Implementing a Worst-Case Optimal Multi-Way Join Algorithm

Firas Abuzaid, James Koppel, Yi Lu

CSAIL, MIT
{fabuzaid,jkoppel,yil}@mit.edu

Abstract. Efficient join processing is one of the most fundamental and well-studied tasks in database research. A worst-case optimal join algorithm was established by Ngo, Porat, Ré and Rudra. However, they left an interesting question: What's the performance of the algorithm for average case complexity? In this work, we implemented these ideas to see how they compare to the algorithms in a modern RDBMS and describe a detailed implementation and possible integration on PostgreSQL and Spark. Experiments on real world graphs verified that NPRR is able to achieve 2-6 times performance improvements over the state-of-the-art RDBMS.

1 Introduction

Efficient join processing is one of the most well-studied problems in database research. Traditional database systems are highly optimized for pair-wise join, where OLAP workloads often consists of star joins with aggregates. There is a fact table which is much larger than the dimension tables, where the fact table is horizontally partitioned and dimension tables are replicated to optimize the queries.

As many large scale data analytics engines emerge such as Spark and F1, queries are becoming more and more complex. People realized that counting different patterns of a graph is helpful to understand a complex network structure. However, the performance of evaluating a join query using a pair-wise manner could be highly suboptimal. Considering listing all triangles in a given graph, as a sequence of two pair-wise join, the size of the intermediate result could be much larger than the number of triangles. The intuition behind this is that there could be much more paths with length two than triangles in a graph.

A full conjunctive query, where there is no projection and every variable in the body appears in the head is useful to formulate many useful queries. For example, we can use the following query to find triangles in a graph: $Q(a,b,c) = R(a,b) \bowtie S(b,c) \bowtie T(a,c)$. Given constraints on the sizes of the input relations such as $|R| \leq n$, $|S| \leq n$, $|T| \leq n$, what is the upper bound of the query result size $|Q|$? This question has practical importance, since a tight bound $|Q| \leq f(n)$ implies an $\Omega(f(n))$ worst-case running time for algorithms answering such queries.

To achieve this goal, Ngo et al. have developed a new algorithm, called NPRR, which is provably worst-case optimal. However, it is interesting to see the performance of this algorithm for average case. We start by introducing the theory behind the algorithm and an description of implementation. In experiments on different types of graphs, we find that NPRR can always answer queries faster using a traditional database engine, e.g., PostgreSQL.

The paper is organized as follows. In Section 2 we review the theory of worst-case optimal join algorithm. We give a detailed description of our implementation of the algorithm in Section 3 and show the results in Section 4. Some interesting future work is listed in Section 5 and we conclude the paper in Section 6.

2 Theory

We briefly summarize the theoretical underpinnings of the NPRR algorithm to provide further understanding and underscore our motivation for studying this algorithm. A detailed survey of this algorithm (and its more general version, known as Generic Join) can be found in [4] and [3].

Traditional mult-way join queries (join queries that involve n relations, where $n > 2$) are evaluated by using a cost-based estimation model to determine the best pair-wise join plan [5]. This, however, is provably suboptimal in the worst-case; to see this, consider the triangle query:

$$Q = R(A, B) \bowtie S(B, C) \bowtie T(A, C)$$

Without loss of generality, suppose this query was executed using the pair-wise join plan $(R \bowtie S) \bowtie T$. If $|R| = |S| = N$, then the intermediate relation $P = R \bowtie S$ will have cardinality $|P| = O(N^2)$ in the worst case. However, the size of the intermediate relation does not provide a lower bound for the size of the final output. If $|T| = N$ as well, then the subsequent join $P \bowtie T$ may, in the best case, generate a final output of N^2 tuples. However, in the worst case, $P \bowtie T$ may only generate $O(N)$ tuples, or even no tuples at all.

