Forecasting the Cost of Processing Multi-join Queries via Hashing for Main-memory Databases

Feilong Liu

Computer Science & Engineering
The Ohio State University
liu.3222@osu.edu

Spyros Blanas

Computer Science & Engineering
The Ohio State University
blanas.2@osu.edu

Abstract

Database management systems (DBMSs) carefully optimize complex multi-join queries to avoid expensive disk I/O. As servers today feature tens or hundreds of gigabytes of RAM, a significant fraction of many analytic databases becomes memory-resident. Even after careful tuning for an in-memory environment, a linear disk I/O model such as the one implemented in PostgreSQL may make query response time predictions that are up to $2\times$ slower than the optimal multi-join query plan over memory-resident data. This paper introduces a memory I/O cost model to identify good evaluation strategies for complex query plans with multiple hash-based equi-joins over memory-resident data. The proposed cost model is carefully validated for accuracy using three different systems, including an Amazon EC2 instance, to control for hardware-specific differences. Prior work in parallel query evaluation has advocated right-deep and bushy trees for multi-join queries due to their greater parallelization and pipelining potential. A surprising finding is that the conventional wisdom from shared-nothing disk-based systems does not directly apply to the modern shared-everything memory hierarchy. As corroborated by our model, the performance gap between the optimal left-deep and right-deep query plan can grow to about $10\times$ as the number of joins in the query increases.

1. Introduction

Ad-hoc analytical queries commonly perform a series of equi-joins between different tables. A database management system needs to accurately forecast the response time of different join evaluation orders to provide timely answers to

these queries. A major component of this prediction is an estimation of the cost to access the data. These I/O cost models have traditionally assumed that all data reside in a single directly-attached hard disk. However, cloud infrastructure providers such as Amazon EC2 offer memory-optimized instance types with dozens of vCPUs and hundreds of gigabytes of memory. In this environment, a significant fraction of the database fits in memory and must be processed in parallel by multiple CPU cores for timely responses to user queries.

The database research community has been investigating query processing techniques for memory-resident datasets for more than three decades [19, 29] and has developed detailed cost models for single-core CPUs with deep cache hierarchies [38, 39]. The commoditization of multi-core, multi-socket systems have rekindled research interest in query execution [11, 27] and efficient processing techniques for modern hardware [32, 42]. In-memory hash-based join algorithms, in particular, have received significant attention [4, 8, 30]. The ramifications of ample single-node parallelism for multi-join query evaluation over in-memory data, however, are not as well understood.

As a consequence, systems implementers find themselves in the dark regarding how to best synthesize query plans for complex multi-join queries that efficiently use all CPU cores, and turn to prior research in parallel query evaluation for parallel databases with a shared-nothing architecture. A multi-join query is commonly evaluated as a series of hash-joins between two tables. For every hash join, the query optimizer chooses which input will be stored in the hash table (henceforth referred to as the *build* or *left* side) and which input will be used to probe the hash table (henceforth referred to as the *probe* or *right* side). Prior work in parallel hash-based multi-join query evaluation has advocated right-deep query trees, because all build ("left") subtrees can be processed in parallel and then probed in a single long pipeline [45]. This conventional wisdom on good query plans is commonly encoded in ad-hoc heuristics in parallel query optimizers to avoid the combinatorial explosion of the search space [41]. Although a single database server

may have as many CPU cores as a parallel database in the past, the hardware architecture does not otherwise resemble a shared-nothing environment. Existing query optimizers need to be augmented to accurately model the resource needs of ad-hoc queries over in-memory data.

This paper focuses on hash-based join evaluation strategies for main-memory databases and contributes a detailed cost model that accurately predicts the query response time of ad-hoc multi-join queries. Our thesis is that the response time for processing a complex multi-join query using all CPU cores is proportional to the number of memory operations weighted by the performance cost for different access patterns. By exhaustively exploring all possible query plans for a join between 4 tables, we find that the memory access cost that is predicted by our model proves to be an accurate proxy for elapsed time. We then demonstrate that linear disk I/O models, such as the current cost model in PostgreSQL, cannot accurately account for the access patterns to and from main memory even after optimally adjusting the model weights through linear regression.

The proposed cost model is conceptually simple as it does not account for the multi-layered cache hierarchy or any NUMA effects. We find that the loss of accuracy due to these two factors is minor for the cache-oblivious non-partitioned hash join algorithm that we model. Yet by only focusing on the top layer of the memory hierarchy, our model gains in versatility and can readily adapt to the underlying hardware. The conceptual simplicity of the model is particularly important in light of the limited topological information about the CPU and cache hierarchy that can be gleaned from a VM in a cloud environment. Experiments with on-premises systems and Amazon EC2 show that our model accurately predicts how the response time of individual query plans will change based on the performance characteristics of different systems.

A surprising finding from our thorough experimental evaluation is that some left-deep query trees can be more efficient than their bushy and right-deep tree counterparts if the join pipeline must run until completion because they result in substantially less memory traffic during execution. As corroborated by our model, the performance gap between the optimal left-deep and right-deep query tree can grow to about $10\times$ as the number of joins in the query increases. This finding challenges the widely held belief among systems implementers that MPP-style optimization is "good enough" for parallel in-memory query processing.

2. Background and Related Work

Query processing techniques for in-memory data

Research in main memory database management systems commenced more than three decades ago. Early work investigated join algorithms for main memory database systems by DeWitt et al. [19] and an investigation of in-memory query processing techniques by Lehman and Carey [29]. As the CPU architecture evolved, additional hardware operations were shown to become performance bottlenecks. Ailamaki et al. [2] studied the performance bottlenecks of four commercial database management systems and highlighted the importance of minimizing L2 data cache stalls and L1 instruction cache stalls. Shatdal et al. [46] proposed to make the join algorithm cache-conscious to improve locality and join performance by adding a partitioning step before the join. Boncz et al. [9] proposed radix-based query processing techniques to improve in-memory performance by reducing cache and TLB misses.

There is substantial prior work in efficient parallel in-memory equi-join techniques. Balkesen et al. [4] compare sort-merge and radix-hash joins in a multi-core, main-memory environment and claimed that radix-hash join was superior to sort-merge join. Leis et al. [30] break input data into small fragments and assign each to a thread to run entire operator pipelines. By controlling the dispatch of data fragments, query execution can be parallelized dynamically in a NUMA-aware fashion. Polychroniou and Ross [42] proposed a family of main-memory partitioning algorithms that includes hash, radix and range partitioning. Li et al. [32] studied data shuffling in a NUMA system and improved the performance by employing thread binding, NUMA-aware thread allocation and relaxed global coordination. Lang et al. [28] optimized the hash join for NUMA architectures by optimizing the physical representation of the hash table. Barber et al. [6] introduced concise hash tables which significantly reduce memory consumption while still offering competitive performance.

Researchers also have proposed cost models for in-memory data processing. Manegold et al. [39] proposed a hierarchical model to capture all levels of a cache hierarchy and show that this can accurately capture the total response time of quick sort, hash join and partitioning operations after calibrating for the CPU cost of each algorithm.

Parallel query evaluation strategies

Multi-join query plan optimization is a difficult problem because of the uncertainty over the cardinality of intermediate results and the large optimization space. Chaudhuri summarized the foundations and significant research findings of query optimization [12], and Lohman has recently identified open problems in query optimization [35].

Prior research has extensively studied parallel and distributed query optimization techniques. R* was an early research prototype that optimized queries to minimize data transfer costs in a shared-nothing environment [36]. Chen et al. [14] proposed to generate parallel query plans in two phases: first generate the optimal query plan for a single node, and then to parallelize it for a multi-processor environment. Hong and Stonebraker [25] showed that two phase optimization produces good query plans for shared-memory systems. Researchers have also studied parallel hash join algorithms. DeWitt and Gerber [18] introduced

multi-processor versions of popular hash join algorithms for a single join and evaluated their advantages and disadvantages in a multi-processor environment. Recently, Chu et al. [16] propose a novel distributed multi-way join algorithm that has been theoretically proven to have better performance than evaluating the multi-way join as a series of binary joins.

