Exploiting Join Cardinality for Faster Hash Joins

Michael Henderson
University of British Columbia
Okanagan
mikeh3@interchange.ubc.ca

Bryce Cutt
University of British Columbia
Okanagan
brycec@interchange.ubc.ca

Ramon Lawrence
University of British Columbia
Okanagan
ramon.lawrence@ubc.ca

ABSTRACT

Hash joins combine massive relations in data warehouses, decision support systems, and scientific data stores. Faster hash join performance significantly improves query throughput, response time, and overall system performance. In this work, we demonstrate how using join cardinality improves hash join performance. The key contribution is the development of an algorithm to determine join cardinality in an arbitrary query plan. We implemented early hash join and the join cardinality algorithm in PostgreSQL. Experimental results demonstrate that early hash join has an immediate response time that is an order of magnitude faster than the existing hybrid hash join implementation. One-to-one joins execute up to 50% faster and perform significantly fewer I/Os, and one-to-many joins have similar or better performance over all memory sizes.

Categories and Subject Descriptors

H.2.2 [**Physical Design**]: Access Methods; H.2.4 [**Systems**]: Query Processing

Keywords

hybrid hash join, cardinality, response time, symmetric, PostgreSQL

1. INTRODUCTION

Database sizes are growing, and the demand to process ever larger data sets and queries is increasing. Applications such as data warehousing, decision support, and scientific data analysis perform hash joins [3] with massive relations. Improving hash join performance results in significantly faster data analysis and processing.

Recently, "early" joins have been proposed that are capable of producing results before having read and partitioned the entire build relation. Early join algorithms were developed primarily for network-based joins in distributed, grid, and integration systems to compensate for network delays and produce results faster. Despite the potential benefits, these algorithms have not been implemented in a conventional database management system.

We have implemented early hash join (EHJ) [8] in the open source database system PostgreSQL. Besides the ability to generate results early, EHJ has the unique feature that its join performance improves for one-to-one joins. To exploit this feature, it is necessary to develop an algorithm to determine the cardinality of a join in an arbitrary query plan. To our knowledge, no production system uses join cardinalities during cost-based optimization. Experimental results show that EHJ outperforms PostgreSQL's hybrid hash join implementation in terms of response time, I/O operations, and overall time, especially for one-to-one joins.

The contributions of this paper are:

- An algorithm for determining join cardinalities in arbitrary query plans.
- A modification of the PostgreSQL DBMS to support early hash join, including new cost formulas for cost-based optimization.
- Experimental results that show significant benefits of using join cardinality detection with EHJ.

The organization of this paper is as follows. In Section 2, we briefly overview existing work on hash joins including early hash join. Our algorithm for determining join cardinalities is in Section 3. The necessary changes to PostgreSQL are covered in Section 4. Experimental results in Section 5 demonstrate the performance improvements of detecting join cardinalities and exploiting them using EHJ. Finally, the paper closes with conclusions and future work.

2. PREVIOUS WORK

A join combines two relations into a single relation. We will 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) [3] is the common hash join algorithm implemented in most database systems. Hybrid hash join selects one build partition to remain memory-resident before the join begins. Thus, any available memory beyond what is needed for partitioning can be used to reduce the number of I/O operations performed. Dynamic hash join [4, 7] can adapt to memory changes by initially keeping all build partitions in memory and then flushing on demand as memory is required. Other optimizations [5] of hash join algorithms such as bit vector filtering, role reversal, and skew handling are orthogonal to this work.

Symmetric hash joins were proposed for network-based joins in distributed and integration systems. Symmetric hash join has a dual hash table structure that partitions and buffers in memory tuples from both inputs. This allows the algorithm to generate results faster and compensate for network-induced delays in tuple arrival rates. The main-memory version of symmetric hash join [6, 11] was extended to handle relations larger than main memory in XJoin [10] and early hash join (EHJ) [8]. We focus on EHJ since it is the fastest algorithm and has the unique feature that its performance differs based on join cardinality.

