Inspector Joins

Shimin Chen[†] Anastassia Ailamaki[†] Phillip B. Gibbons[‡] Todd C. Mowry[†,‡]

[†]Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
{*chensm,natassa,tcm*}*@cs.cmu.edu*

[‡]Intel Research Pittsburgh
4720 Forbes Avenue, Suite 410
Pittsburgh, PA 15213, USA
phillip.b.gibbons@intel.com

Abstract

The key idea behind *Inspector Joins* is that during the I/O partitioning phase of a hash-based join, we have the opportunity to look at the actual data itself and then use this knowledge in two ways: (1) to create specialized indexes, specific to the given query on the given data, for optimizing the CPU cache performance of the subsequent join phase of the algorithm, and (2) to decide which join phase algorithm best suits this specific query. We show how inspector joins, employing novel statistics and specialized indexes, match or exceed the performance of state-of-the-art cache-friendly hash join algorithms. For example, when run on eight or more processors, our experiments show that inspector joins offer 1.1–1.4X speedups over these previous algorithms, with the speedup increasing as the number of processors increases.

1 Introduction

Our ability to minimize the execution time of queries often depends upon the quality of the information we have about the underlying data and the existence of suitable indexes on that data. Thus, database management systems (DBMS) maintain various statistics and indexes on each relation, which fuel all of the optimizer's decisions. Because it is not feasible to maintain statistics and indexes specific to every query, the DBMS must rely on *general* statistics and indexes on the relations in order to optimize and process *specific* queries, often resulting in incorrect decisions and ineffective access methods. This problem is particularly acute for join queries, where (1) characteristics of the join result often must be inferred from statistics on the individual input relations and (2) it is impractical to maintain indexes suitable for all join query and predicate combinations. In this paper, we address this problem in the context of hash joins, one of the most frequent join algorithms.

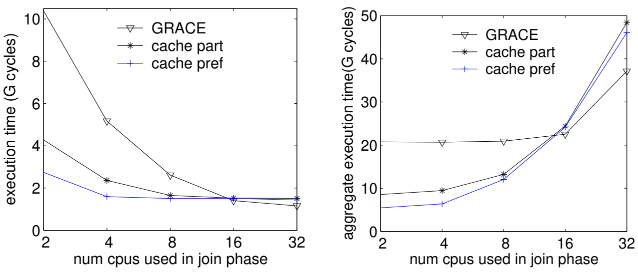
Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005

The key observation in this paper is that because hash-based join algorithms visit all the data in the I/O partitioning phase before they produce their first output tuple, we have the opportunity to *inspect* the data during this earlier pass and then use this knowledge to optimize the subsequent join phase of the algorithm. In particular, we show how statistics and specialized indexes, *specific to the given query on the given data*, can be used to significantly reduce the primary performance bottleneck in hash joins, namely, the poor CPU cache performance caused by the random memory accesses when building and probing hash tables [3, 4, 12, 17]. Although improving hash join cache performance has been the focus of many recent studies [3, 4, 12, 17], we show that our approach, which we call *Inspector Joins*, matches or exceeds the performance of state-of-the-art hash join algorithms, achieving up to 1.4X speedups. Moreover, the specialized indexes created by inspector joins are particularly well-suited to two common join scenarios: foreign key joins and joins between two nearly-sorted relations.[1]

1.1 Previous Cache-Friendly Approaches

Previous studies propose two approaches to improving the CPU cache performance of hash joins: cache partitioning and cache prefetching. Given a pair of build and probe partitions in the join phase, cache partitioning [3, 12, 17] recursively divides the two memory-sized partitions into cache-sized sub-partitions so that a build sub-partition and its hash table fit into the CPU cache, thus reducing the number of cache misses caused by hash table visits. However, the re-partition cost is so significant that cache partitioning is at least 50% worse than cache prefetching for foreign key joins [4]. Moreover, cache partitioning is sensitive to cache interference by other concurrent activities in the system because it assumes exclusive use of the cache. Cache prefetching [4] exploits memory system parallelism in today's processors and uses software prefetch instructions to overlap cache misses with computation. The cache prefetching techniques are effective only when there is sufficient memory bandwidth. However, modern database servers typically run on multiprocessor systems. In an SMP (symmetric multiprocessing) system, the entire mem-

[1] Joins between nearly-sorted relations arise, for example, in the TPC-H benchmark, where the lineitem table and the orders table are nearly sorted on the (joining) order keys. We also observe joins between nearly-sorted relations in a commercial workload.