The performance of different multi-join evaluation strategies has also been extensively studied. Schneider and De-Witt [45] studied multi-way join query processing through hash-based methods in the Gamma parallel database system. They observed that trees with shallow hash-join build phases and a long probe pipeline ("right-deep" trees) can provide significant performance advantages in large multiprocessor database machines. The key advantages of right-deep query trees are the ample parallelization potential, as each build subtree can be executed concurrently, and that no intermediate results need to be materialized in the long probe pipeline because results are streamed from one operator to the next.

Right-deep trees introduce the problem of processor allocation to each join sub-tree during parallel evaluation, and Lu et al. [37] use a greedy algorithm to generate query plans for multi-join queries that determines the order of each join, but also the number of parallel joins and the processor assignment to each join. Ioannidis and Kang [26] show that optimization for bushy and left-deep trees is easier than optimizing left-deep trees alone. Chen et al. [13] proposed bushy trees with right-deep pipelines, which they call segmented right-deep trees, for efficient execution of multi-join queries. Query evaluation strategies for shared-nothing main-memory database management systems have been examined in the context of the PRISMA/DB system [49]. Wilschut and Apers have identified synchronization costs as the performance bottleneck from parallelism [48]. Recently, Giceva et al. [21] have studied how to allocate resources and deploy query plans on multi-cores in the context of their operator-centric SharedDB system.

Liu and Rundensteiner [33] introduced segmented bushy query trees for the efficient evaluation of multi-join queries. Pipeline delays have also been studied in research by Deshpande and Hellerstein [17] who propose to reduce pipeline delays by plan interleaving between plan fragments. Ahmed et al. [1] have proposed a technique to optimize bushy join trees for snowstorm queries in the Oracle database system.

Machine learning techniques have also been explored to predict the response time of a query. IBM's DB2 introduced LEO, a learning optimizer, to adjust the query optimizer's prediction based on the observed query cost [40, 47]. Zhang et al. [51] proposed a statistical learning approach, COMET, to adapt to changing workloads for XML operations. Duggan et al. [20] used machine learning techniques to predict the performance of concurrent workloads. Akdere et al. [3] introduced two learning-based models, a plan-level model and an operator-level model, and contributed a hybrid model to predict query performance. Li et al. [31] combined re-
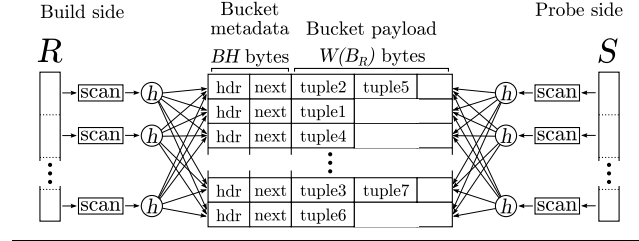

Figure 1. The non-partitioned hash join algorithm.

gression tree models and scaling functions to estimate the resource consumption of individual operators.

The non-partitioned in-memory hash join algorithm

Our proposed memory I/O model captures the fundamental memory access operations from the popular non-partitioned in-memory hash join algorithm and the data structure layout optimization by Balkesen et al. [5]. The main data structure is a hash table that is shared by all threads (see Figure 1). The hash table is a contiguous array of buckets. Each bucket contains metadata and the payload. Metadata are header information and a pointer to overflow buckets if this bucket has spilled. The hash join is divided into two phases: the build phase and probe phase. During the build phase, an empty hash table is created and the build table R is divided into equi-sized parts which are assigned to different threads. All threads then scan their input sequentially, hash the join key of every tuple, lock the bucket, write the tuple into the bucket, and then unlock the bucket. The probe phase starts after the entire build table R has been consumed. In the probe phase, all threads sequentially scan the probe table S, hash the join key of every tuple, and then join with the tuple in the corresponding bucket if the join keys are equal.

3. Modeling memory I/O

A fundamental challenge in building memory I/O models is the pull-based nature of data movement throughout the memory hierarchy. In stark contrast to the disk I/O hierarchy where data movement is explicitly initiated by the application, application control of memory I/O is limited to prefetch, flush and non-temporal access hints. Complicating matters further, memory loads and stores are not only triggered when retrieving user data for query processing, but also when manipulating local variables, accessing the call stack and co-ordinating with other CPU cores.

Determining the components of a memory I/O cost model is therefore a delicate balancing act. It is equally important to include sufficient detail to make accurate predictions but also avoid adding too many parameters that may exaggerate the effect of minor fluctuations in the model inputs because of over-fitting. The latter is particularly important for memory I/O cost models that need to be generic to avoid obsolescence in light of the quick pace of innovation in hardware.

When constructing performance models for general algorithms it is important to model every level of the cache hier-

Mnemonic	Access pattern
SR	Read one cache line sequentially
RR	Read one random cache line
SW	Write one cache line sequentially
RW	Write one random cache line

Table 1. Access patterns captured by the model.

archy. The non-partitioned hash join algorithm, however, is cache-oblivious and memory-bound. Therefore, our thesis is that when modeling a sequence of non-partitioned hash joins *the query response time is dominated by the cost of accessing main memory* and not individual caches.

Not all access patterns to memory are equally expensive, however. Sequential access patterns are more efficient because they leverage prefetching and result in few TLB misses. Similarly, writes are more expensive than reads because they require exclusive access to the cache line under the MESIF cache coherence protocol, which may require cache line invalidations in other CPU cores. (We outline the four access patterns in Table 1.) We thus model the response time for processing query Q as being proportional to the number of operations $N(\mathsf{A})$ weighted by the performance cost w_A for each access pattern $\mathsf{A} \in \{\mathsf{SR}, \mathsf{RR}, \mathsf{SW}, \mathsf{RW}\}$:

$$
\begin{aligned}
Time(Q) \propto\ & w_\mathsf{SR} \cdot N(\mathsf{SR}) + w_\mathsf{RR} \cdot N(\mathsf{RR}) \\
& + w_\mathsf{SW} \cdot N(\mathsf{SW}) + w_\mathsf{RW} \cdot N(\mathsf{RW})
\end{aligned}
\tag{1}
$$

The performance cost w_A for each access pattern A is obtained experimentally for each system through a controlled microbenchmark procedure, which is described in detail in Section 3.1. The number of operations $N(\cdot)$ is calculated from our model in Sections 3.2 and 3.3. Section 3.4 summarizes key insights obtained by our model.

Modeling assumptions and limitations

Our model assumes that memory is shared by all processing cores, it is automatically kept coherent by the hardware and that data transfer operations are performed at the granularity of a cache line. We also assume that the database and the working memory for a query can fit in memory and no paging to secondary storage occurs. We only consider multijoin queries that are evaluated in parallel from all threads using the non-partitioned hash-join algorithm by Balkesen et al. [5], which has been shown to offer competitive performance while judiciously using memory [8].

The fact that our proposed model does not account for any caching or NUMA effects does not imply that architecture-awareness is not important. On the contrary, the memory I/O cost model is an abstraction of a well-designed query engine that is not compute-bound and will not trigger redundant memory I/O during query processing. Inefficient implementations of the non-partitioned hash join will result in higher memory traffic, which will not be captured by the model. In particular, we assume that the hash table is striped on all available NUMA nodes.

Finally, a limitation of join cost modeling is the assumption that the cardinality of intermediate join results is known or can be estimated reasonably accurately. We acknowledge that cardinality estimation is not a solved problem [35]. Inaccuracies in estimating the cardinality or the skew of the build side are particularly problematic. Improperly sized hash tables will result in the creation and probing of overflow hash buckets. Some DBMSs will track excessive bucket overflows and pause the hash table build process to resize the hash table and accommodate more tuples. New join algorithms have recently been proposed to avoid populating the hash table with heavily-skewed data [43]. Bucket overflows and hash table resizes are not accounted for by the proposed model.

3.1 Bootstrapping the model

The model depends on the performance cost w_A for each access pattern $\mathsf{A} \in \{\mathsf{SR}, \mathsf{RR}, \mathsf{SW}, \mathsf{RW}\}$ for accurate predictions. These general performance costs encapsulate (1) differences in the underlying hardware of each system such as different CPU models or different memory modules, (2) some hardware configuration options such as memory timings and NUMA settings that can be changed in the BIOS, and (3) OS-specific configuration options, such as the number and size of huge TLB pages. Bootstrapping consists of four specific microbenchmarks where all threads transfer data from memory such that only one specific access pattern is triggered at a time. The microbenchmarks do not perform any computation, so the derived performance costs can be thought as a "worst-case" scenario where all threads are waiting on memory I/O.