Early hash join uses the dual hash table structure that allows a tuple to be processed from either input at any time. When a tuple is processed, its join attributes are passed through a hash function to determine a bucket index. Each bucket index stores a linked list of tuples that hash to that location. The arriving tuple first probes the bucket index in the other side of the hash table to generate results. Then, it is inserted in its side of the hash table. When memory is required, a partition (a group of buckets), is chosen to be flushed to disk. In EHJ, once a bucket of a partition has been flushed to disk, it can no longer receive new tuples in memory. Any tuples arriving to that partition are directly flushed to disk. This has been referred to as being *frozen*. During the cleanup phase of the join, partitions that were flushed to disk are joined together to produce remaining results. For this work, we ignore the background processing feature of EHJ that is not relevant in a centralized system.

An early join algorithm must have a duplicate detection strategy to guarantee that a join result cannot be generated twice (once in the initial in-memory phase and later during the cleanup phase). EHJ's duplicate detection strategy depends on the join cardinality. The join cardinality may be one-to-one (1:1), one-to-many (1:N), or many-to-many (M:N). In a one-to-many join (common with primary key to foreign key joins), when a tuple in the many input finds its only match in the one input, it can be discarded. A *tuple discard* occurs when a tuple currently in the hash table produces a join result and then is removed from the hash table. An *insert avoided* occurs when the tuple was being used as a probe tuple, generated a match, then is discarded and never inserted in the hash table. In a one-to-one join, when any tuple generates an output, it can be discarded. Discarding tuples saves memory and gives the potential to generate more results than hybrid hash join. In the many-to-many case, timestamps are needed to track when tuples arrived in order to avoid duplicate results.

PostgreSQL was chosen as the experimental database system because its open source implementation closely follows conventional practice. PostgreSQL has a heuristic and cost-based optimizer, iterator-based query execution model, and implementations of all joins including hybrid hash join. In comparison, MySQL has a limited optimizer and no hash join implementation. The hybrid hash join implementation in PostgreSQL is almost identical to [3] and has a form of dynamic partitioning to handle poor estimates. PostgreSQL has no support for determining join cardinality. This is expected as hybrid hash join always functions the same regardless of join cardinality.

3. DETERMINING JOIN CARDINALITIES

Our recursive algorithm for determining the cardinality of a join in a query plan is in Figure 1. The join cardinality algorithm relies on an algorithm (Figure 2) to determine the candidate keys of a relation produced by an operator.

```
int getJoinCard(JoinOperator J)
// Returns 0 if 1:1, 1 if 1:N, 2 if N:1, 3 if M:N
{
    Set JK1 = set of join attributes for build relation (input 1)   (1)
    Set JK2 = set of join attributes for probe relation (input 2)   (2)
    Set K1 = getCandidateKeys(J.getChild(1));                       (3)
    Set K2 = getCandidateKeys(J.getChild(2));                       (4)
    if (in(JK1,K1) and in(JK2,K2))                                  (5)
        return 0; // 1:1 (one-to-one)                               (6)
    if (in(JK1,K1))                                                 (7)
        return 1; // 1:N (one-to-many)                              (8)
    if (in(JK2,K2))                                                 (9)
        return 2; // N:1 (many-to-one)                              (10)
    return 3; // M:N (many-to-many)                                 (11)
}

boolean in(Set target, Set keys)
// Returns true if the target set of attributes is a superset of any
// of the keys (sets of attributes) in the Set keys
{
    for (all sets S in keys)                                        (1)
        if (target ⊇ S)                                             (2)
            return true;                                            (3)
    return false;                                                   (4)
}
```

Figure 1: Join Cardinality Algorithm

For example, consider four relations (primary keys underlined): $A(\underline{a})$, $B(\underline{b}, a)$, $C(\underline{c}, b)$, and $D(\underline{a}, \underline{b}, d)$. Two different query plans are given in Figure 3. In these diagrams, the join cardinality is displayed beside each join, and the candidate keys of each operator are also displayed. Since all joins are equi-joins, only the common join attribute is displayed under the join rather than the full join clause. For example, the join of B and D in the first plan would have a join clause of $B.b = D.b$ but only the common attribute b is displayed.

