Are Multi-Way Joins Actually Useful?

Michael Henderson and Ramon Lawrence
Department of Computer Science, University of British Columbia Okanagan, Kelowna, BC, Canada
mikecubed@gmail.com, ramon.lawrence@ubc.ca

Keywords: Multi-way Join : Hybrid Hash : Join Ordering : Query Optimization : PostgreSQL

Abstract: Multi-way joins improve performance by avoiding extra I/Os from multiple partitioning steps. There are several multi-way join algorithms proposed, and the research results are encouraging. However, commercial database systems are not currently using multi-way joins. Practical issues include modifying the optimizer and execution system to support multi-way operators and ensuring robust and reliable performance. The contribution of this work is an implementation and experimental evaluation of multi-way joins in PostgreSQL. We provide algorithms that modify the optimizer to cost multi-way joins and create and execute query plans that have more than two input operators. Experimental results show that multi-way joins are beneficial for several queries in a production database system and can be effectively exploited by the optimizer, however there are implementation issues that must be resolved to guarantee robust performance.

1 INTRODUCTION

There has been recent research on the development of multi-way join algorithms that are capable of joining more than two relations at a time. Despite the potential performance benefits, database systems are not using multi-way joins. The objective of this paper is to attempt to answer the question: Are multi-way joins useful in database systems?

The positive argument for multi-way joins is supported by a significant amount of research literature that demonstrates performance benefits. Hash teams (Graefe et al., 1998a) implemented in Microsoft SQL Server 7.0 and generalized hash teams (Kemper et al., 1999) improve I/O efficiency compared to binary plans by avoiding multiple partitioning steps. These algorithms can be used for joins of relations on the same join attributes (hash teams) or join relations related by a chain of functional dependencies/foreign keys (generalized hash teams). It has been shown by the SHARP (Bizarro and DeWitt, 2006) query processing system that a multi-way operator adapts to estimation inaccuracies, has the ability to dynamically share memory easier than binary plans, and avoids redundant `next()` calls in the iterator model. Further, streaming multi-way joins such as MJoin (Viglas et al., 2003) and slice join (Lawrence, 2008) compensate for network delays and blocking. A streaming multi-way join operator allows tuples to be processed from any input at any time which simplifies optimization issues such as join order selection and handles changing input arrival rates.

However, there are issues with robustness of multi-way operators. A multi-way operator still must select a probe ordering internally which has similar complexity as join order selection with the additional goal that it is adaptable to the data characteristics during join processing. The one known commercial implementation, hash teams in Microsoft SQL Server 7.0 (Graefe et al., 1998b), was later dropped in SQL Server 2000 SP1 (Microsoft Corporation, 2001) in order to improve stability and due to limited performance benefits. There are limitations on the types of joins possible using a multi-way operator. Hash teams are limited to joins of relations all on the same join attributes, generalized hash teams support chains of foreign key joins, and SHARP supports star queries. Finally, although it has been argued that the changes to the query optimizer to support multi-way joins are straightforward, in practice that is not the case. A typical query optimizer (Moerkotte and Neumann, 2008) is only capable of exploring binary plans. Without modifying the plan generator to cost multi-way plans simultaneously with binary plans, multi-way operators must be constructed after binary optimization is complete.

To gain insights on the capability of multi-way joins in a database system, we implemented three multi-way join algorithms: hash teams, generalized hash teams, and SHARP. We have developed various implementations of generalized hash teams and examined their tradeoffs. A new post-optimization method for creating multi-way join plans takes as input an optimized binary query plan and applies transformations and costing to determine if a multi-way plan is more efficient than the input binary plan.

The experimental test bed is PostgreSQL. PostgreSQL is an open-source database system that closely follows conventional practice. PostgreSQL has an iterator-based execution model, cost-based optimizer, and an implementation of hybrid hash join. Queries with multi-way joins show performance benefits over binary hash joins in many cases. The performance improvements vary by query type and implementation approach. In general, multi-way algorithms are faster as they perform fewer I/Os, but implementations must avoid increasing the CPU time for probing and join clause evaluation. The post-optimization method implemented finds a significant number of the beneficial plans, has insignificant runtime cost, and requires minimal modifications to the optimizer.

The contributions of this paper are:

- Implementations of the current state-of-the-art multi-way join algorithms.
- Experimental analysis of some implementations of generalized hash teams.
- A post-optimization algorithm for converting an optimized binary plan into a multi-way execution plan.
- An experimental evaluation of all algorithms in the production database system PostgreSQL.

The organization of this paper is as follows. In Section 2, we briefly overview existing work on multi-way hash joins. The implementation of generalized hash teams is described in Section 3. The post-optimization algorithm for constructing multi-way plans is described in Section 4. The issues in implementing the join algorithms in PostgreSQL are covered in Section 5. Experimental results in Section 6 demonstrate benefits but also implementation challenges of multi-way joins. The paper closes with conclusions and future work.

2 PREVIOUS WORK

This section contains background information on binary and multi-way hash join algorithms.