Thus, we can always construct a family of instances that will provably have $\Omega(N^2)$ runtime; figure 2 illustrates this in more detail. Moreover, we know by the `AGM` bound that the upper bound for the output for the triangle query is $O(N^{3/2})$ [1]. Ngo et al. demonstrate that there is a family of algorithms that does achieve this bound. We discuss the intuition for this algorithm strictly for the case of triangle queries below; a more general and thorough treatment of the algorithm (as well as detailed pseudocode) can be found in [3] and [4].

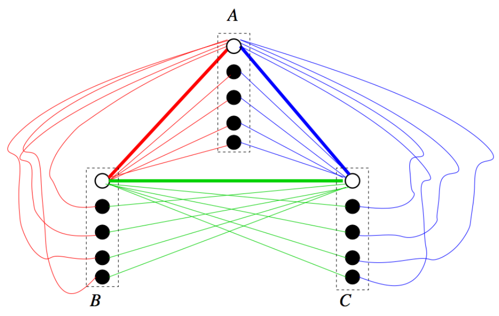
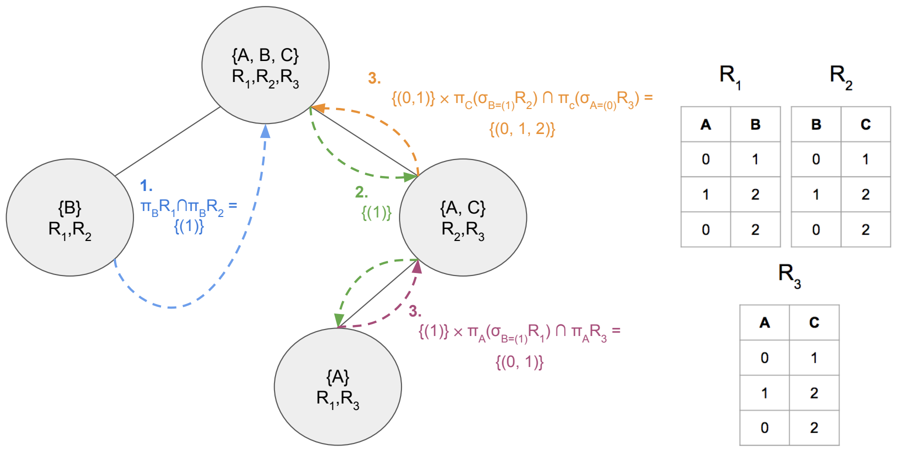


Fig. 1. A counter-example which demonstrates why pair-wise plans for the triangle query are suboptimal. See [4] for a thorough explanation.

First, rather than fix an order of relations to join, we instead fix some order of *attributes*. Doing so enables us to "look ahead" and ensure that we don't unnecessarily materialize an intermediate relation that could prove to be wasteful later on in the computation. Without loss of generality, let our order $U = (A, B, C)$. We first compute candidates for attribute A, which we denote as C_A. To do this, we examine each relation with A in its attribute set, and then compute the following: $C_A = \pi_A(R) \cap \pi_A(T)$. We can now guarantee that, for every possible triangle (a_i, b_j, c_k) in the final output, $a_i \in C_A$ will always hold true. In other words, C_A gives us the list of candidate values for attribute A that may participate in the final output.

Next, we compute candidates for attribute B, which is next in our order U. We use the same approach as before, but we now incorporate our C_A candidate values. For each $a_i \in C_A$, we evaluate $C_{AB} = \{a_i\} \times \big(\pi_B(\sigma_{A=a_i} R) \cap \pi_B(T)\big)$. Just as before, we maintain the invariant that every tuple $(a_i, b_j) \in C_{AB}$ is a candidate tuple for the final output.

Lastly, for attribute C, we iterate through each candidate $(a_i, b_j) \in C_{AB}$, and evaluate $C_{ABC} = \{a_i, b_j\} \times \big(\pi_C(\sigma_{B=b_j} S) \cap \pi_C(\sigma_{A=a_i} T)\big)$. The candidate set C_{ABC} now equals our final output.