In the first plan, the bottom join of B and D has cardinality 1:N because with the join clause $B.b = D.b$, only input 1 (B) has the join attributes as a superset of its candidate key. The output relation candidate key is (a, b). The join above with A is also 1:N for the same reason. Finally, the join with C is M:N because the join attribute b is not a superset of the candidate keys of either input relation. The candidate key (a, c) of the top output relation is the result of combining the candidate key of the first input, (c), with the candidate key of the second input, (a, b), while removing the join attribute (b). This result can also be seen using functional dependencies. In input 1, $c \rightarrow b$ and in input 2, $a, b \rightarrow a, b, d$. Combining the functional dependencies gives $a, b, c \rightarrow a, b, c, d$ which simplifies to $a, c \rightarrow a, b, c, d$ as $c \rightarrow b$.

In the second join plan, the join of B and D is on (a, b) and is a 1:1 join. The join attributes are a superset of the candidate key attributes of B, and the output candidate keys are $(b), (a, b)$. The join with C is N:1 as the join attribute b is a superset of one of the keys in the probe relation. Finally, the join with A is also N:1 as the join attribute a is the candidate key of A.

```
Set getCandidateKeys(Operator op)
// Returns a set of candidate (unique) keys in the output relation
// produced by Operator op
{
    Set K1, K2;
    if (op.numChildren ≥ 1)
        K1 = getCandidateKeys(op.getChild(1));
    if (op.numChildren == 2)
        K2 = getCandidateKeys(op.getChild(2));
    if (op is a base relation scan on Relation R)
        return set of all unique indexes on R;
    if (op is a projection)
    {
        Set P = project attributes in op
        return $\{k | k \in K1$ and $k \subseteq P\}$;
    }
    if (op is a duplicate elimination/grouping)
        return new Set of grouping attributes;
    if (op is a join with a non-equijoin condition)
        return $\emptyset$;
    if (op is join with only equi-join conditions)
    {
        Set JK1 = set of join attributes for input 1
        Set JK2 = set of join attributes for input 2
        if (in(JK1,K1) and in(JK2,K2)) // 1:1
            return K1 ∪ K2;
        if (in(JK1,K1)) // 1:N
            return K2;
        if (in(JK2,K2)) // N:1
            return K1;
        if (K1 is null or K2 is null)
            return $\emptyset$;
        return $\{(k1 - JK1) \cup (k2 - JK2) | k1 \in K1$
                and $k2 \in K2\}$;
    }
    return K1; // Selection, other operators
}
```

Figure 2: Candidate Key Determination Algorithm

The B and D join in query plan 2 demonstrates an issue in the algorithm implementation. The candidate keys of the output relation listed as $(b), (a, b)$ are more precisely $(B.b), (D.a, D.b)$. With equality on the attributes, this expands to $\{(B.b), (D.b), (D.a, D.b), (D.a, B.b)\}$ and then condenses to $\{(B.b), (D.b)\}$ as the last two are superkeys. At the implementation level each candidate key is represented as a set of attribute indexes in the output relation. For example, the last set would be represented as $\{(1), (4)\}$ ($B.b$ is the first attribute in output relation, and $D.b$ is the fourth attribute).

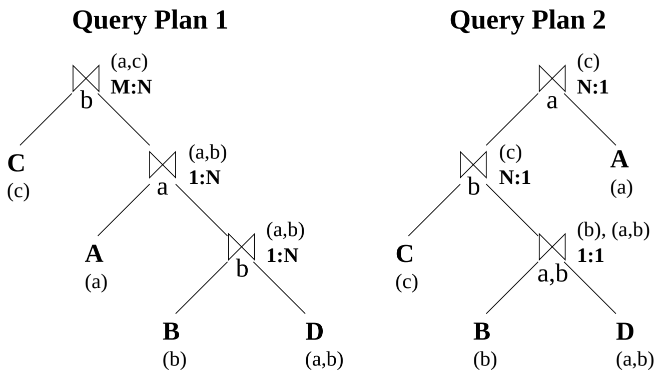

Figure 3: Example Join Plans with Cardinalities

4. SYSTEM MODIFICATIONS

The modifications performed are in three categories: early hash join implementation, changes to the query execution system, and modification of the cost-based optimizer.