2.1 Binary Hash Joins

A join combines two relations into a single relation. We refer to the smaller relation as the *build relation*, and the larger relation as the *probe relation*. A hash join first reads the tuples of the build relation, hashes them on the join attributes to determine a partition index, and then writes the tuples to disk based on the partition index. It then repeats the process for the probe relation. The partitioning is designed such that each build partition is now small enough to fit in a hash table in available memory. This hash table is then probed with tuples from the matching probe partition. Hybrid hash join (HHJ) (DeWitt et al., 1984) is a common hash join algorithm implemented in most database systems. Hybrid hash join selects one build partition to remain memory-resident before the join begins. Any available memory beyond what is needed for partitioning is used to reduce the number of I/O operations performed. Dynamic hash join (DHJ) (DeWitt and Naughton, 1995; Kitsuregawa et al., 1989) can adapt to memory changes by initially keeping all build partitions in memory and then flushing on demand as memory is required. Hash join optimizations (Graefe, 1992) include bit vector filtering and role reversal.

Skew occurs in data when certain values occur more frequently than others. Skew can be classified (Walton et al., 1991) as either *partition skew* or *intrinsic data skew*. Partition skew is when the partitioning algorithm constructs partitions of non-equal size (often due to intrinsic data skew but also due to the hash function itself). Partition skew can be partially mitigated by using many more partitions than required, as in DHJ (Kitsuregawa et al., 1989; Nakayama et al., 1988), and by producing histograms on the data when recursive partitioning is required.

2.2 Multi-Way Joins

A multi-way join can join two or more relations at the same time. The common issues in all multi-way join implementations are the hash table structure, the probe ordering, and the join types supported.

The hash table structure must support the ability to partition the input relations such that only tuples at the same partition index can join together. Each input typically has its own hash table and associated partition file buffers. A multi-way operator that has multiple hash tables can use the memory available to buffer tuples from any input. Internally, the algorithm must select a probe ordering. The probe ordering specifies the order of inputs to probe given a tuple of one input. The probe ordering may differ based on the input tuple and may adapt as the join progresses.

Not all joins can be efficiently executed using multi-way hash joins. If all inputs cannot fit in memory, the only join plans that can be executed using one partitioning step are those where the inputs all have common hash attributes or joins where indirect partitioning is possible. One partitioning step is sufficient as the cleanup can be performed by loading all partitions at the same index in memory and then probing. Star joins can be processed using multi-dimensional partitioning. Multi-dimensional partitioning requires the build relations be read multiple times during the cleanup phase, but this still may be more efficient than the corresponding binary plans. The rest of this section provides background on existing multi-way join implementations.

2.2.1 Hash Teams

Hash teams (Graefe et al., 1998a) used in Microsoft SQL Server 7.0 perform a multi-way hash join where the inputs share common hash attributes. A hash team consists of hash operators and a team manager that coordinates the memory management for all inputs. Performance gains of up to 40% were reported. The probe ordering is the same as the original binary plan. The team manager is separate from the regular plan operators. The operators are binary and are co-ordinated externally by the team manager.

2.2.2 Generalized Hash Teams

Hash teams were extended to generalized hash teams (Kemper et al., 1999) that allows tables to be joined using indirect partitioning. Indirect partitioning partitions a relation on an attribute that functionally determines the partitioning attribute. A TPC-H (TPC, 2013) query joining the relations *Customer*, *Orders*, and *LineItem* can be executed using a generalized hash team that partitions the first two relations on *custkey* and probes with the *LineItem* relation by using its *orderkey* attribute to indirectly determine a partition number using a mapping constructed when partitioning *Orders*. This mapping provides a partition number given an *orderkey*.

The major issue with indirect partitioning is that storing an exact representation of the mapping function is memory-intensive. In (Kemper et al., 1999), bitmap approximations are used that consume less space but introduce the possibility of mapping errors. These errors do not affect algorithm correctness but do affect performance. The bitmap approximation works by associating a bitmap of a chosen size with each partition. A key to be stored or queried with the mapping function is hashed based on the bitmap size to get an index I in the bitmap. The bit at index I is set to 1 in the bitmap for the partition where the tuple belongs. Note that due to collisions in the hashing of the key to the bitmap size, it is possible for a bit at index I to be set in multiple partition bitmaps which results in *false drops*. A false drop is when a tuple gets put into a partition where it does not belong.

The generalized hash team algorithm does not have a "hybrid step" where it uses additional memory to buffer tuples beyond what is required for partitioning. Further, the bitmaps must be relatively large (multiples of the input relation size) to reduce the number of false drops. Consequently, even the bitmap approximation is memory intensive as the number of partitions increases.

2.2.3 SHARP

SHARP (Bizarro and DeWitt, 2006) is a multi-way join algorithm for star joins that performs multi-dimensional partitioning. An example TPC-H star query involves *Part*, *Orders* and *LineItem*. In multi-dimensional partitioning, *Part* and *Orders* are the build tables and are partitioned on *partkey* and *orderkey* respectively. *LineItem* is the probe relation and is partitioned simultaneously on *(partkey,orderkey)* (in two dimensions). The number of partitions of the probe table is the product of the number of partitions in each build input. For example, if *Part* was partitioned into 3 partitions and *Orders* partitioned into 5 partitions, then *LineItem* would be partitioned into 5*3=15 partitions.