(a) Execution time varying the number of CPUs

(b) Aggregate time of all CPUs used in join phase

Figure 1: Impact of memory bandwidth sharing on join phase performance in an SMP system

ory bandwidth is shared across all the processors. Because cache prefetching essentially trades off bandwidth for reduced execution time, its benefit gradually disappears as more and more processors eagerly compete for the limited memory bandwidth.

Figure 1 shows the join phase performance of joining a 500MB build relation with a 2GB probe relation, varying the number of CPUs used in the join phase. In each experiment, the number of I/O partitions generated is a multiple of the number of CPUs. Then the same join phase algorithm is run on every CPU to process different partitions in parallel. (Please see Section 6.1 for setup details.) As shown in Figure 1(a), both cache partitioning (*cache part*) and cache prefetching (*cache pref*) perform significantly better than the original GRACE hash join. Cache partitioning is worse than cache prefetching because of the re-partition cost. The effect of memory bandwidth sharing is more clearly shown in Figure 1(b), which reports the total aggregate time of all CPUs for the join phase for this same experiment. We can see that the benefit of cache prefetching gradually disappears as more and more processors are competing for the memory bandwidth. Cache prefetching becomes even worse than the GRACE hash join when there are 16 processors or more. Interestingly, the GRACE hash join also suffers from memory bandwidth sharing when there are 32 processors.

1.2 The Inspector Join Approach

To achieve good performance even when memory bandwidth is limited, we need to reduce the *number* of cache misses of the join phase algorithm, in addition to applying prefetching techniques to hide cache miss latencies. Our approach exploits the multi-pass structure of the hash join algorithm. During the I/O partitioning phase, inspector joins create a special multi-filter-based index with little overhead; this index will enable us to have "in-place" cache-sized sub-partitions of the build table. Unlike cache partitioning our approach reduces the number of cache misses without moving tuples around. The join phase, which we refer to as a *cache-stationary join phase* because of its in-place nature, is performed using the index.

Our cache-stationary join phase is specially designed for joins with nearly unique build join keys, which include primary-foreign key joins, the majority of all the real-world joins. On the other hand, if probe tuples frequently match multiple build tuples in a given join query, the cache-stationary join phase is not the best choice. An inspector join can detect this condition during its inspection and switch to use a different join phase algorithm (see Section 6.5 for details). Moreover, as mentioned above, inspector joins can detect nearly-sorted relations (after any predicates being applied before the join). Our initial intuition was that a sort-merge based join phase algorithm should be applied in this case. However, surprisingly, the cache-stationary join phase performs equally well, due to the effectiveness of its multi-filter-based index.

1.3 Contributions

This paper makes the following contributions. First, we propose and study the inspector join, which to our knowledge, is the first hash join algorithm that exploits the free information obtained from one pass of the algorithm to improve the performance of a later pass. Second, we propose a specialized index that addresses the memory bandwidth sharing problem, and can take advantage of nearly-sorted relations. Moreover, we utilize cache prefetching to improve the robustness of inspector joins in the face of cache interference. Third, we present an illustrative example of how inspector joins can use its collected statistics to select between two join phase algorithms for the given query and data. Finally, our experiments demonstrate that as we run on 8 or more processors, inspector joins achieve 1.1–1.4X speedups over previous state-of-the-art cache prefetching and cache partitioning algorithms, with the speedup increasing as the number of processors increases.

The paper is organized as follows. Section 2 discusses related work. Section 3 illustrates the high level ideas in our solution. Section 4 and 5 describe our algorithms in detail. Section 6 presents our experimental results. Finally, Section 7 concludes the paper.