{A, B, C}
R_1,R_2,R_3
{B}
R_1,R_2
{A, C}
R_2,R_3
{A}
R_1,R_3
1.
$\pi_B R_1 \cap \pi_B R_2 =$
{(1)}
2.
{(1)}
3.
$\{(0,1)\} \times \pi_C(\sigma_{B=(1)} R_2) \cap \pi_c(\sigma_{A=(0)} R_3) =$
{(0, 1, 2)}
3.
$\{(1)\} \times \pi_A(\sigma_{B=(1)} R_1) \cap \pi_A R_3 =$
{(0, 1)}

R_1

A	B
0	1
1	2
0	2

R_2

B	C
0	1
1	2
0	2

R_3

A	C
0	1
1	2
0	2

Fig. 2. A walkthrough of the NPRR algorithm on the triangle query. Here the attribute order $U = (B, A, C)$.

3 Implementation

We implemented two versions of NPRR – a single-node, single-threaded implementation in C++, and a distributed implementation in Spark. For the sake of brevity, we will only discuss the former in this paper; the Spark implementation can be found at `https://github.com/fabuzaid21/distributed-generic-join`.

To evaluate the NPRR algorithm efficiently, we first build a query tree plan, following Algorithm 3 from [3]. This query tree represents the recursive path we take during the execution of the algorithm.

Next, we construct a family of indexes to efficiently compute the selection predicates, projections, and intersections. For each relation, we examine the total order for the join along the relations of that attribute and compute a series of keys that we will use in the evaluation of the join. For example, if relation $R(A_2, A_4, A_6)$ was part of our join query, and the total order we computed is $U = (A_1, A_4, A_2, A_5, A_6, A_3)$, then we would compute the following keys: $[(), (A_4)], [(), (A_4, A_2)], [(), (A_4, A_2, A_6)], [(A_4), (A_2)],$ $[(A_4), (A_2, A_6)]$,$[(A_4, A_2), (A_6)]$, and $[(A_4, A_2, A_6), ()]$. Once we compute the keys $K = (k_1, k_2)$ for the relations R, we then construct two indexes for each key-relation pair:

1. A hash set that indicates whether a tuple $t \in \pi_{k_1 \cup k_2} R_e$.
2. A hash map that maps a tuple t with attributes in k_1 to values v with attributes k_2 in R_e. This hash map lets us compute $\pi_{k_2}(R_e \ltimes t)$.

To build these indexes efficiently, we modeled attributes as bitvectors in C++. Each relation had an array of sets and hash maps; we computed an index into the array using the key pair (k_1, k_2). While the overhead of constructing these indexes is substantial, it would be practically impossible to acheive any sort of efficiency without them.

4 Results

We now evaluate the performance of NPRR, PostgreSQL and an in-memory triangle counting baseline. We release all the source codes of the algorithms used in our evaluation in `https://github.com/fabuzaid21/nprr`.

4.1 Choice of Datasets

We used four large real-world datasets, which are from four different domains as shown in Table 1. Facebook consists of 'circles' (or 'friends lists') from Facebook. Gnutella is a sequence of snapshots of the Gnutella peer-to-peer file sharing network from August 2002. Wikivote consists of votes deciding who is going to be promoted as adminship. Condmat (Condense Matter Physics) collaboration network covers scientific collaborations between authors papers submitted to Condense Matter category.

Data	$\lvert V\rvert$	$\lvert E\rvert$
facebook	4039	88234
gnutella	36682	88328
wikivote	7115	103689
condmat	23133	93479

Table 1. Datasets

4.2 Experimental Settings

We ran our experiments on a machine with a 2.4GHz Intel(R) Core(R) i5-4258U CPU and 8 GB DDR3-1,6000 RAM, running 64-bit MacOSX 10.11.2. NPRR was compiled with clang-700.1.81(Apple LLVM version 7.0.2). The optimization flag O2 option was enabled. PostgreSQL 9.4.5 was used during evaluation. All running time reported is the wall-clock time elapsed during computing.