For a tuple to be generated in the memory phase, the tuple of *LineItem* must have both its matching *Part* and *Orders* partitions in memory. Otherwise, the probe tuple is written to disk. The cleanup pass involves iterating through all partition combinations. The algorithm loads on-disk partitions of the probe relation once and on-disk partitions of the build relation i a number of times equal to $\prod_{j=1}^{i-1} X_j$, where X_i is the number of partitions for build relation i. Reading build partitions multiple times may still be faster than materializing intermediate results, and the operator benefits from memory sharing during partitioning and the ability to adapt during its execution.

2.3 Other Join Algorithms

There are two other related but distinct areas of research on join algorithms. There is work on parallelizing main-memory joins based on hashing (Blanas et al., 2011) and sorting (Albutiu et al., 2012). The assumption in these algorithms is that there is sufficient memory for the join inputs such that the dominate factor is CPU time rather than I/O time. In this work, we are examining joins where I/O is still a major factor. The main-memory and cache-aware optimization techniques employed in these works could also be applied to I/O bound joins.

Another related area is performing multi-way joins on Map-Reduce systems such as in (Zhang et al., 2012; Afrati and Ullman, 2011). Map-Reduce systems are designed for very large-scale queries over a commodity cluster. The join algorithms used and how they apply to multi-way joins are distinct from traditional relational database systems.

3 IMPLEMENTING HASH TEAMS

We have implemented several variations of generalized hash teams. All implementations support both direct and indirect partitioning. The different implementation features can be classified in three areas:

- **Partitioning** - The standard generalized hash team algorithm does not have a hybrid step. Our implementation calculates the expected number of partitions required and uses a multiple of this number to partially compensate for skew. Dynamic partition flushing allows a “hybrid” component to improve performance.
- **Materialization** - An algorithm may either use lazy materialization of intermediate results (Bizarro and DeWitt, 2006; Lawrence, 2008) where no intermediate tuples are generated or eager materialization by generating all intermediate tuples.
- **Mapping** - The algorithm uses either an exact mapping or bit mapping for indirect partitioning.

These features are discussed in the following sections. This discussion only applies to the implementation of hash teams. The implementation of SHARP is distinct and follows closely to (Bizarro and DeWitt, 2006) except that dynamic probe orderings and adaptive features are not implemented.

3.1 Partitioning

The partitioning algorithm must divide tuples such that only tuples at the same partition index can join together. The algorithm must also guarantee that for each partition index the partitions for the $N-1$ build inputs can be memory resident at the same time during cleanup in order to be probed by the probe input. The minimum number of partitions (with lazy materialization), P_{min}, required is:

$$P_{min} = \frac{\sum_{i=1}^{N-1} |R_i| * size(R_i)}{(M - overhead)} \tag{1}$$

$|R_i|$ is the number of tuples of input R_i, and $size(R_i)$ is the average size in bytes of each tuple. M is the join memory size in bytes and N is the total number of inputs in the join. *Overhead* is the overhead in bytes for mapping functions which consume memory available for the join (details in Section 3.3). The partitioning is done in order of increasing size of the build relations. Determining a partition ordering for indirect partitioning joins is similar except that a partial ordering must be respected such that an input cannot be partitioned before the input that generates its mapping is partitioned. Consequently, the last relation in the functional dependency chain (which we refer to as the *determinant* relation) is always the last partitioned relation.

Similar to DHJ (DeWitt and Naughton, 1995), the number of partitions P is a multiple of the minimum required (currently $P = 4 * P_{min}$). This will partially compensate for skew and allow the in-memory partitions to be determined dynamically. Additional partitions and dynamic de-staging do not solve all partition skew issues. Similar to binary joins, recursive partitioning can be applied on a partition where all build relations cannot fit in memory simultaneously.

3.2 Materialization

There are two approaches to generating output tuples. *Eager materialization* builds intermediate tuples for each binary join in the operator. When joining the relations *Customer*, *Orders*, and *LineItem*, this would result in materializing the intermediate relation *Customer-Orders* before probing with *LineItem*. In general, eager materialization is not a good choice because of the cost of generating the intermediate relations. Further, partitioning becomes more difficult as estimating the size of the intermediate relations may be difficult, and they must fit in memory during cleanup. The benefit is that the standard binary join code can be used unchanged after partitioning is complete, and every intermediate tuple is only materialized once.

With *lazy materialization* (Bizarro and DeWitt, 2006; Lawrence, 2008) pointers to the component tuples are kept and only when all components tuples are available is a result generated. Lazy materialization saves the cost of materializing intermediate tuples and requires fewer partitions (and less risk of partition skew) during partitioning. However, this is the classic time versus space trade-off. Space is saved by not materializing intermediate relations with a trade off in time because a given combination of component tuples may need to be generated multiple times. For instance, a given pair of *Customer* and *Orders* tuples with matching *custkey* is not materialized as in a binary plan, however it must be rebuilt every time for the corresponding *LineItem* tuples. Note that lazy materialization is ideal for SHARP or one-to-one cardinality joins as the join would never materialize intermediate tuples more than once.