2 Related Work

Hash join cache performance. Hash join has been studied extensively over the past two decades [5, 10, 11, 16]. Because of its I/O partitioning approach, a hash join sequentially visits the disk pages in source relations and intermediate partitions. By using advanced I/O techniques, such as I/O prefetching, hash join is CPU bound if there is sufficient I/O bandwidth [4]. Therefore, this paper focuses on the CPU cache performance of the algorithm.

Previous studies show how cache partitioning and/or cache prefetching can be used to improve the CPU cache performance of hash joins. Shatdal *et al.* show that cache partitioning achieves 6-10% improvement for joining memory-resident relations with 100B tuples [17]. Boncz, Manegold and Kersten propose using multiple passes in cache partitioning to avoid cache and TLB thrashing when joining vertically-partitioned relations (essentially joining two 8B columns) [3, 12]. However, Chen *et al.* show that when the tuple size is more than 20B, the re-partition cost of cache partitioning is so significant that cache partitioning is at least 50% worse than cache prefetching [4]. Chen *et al.* propose exploiting the inter-tuple parallelism to overlap the cache misses of a tuple with the processing of multiple tuples [4]. They propose two prefetching algorithms, group prefetching and software-pipelined prefetching, and evaluate their performance through detailed cycle-by-cycle simulations.[2]

As shown in Figure 1, the performance of cache prefetching degrades significantly when more and more CPUs are eagerly competing for the memory bandwidth in a multiprocessor system. Our inspector joins exploit information collected in the I/O partitioning phase to fit address-range-based sub-partitions in cache, thus reducing the number of cache misses without incurring additional copying cost. Our approach is effective for tuples that are 20B or more. For smaller tuples, we revert to cache partitioning.

Inspection concept. Several studies exploit information collected while processing queries previously submitted to the DBMS: reusing partial query results in multi-query optimization [15], maintaining and using materialized views [1], creating and using join indices [19], and collecting up-to-date statistics for future query optimizations [18]. Unlike these studies the inspection and use of the information in our approach are specific to a single query. Therefore, we avoid the complexities of deciding what information to keep and how to reuse data across multiple related queries. Moreover, our approach is effective for any join query and predicate combinations.

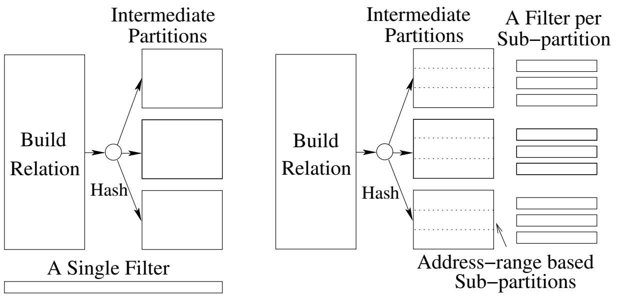
Dynamic re-optimization techniques augment query plans with special operators that collect statistics about the actual data during the execution of a query [9, 13]. If the operator detects that the actual statistics deviate considerably from the optimizer's estimates, the current execution plan is stopped and a new plan is used for the remainder of the query. Compared to the *global* re-optimization of query plans, our inspection approach can be regarded as a *complementary, local* optimization technique inside the hash join operator. When hash joins are used in the execution plan, our inspection approach creates specialized indexes to enable the novel cache-stationary optimization and allows informed choice of join phase algorithms. Because the indexes and informed choice account for which tuples will actually join as well as their physical layout within the intermediate partitions, this functionality cannot be achieved by operators *outside* the join operator.

3 Inspector Joins: Overview

In this section, we describe (i) how we create the multi-filters as a result of data inspection, (ii) how we minimize the number of cache misses without moving any tuples around, (iii) how we exploit cache prefetching to hide the remaining cache misses and to improve robustness against cache interference, and (iv) how we choose join phase algorithms based on obtained information about the data.