4.1 Early Hash Join Implementation

The early hash join algorithm [8] was provided by the authors as a standalone Java implementation. PostgreSQL is written in C, so the algorithm was ported from Java to C. Some features not relevant to the project, such as the background processing thread, were not ported. The major challenge was that a production DBMS has specific APIs for accessing the buffer, reading and writing from temporary files, performing hashing, and allocating and deallocating memory. This resulted in major changes from the original Java code. Further, the dual hash table structure used by early hash join was implemented from scratch using the same APIs as used by PostgeSQL's current hybrid hash join table. These changes resulted in the creation of two files in the `backend/executor` package `nodeEarlyHashJoin.c` (EHJ algorithm) and `dualHashTable.c` (dual hash table structure). The new functions to determine join cardinality were added to the file `selfuncs.c`. Similar to the existing HHJ implementation, partition skew is handled by dynamic repartitioning if a partition becomes too large to fit in memory.

4.2 Query Execution

Another challenge was that early join algorithms are implicitly "push" algorithms as they process tuples as they arrive at either input. This implies that the inputs are producing tuples as a separate process or thread. An iterator-based DBMS, such as PostgreSQL, is inherently "pull" based as an operator requests a tuple from an input below it in the tree and blocks until that tuple arrives. Although EHJ was designed for a centralized system, its iterator implementation was based on the ability to dynamically change inputs when an input was blocked, which is not possible without major modifications to the query execution system.

The compromise implemented is that EHJ has the ability to request tuples from either input, but is forced to block if the iterator below takes time to produce that tuple. This solution was chosen as there has been a conscious effort in the PostgreSQL implementation to avoid creating multiple threads and/or processes within a query execution. To avoid random I/O costs, the EHJ implementation reads multiple pages (e.g. 1000 tuples) at a time from one input before switching to the other. EHJ uses the optimal alternating reading strategy [9] until memory is full. For one-to-one joins, EHJ continues alternating between inputs. For the other joins, EHJ switches to reading all the build relation first then the probe relation (like HHJ). The cost formulas in Section 4.3 are based on these reading strategies.

4.3 Cost-based Optimizer

PostgreSQL has a sophisticated cost-based optimizer that generates and evaluates alternative query plans and selects the plan of lowest cost. The optimizer was modified to allow it to cost plans consisting of early hash join operators in addition to nested loop, sort, and hybrid hash joins. We added an environment variable that allows the user to dynamically turn off EHJ similar to how current join algorithms can be enabled or disabled as desired.

Although a cost function was provided for many-to-many joins for EHJ [8], this cost function does not accurately model the cost of one-to-one and one-to-many joins. We created cost functions to be consistent with the cost of hybrid hash join in PostgreSQL, which considers CPU time as well as I/O cost. PostgreSQL computes both a startup cost and overall cost. EHJ's startup cost is considerably less than HHJ's since it does not need to partition the whole build relation, hence it will be selected over HHJ if the overall costs are the same. The overall cost for EHJ is higher than HHJ for many-to-many joins, about the same for one-to-many joins, and significantly lower for one-to-one joins.

Figure 4 shows simplified cost functions that only consider the number of I/Os performed. $|R|$ is the size of the build relation R (in tuples). $|S|$ is the size of the probe relation S. M is the join memory size. $f = M/|R|$ and is the fraction of the build relation that can remain memory resident. σ is the join selectivity. N is the number of tuples read in total from both inputs before memory is full for the first time. The many-to-many formula is derived from [8]. All formulas do not include input reading costs and were verified by simulations and experiments in PostgreSQL.