3.3 Mapping

Indirect partitioning requires the construction and use of mappers that given an attribute in the relation being indirectly partitioned return the partition where that tuple belongs. In addition to the bit mapper proposed in (Kemper et al., 1999), we implemented a simple exact mapper. The exact mapper records an integer hash value (or the join attribute if it is an integer key) as a lookup value and the data value is the hash key of the partitioning attribute. This reduces the number of false drops to only instances of collisions of hash values. The exact mapper is implemented as a hash table. If a mapping lookup fails to find an entry, that implies the mapping was never generated and the tuple requesting the mapping can be discarded. Each mapping occupies approximately 12 bytes of space (4 for lookup hash value, 4 for data hash value, and 4 for pointer in hash table structure).

In certain cases, the exact mapper is more space efficient than an array of bitmaps (one for each partition) when the number of partitions increases. For one indirect partitioning step, the percentage of false drops (Kemper et al., 1999) for a bit mapper is $\frac{p-1}{p} * \frac{t-1}{b}$ where p is the number of partitions, t is the size in tuples of the mapping generator relation, and b is the length of the bitmap. The number of false drops increases with the number of partitions p and is directly proportional to the ratio of the mapping relation size to the bitmap size. For a reasonable percentage of false drops, the bitmap size must be 4 or 8 times the number of tuples in the mapping relation.

Consider indirectly partitioning *LineItem* with a mapping built from *Orders.orderkey*. Let $t = 15$ million tuples of *Orders* and $p = 8$. Assume the bit vector is 4 times the size of orders, i.e. $b = 60$ million bits. Then the percentage of false drops is $\frac{7}{8} * \frac{15}{60} = 21.9\%$. Since we need a bitmap for each partition, the total mapping cost is $p * b = 480$ million bits. An exact mapper storing the hash value of *Orders.orderkey* for lookup and a data member that is the hash value of *Orders.custkey* would occupy 12 bytes of space (size of two integers plus a pointer). Based on the previous example, the cost of this mapper is 960 million bits with no false drops. For $p > 16$, the exact mapping consumes less space than the bitmap approach.

The mapper size limits the use of indirect partitioning as the mapper must fit in memory to be worthwhile. Otherwise, the random I/Os incurred by mapping destroy any cost advantage of the multi-way join.

4 CONSTRUCTING MULTI-WAY PLANS USING POST-OPTIMIZATION

The post-optimization approach to create multi-way join plans occurs after an optimized logical query tree has been produced. During conversion into an execution plan, hash join nodes are examined to see if they can be merged into multi-way join nodes with their children. In the greedy algorithm in Figure 2, a hash join node is considered for conversion into a multi-way join node if one or both of its children are either a binary hash join or a multi-way hash join. A multi-way join is only created if all of the children of the given node can be combined into a multi-way join operator. Only three possibilities must be considered (see Figure 1):

- Merge with left child hash node - The inputs are the inputs of the left child and the right input.
- Merge with right child hash node - The inputs are the inputs of the right child and the left input.
- Merge with both children hash nodes - The inputs are all the inputs of both children.

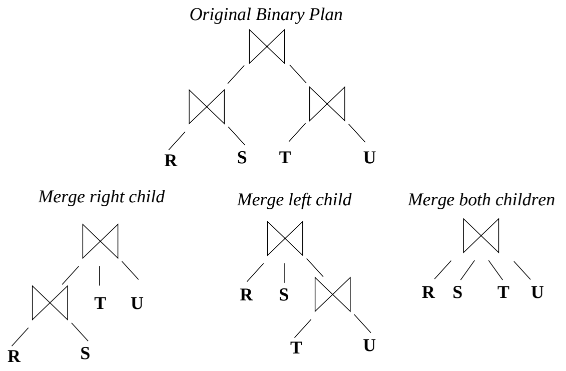

Figure 1: Building a Multi-Way Join via Merging

The method `buildNWayNode` takes an array of logical query tree input nodes and attempts to construct a multi-way hash join execution node. The inputs are analyzed to see if they fall into the three query types supported:

- Direct partitioning join - All inputs must have the attributes required by the hash function.

```
PlanNode* convertHashNode(HashNode *current)
// Given a HashNode in the logical query tree converts it into a
// binary or multi-way hash join plan (execution) node.
{
    Node* left = current→getLeft();
    Node* right = current→getRight();
    int leftType = left→getType();
    int rightType = right→getType();
    double binaryCost = current→getCost();
    double bestCost = binaryCost;
    PlanNode* bestPlan = null;
    NWayNode* nway;

    if (leftType is binary or nway hash join)
    { // Merge with left child
        nway = buildNWayNode(left.getChildren(),right);
        if (nway != null)
            if (nway→getCost() < bestCost)
            {   bestPlan = nway;
                bestCost = nway→getCost();
            }
    }
    if (rightType is binary or nway hash join)
    { // Merge with right child
        nway = buildNWayNode(left, right.getChildren());
        if (nway != null)
            if (nway→getCost() < bestCost)
            {   bestPlan = nway;
                bestCost = nway→getCost();
            }
    }
    if (leftType and rightType are binary or nway hash joins)
    { // Merge with both children
        nway = buildNWayNode(left.getChildren(), right.getChildren());
        if (nway != null)
            if (nway→getCost() < bestCost)
            {   bestPlan = nway;
                bestCost = nway→getCost();
            }
    }
    if (bestPlan != null)
    { // One of the multi-way plans is better
        Construct a multi-way join node for the logical query tree.
        Update pointer in parent node to point to new multi-way join
        node as we may do merge recursively.
        return nway; // Return multi-way join execution node
    }
    Otherwise convert to binary hash join node as usual
}
```

Figure 2: Post-Optimization Multi-Way Join Algorithm

- Indirect partitioning join - All inputs must have the partitioning attributes or have attributes that functionally determine the partitioning attributes.
- Star join - One of the inputs must be a relation that joins with all the other inputs.