3.1 Inspecting the Data: Multi-Filters

While partitioning tables for a hash join, commercial DBMS often construct a filter to quickly discard probe tu-



(a) Single-filter scheme (b) Multi-filter scheme

Figure 2: Using multiple filters to inspect the data

ples that do not have any matches [11]. Such filters may improve join performance significantly when a large number of probe tuples do not match any tuples in the build relation (e.g., there is a predicate on the build relation in a foreign-key join). As shown in Figure 2(a), a single filter is computed from all the build tuples to approximately represent all the join attribute values in the build relation. Testing a value against the filter is conservative: while a negative answer means that the value is not in the filter, false positives may occur with a low probability. (Bloom filters, detailed in Section 4, are a typical example.) When partitioning the probe relation, the algorithm tests every probe tuple against the filter. If the test result is negative for a tuple, the algorithm simply drops the tuple, thus saving the cost of writing it to disk and processing it in the join phase.

Instead of using a single large filter that represents the entire build relation, the inspector join creates multiple shorter filters (illustrated in Figure 2(b)), each representing a disjoint subset of build tuples. Testing a probe tuple against the filters will (conservatively) show which subsets the probe tuple has matches in. The build relation subsets are address-range-based sub-partitions; that is, a subset represents all build tuples in K consecutive pages in a build partition. K is chosen to make the sub-partition fit in the cache in the join phase, as will be described in Section 3.2.

The inspector join builds the set of small filters by inspecting the build relation during the partitioning phase. To keep track of the sub-partition boundaries, we use a page counter for every partition. Then, every build tuple is used to compute the filter corresponding to the sub-partition the tuple belongs to. Note that the multi-filter scheme tests filters differently than the single-filter scheme. For every probe tuple, after computing its destination partition, the algorithm checks the join attribute value in the tuple against *all* the filters in the partition. The algorithm drops the probe tuple only if *all* filter tests for all sub-partitions are negative. The positive tests show which sub-partition(s) may contain matching build tuples of the probe tuple, and this information is used in the join phase of the inspector algorithm. Section 4 demonstrates that our multi-filter scheme incurs the same number of cache misses as the single-filter scheme during the inspection and filter-construction phase, and it can achieve the same aggregate false positive rate with moderate memory space requirements.

[2] The cache prefetching algorithms require faulting prefetch instructions, meaning that a prefetch should succeed even if it incurs a TLB miss. IA64 processors support faulting prefetches [7]. We believe that since faulting prefetches are very important to database applications, they will be supported in more and more processors.

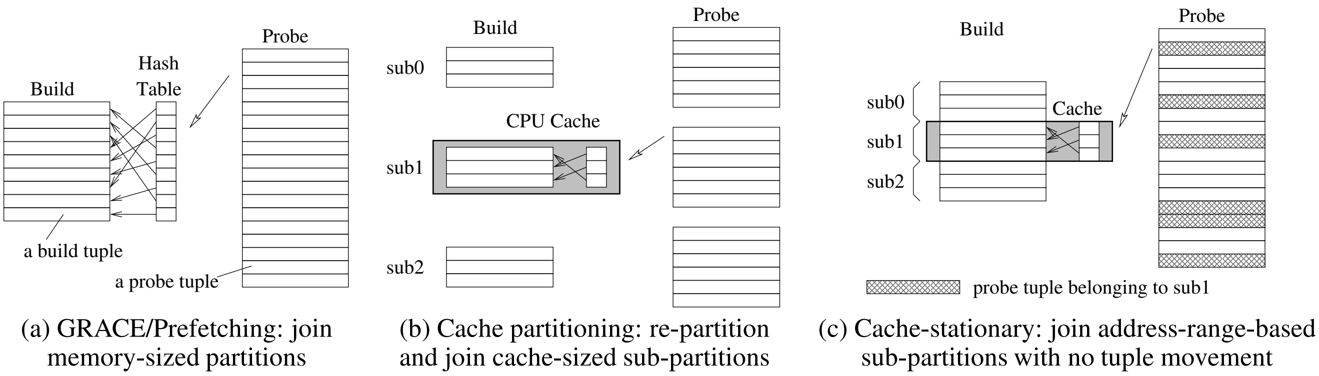


Figure 3: Comparing the cache behaviors of different join phase algorithms

Table 1: Terminology used throughout this paper

Name	Definition
P	page size (in bytes)
N	# of build tuples in build relation
n	# of build tuples per partition page
H	# of bytes in hash table for every key
C	effective cache size (in bytes)[3]
L	cache line size (in bytes)
K	# of build pages per sub-partition
S	# of sub-partitions per build partition
bpk	# of bits per key for a single Bloom filter
fpr	Bloom filter false positive rate