To derive the performance cost for the SR and SW patterns each thread allocates a NUMA-local contiguous array and populates it with random values. Each array is sized to a power of two such that as much of the available memory in the system is used, but no paging to secondary storage occurs. Every thread then concurrently reads its local array sequentially and either accumulates an 8-byte value if the access pattern is SR, or accumulates and writes an 8-byte value back if the access pattern is SW. (Notice that the SW access pattern includes an explicit memory read.)

The RR and RW microbenchmarks use an array of the same size as for SR and SW that is now shared between all threads. Before the benchmarking, each thread first generates a random sequence of cache-aligned array offsets in NUMA-local memory and waits on a barrier. The benchmarking then starts, each thread accesses an entire cache line at every offset in the randomly generated sequence. If the access pattern is RR, every thread only accumulates values in an 8-byte local variable while reading the cache line. If the access pattern is RW, every thread reads and writes the entire cache line in 8-byte units.

The performance cost w_A is calculated by dividing the execution time of the $\mathsf{A} \in \{\mathsf{SR}, \mathsf{RR}, \mathsf{SW}, \mathsf{RW}\}$ microbenchmark with the number of cache lines that were accessed.

Symbol	Meaning
A	Access pattern, one of SR, RR, SW, RW.
w_A	Performance cost for access pattern A
$N(A)$	Number of operations of access pattern A
CL	Cache line size
$|R|$	Cardinality of relation R
$W(R)$	Size of each tuple in relation R
B_R	Hash table on R
$|B_R|$	Number of hash buckets in B_R
$W(B_R)$	Hash bucket size in B_R
T	Tuples per bucket (ie. hash table load factor)
BH	Size of metadata per hash bucket
$Q_{m,k}$	Query subtree k at depth m
$I_{m,k}$	Intermediate output of $Q_{m,k}$ (cf. Figure 3)

Table 2. Notation.

3.2 Modeling individual operators

In this section we analyze the memory access cost of individual operations. The notation is summarized in Table 2.

Sequential scan

All threads read data sequentially from memory during the scan. Scanning entails no random reads or memory writes. For a relation R, the number of SR memory operations is determined by the size of the input relation and cache line size. Therefore:

$$N_{scan(R)}(\text{SR}) = \left\lceil \frac{|R| \cdot W(R)}{CL} \right\rceil \qquad (2)$$

$$N_{scan(R)}(\text{RR}) = N_{scan(R)}(\text{SW}) = N_{scan(R)}(\text{RW}) = 0 \qquad (3)$$

Hash join build phase

During the hash join build phase every thread concurrently reads tuples from the build relation, hashes the tuple on the join key and inserts it into a shared hash table. Hashing the join key only involves computation, thus there will be no memory accesses. As a good hash function is designed to assign different keys into different hash buckets, the hash buckets will be accessed in random order. Once the appropriate hash bucket has been identified, the thread needs to latch the bucket before writing into it to prevent insertion races with other threads. The latch and other metadata information are stored in a bucket header structure at the beginning of the bucket and have size BH. Acquiring the latch will result in a local modification to the cache line. In the absence of any latch contention, releasing the latch will modify the same cache line again. Since the latch has likely already been brought into the cache after the latch acquire operation, the latch release operation will hit the same cache line and will not trigger additional memory accesses. Therefore one RW memory accesses will occur per tuple in the build relation R:

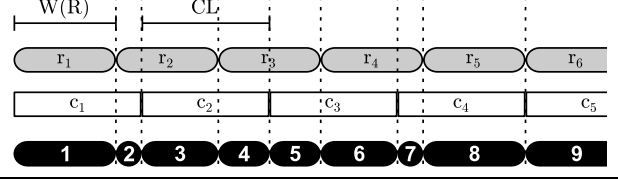$$N_{build(R)}(\text{RR}) = N_{build(R)}(\text{SR}) = 0 \qquad (4)$$

Figure 2. Modeling the cache line (c_i) writes in a hash bucket, when inserting tuples r_i.

$$N_{build(R)}(\text{RW}) = |R| \qquad (5)$$

Writing each R tuple in the hash bucket will require additional writes. The number of memory writes depends on how many cache lines overlap with the modified memory region and the write pattern. Let us consider the insertion of T tuples from relation R (denoted by $r_1, r_2, \ldots$) into the hash bucket that consists of the cache lines $c_1, c_2, \ldots$ that is shown in Figure 2. When the first tuple r_1 is inserted it fits entirely into c_1, causing one memory write. When r_2 is inserted, it spans both the c_1 and the c_2 cache lines and requires two memory writes. Similarly, both r_3 and r_4 trigger two memory writes each because they span cache lines, but inserting r_5 only requires one memory write as it fits completely within the c_4 cache line.

We can derive a closed form solution for the memory writes that are triggered by inserting T tuples of width $W(R)$ into cache lines of size CL by calculating the boundaries of the cache lines and the tuples, which are shown as vertical dashed lines in Figure 2. We apply the inclusion/exclusion principle and first add the tuple boundaries (k_1) and cache boundaries (k_2). We will then subtract the boundaries that are both tuple and cache line boundaries (k_3), such as the boundary between r_5 and r_6 in Figure 2. The $r_1, \ldots, r_T$ tuples have $k_1 = T - 1$ boundaries and the cache lines have $k_2 = \left\lfloor \frac{T \cdot W(R) - 1}{CL} \right\rfloor$ boundaries. The common boundaries can be found in the locations that are both a multiple of $W(R)$ and a multiple of CL. If $lcm(a, b)$ is the least common multiple of a and b, these locations are precisely at $1 \times lcm(W(R), CL)$, $2 \times lcm(W(R), CL)$, ... Thus, the number of common boundaries is $k_3 = \left\lfloor \frac{T \cdot W(R) - 1}{lcm(W(R), CL)} \right\rfloor$. We conclude that the number of memory accesses to insert T tuples of size $W(R)$ into one hash bucket will be $\mathcal{M}(T, W(R)) = k_1 + k_2 - k_3 + 1$, which is:

$$\mathcal{M}(T, W(R)) = $$
$$T + \left\lfloor \frac{T \cdot W(R) - 1}{CL} \right\rfloor - \left\lfloor \frac{T \cdot W(R) - 1}{lcm(W(R), CL)} \right\rfloor \qquad (6)$$

If the hash table B_R on relation R has $|B_R|$ hash buckets and each bucket is sized to fit $\lceil |R|/|B_R| \rceil$ tuples, the total number of SW memory accesses is:

$$N_{build(R)}(\mathsf{SW}) = |B_R| \times \mathscr{M}\left(\left\lceil \frac{|R|}{|B_R|} \right\rceil, W(R)\right) \qquad (7)$$

Hash join probe phase

After the build phase completes, each thread will read tuples from the probe relation S, hashes each tuple on the join key and compares the join key of the tuple from S with the join keys of all the tuples in the corresponding hash bucket. No memory access is involved when evaluating the hash function. The join key will then need to be compared with every key in the hash bucket. (If the hash bucket is empty, the hash bucket metadata will still need to be retrieved to discover this.) As the bucket is determined by the hash of the join key of the input tuple, the first access to the hash bucket is RR for every tuple in the probing relation S:

$$N_{probe(S)}(\mathsf{RR}) = |S| \qquad (8)$$

All tuples in the hash bucket will be retrieved from memory for the join key comparison using the SR access pattern. The number of sequential reads is determined by the number of additional cache lines the bucket occupies. Each bucket in the hash table B_R will contain $W(R) \cdot T + BH$ bytes of data, but the first CL bytes have already been accounted for in the RR access pattern. Assuming that each hash bucket fits $T = \lceil |R|/|B_R| \rceil$ tuples, let $\mathscr{L}(R)$ be the number of SR memory accesses per probe:

$$\mathscr{L}(R) = \left\lceil \frac{W(R) \cdot \left\lceil \frac{|R|}{|B_R|} \right\rceil + BH}{CL} \right\rceil - 1 \qquad (9)$$

$$N_{probe(S)}(\mathsf{SR}) = |S| \times \mathscr{L}(R) \qquad (10)$$

The output of the join will be written to an output buffer of each thread so as to be used by the next operator. Since the queue buffer is reused between operators, it can be sized to fit in the cache and not generate any memory access when populating the output buffer in a well-tuned system. Our model of the probe phase of the hash join therefore includes no SW or RW operations:

$$N_{probe(S)}(\mathsf{SW}) = N_{probe(S)}(\mathsf{RW}) = 0 \qquad (11)$$

3.3 Modeling multiple joins in a query tree

The number of memory accesses for the subtree rooted at depth i and position j (see the bushy tree in Figure 3) is obtained by adding the memory accesses of the operator at depth i and position j with the memory accesses of the left and right subtrees for each access pattern $\mathsf{A} \in \{\mathsf{SR}, \mathsf{RR}, \mathsf{SW}, \mathsf{RW}\}$. If $I_{i,j}$ denotes the intermediate result from the operator at depth i and position j, the number of memory accesses with pattern A for the entire query is $N(\mathsf{A}) = N_{0,0}(\mathsf{A})$, where:
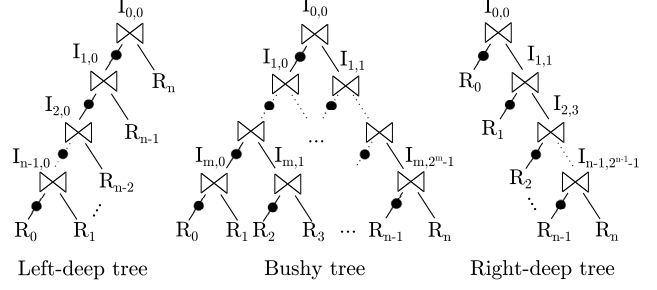

Figure 3. Left-deep tree, bushy tree and right-deep tree. The output of a join is pipelined into the next operator. The black dot denotes the build side, that is the join input that will be buffered in a hash table before processing the probe side.

$$N_{i,j}(\mathsf{A}) = \begin{cases} N_{scan(R_k)}(\mathsf{A}) & \text{if seq. scan,} \\[2ex] N_{build(I_{i+1,2j})}(\mathsf{A}) + N_{probe(I_{i+1,2j+1})}(\mathsf{A}) \\ + N_{i+1,2j}(\mathsf{A}) + N_{i+1,2j+1}(\mathsf{A}) & \text{if hash join.} \end{cases} \qquad (12)$$

Equation 12 does not account for the cost of materializing the output $I_{i,j}$. For intermediate results, we have experimentally found that this is not a significant cost if the output buffer has been sized appropriately [10] — the buffer that stores the materialized output batch for the next operator stays cached and does not trigger memory accesses. (Note, however, that the cost of processing a larger intermediate result is captured in the cardinality and width of $I_{i,j}$ when accounting for the memory accesses of subsequent operations.) We validate this modeling choice in Section 4.2.2. The final join output $I_{0,0}$ is also materialized incrementally if it is consumed by an aggregation or a top-k operation. (This pattern, for instance, occurs in every query in the TPC-H decision support benchmark.) If the entire query output is to be materialized, the model can be trivially extended with additional SW accesses to reflect the materialization cost. This cost would be identical for all query plans.

3.4 Insight on join evaluation orders

This model can offer valuable insights into the memory I/O requirements of different query plans. We illustrate this by modeling the common case of a linear primary key–foreign key join between $n+1$ relations. Let that the multi-join operation be $R_0 \bowtie R_1 \bowtie \cdots \bowtie R_{n-1} \bowtie R_n$, where $|R_i| \leq |R_{i+1}|$, for every $i \in [0, n-1]$, The join sequence for a left-deep evaluation strategy Q_L is:

$$Q_L = \left(\left(\left((R_0 \bowtie R_1) \bowtie \cdots \right) \bowtie R_{n-1} \right) \bowtie R_n \right)$$

While the right-deep evaluation strategy Q_R is:

$$Q_R = \left(R_0 \bowtie \left(R_1 \bowtie \left(\cdots \bowtie (R_{n-1} \bowtie R_n) \right) \right) \right)$$

Let us consider the common case of a join that produces as many tuples as the probe side, as is commonly the case for primary key–foreign key joins. Closed formulas for the memory access cost of the right-deep query tree Q_R that is shown in Figure 3 can be obtained by setting $I_n = R_n$, as follows:

$$N(\mathsf{SR}) = \sum_{i=0}^{n} \left(\left\lceil \frac{|R_i| \cdot |W(R_i)|}{CL} \right\rceil \right) + \sum_{i=1}^{n} (|I_i| \times \mathscr{L}(R_{i-1}))$$
$$(13)$$

$$N(\mathsf{SW}) = \sum_{i=0}^{n-1} \left(|B_{R_i}| \times \mathscr{M} \left(\left\lceil \frac{|R_i|}{|B_{R_i}|} \right\rceil, W(R_i) \right) \right) \quad (14)$$

$$N(\mathsf{RR}) = \sum_{i=1}^{n} |I_i| \qquad N(\mathsf{RW}) = \sum_{i=0}^{n-1} |R_i| \qquad (15)$$

The cost for a left-deep tree Q_L, with $I_n = R_0$ as shown in Figure 3, is:

$$N(\mathsf{SR}) = \sum_{i=0}^{n} \left(\left\lceil \frac{|R_i| \cdot |W(R_i)|}{CL} \right\rceil \right) + \sum_{i=1}^{n} (|R_i| \times \mathscr{L}(I_{n-i+1}))$$
$$(16)$$

$$N(\mathsf{SW}) = \sum_{i=1}^{n} \left(|B_{I_i}| \times \mathscr{M} \left(\left\lceil \frac{|I_i|}{|B_{I_i}|} \right\rceil, W(I_i) \right) \right) \quad (17)$$

$$N(\mathsf{RR}) = \sum_{i=1}^{n} |R_i| \qquad N(\mathsf{RW}) = \sum_{i=1}^{n} |I_i| \qquad (18)$$

Insight 1: The left-deep (Q_L) and right-deep (Q_R) evaluation strategy perform the same number and type of memory writes. $N(\mathsf{RW})$ and $N(\mathsf{SW})$ are identical for the two query trees, because the cardinality of the k-th intermediate output is $|I_k| = |R_{n-k}|$ for the left-deep strategy Q_L, and $|I_k| = |R_n|$ for the right-deep strategy Q_R.

Insight 2: The left-deep (Q_L) evaluation strategy performs $n|R_n| - \sum_{i=1}^{n} |R_i|$ fewer RR memory accesses than the right-deep (Q_R) strategy. The RR accesses for the left-deep strategy are $N_L(\mathsf{RR}) = \sum_{i=1}^{n} |R_i|$, while the RR accesses are $N_R(\mathsf{RR}) = n|R_n|$ for the right-deep strategy. $N(\mathsf{SR})$ is identical in both strategies if $\mathscr{L}(\cdot) = 0$, namely if the tuples are thinner than the last-level cache line. This is commonly the case due to projection push down or columnar storage.

3.5 Possible extensions to the model

The proposed model accounts for wide tuples but assumes that the tuple width is fixed per input relation. The model could crudely be applied on variable-length records by using the *average* tuple size, but accurate results would require

a complete histogram of the attribute size per join key. A research challenge is to devise techniques to collect such metadata information efficiently.

Another promising direction is to extend the model for distributed in-memory joins. We foresee two challenges in generalizing our model to distributed in-memory hash joins. First, it seems unlikely that a model that does not account for locality and data placement can remain accurate when some accesses become remote and thus an order of magnitude more expensive than what is currently modeled. Second, one would need to model the cost of remote data transfers during the join. A first step towards this direction is the performance model by Barthels et al. [7] that models the cost of transferring data using RDMA primitives.

Hash tables are widely used for storing data for in-memory caching or in-memory transaction processing. Modeling the cost of each access is a first step towards understanding the performance characteristics of complex operations (such as transactions) that consist of a series of reads and writes on a hash table. We speculate that the main limitation of our model when applied to ad-hoc operations is to account for the effect of concurrent writes and reads. The hash join algorithm naturally separates the write-intensive and read-intensive operations to different phases of the algorithm. Were these operations concurrent, contention would diminish performance. A promising research direction is to construct contention-aware performance models for in-memory data processing.