The approach to determine if a multi-way join is possible is to start with a single join clause and iteratively add one join clause at a time and evaluate if the join is still one of the three types supported. It is possible for a join to be both a direct partitioning and a star join. In that case, the direct partitioning join is selected as it is more efficient.

The advantage of the post-optimization approach is that it requires fewer modifications to the query optimizer and can be performed in time linear with the number of hash joins in the plan. The post-optimization conversion replaces sequences of binary joins with more efficient multi-way joins for execution but does not alter the query plan in any other way. Thus, if the original binary plan is good, the multi-way should be better if the multi-way operator executes more efficiently than the sequence of binary operators.

5 IMPLEMENTING MULTI-WAY JOINS IN POSTGRESQL

Adding the multi-way join algorithms, generalized hash teams and SHARP, involved the addition of six source code files. Each join had a file defining its hash table structure and operations and a file defining the operator in iterator form. Generalized hash teams (GHT) had two mapper implementations: exact mapper and bit mapper.

In comparison to implementing the join algorithms themselves, a much harder task was modifying the optimizer and execution system to use them. The basic issue is both of these systems assume a maximum of two inputs per operator, hence there are many changes required to basic data structures to support a node with more than two inputs. The changes can be summarized as follows:

- Create a multi-way hash node structure for use in logical query trees and join optimization planning.
- Create a multi-way execution node that stores the state necessary for iterator execution.
- Modify all routines associated with the planner that assume two children nodes including EXPLAIN feature, etc.
- Create multi-way hash and join clauses (*quals*) from binary clauses.
- Create cost functions for the multi-way joins that conform to PostgreSQL cost functions which include both I/O and CPU costs.
- Modify the mapping from logical query trees to execution plan (specifically for hash join nodes as given in Figure 2) to support post-optimization creation of multi-way join plans.

The changes were made as general as possible. However, there are limitations on what queries can be successfully executed with multi-way joins.

6 EXPERIMENTAL RESULTS

The experiments were executed on a dual processor AMD Opteron 2350 Quad Core at 2.0 GHz with 32GB of RAM and two 7200 RPM, 1TB hard drives running 64-bit SUSE Linux. Similar results were demonstrated when running the experiments on a Windows platform. PostgreSQL version 8.3.1 was used, and the source code modified as described.

The data set was TPC-H benchmark (TPC, 2013) scale factor 10 GB[1] (see Figure 3) generated using Microsoft's TPC-H generator (Chaudhuri and Narasayya,), which supports generation of skewed data sets with a Zipfian distribution. The results are for a skewed data set with z=1. Experiments tested different join memory sizes configured using the `work_mem` parameter. The memory size is given on a per join basis. Multi-way operators get a multiple of the join memory size. For instance, a three-way operator gets 2*`work_mem` for its three inputs.

Relation	Tuple Size	#Tuples	Relation Size
Customer	194 B	1.5 million	284 MB
Supplier	184 B	100,000	18 MB
Part	173 B	2 million	323 MB
Orders	147 B	15 million	2097 MB
PartSupp	182 B	8 million	1392 MB
LineItem	162 B	60 million	9270 MB

Figure 3: TPC-H 10 GB Relation Sizes

6.1 Direct Partitioning

One experiment was a three-way join of *Orders* relations. The join was on the *orderkey* and produced 15 million results. The results are in Figure 4 (time) and Figure 5 (IOs).

The results clearly show a benefit for a multi-way join with about a 60% reduction in I/O bytes for the join and approximately 12-15% improvement in overall time. The multi-way join performs fewer I/Os by saving one partitioning step. It also saves by not materializing intermediate tuples in memory and by reducing the number of probes performed. The multi-way join continues to be faster even for larger memory sizes and a completely in-memory join.

[1]The TPC-H data set scale factor 100 GB was tested on the hardware but run times of many hours to days made it impractical for the tests.