3.2 Improving Locality for Stationary Tuples

During the join phase, the inspector join algorithm knows which probe tuples match every address-range-based sub-partition of the build relation, and therefore processes tuples one sub-partition at a time. For every sub-partition, the algorithm builds a cache-resident hash table on the build tuples, and probes it with all the probe tuples associated with this sub-partition. We ensure that the build tuples of a sub-partition and its hash table fit into the cache by choosing the number of pages per build sub-partition, K, as follows:

$$K \cdot P + K \cdot n \cdot H \leq C$$

The variables used above and throughout the paper are summarized in Table 1.

Figure 3 compares the cache behaviors of all the join-phase algorithms that we are considering. As shown in Figure 3(a), the GRACE algorithm joins memory-sized partitions. It builds an in-memory hash table on all the build tuples, then probes this hash table using every tuple in the probe partition to find matches. Because of the inherent randomness of hashing, accesses to the hash table have little temporal or spatial locality. Since the build partition and its hash table are typically much larger than the CPU cache size, these accesses often incur expensive cache misses, resulting in poor CPU cache performance.

Figure 3(a) shows that the cache prefetching algorithms [4] perform the join in the same way as the GRACE algorithm [10]. The prefetching algorithms do not reduce the number of cache misses; rather, they use prefetch instructions to hide the time needed to service cache misses when repeatedly visiting the hash table. To achieve that, they rely on sufficient memory bandwidth to quickly service cache misses and prefetch requests. When multiple processors compete for limited bandwidth, however, the performance of the prefetching algorithms is likely to degrade significantly, as shown previously in Figure 1.

Figure 3(b) illustrates how cache partitioning joins pairs of cache-sized partitions to avoid excessive cache misses because of hash table visits. The algorithm recursively partitions memory-sized partitions into cache-sized sub-partitions, and then joins the sub-partitions using cache-resident hash tables. Essentially, cache partitioning visits every tuple at least once more than the GRACE algorithm, thereby incurring significant re-partition cost, as shown previously in Figure 1.

Figure 3(c) shows the cache behavior of the cache-stationary join phase of inspector join. It reads memory-sized partitions into memory and processes one cache-sized partition at a time, avoiding the cache misses caused by hash table visits. It simply visits the consecutive pages of a build sub-partition to build a hash table. Random memory accesses are avoided while building the hash table because the build sub-partition and the hash table fit into the cache. Since the algorithm already knows which probe tuples may have matches in the given build sub-partition, it can directly visit these probe tuples in place without moving them. Compared to cache partitioning, inspector joins eliminate unnecessary cache misses without moving any tuples, thereby avoid the excessive re-partitioning overhead. The algorithm almost never revisits probe tuples when join attribute values in the build relation are unique (or almost unique). Values in the build relation are unique, for instance, in foreign-key joins, which constitute most of the real-world joins. (As detailed below, the inspector join verifies the assumption and selects one of the other join phase algorithms when the assumption does not hold.) Moreover, the algorithm utilizes prefetching techniques to further hide the latency for the probe tuples, as we describe in the next subsection.