Figure 3 shows the performance of NPRR and PostgreSQL on facebook, gnutella, wikivote and condmat. On gnutella, the performance of PostgreSQL is comparable to NPRR. On facebook, wikivote and condmat, the NPRR is about 2 - 6 times faster than PostgreSQL.

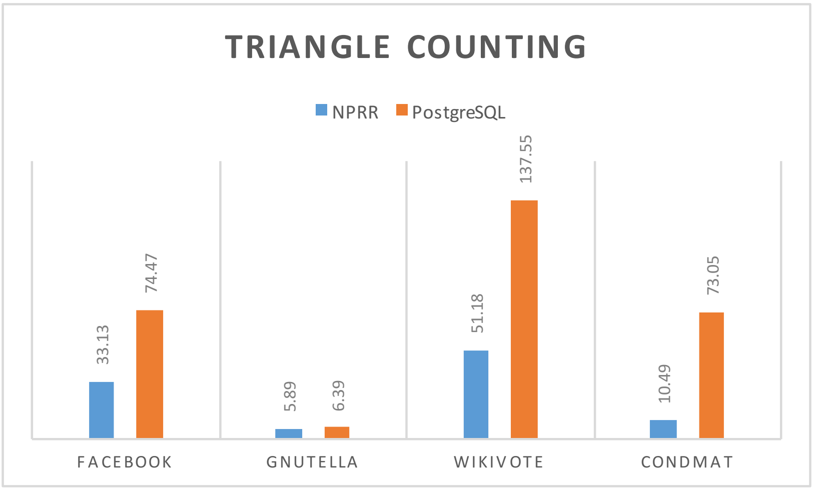


Fig. 3. Performance of NPRR and PostgreSQL

4.3 Native implementation

We also provide a native high optimized implementation for triangle counting in C++. Interested readers can find the code in Appendix A.

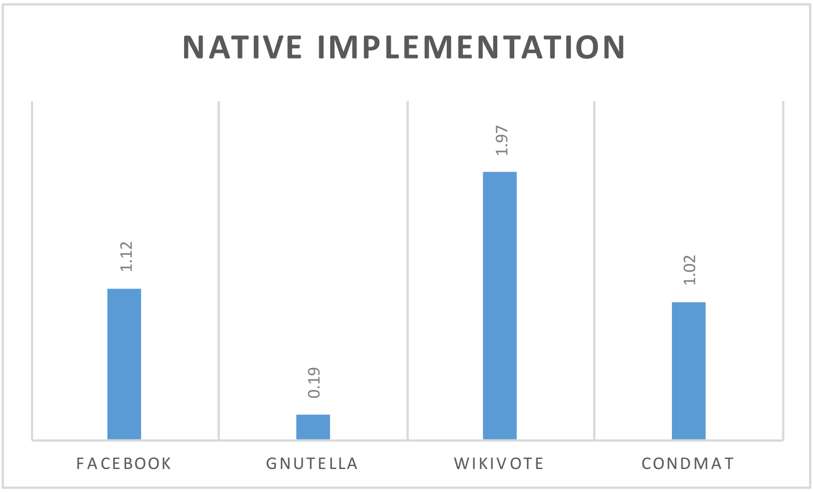


Fig. 4. Performance of highly optimized native implementation

Figure 4 shows the performance of our highly optimized native implementation on facebook, gnutella, wikivote and condmat. This provides a very clear reference point that shows the limit of performance. Our native implementation is about 10 - 25 times faster than NPRR.

4.4 Discussion

We see the following key inferences: (1) Native code, as expected, delivers best performance since it is highly optimized for triangle counting. (2) NPRR performs better than PostgreSQL.

5 Future Work

Our implementation of the NPRR algorithm has room for many optimizations. Notable potential optimizations include:

- The computation of indices on demand, to avoid the creation of unused indices
- The use of constant-width bitmasked vectors to repreest tuples, to avoid the expensive copy operations needed for projection

However, the most important next step in the development of this algorithm would be an implementation into an RDBMS such as Postgres.