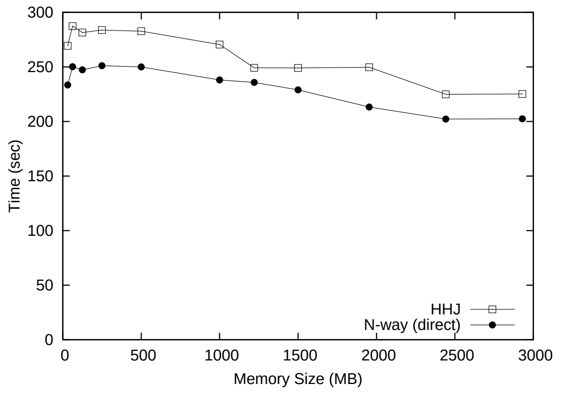

Figure 4: Three Way Orders Join (Time)

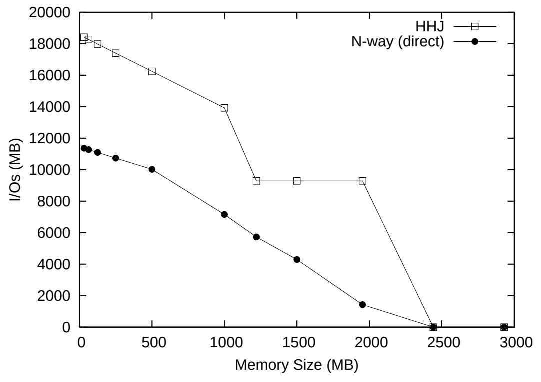

Figure 5: Three Way Orders Join (I/O bytes)

Another direct partitioning join hashes *Part*, *PartSupp* and *LineItem* on *partkey* and joins *PartSupp* and *LineItem* on both *partkey* and *suppkey*. The results are in Figure 6 (time) and Figure 7 (IOs).

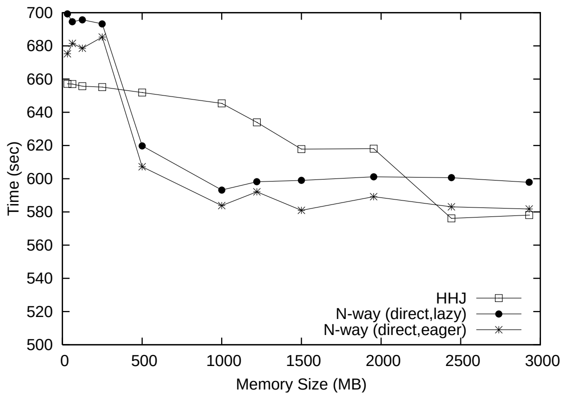

Figure 6: Part-PartSupp-LineItem (Time)

Unlike the one-to-one join, this join exhibited different performance based on the implementations. The original implementation (not shown) had the

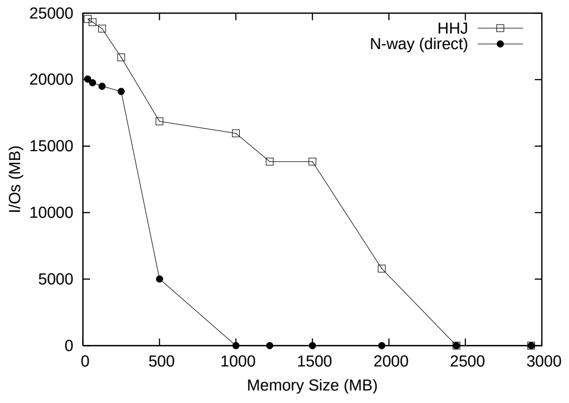

Figure 7: Part-PartSupp-LineItem (I/O bytes)

multi-way join being slower over all memory sizes by 5-20% even though it had performed significantly less I/O. The difference turned out to be significantly more hash and join qualifier (clause) evaluations for the multi-way operator. Several optimizations were made to reduce the number of qualifier evaluations and probes to below that of HHJ. The multi-way join does not have superior performance over all memory sizes. The major improvement in I/Os at 500 MB is due to the multi-way operator sharing memory between the inputs as the smaller input *Part* fits in its 500 MB allocation and can provide an extra 187 MB to buffer *PartSupp* tuples.

To test the potential benefit of eager materialization, we modified the implementation to allow for materialization of intermediate tuples *unconstrained* by memory limitations. Thus, the materialization implementation is unrealistically good as it could exceed the space allocated for the join considerably without paying any extra I/O or memory costs. The result was only a 2% improvement in time.

The clear impact of probing cost on the results motivate the benefit of adaptive probe orders which may improve results. CPU costs are often considered a secondary factor to I/Os for join algorithms, although in practice the costs can be quite significant.

6.2 Indirect Partitioning

Indirect partitioning was tested with a join of *Customer*, *Orders*, and *LineItem*. We tested the original bit mapper with no hybrid component, an exact mapper with a hybrid component, a bit mapper with a hybrid component, and HHJ. The bit mapper with no hybrid component used its entire memory allocation during partitioning for the bit mapper. The hybrid bit mapper used 12 bytes * number of tuples in the *Orders* relation as its bit map size which is the same amount of space used by the exact mapper. The results are in Figure 8 (time) and Figure 9 (IOs).