Join	Cost Function
HHJ	$2 * (\lvert R\rvert + \lvert S\rvert - f * \lvert R\rvert - \sigma * M * \lvert S\rvert)$
EHJ 1:1	$2 * (\lvert R\rvert + \lvert S\rvert - f * \lvert R\rvert - 2 * gen)$ $gen = gen1 + gen2 + gen3$ $N = (1 - sqrt(1 - 2 * \sigma * M))/\sigma$ $gen1 = \sigma \frac{N^2}{4}$ $gen2 = \sigma * [M * \frac{N}{4} + M * (\frac{N}{2} + M)$ $+M * (gen1 + gen2) * (\lvert R\rvert - M)]$ $gen3 = \sigma * M * (\lvert S\rvert - \lvert R\rvert)$
EHJ 1:N	$2 * (\lvert R\rvert + \lvert S\rvert - f * \lvert R\rvert - gen)$ $gen = gen1 + gen2 + gen3$ $N = (1 - sqrt(1 - \sigma * M))/(\sigma/2)$ $gen1 = \sigma \frac{N^2}{4}$ $gen2 = \sigma(\frac{M+\frac{N}{2}}{2})(\frac{N}{2}) - gen1$ $gen3 = \sigma * M * (\lvert S\rvert - \frac{N}{2})$
EHJ M:N	$2 * (\lvert R\rvert + \lvert S\rvert - f * \lvert R\rvert - \sigma * M * (\lvert S\rvert - 0.5M))$

Figure 4: EHJ Cost Functions

Briefly, the standard HHJ cost function is two times the size of the input relations minus the fraction (f) of the build relation (R) that is memory-resident and the fraction of the probe relation (S) that matches with the in-memory partition of size M. The EHJ many-to-many cost is slightly higher because the initial alternate reading results in half of memory ($0.5M$) being filled with probe tuples that must always be flushed to disk and thus do not have the potential of being discarded when matching with the in-memory build tuples. The EHJ one-to-many and one-to-one costs depend on the number of tuples generated before the cleanup phase as a generated tuple results in tuples being discarded and not flushed to disk. One-to-one joins discard two tuples for every one generated compared to one for one-to-many joins.

The total tuples generated (gen) are divided into three parts: tuples generated before memory is full ($gen1$), tuples generated until the build relation is completely read ($gen2$), and tuples generated until the probe relation is completely read ($gen3$). N used in $gen1$ is a quadratic function that is derived from calculating the point when memory is full considering discards: $M = \frac{N}{2} + \frac{N}{2} - \sigma * \frac{N}{2} * \frac{N}{2}$ (one-to-many version). For one-to-many joins, $gen2$ multiples how many build tuples a probe tuple is expected to see before it is flushed by the $\frac{N}{2}$ build tuples currently in the hash table. $gen3$ is the matches found by comparing the remainder of the probe relation not yet read with the complete fraction of the build relation in memory.

For one-to-one joins, the situation is more complicated because the algorithm alternates throughout and discards can occur on the build relation as well. $gen2$ consists of three terms. The first term is the expected tuples generated when probing with R tuples. This tends to be a smaller value as biased flushing will eventually flush all of S and only leave R tuples in memory. The second term calculates for the first M tuples of S, the average number of tuples of R matched with. The third term estimates the effective number of tuples seen by the next $R - M$ tuples of S. $gen3$ estimates the results produced by probing with the remainder of S.

5. EXPERIMENTAL RESULTS

The experiments were conducted on an Intel Core 2 Quad Q6600 at 2.4GHz (4 core processor) with 8GB of RAM and a 7200 RPM HDD running 64-bit Debian Linux. PostgreSQL version 8.3.1 was used, and the source code modified as described. The data set was TPC-H benchmark [1] scale factor 1 GB (see Figure 5) generated using Microsoft's TPC-H generator [2], which supports generation of skewed data sets with a Zipfian distribution. The results are for a skewed data set with z=1, although the relative difference between the algorithms is unaffected by skew (especially for 1:1 joins which have no skew). Non-skewed data sets gave similar results. All experiments are the average of five runs. In addition to the default tuple ordering, relations were also randomly permuted in some experiments to test different relation orderings.[1] Experiments tested different join memory sizes configured using the `work_mem` parameter. The charts show memory fractions (relative size of `work_mem/build_rel_size` (f in cost functions) as it is the relative size of the build relation to memory that is important, not the absolute sizes.[2]

Relation	Avg. Tuple Size	#Tuples	Relation Size
Customer	196 B	150,000	28 MB
Supplier	184 B	10,000	1.8 MB
Part	173 B	200,000	33 MB
Orders	147 B	1,500,000	210 MB
PartSupp	182 B	800,000	139 MB
LineItem	162 B	6,000,003	926 MB

Figure 5: TPC-H 1 GB Relation Sizes

5.1 Join Cardinality Algorithm

We verified the join cardinality algorithm by testing it with the standard 22 TPC-H queries. These queries contain complex subqueries, aggregation, ordering, and joins. The

[1]A randomly permuted relation is physically materialized and appropriate indexes and keys are created.