4. Experimental evaluation

This section describes the experimental evaluation and validation of our model. Sections 4.2.1 and 4.2.2 focus on a single join and evaluate the accuracy of the model in predicting the memory I/O activity. We validate that I/O activity is linearly correlated with query response time, and we find that the impact of incrementally materializing intermediate results to the next pipeline stage is negligible. Section 4.2.3 evaluates how parallel subtree evaluation affects performance, and validates that sequential subtree evaluation results in similar query response times to the fastest parallel evaluation strategy but with less memory pressure.

We then turn our attention to multi-join queries, and we exhaustively compare all possible multi-join query plans for a join with four input relations. Section 4.3.1 discusses how our simple memory I/O model compares with a linear disk-based I/O model, such as the Haas et al. model [24], and

Vendor, CPU or instance type	Cost per access pattern			
	w_{SR}	w_{RR}	w_{SW}	w_{RW}
Intel, 2×E5-2695v2	1.00	3.79	5.03	6.25
AMD, 2×Opteron 6172	1.00	6.44	1.88	8.42
Amazon EC2, c4.4xlarge	1.00	6.81	5.21	13.86

Table 3. Measured cost per access pattern for all systems. (See Section 3.1 for details on the bootstrapping process).

	Predicted memory I/O					Observed memory I/O				Response time slowdown		
Load factor	RW Build(R)	SW Build(R)	Total writes	SR Scan(R)	Total reads	Writes Total	Error	Reads Total	Error	Pred.	Observ.	Error
1.0	512	0	512	128	640	505	**1.4%**	634	**0.9%**	—	—	—
2.0	1,024	0	1,024	256	1,280	1,019	**0.5%**	1,277	**0.2%**	2.00×	1.97×	**1.5%**
3.0	1,536	0	1,536	384	1,920	1,531	**0.3%**	1,921	**-0.1%**	3.00×	2.94×	**2.0%**
4.0	2,048	512	2,560	512	3,072	2,555	**0.2%**	3,081	**-0.3%**	4.77×	4.77×	**0.0%**
5.0	2,560	1,024	3,584	640	4,224	3,580	**0.1%**	4,234	**-0.2%**	6.55×	6.56×	**-0.2%**
6.0	3,072	1,536	4,608	768	5,376	4,605	**0.1%**	5,388	**-0.2%**	8.32×	8.36×	**-0.5%**
7.0	3,584	2,048	5,632	896	6,528	5,629	**0.1%**	6,548	**-0.3%**	10.09×	10.14×	**-0.5%**
8.0	4,096	2,560	6,656	1,024	7,680	6,656	**0.0%**	7,701	**-0.3%**	11.87×	11.98×	**-0.9%**

Table 4. Model accuracy for different hash table load factors. I/O events in millions.

$\frac{|S|}{|R|}$	Join selectivity	$|S \bowtie R|$ (millions)	Predicted memory I/O				Total	Observed I/O		Response time slowdown		
			RW Build(R)	RR Probe(S)	SR Scan(R)	SR Scan(S)		Total	Error	Pred.	Observ.	Error
4×	25%	512	512	2,048	128	512	3,200	3,136	**2.0%**	2.15×	2.04×	**5.4%**
3×	33%	512	512	1,536	128	384	2,560	2,625	**-2.5%**	1.77×	1.71×	**3.5%**
2×	50%	512	512	1,024	128	256	1,920	1,957	**-1.9%**	1.38×	1.37×	**0.7%**
1×	100%	512	512	512	128	128	1,280	1,299	**-1.5%**	—	—	—
2×	100%	1,024	512	1,024	128	256	1,920	1,876	**2.3%**	1.38×	1.37×	**0.7%**
4×	100%	2,048	512	2,048	128	512	3,200	3,270	**-2.1%**	2.15×	2.10×	**2.4%**
6×	100%	3,072	512	3,072	128	768	4,480	4,583	**-2.2%**	2.92×	2.84×	**2.8%**
8×	100%	4,096	512	4,096	128	1,024	5,760	5,907	**-2.5%**	3.68×	3.56×	**3.4%**

Table 5. Model accuracy for larger probe relations and different join selectivities. I/O events in millions.

whether parameter tuning of disk-based models is sufficient to predict the response time of queries on memory-resident data. Section 4.3.2 investigates how accurate is our model in predicting the response time of a particular query plan on different hardware. We then evaluate the sensitivity of our model to changes in the database size (Section 4.3.3) and join skew (Section 4.3.4). Finally Section 4.3.5 evaluates the predictive power of our model in forecasting the cost of deeper multi-join pipelines. An extended version of this paper contains experiments with varying join selectivities and input cardinality ratios [34].

We have extended Pythia [44], a prototype open-source in-memory query engine to support multi-way joins and complex query pipelines. We started with the optimized non-partitioned hash join implementation of Balkesen et al. [5] to create a hash join operator that can be placed in complex query pipelines using a parallel, vectorized, pull-based execution model [10, 22]. Building on prior work [15], we prefetch hash buckets before they are accessed by "peeking ahead" within the batch of input tuples. We have built a custom memory allocator to use huge pages for memory requests larger than 1 GB. By tightly controlling memory allocation we can enforce NUMA placement, support pre-allocation during query planning time, and precisely monitor the memory consumption of a query during processing.

4.1 Hardware setup and methodology

We use three systems for our experimental evaluation. The first system (which we will refer to as "Intel") is a server with two NUMA nodes and 256 GB of DDR3 memory, with two Intel Xeon E5-2695v2 12-core processors. Each core in an E5-2695v2 processor shares the 30 MB L3 cache. We have enabled hyper-threading, so the operating system can simultaneously execute 48 threads. The second system (which we will refer to as "AMD") is a server with four NUMA nodes and 32 GB of memory, with two AMD Opteron 6172 12-core processors (in total, 24 concurrent threads). Each core in an Opteron 6172 processor shares the 12 MB L3 cache.

In a cloud environment, the precise hardware specifications may not be known before the VM is instantiated. An advantage of the proposed cache-oblivious model is its adaptability to hardware with unknown specifications. The third system (which we refer to as "EC2") is an Amazon EC2 instance of type c4.4xlarge. The instance has 30 GB of memory and 16 vCPUs. To control for cyclical variations in performance, we bid for a single spot EC2 instance around 10am every morning (US Eastern time) for 7 consecutive days. The observed day-to-day variation in performance is less than 2% so we only report averages over all seven days for all EC2 experiments.

The measured performance cost w_A for each access pattern A among SR, RR, SW and RW for the three systems is shown in Table 3. (Please refer to Section 3.1 for details on the bootstrapping process.) We use the built-in "perf" tool of the Linux kernel to tap into hardware counters of performance events related to memory accesses. To minimize reporting overheads from "perf" we only inspect these counters after the completion of the initialization, the build and the probe phases (that is, three times per join operation). Every experiment is repeated at least ten times, and we report the average. We assume that a memory manager preallocates memory when the DBMS starts, and queries can quickly reuse pre-allocated memory. Thus, memory allocation time is not included in the reported query response time.

4.2 Validating modeling assumptions

This section validates key modeling assumptions using the Intel system that is described in Section 4.1. We first dissect a single hash join to evaluate how accurately we model memory I/O traffic and whether memory traffic is a good predictor of the join execution time. We then turn our attention to how parallel evaluation of distinct build subtrees of a query plan affects performance.

4.2.1 Hash join build phase

The first experiment evaluates the accuracy of the model in capturing the memory I/O activity and predicting the completion time of the hash join build phase. We create one relation R with two attributes. The first attribute is an 8-byte integer which ranges from 1 to the cardinality of relation $|R|$, and the second attribute is a random 8-byte integer. Narrow tuples are commonly encountered in column-stores and have been extensively used for experimental evaluation in prior work [5]. The input relation has been randomly permuted so that the keys are in random order, and is striped among both NUMA nodes of our system. For this experiment we fix the number of hash buckets $|B|$ to 512 million and use 16 bytes per bucket as the bucket header BH; the first cache line can therefore hold 3 tuples.