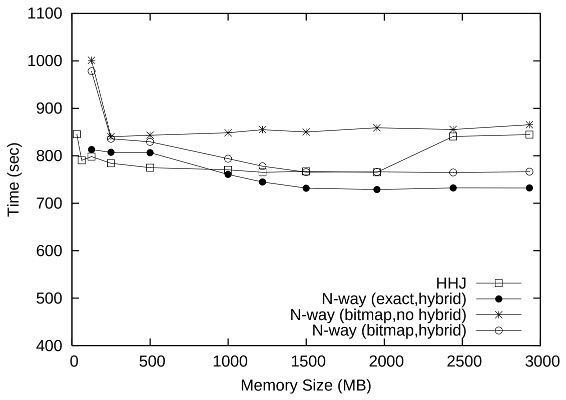

Figure 8: Customer-Orders-LineItem (Time)

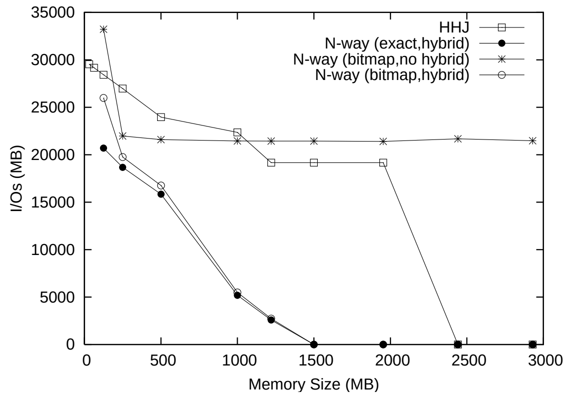

Figure 9: Customer-Orders-LineItem (I/O bytes)

For this join, the multi-way algorithms had fewer I/Os but that did not always translate to a time advantage unless the difference was large. The hybrid stage is a major benefit as the join memory increases. HHJ had worse performance on a memory jump from 2000 MB to 2500 MB despite performing 20GB fewer I/Os! The difference was the optimizer changed the query plan to join *Orders* with *LineItem* then the result with *Customer* at 2500 MB where previously *Customer* and *Orders* were joined first. This new ordering produced double the number of probes and join clause evaluations and ended up being slower overall.

The major limitation was the mapper size. The mappers did not produce results for the smaller memory sizes of 32 MB and 64 MB as the mapper could not be memory-resident. For 128 MB, the bit mapper performed significantly more I/Os and had larger time than the exact mapper due to the number of false drops. The number of false drops was greatly reduced as the memory increased. The bit mapper without a hybrid component continued to read/write all relations and was never faster than HHJ.

6.3 Multi-Dimensional Partitioning

One of the star join tests combined *Part*, *Orders*, and *LineItem*. The performance of the SHARP algorithm versus hybrid hash join is in Figures 10 and 11. SHARP performed 50-100% fewer I/Os in bytes and was about 5-30% faster. Only at very small memory sizes did the performance become slower than HHJ, and it was faster in the full memory case.

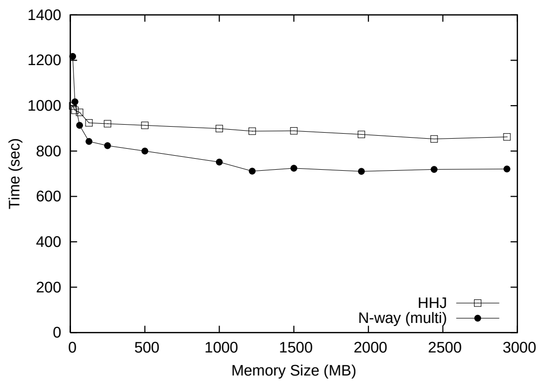

Figure 10: Part-Orders-LineItem (time)

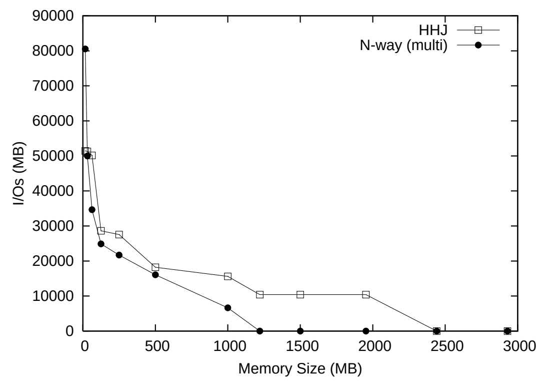

Figure 11: Part-Orders-LineItem (I/O bytes)

6.4 Results Discussion

Multi-way joins can be added to the optimizer using post-optimization. Post-optimization has a low barrier to entry and catches many of the opportunities for exploiting multi-way joins. The cost functions allow the optimizer to choose between the binary and multi-way operators.

Implementing efficient, robust, and scalable multi-way joins is a non-trivial challenge. The experimental results clearly show that multi-way performance depends on the join type. Direct partitioning joins are efficient and are clearly superior when the hash attributes uniquely identify tuples in each input. However in that case, it is also likely that interesting orders based on sorting may apply (as the relations may be sorted on the primary/unique attribute) which would have even better performance. Direct partitioning joins where a tuple in one input may match with many in the other inputs has better performance in some cases primarily due to sharing memory over all inputs, but despite numerous optimizations, we have been unable to demonstrate improved performance over all memory sizes, especially smaller memory sizes. Lazy materialization is superior to eager materialization as it saves both memory and I/Os for materializing intermediate results and also the time to construct and probe them in intermediate hash tables.