[2]Note that a feature of the HHJ implementation is that it can only have a number of partitions that is a power of 2. Thus, memory percentages tested were 6.4, 12.8, 25.6, 51.2, and 100% and also 40, 60 and 80%. For a memory fraction of 40%, PostgreSQL's HHJ will allocate 4 partitions ($\frac{1}{0.4} = 2.5$ rounded up to the nearest power of 2). This wastes the extra memory (as only 1 partition is memory-resident) and causes similar performance to the 25.6% case. This is visible in the figures as periods of flat performance for HHJ. EHJ does not suffer from this issue as it dynamically manages its partitions based on the memory allocated.

join cardinality algorithm successfully determined the join cardinalities in all cases. All joins were either one-to-one or one-to-many in these queries.

5.2 EHJ vs. HHJ Performance

The experiments tested the performance of EHJ versus hybrid hash join (HHJ) and merge join (MJ). The one-to-one join is a self-join of *LineItem* with itself on its primary key (*l_ordernum*, *l_linenumber*). *LineItem* is sorted on these attributes and EHJ performs the join in-memory for all memory sizes (see Figure 6) as it generates all tuples by alternate reading the inputs. HHJ is considerably slower, and even MJ is slower as Postgres forces a sort on the relations as its sequential scan does not guarantee sorted output.[3]

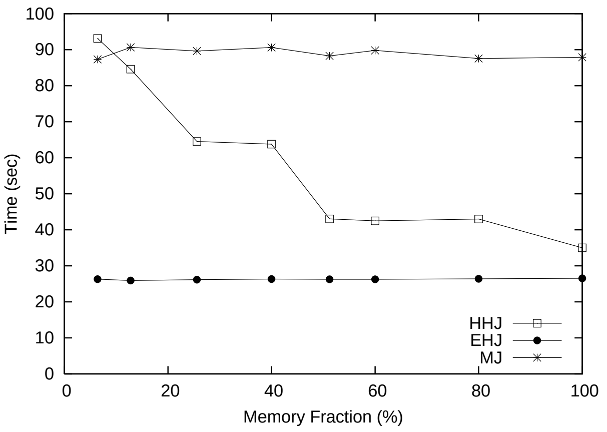

Figure 6: Time for Lineitem Self Join

We created two randomly permuted versions of *LineItem* and performed the self-join (see Figure 7). EHJ produces the first 1000 results in 0.15 to 0.20 sec., compared with 6 to 10 sec. for HHJ and 40 to 60 sec. for MJ. EHJ has an immediate response time whereas HHJ must partition the build relation and MJ must sort both relations. Figure 8 shows the difference in I/Os between EHJ and HJ.

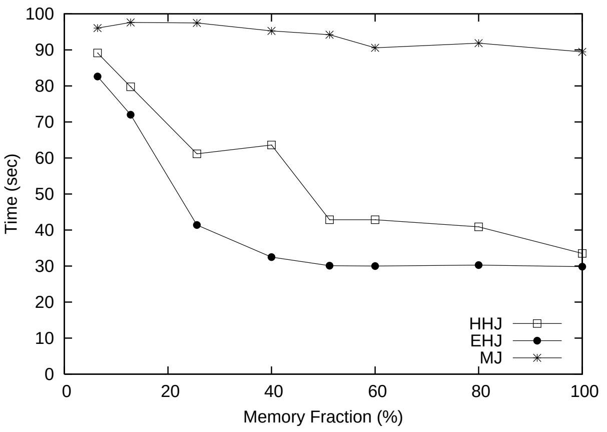

Figure 7: Time for LineItem Self Join (random)

[3]By forcing the optimizer to select index scans over the default sequential scans, merge join has similar performance to EHJ as it does not sort the input relations.