3.3 Exploiting Cache Prefetching

We exploit cache prefetching techniques in addition to using cache-sized sub-partitions for two reasons. First, cache prefetching can hide the latency of the remaining cache

[3] Note that we usually set C to be a fraction (e.g. 0.5) of the total cache size so that call stacks and other frequently used data structures can stay in the cache as well.

misses, such as the cold cache misses that bring a build sub-partition and its hash table into the CPU cache, and the cache misses for accessing the probe tuples. Second, cache prefetching can improve the robustness of our algorithm when there is interference with other processes running concurrently in the system. As shown in [4], cache partitioning performance degrades significantly when the CPU cache is flushed every 2-10 ms, which is comparable to typical thread scheduling time. To cope with this problem, we issue prefetch instructions as a safety net for important data items that should be kept in the cache, such as the build tuples in a build sub-partition. If the data item is in cache, there is no noticeable penalty. On the other hand, if the data item has been evicted from cache, the prefetch instruction brings it back into the cache significantly earlier, making this approach worthwhile. In a sense, we use double measures to maximize cache performance when accessing important data items.

3.4 Choosing the Best Join Phase Algorithm

Based on the statistics collected from the actual data in the partition and inspection phase, inspector joins can choose the join phase algorithm best suited to the given query. For example, we detect duplicate build keys by counting the number of sub-partitions each probe tuple matches. Since a probe tuple must be tested against all the possible matching sub-partitions for correctness, the time for the cache-stationary join phase of the inspector join increases with the number of duplicate build keys. When this number is above a threshold, inspector joins select a different join phase algorithm, as will be shown in Section 6.5.

Our inspection approach can also detect relations that are nearly-sorted on the join key. Our initial intuition is that a sort-merge based join phase should be applied in this case. To verify our intuition, we implemented an inspection mechanism to detect nearly-sorted tuples. The basic idea is to keep tuples that are out of order in a memory buffer when partitioning an input relation. The input is nearly sorted if the memory buffer does not overflow when all the tuples are read. At this point, all the intermediate partitions contain in-order tuples. We then partition the (small number of) out-of-order tuples and store them separately from the in-order tuples. In the join phase, given four inputs per partition (out-of-order and in-order build and probe inputs), the sort-merge algorithm first sorts the out-of-order inputs and then merges all four inputs to find matching tuples. Surprisingly, we find in our experiments that the cache-stationary join phase performs as well as the sort-merge implementation. We will discuss the results in Section 6.5.

4 I/O Partition and Inspection Phase

In this section, we begin by introducing a typical filter implementation: Bloom filters. Then, we discuss the memory space requirement of our multi-filter scheme, and we illustrate how our scheme achieves the same number of cache misses as the single-filter scheme. Finally, we describe the I/O partition and inspection algorithm that uses the multi-filter scheme to determine the sub-partition information for probe tuples.

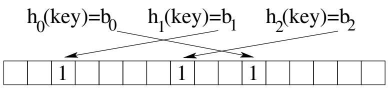


Figure 4: A Bloom filter with three hash functions

Table 2: Number of bits per key (bpk) under different false positive rates (fpr) with $d = 3$

fpr	0.1	0.05	0.01	0.005	0.001	0.0005
bpk	4.808	6.529	12.364	15.997	28.474	36.277

4.1 Bloom Filters: Background

A Bloom filter represents a set of keys and supports membership tests [2]. As shown in Figure 4, a Bloom filter consists of a bit vector and d independent hash functions, $h_0, h_1, \ldots, h_{d-1}$ ($d = 3$ in the figure). To represent a set of keys, we first initialize all the bits in the bit vector to be 0. Then, for every key, we compute d bit positions using the hash functions and set the bits to 1 in the bit vector. (A bit may be set multiple times by multiple keys.)

To check whether a test key exists in the set of known keys, we compute d bit positions for the test key using the hash functions and check the bits in the bit vector. If some of the d bits are 0, the set of known keys can not contain the test key. If all of the d bits are 1, the test key may or may not exist in the set of known keys. Therefore, Bloom filter tests may generate false positives but may never generate false negative results.

Intuitively, the larger the Bloom filter vector size, the smaller the probability that a test generates a false positive, which is called the false positive rate. In fact, the false positive rate fpr and the number of bits per key bpk of the bit vector are closely related to each other [2]:

$$fpr \approx (1 - e^{-d/bpk})^d, \text{when bit vector size} \gg 1$$

Table 2 shows the bpk values for various fpr. In this paper, we only consider Bloom filters with $d = 3$. We point out, however, that our algorithm works for any choice of d.