Indirect partitioning has benefits, but the mapping functions consume both space and CPU time. Indirect partitioning is inapplicable if the mappers cannot be completely memory-resident during partitioning. Indirect partitioning without a hybrid step is not competitive with binary plans. Note that this does not contradict the results in (Kemper et al., 1999) as the join algorithm was not used in conjunction with the space saving hash aggregation structure proposed. The results clearly show both a benefit and an issue with direct and indirect partitioning joins. The number of I/Os performed is less but that does not always translate into a time advantage. The I/O cost is often reduced by intelligent operating system buffering. Overall, indirect partitioning joins are not as robust as binary joins and the high CPU cost often outweighs any I/O advantages. Similar to main-memory optimized joins, optimizing algorithms for cache-awareness is very beneficial for performance. A cache-aware multi-way join may have even better performance.

Multi-dimensional partitioning as implemented in SHARP is a much more consistent winner. The major bad case relates to the "curse of dimensionality" when build inputs are read multiple times and more I/Os are performed than the corresponding binary plan. This case is identified by the cost functions and will be avoided by the optimizer. Like the one-to-one direct partitioning joins, multi-way partitioning is faster than binary plans for the fully in-memory case. This speed improvement occurs as it does not materialize intermediate results and performs the same (or fewer) number of probes than binary plans. Further, star queries are much more common than the types of queries that would be beneficial for direct or indirect partitioning. Multi-dimensional partitioning is a beneficial addition to the set of join operators and in its most basic implementation (without any adaptability) is fairly straightforward to implement.

7 CONCLUSIONS

The goal of this paper was to determine if multi-way joins are useful in a database system. The answer is yes, especially for star queries. Direct and indirect partitioning multi-way joins improve performance in some cases, especially for one-to-one joins. The two major issues are the relatively limited number of queries affected, and the care that must be taken to guarantee good performance. Multi-dimensional partitioning is beneficial for a larger number of queries, more efficient except for known cases, and more stable to implement. Multi-dimensional partitioning joins should be implemented in commercial database systems.

Future work includes allowing the optimizer to cost multi-way joins simultaneously with binary plans and conducting experiments to evaluate multi-way joins in conjunction with aggregation operators.

REFERENCES

Afrati, F. N. and Ullman, J. D. (2011). Optimizing Multiway Joins in a Map-Reduce Environment. *IEEE Trans. Knowl. Data Eng.*, 23(9):1282–1298.

Albutiu, M.-C., Kemper, A., and Neumann, T. (2012). Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *PVLDB*, 5(10):1064–1075.

Bizarro, P. and DeWitt, D. J. (2006). Adaptive and Robust Query Processing with SHARP. Technical Report Technical Report 1562, University of Wisconsin.

Blanas, S., Li, Y., and Patel, J. M. (2011). Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD Conference*, pages 37–48.

Chaudhuri, S. and Narasayya, V. TPC-D data generation with skew. Technical report, Microsoft Research, Available at: *ftp.research.microsoft.com/users/viveknar/tpcdskew*.

DeWitt, D., Katz, R., Olken, F., Shapiro, L., Stonebraker, M., and Wood, D. (1984). Implementation Techniques for Main Memory Database Systems. In *ACM SIGMOD*, pages 1–8.

DeWitt, D. and Naughton, J. (1995). Dynamic Memory Hybrid Hash Join. Technical report, University of Wisconsin.

Graefe, G. (1992). Five Performance Enhancements for Hybrid Hash Join. Technical Report CU-CS-606-92, University of Colorado at Boulder.

Graefe, G., Bunker, R., and Cooper, S. (1998a). Hash Joins and Hash Teams in Microsoft SQL Server. In *VLDB*, pages 86–97.

Graefe, G., Ewel, J., and Galindo-Legaria, C. (September 1998b). Microsoft SQL Server 7.0 Query Processor at *msdn.microsoft.com/en-us/library/aa226170(SQL.70).aspx*. Technical report, Microsoft Corporation.

Kemper, A., Kossmann, D., and Wiesner, C. (1999). Generalised Hash Teams for Join and Group-by. In *VLDB*, pages 30–41.

Kitsuregawa, M., Nakayama, M., and Takagi, M. (1989). The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method. In *VLDB*, pages 257–266.

Lawrence, R. (2008). Using Slice Join for Efficient Evaluation of Multi-Way Joins. *Data and Knowledge Engineering*, 67(1):118–139.

Microsoft Corporation (May 2001). Description of Service Pack 1 for SQL Server 2000 at *http://support.microsoft.com/kb/889553*. Technical report, Microsoft Corporation.

Moerkotte, G. and Neumann, T. (2008). Dynamic programming strikes back. In *ACM SIGMOD*, pages 539–552.

Nakayama, M., Kitsuregawa, M., and Takagi, M. (1988). Hash-partitioned join method using dynamic destaging strategy. In *VLDB*, pages 468–478.

TPC (2013). TPC-H Benchmark. Technical report, Transaction Processing Performance Council.

Viglas, S., Naughton, J., and Burger, J. (2003). Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources. In *VLDB*, pages 285–296.

Walton, C. B., Dale, A. G., and Jenevein, R. M. (1991). A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. In *VLDB*, pages 537–548.

Zhang, X., Chen, L., and Wang, M. (2012). Efficient Multi-way Theta-Join Processing Using MapReduce. *PVLDB*, 5(11):1184–1195.