8 is used to gather the values of columns from L, and J9 is used to gather the values of columns from R.

Challenges and Opportunities

The algorithm and simple example have been described where the join key is a (scalar) primary key and the associated value is a single value. Yet, the algorithm may be scaled to composite keys with many associated values. It is even possible to apply the algorithm to joins involving more than two tables.

At some point, a composite key becomes too large to use as the sorting key efficiently. In such a case, a scalar key may be derived by hashing the composite key. Additionally, the table ID can be represented as the least significant bit of the derived key. However, the hashed value will likely produce hash collisions resulting in false matches in the join. For inner joins, the false matches can simply be filtered out of the join. For outer joins, the inner join must be partitioned into true matches and false matches. Then, falsely matched derived keys are looked up in the set of true matches and discarded if the composite key is found. The remainder is added to the set of outer join components.

Similarly, at some point there are too many associated values to sort efficiently. In such a case, records may be enumerated to produce a row ID. The row ID can implicitly represent the table ID, since the row ID for each record in the second table will be larger than the number of records in the first table. The row IDs instead of the records are then sorted by key, and the join result is represented by the three tuple record set (key, L row ID, R row ID). The final join result can then be materialized by a simple gather operation.

The join algorithm is described using the assumption that only two tables are being joined. The algorithm may be trivially adapted to accomodate more than two tables. Enumerate the tables to derive a table ID. If the join uses a derived key but not a row ID, set the table ID as the least significant `ceil(log2(N))` bits of the derived key for N tables. When applying the join criteria, flag inner join components by computing the equality among N consecutive keys.

Performance

The following table provides timings for inner joins on tables L, R and J with 3 column composite key and no associated value column with N rows each (approximately one to one cardinality).

N	Interleave	PyGDF
1024	0.5863	431.33015632629395
2048	0.5959	439.1122817993164
4096	0.7622	437.436580657959
8192	0.7647	433.8551998138428
16384	0.7644	438.488245010376
32768	0.9312999999999999	476.58846378326416
65536	1.6437000000000002	459.77327823638916
131072	2.4196999999999997	466.7545557022095
262144	1.9535999999999998	460.03315448760986
524288	2.532	459.456467628479
1048576	6.0266	447.4830150604248
2097152	5.7803	484.2724800109863
4194304	8.0931	485.1621150970459
8388608	10.7327	474.75810050964355
16777216	19.964499999999997	524.4124889373779
33554432	38.259100000000004	685.5380058288574
67108864	82.72160000000001	969.7905540466309

Summary

A flexible many-thread SIMD algorithm for computing joins linearly has been proposed. Sort-merge join can use memory access coalescing, but it is difficult to distribute accross many SIMD threads. Hash joining can be highly parallel, but requires random memory access that is difficult to coalesce. The algorithm proposed here both enables simple memory coalescing and simple work distribution accross many SIMD threads making it uniquely appropriate for GPU-accelerated join computation.