4.2 Memory Space Requirement

In a single filter scheme, the total size of the filter in bytes can be computed as follows, where N is the total number of build tuples (assuming that keys are unique):

$$total\_filter\_size_{single} = bpk \cdot N/8$$

Our multi-filter scheme constructs a filter per sub-partition in every build partition. However, the filters represent disjoint subsets of build tuples; every build tuple belongs to one and only one sub-partition. Therefore, every build tuple is still represented by a single filter. Let bpk' be the number of bits per key for an individual filter. Then the total filter size of the multi-filter scheme is:

$$total\_filter\_size_{multi} = bpk' \cdot N/8$$

We can quantify the increase in memory space by using the ratio between the filter sizes of the multi-filter and the single-filter schemes:

$$space\_increase\_ratio = \frac{total\_filter\_size_{multi}}{total\_filter\_size_{single}} = \frac{bpk'}{bpk}$$

Table 3: Total filter size varying build tuple size (1GB build relation, $fpr = 0.05$, $S = 50$ sub-partitions)

tuple size	20B	60B	100B	140B
# of tuples	50M	16.7M	10M	7.1M
single-filter	40.8MB	13.6MB	8.2MB	5.8MB
multi-filter	178.0MB	59.4MB	35.6MB	25.3MB

To obtain bpk', we need to compute the false positive rate fpr' for an individual filter in the multi-filter scheme.

Suppose there are S sub-partitions per build partition. Then, a probe tuple will be checked against all the S filters in the partition to which the probe tuple is hashed. If any filter test is positive, the join phase algorithm has to join the probe tuple with the corresponding build sub-partition for matches. In order to keep the number of additional probes caused by false positives the same as the single-filter scheme, the single-filter scheme fpr and the individual fpr' of the multi-filter scheme should satisfy:

$$fpr' = fpr/S$$

For example, if the single-filter scheme's fpr is 0.05, we can compute the space increase ratio as follows. Since $fpr' = fpr/S$, $fpr' = 0.001$ if $S = 50$. Then, $bpk = 6.529$ and $bpk' = 28.474$, according to Table 2. Therefore, $space\_increase\_ratio$ is 4.4. Similarly, if $S = 100$, we can compute that $space\_increase\_ratio$ is 5.6.

Table 3 compares the filter size of the multi-filter scheme with the single-filter scheme when the aggregate false positive rate is 0.05 and there are 50 sub-partitions per partition.[4] The build relation is 1GB large, and we vary the tuple size from 20 to 140 bytes. We can see that the space requirement is moderate when the tuple size is greater than or equal to 100B, which is typical in most real-world applications. Even if the tuple size is as small as 20B, the memory requirement of 178MB can still be satisfied easily in today's database servers.[5]

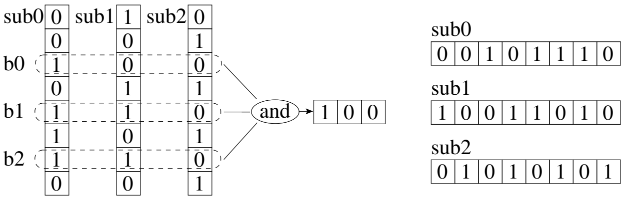
4.3 Minimizing the Number of Cache Misses

The single-filter scheme writes three bits in the Bloom filter for every build tuple. For every probe tuple, it reads three bits in the Bloom filter. Since the bit positions are random because of the independent hash functions, the single-filter scheme potentially incurs three cache misses for every build tuple and for every probe tuple, assuming the total filter size is larger than the CPU cache size. (We do not use our algorithm if the relation is so small that the computed single filter size is smaller than cache, but the total size of the multiple filters may be larger than cache.)

In the multi-filter scheme, a build tuple is still represented by a single filter corresponding to its sub-partition. Therefore, the multi-filter scheme still writes three bits for every build tuple, incurring the same number of cache misses as the multi-filter scheme.

[4] $S = 50$ is a reasonable choice. Even if the cache size is as small as 1MB, and the I/O partition phase can produce up to 500 partitions (limited by the capability of the storage manager), it allows the build relation size to be as large as 25GB.