The experimental results and the model predictions for building a hash table are shown in Table 4. We vary the cardinality of the build relation R from 512 million ($1\times$) to 4 billion ($8\times$) tuples. When the build relation size varies from $1\times$ to $3\times$, all tuples will be in the same cache line with bucket header, and $N(\text{SW}) = 0$. When the build relation size reaches or exceeds $4\times$, tuples will be inserted in the next available cache line and will cause SW traffic. When comparing the model prediction with the observed I/O activity, we notice that the read and the write prediction is very accurate and commonly deviates less than 1% from the observed memory traffic. When comparing the model prediction with the experimentally observed execution time, we observe that the memory I/O activity is an accurate predictor of the execution time of the build phase, as it commonly deviates less than 2% from the observed time.
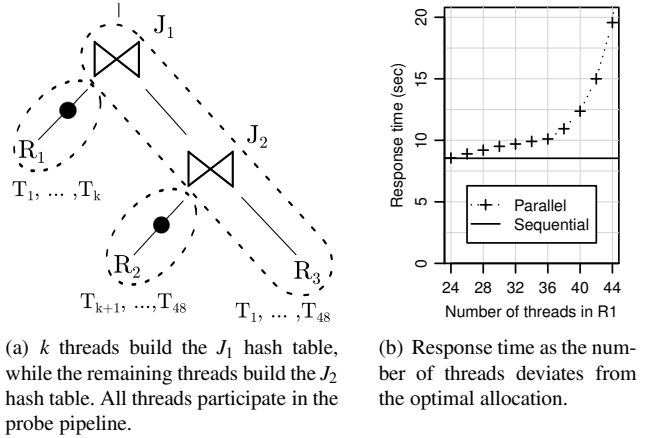
(a) k threads build the J_1 hash table, while the remaining threads build the J_2 hash table. All threads participate in the probe pipeline.

(b) Response time as the number of threads deviates from the optimal allocation.

Figure 4. Exploring sequential or parallel build subtree evaluation for right-deep query trees.

4.2.2 Hash join probe phase

We now evaluate the accuracy of our model in capturing the hash join probe phase. We use the same build relation R as in Section 4.2.1 and fix the cardinality of R to 512 million tuples ($1\times$). As shown in Table 5, the cardinality of the probe relation $|S|$ varies from $1 \times 512M$ to $8 \times 512M$, and the probability that an S tuple finds a matching R tuple in the hash table varies from 25% to 100%. We observe that the model's memory activity prediction is accurate and within 3% of the observed memory read traffic. Memory I/O activity is again linearly correlated with query response time. In fact, the model's prediction slightly overpredicts the query response time. Finally, the cardinality of the join output $|S \bowtie R|$ does not noticeably affect the memory I/O activity and the response time, because the join output is materialized incrementally in the same output buffer for pipelined operations. This validates our design choice to not explicitly account for the output materialization cost of intermediate results.

4.2.3 Parallel vs. sequential build subtree evaluation

Schneider and DeWitt [45] argued that one aspect of superiority of right deep trees over left deep trees is their potential for exploiting parallelism in their independent build subtrees. In this section, we study the performance of utilizing this inter-subtree parallelism for memory-resident datasets. We create three relations that contain 1 billion tuples each, and evaluate the query plan shown in Figure 4(a). Every probe tuple matches one build tuple in the J_1 and J_2 joins. We execute the query plan both sequentially and in parallel. When we execute the query sequentially we use all available threads to build R_1 first, and then move on to build R_2 after completing the build of R_1. When we execute the query in parallel, the threads are divided into two groups: k threads build a hash table on R_1 while the other $48 - k$ concurrently build a hash table on R_2. All threads participate in the probe phase for both evaluation strategies.
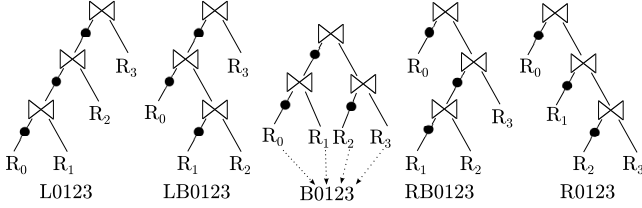
Figure 5. Query plan notation for the exhaustive 4-relation join experiment. The letters denote the tree shape, followed by the input relations when reading from left to right.

The response time of the two evaluation strategies is shown in Figure 4(b) as a function of the number of threads assigned to R_1. Because the two tables have the same size, the optimal allocation ($k = 24$) is to divide the threads into two equally-sized groups. Performance will degrade if the optimal thread assignment is not predicted correctly by the query optimizer. However, because the communication costs of a main-memory environment are negligible compared to shared-nothing architectures, the performance of building the hash tables on R_1 and R_2 sequentially is equal to the performance of parallel evaluation with the best thread allocation. In light of these findings, we process independent build subtrees sequentially in the experiments that follow.

4.3 Evaluation with multi-join query plans

In order to exhaustively evaluate our model's prediction for multi-join query plans, we generate a synthetic database with four relations R_0, R_1, R_2 and R_3. All relations contain two long integer attributes a and b, and the tuple size is 16 bytes. The attribute a ranges from 1 to the cardinality of the relation and the values are randomly distributed. We evaluate the following SQL query:

```
SELECT SUM(R0.a + R3.b)
FROM R0, R1, R2, R3
WHERE R0.b = R1.a, R1.b = R2.a, R2.b = R3.a
```

This synthetic database with 4 relations allows us to explore the five query plan shapes which are shown in Figure 5, namely the left-deep (L), left-bushy (LB), bushy (B), right-bushy (RB), and right-deep (R) tree. We use "TXXXX" to refer to a query plan, where "T" is the query type, followed by a sequence of numbers denoting the input relations when reading from left to right. By focusing on a database with 4 relations, we can systematically explore the entire query plan space, while keeping the number of query plan candidates within a reasonable limit for presentation purposes. (We relax this constraint in Section 4.3.5 where we evaluate the model with query plans that involve more than 3 joins.)

4.3.1 Comparison with a popular disk-based model

Do we need a new cost model for memory I/O, or is adjusting the weights of existing linear disk-based models sufficient? We answer this question by comparing the accuracy

of our model with the PostgreSQL 9.3 disk I/O model which mimics the Haas et al. [24] model:

$$Cost(Q) \propto n_s \cdot c_s + n_r \cdot c_r \qquad (19)$$

The number of I/O operations n_s and n_r for a given query plan are predicted by the query optimizer and is a function of the cardinalities of the join inputs and the join output. The cost weights c_s and c_r are user-adjustable parameters.

We exhaustively explore all viable multi-join query plans with 4 relations for the query that is shown in Section 4.3. We set the relation cardinality of R_0, R_1, R_2, R_3 to 2 billion, 512 million, 128 million and 32 million tuples respectively. The join ratio is 1:4, that is for every value of $R_{k+1}.a$, there are 4 tuples with matching $R_k.b$ values. We force PostgreSQL to use a specific join order and explicitly set the cardinality estimates to the actual join cardinality instead of relying on the prediction by the query optimizer. We obtain n_s by setting w_s to 1 and w_r to 0, and we report the total cost for the query plan, as shown by PostgreSQL's `EXPLAIN` command. (We do likewise to find n_r.) We finally perform linear regression to find the *optimal* non-negative costs to minimize the error between the cost prediction and the observed execution time.

Figure 6(a) shows the prediction of the PostgreSQL disk I/O model on the horizontal axis (after parameter fitting using linear regression) and the observed response time on the Intel system on the vertical axis. Even with optimal cost weights, the predictions of the PostgreSQL disk I/O model are clustered in five points (shown as vertical bands) on Figure 6(a). PostgreSQL gives the same cost prediction for plans with very different execution times: it predicts that the best two plans ("L3210" and "L2310") will have the same response time as the "R0132" and "R0123" plans which are in reality twice as expensive. Making matters worse, "R3210" is the optimal right-deep query plan but is predicted to be twice as expensive as the "R0123" query plan.

In comparison, Figure 6(b) shows the prediction from our model on the Intel system, which successfully predicts the cheapest plans "L3210" and "L2310". In addition, all observed response times lie within a narrow ±15% band of their prediction. A disk I/O model that only considers the size of the data accessed and whether this access is sequential or random cannot account for the access patterns that arise during an in-memory hash join.