5.1 Integration into Postgres

We conducted extensive study into the feasibility of integrating the NPRR join into Postgres.

The complexity of adding new types of joins to Postgres has been greatly reduced by the addition of Custom Scan Providers [[2]] in the development branch of Postgres 9.5. Custom scan providers allow the insertion of nodes tagged "custom" into Postgres's internal data structures. When encountering one of these nodes, Postgres will call certain externally-defined functions when it needs to do things such as render the node or execute a scan. A Custom Scan Provider provides implementations for these functions, enabling the addition of new join and scan types simpling by linking in a dynamic library.

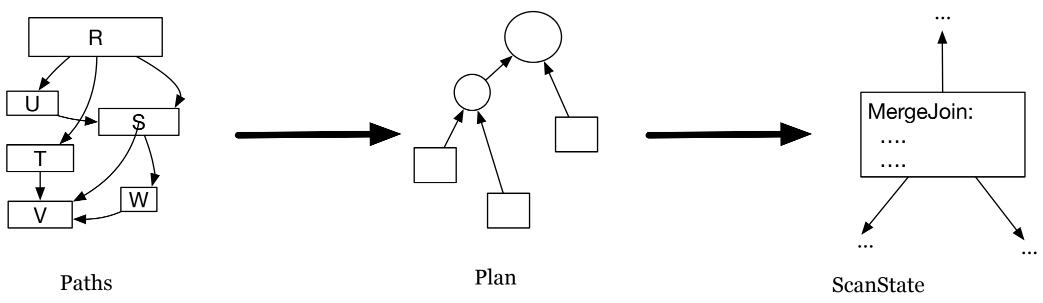


Fig. 5. Lifecycle of a join in Postgres

Lifecycle of a Join Whatever the integration mechanism, Implementing a join in Postgres ultimately requires integrating into the three subsystems involved in join processing: query planning, plan representation, and query execution. The query planner constructs a structure of type `RelOptInfo` for each relation to be computed during the query, and then builds a directed acyclic graph of "paths" presenting various ways these relations can be computed from each other. Once the paths have been considered, an optimal path is selecetd and converted into a query plan tree. During query execution, every node of the plan tree is given an associated "ScanState" structure. The executor function for the node is then responsible for computing the next tuple, updating the scan state and executing the plans for its child nodes. Figure 5.1 illustrates the lifecycle of a join in Postgres. While extending the representation of query plans is straightforward, the unusual nature of the NPRR join presents problems for integrating into both the executor and the planner.

Integrating into the Query Optimizer Conventional query planning as implemented in Postgres assumes at most two children per node, which makes adding the NPRR join difficult. Postgres contains two query planners: the Genetic Query Optimizer (GEQO), and a standard dynamic-programming based optimizer.

The Genetic Query Optimizer runs a genetic algorithm to iteratively optimize a pool of potential query plans. Its mechanism for considering new joins fundamentally only considers pairwise joins. During its process for creating new individuals, it contains a list of "clumps," or subtrees together. It is presented with new clumps according to an order suggested by the genetic algorithm, and considers the cost of joining the new clump pairwise to any existing clump.

The "standard join search" is a Selinger-type DP-based optimizer. Like other Selinger-type optimizers and similar to the Genetic Query Optimizer, it maintains optimal subplans for executing subsets of the query, and iteratively adds new base relations to the plan by joining individual base relations to an existing subtree with a pairwise join.

Both optimizers hence do not admit a non-binary join in their search process. We do however have one saving grace that makes the integration of the NPRR join into the query planner possible, if not feasible. New plans are considered in Postgres using an imperative API. That is, rather than e.g.: invoking a planning routine which returns a new path to be considered, each `RelOptInfo` maintains a list of optimal paths for computing that relation. A planning routine is expected to modify this list to add new potential joins, possibly pruning strictly inferior paths. At certain points during query optimization, Postgres invokes a routine to be implemented by a Custom Scan Provider. A Custom Scan Provider that offers the NPRR join could use this to replace all existing paths with one representing an NPRR join. Properly considering trees that include NPRR joins alongside traditional pairwise joins would unfortunately require the implementation of a new search algorithm.