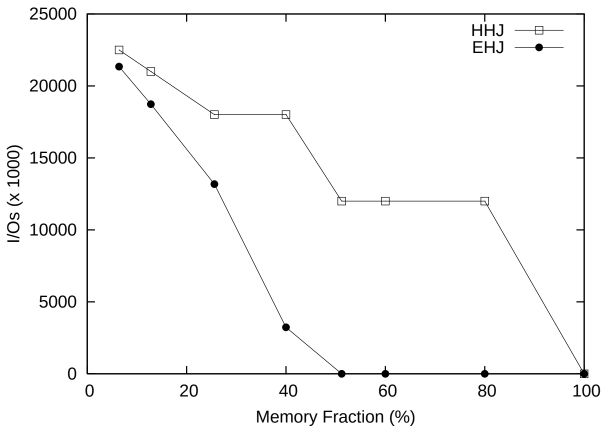

Figure 8: Join I/Os for Lineitem Self Join (random)

The one-to-many join tested is *Orders* and *LineItem* on *orderkey*. EHJ is faster for both the default (sorted) ordering (Figure 9) and when randomizing *LineItem* (Figure 10). EHJ produces the first 1000 results in 0.02 to 0.06 sec. compared with 1.5 to 2 sec. for HHJ and 20 to 30 sec. for MJ. EHJ is faster than HHJ as it performs fewer probes due to its hash table organization.

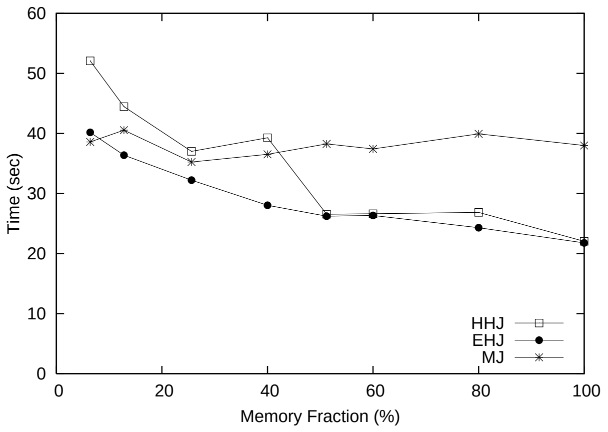

Figure 9: Time for Orders-LineItem Join

The many-to-many join was *LineItem* self-join on *orderkey* (see Figure 11) with randomly permuted relations. EHJ has a faster response time, but is now performing slightly more I/Os than HHJ. Its overall time remains competitive.

For the TPC-H query set, EHJ is slightly faster than HHJ for queries with one-to-many joins that are larger than main memory. EHJ's response time is noticeably faster for queries without ordering and aggregation. The most dramatic effect is for queries with 1:1 joins. Query number 18 (below) of the standard 22 queries contains a 1:1 join between *Orders* and *LineItem* aggregated on *l_orderkey*.

```
SELECT c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice,
sum(l_quantity) FROM customer, orders, lineitem
WHERE c_custkey = o_custkey and o_orderkey = l_orderkey and
o_orderkey IN (SELECT l_orderkey FROM lineitem
    GROUP BY l_orderkey HAVING sum(l_quantity) > 50)
GROUP BY c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice
ORDER BY o_totalprice desc, o_orderdate
```

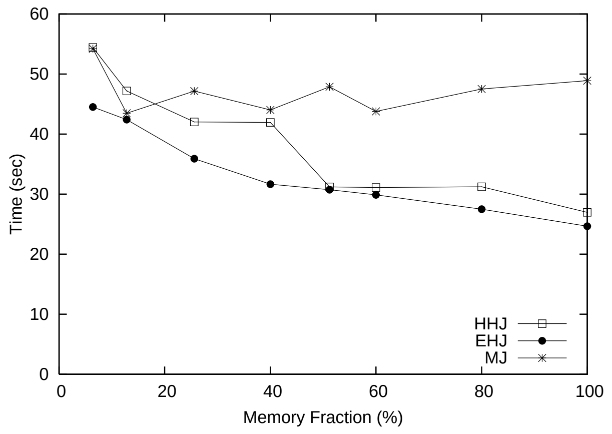

Figure 10: Time for Orders-LineItem Join (random)