4.3.2 Model adaptability to different hardware

In this section, we test the model's accuracy on the different systems that were described in Section 4.1. Because of limited memory size, we set the cardinality of R_0, R_1, R_2, R_3 to be 256 million, 64 million, 16 million, and 4 million tuples respectively. The join ratio is set to be 1:4 as in Section 4.3.1, and the query is the 4-relation join described in Section 4.3. The predictions are more "noisy" due to caching effects that arise because of the smaller database size — for instance, R_3

(a) PostgreSQL disk I/O cost model, (b) Our model on the Intel system, (c) Our model on the AMD system, (d) Our model on the EC2 instance, r_p=0.740, r_s=0.724 r_p=0.970, r_s=0.982. r_p=0.881, r_s=0.903. r_p=0.939, r_s=0.989.
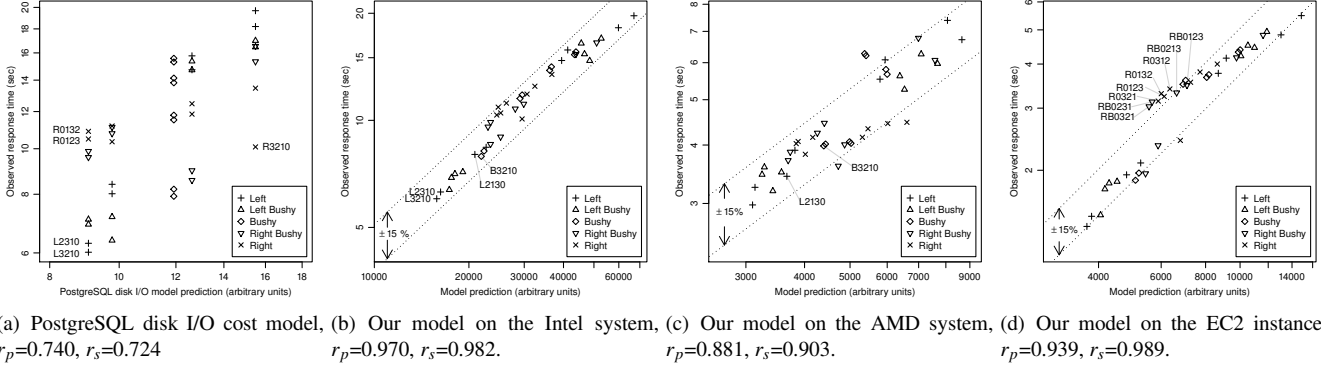
Figure 6. Query response time vs. predicted cost from the PostgreSQL I/O cost model with least-squares fit, and our model for the Intel, AMD and EC2 systems. r_p denotes the Pearson correlation coefficient, and r_s the Spearman correlation coefficient.

can be cached in its entirety in all systems. (We explore the model sensitivity to the database size in Section 4.3.3.)

Figure 6(c) compares the model's cost prediction with the observed query response time on the AMD system. Our model remains accurate as only a few of the 40 query plans fall outside the ±15% band. The proposed model also adapts to the hardware architecture of the underlying platform. Observe the "L2130" and "B3210" query plans on the Intel system at Figure 6(b). Both plans have been accurately predicted to have comparable performance. Yet for the AMD system at Figure 6(c), the model accurately predicts that the "B3210" query plan is 15% more expensive than the "L2130" plan. This difference can be attributed to the proportion of the RR activity of the two plans, which is 76% for the "B3210" plan but only 52% for the "L2130" plan. As the w_{RR} weight changes from 3.79 on the Intel system to 6.44 on the AMD system (cf. Table 3), our model captures this performance difference and adjusts the cost predictions for all plans accordingly.

We also test our model on the Amazon EC2 instance using the same dataset as for the AMD system and show the results in Figure 6(d). The EC2 results are similar to the results from the Intel system, except for all the plans with prefixes "RB0" and "R0" that construct a hash table on R_0. Our model predicts that these plans are much faster than their actual response time. A closer look at the performance counters reveals that the discrepancy is due to the cost to translate virtual to physical memory addresses when accessing this large hash table. The Intel system was configured with 1GB huge pages that Pythia can readily use, while the EC2 instance had no huge page support. Modeling the hierarchical nature of the page table could improve the accuracy of the model when huge pages are not enabled or not supported.

We now adopt more concise metrics to quantify model accuracy. Prior work [23, 50] has used correlation metrics to measure the accuracy of the model prediction. The Pearson correlation coefficient r_p is used to measure the linear correlation between two variables (namely, the prediction and the
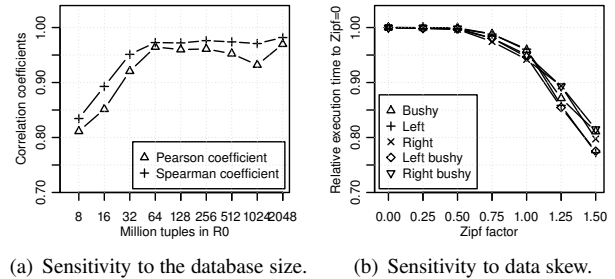
(a) Sensitivity to the database size. (b) Sensitivity to data skew.

Figure 7. Model sensitivity to changes in the database size and skew for the Intel system.

actual query response time). A deficiency of the Pearson coefficient is that it can assign low coefficient scores to models that correctly predict the relative ordering of different query plans, but the correlation with the observed response time is non-linear. The Spearman correlation coefficient r_s accommodates non-linear models by measuring the linear correlation between the *ranks* of different query plans. For the remainder of the paper we will use both the Pearson (r_p) and Spearman (r_s) coefficients as metrics of model accuracy.

4.3.3 Model sensitivity to the database size

The proposed model assumes that caching affects all query plans equally. In this section we validate this design assumption by showing the accuracy of the model as a growing fraction of the database fits in the cache (that is, the database shrinks in size). We use the multi-join query in Section 4.3 on the Intel system, and fix the join ratio to be 1:4. Figure 7(a) plots the Pearson and Spearman correlation coefficients as $|R_0|$ ranges from 2 billion tuples to as little as 8 million tuples. The data points for 2 billion tuples reflect the coefficients for Figure 6(b). The dip for 1 billion tuples is caused by three outliers, namely the "LB2103", "L1203" and "L2103" plans, whose response time is in the slowest quartile (25%) among all query plans. These very expensive query plans would realistically be never considered as viable
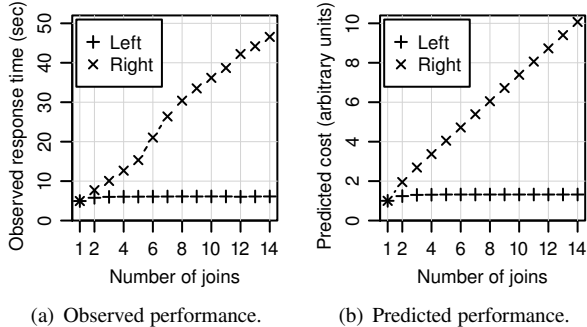
(a) Observed performance.   (b) Predicted performance.

Figure 8. Comparing the optimal left-deep and right-deep query plans as the number of joins increases. The performance gap grows to almost $10\times$, as predicted by the model.

alternatives by the query optimizer. The model remains very accurate with a Spearman coefficient of at least 0.95, until the database size becomes less than 1 GB (that is, when R_0 has fewer than 32 million tuples). When $|R_0|$ is 8 million tuples, R_1, R_2, and R_3 fit entirely in cache. Even then, both the Pearson and Spearman correlation coefficient for our model remain above 0.8 and 26 out of the 40 query plans have response times that are within $\pm15\%$ of the model prediction.

4.3.4 Model sensitivity to data skew

Data skew has been shown to have a profound effect on performance even for a single join [5]. We now evaluate the effect of data skew on our model's prediction. We use the multi-join query discussed in Section 4.3 and we fix the cardinality of R_0, R_1, R_2 and R_3 to be 2 billion, 512 million, 128 million and 32 million tuples, respectively. Attribute $R_k.b$ ranges from 1 to $|R_{k+1}|$, and the probability for $R_k.b$ to reference individual keys in $R_{k+1}.a$ follows the Zipf distribution. We vary the Zipf factor from 0 (no skew) to 1.5.

Figure 7(b) plots the normalized query response time of query plans that exhibit probe-side data skew. Since our model doesn't account for data skew, it produces the same prediction for all Zipf factors. The query plan response time remains stable when the Zipf factor is smaller than 0.5. When Zipf factor is 1, the most frequent value for R_0 occurs more than 100 million times but the drop in query response time is less than 6%. More skew affects all query plans almost proportionally for Zipf factors as high as 1.5. Given that different query plans have up to $4\times$ different response times, the relative order between the query plans does not change: our model perfectly ranks all the queries that are shown in Figure 7(b) (that is, $r_s = 1.0$).