Integrating into the Query Executor The NPRR join requires more substantial preprocessing than other join, involving the creation of quadratically many hash tables. While Postgres has built-in infrastructure for creating hash indices, the existing hash index support is hardcoded to only index an entire table, and to only work on base relations. We would thus need to implement the indexing ourselves, including handling the buffer management policies. Joins in Postgres are given great leeway to manage their internal state, so this issue is less prohibitive than with the query optimizer. Proper index management would still nonetheless be very difficult.

5.2 Distributed NPRR in Spark

We also implemented a basic implementation of NPRR in Spark which computes the join across a distributed cluster of nodes. This raises an interesting question: do the worst-case optimality guarantees of NPRR translate to the distributed setting? Moreover, what does it mean for a join algorithm to be worst-case optimal in the distributed setting? This is a future area of research that could impact large-scale data processing in cluster computing environments and is certainly worth pursuing.

6 Conclusion

As many large scale data analytics engines emerge such as Spark and F1, queries are becoming more and more complex. However, the performance of evaluating a join query using a pair-wise manner could be highly suboptimal. In this paper, we studied a worst-case optimal join algorithm was established by Ngo, Porat, Ré and Rudra. We implemented these ideas to see how they compare to the algorithms in a modern RDBMS and describe a detailed implementation and possible integration on PostgreSQL and Spark. Experiments on real world graphs verified that NPRR is able to achieve 2-6 times performance improvements over the state-of-the-art RDBMS.

References

1. A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. In *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*, pages 739–748. IEEE, 2008.
2. K. Kohei. Writing a Custom Scan Provider. `http://www.postgresql.org/docs/9.5/static/custom-scan.html`, 2015. Accessed: 10-Dec-2015].
3. H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms:[extended abstract]. In *Proceedings of the 31st symposium on Principles of Database Systems*, pages 37–48. ACM, 2012.
4. H. Q. Ngo, C. Re, and A. Rudra. Skew strikes back: New developments in the theory of join algorithms. *ACM SIGMOD Record*, 42(4):5–16, 2014.
5. L. D. Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems (TODS)*, 11(3):239–264, 1986.

A Native implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#include<vector>
#include<map>

using namespace std;

vector<pair<int, int> > rel1;
map<int, vector<int> > rel2;
map<int, vector<int> > rel3;
```

```
double get_wall_time(){
    struct timeval time;
    if (gettimeofday(&time,NULL)){
        //  Handle error
        return 0;
    }
    return (double)time.tv_sec + (double)time.tv_usec * .000001;
}

int main(int argc, char **argv) {
  FILE *fin = fopen(argv[1], "r");

  double t1 = get_wall_time();

  while (true) {
    int a,b;
    int nRet = fscanf(fin, "%d %d", &a, &b);
    if (nRet != 2) {
      break;
    }


    rel1.push_back(make_pair(a,b));

    if (!rel2.count(a)) {
      vector<int> v;
      rel2[a] = v;
    }

    rel2[a].push_back(b);


    if (!rel3.count(a)) {
      vector<int> v;
      rel3[a] = v;
    }

    rel3[a].push_back(b);
  }

  double t2 = get_wall_time();

  int nTriangles = 0;

  for (auto x : rel1) {
    int a = x.first;
    int b = x.second;

    if (rel2.count(b)) {
      for (int c : rel2[b]) {
        if (rel3.count(a)) {
          for (int cp : rel3[a]) {
            if (c == cp) {
              nTriangles++;
            }
          }
        }
      }
    }
  }

  printf("%d\n", nTriangles);

  double t3 = get_wall_time();

  printf("Reading time: %f\n", t2-t1);
  printf("Computation time: %f\n", t3-t2);


  return 0;
}
```