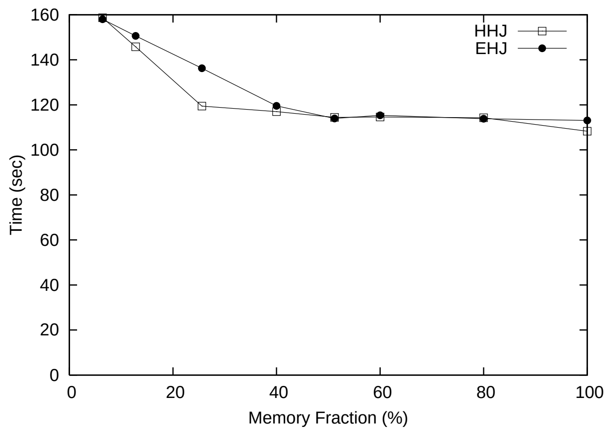

Figure 11: Time for LineItem M:N Join (random)

EHJ is 5 to 30% faster than HHJ (see Figure 12).[4]

In summary, EHJ has an immediate response time for all join cardinalities that is an order of magnitude faster than HHJ. One-to-one joins are considerably faster for all memory sizes due to the ability to discard matching tuples which reduces I/Os. EHJ has similar or better performance for one-to-many joins. EHJ excels for relations that are in sorted or near sorted order as it can exploit alternate reading between inputs to generate more results in memory. For many-to-many joins, the overhead of timestamps and more I/Os result in marginally slower performance. However, many-to-many joins are very rare in practice.

6. CONCLUSIONS

This paper described the implementation of early hash join in PostgreSQL. We developed a new join cardinality algorithm and EHJ cost formulas in addition to implementing the EHJ algorithm. EHJ is faster and more efficient for one-to-one and one-to-many joins, and nicely integrates into the cost-based optimizer. Detecting join cardinality has other benefits in query optimization (such as for selectivity and join size estimation) that can also be exploited. Future work includes submitting the code to the PostgreSQL community for potential inclusion into a future release. We are also examining how the ability to read from any input may allow EHJ to improve performance by pro-actively requesting pages that have been buffered by other queries.

[4]Absolute memory sizes are shown as memory fraction is undefined for queries with multiple joins.

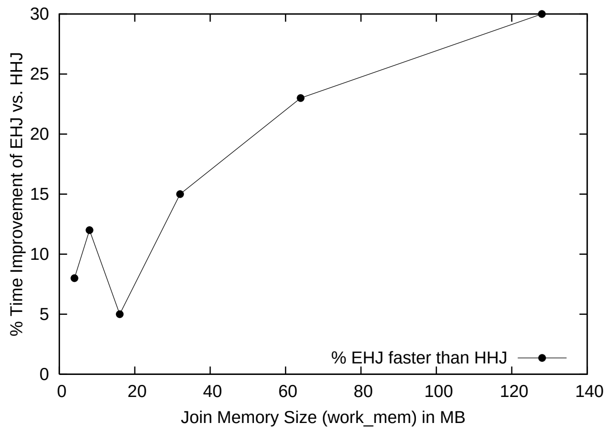

Figure 12: Time for TPC-H Query 18

7. REFERENCES

[1] TPC-H Benchmark. Technical report, Transaction Processing Performance Council.

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

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

[4] D. DeWitt and J. Naughton. Dynamic Memory Hybrid Hash Join. Technical report, University of Wisconsin, 1995.

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

[6] W. Hong and M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. *Distributed and Parallel Databases*, 1(1):9–32, 1993.

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

[8] R. Lawrence. Early Hash Join: A Configurable Algorithm for the Efficient and Early Production of Join Results. In *VLDB*, pages 841–842, 2005.

[9] R. Lawrence, R. P. Russo, and N. D. Shyamalkumar. The Effect of Reading Policy on Early Join Result Production. *Information Sciences*, 177(19):3939–3956, Oct. 2007.

[10] T. Urhan and M. Franklin. XJoin: A Reactively Scheduled Pipelined Join Operator. *IEEE Data Engineering Bulletin*, 23(2):7–18, 2000.

[11] A. N. Wilschut and P. M. G. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. In *PDIS*, pages 68–77, 1991.