4.3.5 Modeling longer join pipelines

After focusing on the exhaustive exploration of the query plan space for a query with three joins, we now will explore how the performance of the best right-deep tree ("R3210") and the best left-deep tree ("L3210") changes as the number of joins increases, and whether our model can predict this accurately. We generate more relations with a join ratio of

1:4 as in Section 4.3.1, and we grow each query tree accordingly. We plot the query response time when the number of joins increases for both the left-deep tree and the right-deep tree in Figure 8(a), and contrast this with the result from our prediction in Figure 8(b). The left-deep tree and the right-deep tree have the same performance when the number of joins is 1, as the trees are identical. As the number of joins increases, the performance of right-deep trees linearly worsens over the performance of the left-deep tree. The key drawback of the right-deep tree is that by using the largest relation as the probing relation the data of the largest relation "flows" through all pipeline stages and results in additional RR memory accesses in every stage. Our model accurately forecasts this performance trend for left-deep trees and right-deep trees as the number of joins increases.

5. Concluding remarks

A significant fraction of a database may fit entirely in main memory and can be analyzed in parallel using multiple CPU cores to keep query response times short. Accurately forecasting the response time of different in-memory query plans is becoming important for query optimization in this environment. Towards this goal, we contribute a cost model that can accurately predict the query response time of ad-hoc query plans with multiple hash-based joins.

A surprising insight from our model is that some left-deep query trees can be more efficient than their bushy and right-deep tree counterparts because they result in less memory I/O during execution. Prior work in parallel hash-based multi-join query evaluation has advocated right-deep query trees because all build subtrees can be processed in parallel and then probed in a single pipeline [45]. Our experimental evaluation corroborates that in a main-memory setting evaluating query subtrees sequentially using all threads is as fast as evaluating separate query subtrees concurrently. This finding suggests that evaluating one join at a time and storing the intermediate result in a hash table may be a viable query execution strategy over a memory-resident dataset as it can ameliorate the cascading effect of errors in join cardinality estimation. Query processing techniques that are adaptive to cardinality estimation errors are a promising research area for future work.

References

[1] R. Ahmed, R. Sen, M. Poess, and S. Chakkappen. Of snow-storms and bushy trees. *PVLDB*, 7(13):1452–1461, 2014.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *VLDB*, pages 266–277, 1999.

[3] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. Learning-based query performance modeling and prediction. In *ICDE*, pages 390–401, 2012.

[4] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, 2013.

[5] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *ICDE*, pages 362–373, 2013.

[6] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. Memory-efficient hash joins. *Proc. VLDB Endow.*, 8(4):353–364, Dec. 2014.

[7] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using RDMA. In *SIGMOD*, pages 1463–1475, 2015.

[8] S. Blanas and J. M. Patel. Memory footprint matters: efficient equi-join algorithms for main memory data processing. In *SOCC*, pages 19:1–19:16, 2013.

[9] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, pages 54–65, 1999.

[10] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[11] G. Candea, N. Polyzotis, and R. Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *PVLDB*, 2(1):277–288, 2009.

[12] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43, 1998.

[13] M.-S. Chen, M.-L. Lo, P. S. Yu, and H. C. Young. Using segmented right-deep trees for the execution of pipelined hash joins. In *VLDB*, pages 15–26, 1992.

[14] M.-S. Chen, P. S. Yu, and K.-L. Wu. Scheduling and processor allocation for parallel execution of multi-join queries. In *ICDE*, pages 58–67, 1992.

[15] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. In *ICDE*, pages 116–127, 2004.

[16] S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD*, pages 63–78, 2015.

[17] A. Deshpande and L. Hellerstein. Flow algorithms for parallel query optimization. In *ICDE*, pages 754–763, 2008.

[18] D. J. DeWitt and R. H. Gerber. Multiprocessor hash-based join algorithms. In *VLDB*, pages 151–164, 1985.

[19] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *SIGMOD*, pages 1–8, 1984.

[20] J. Duggan, U. Çetintemel, O. Papaemmanouil, and E. Upfal. Performance prediction for concurrent database workloads. In *SIGMOD*, pages 337–348, 2011.

[21] J. Giceva, G. Alonso, T. Roscoe, and T. Harris. Deployment of query plans on multicores. *PVLDB*, pages 233–244, 2014.

[22] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD*, pages 102–111, 1990.

[23] Z. Gu, M. A. Soliman, and F. M. Waas. Testing the accuracy of query optimizers. In *DBTest*, pages 11:1–11:6, 2012.

[24] L. M. Haas, M. J. Carey, M. Livny, and A. Shukla. Seeking the truth about ad hoc join costs. *VLDB J.*, 6(3):241–256, 1997.

[25] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in XPRS. *Distributed and Parallel Databases*, 1(1):9–32, 1993.

[26] Y. E. Ioannidis and Y. C. Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *SIGMOD*, pages 168–177, 1991.

[27] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624, 2010.

[28] H. Lang, V. Leis, M. Albutiu, T. Neumann, and A. Kemper. Massively parallel numa-aware hash joins. In *IMDM*, pages 1–12, 2013.

[29] T. J. Lehman and M. J. Carey. Query processing in main memory database management systems. In *SIGMOD*, pages 239–250, 1986.

[30] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754, 2014.

[31] J. Li, A. C. König, V. R. Narasayya, and S. Chaudhuri. Robust estimation of resource consumption for SQL queries using statistical techniques. *PVLDB*, 5(11):1555–1566, 2012.

[32] Y. Li, I. Pandis, R. Müller, V. Raman, and G. M. Lohman. NUMA-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[33] B. Liu and E. A. Rundensteiner. Revisiting pipelined parallelism in multi-join query processing. In *VLDB*, pages 829–840, 2005.

[34] F. Liu and S. Blanas. Forecasting the cost of processing multi-join queries via hashing for main-memory databases (extended version). arXiv:1507.03049.

[35] G. M. Lohman. Is query optimization a "solved" problem? The ACM SIGMOD Blog, Apr. 2014.

[36] G. M. Lohman, C. Mohan, L. M. Haas, D. Daniels, B. G. Lindsay, P. G. Selinger, and P. F. Wilms. Query processing in R*. In *Query Processing in Database Systems*, pages 31–47. Springer, 1985.

[37] H. Lu, M.-C. Shan, and K.-L. Tan. Optimization of multi-way join queries for parallel execution. In *VLDB*, pages 549–560, 1991.

[38] S. Manegold, P. A. Boncz, and M. L. Kersten. What happens during a join? Dissecting CPU and memory optimization effects. In *VLDB*, pages 339–350, 2000.

[39] S. Manegold, P. Boncz, and M. L. Kersten. Generic database cost models for hierarchical memory systems. In *VLDB*, pages 191–202, 2002.

[40] V. Markl and G. M. Lohman. Learning table access cardinalities with LEO. In *SIGMOD*, page 613, 2002.

[41] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In *VLDB*, pages 314–325, 1990.

[42] O. Polychroniou and K. A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *SIGMOD*, pages 755–766, 2014.

[43] O. Polychroniou, R. Sen, and K. A. Ross. Track join: Distributed joins with minimal network traffic. In *SIGMOD*, SIGMOD '14, pages 1483–1494, 2014.

[44] Pythia. `http://github.org/sblanas/pythia`.

[45] D. A. Schneider and D. J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *VLDB*, pages 469–480, 1990.

[46] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *VLDB*, pages 510–521, 1994.

[47] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO – DB2's LEarning Optimizer. In *VLDB*, pages 19–28, 2001.

[48] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *PDIS*, pages 68–77, 1991.

[49] A. N. Wilschut, J. Flokstra, and P. M. Apers. Parallel evaluation of multi-join queries. In *ACM SIGMOD Record*, volume 24, pages 115–126. ACM, 1995.

[50] W. Wu, X. Wu, H. Hacigümüs, and J. F. Naughton. Uncertainty aware query execution time prediction. *PVLDB*, 7(14): 1857–1868, 2014.

[51] N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, and C. Zhang. Statistical learning techniques for costing XML queries. In *VLDB*, pages 289–300, 2005.