[5] Hash join may choose to hold intermediate partition pages in memory. Therefore, the above additional memory space requirement may result in extra I/Os. However, hash join is CPU bound with reasonable I/O bandwidth [4], and this is a minor effect.



(a) Vertical layout (b) Horizontal layout

Figure 5: Layouts of multiple Bloom filters

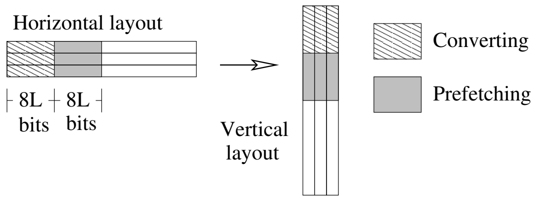


Figure 6: Horizontal to vertical layout conversion

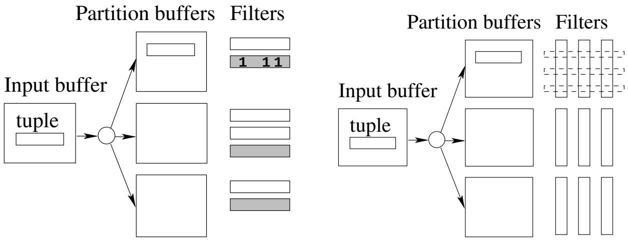
However, the multi-filter scheme checks S filters for every probe tuple, where S is the number of sub-partitions per partition. We ensure that the filters are of the same size. Given a probe tuple, the multi-filter scheme still computes the same number of bit positions as the single-filter scheme, then it simply checks the same bit positions for all S filters. However, the filter tests may incur $3S$ cache misses, which is much more than the single-filter scheme.

This problem can be solved by laying out the filters vertically for every partition. As shown in Figure 5(a), the bits at the same bit position in all the filters of a partition are consecutive in memory. That is, the first bits of all the filters are stored together, which are followed by the second bits of all the filters, so on so forth. Note that the cache line size is typically 32B to 128B, or 256-1024 bits, which is much larger than the number of filters per partition S. Therefore, under the vertical layout, we can read the bits of a given position from all the filters while incurring only a *single* cache miss. In this way, the multi-filter scheme can keep the number of cache misses the same as the single-filter scheme for testing a probe tuple.

Figure 5(a) shows that we can test all the filters for a given probe tuple using a bit operation under the vertical layout. We simply compute a bit-wise AND operation of the b_0 bits, the b_1 bits, and the b_2 bits. A 1 in the result means all three bits for the corresponding filter are 1. Therefore, a 1/0 resulting bit means a positive/negative test result for the corresponding filter.

A new problem occurs when we lay out the filters vertically: new filters can not be easily allocated and the number of filters in a partition must be determined before allocating the memory space for the vertical filters. Since the actual partition size may vary due to data skew, using the maximal possible number of sub-partitions may waste a lot of memory space.

We solve this dynamic allocation problem by using horizontal layout when partitioning the build relations and generating the filters, as shown in Figure 5(b). Then, we con-



(a) Partition build relation (b) Partition probe relation

Figure 7: I/O partition algorithm

vert the horizontal layout into an equivalent vertical layout before partitioning the probe relation.

Figure 6 illustrates the conversion algorithm. Horizontal filters are allocated at cache line boundaries. We transpose the filters one block at a time. Every block consists of a cache line ($8L$ bits) for all the filters. The source cache lines of different filters in the horizontal layout are not contiguous in memory, while the destination block is a continuous chunk of memory. Every outer-loop iteration of the algorithm prefetches the next source and destination blocks in addition to converting the current block. In this way, we hide most of the cache miss latency of accessing the source and destination filters.

4.4 Partition and Inspection Phase Algorithm

The algorithm consists of the following three steps:

1. Partition build relation and compute horizontal filters;
2. Convert horizontal filters to vertical layout;
3. Partition probe relation and test vertical filters.

We have already described the algorithm for step 2. This subsection describes the other two steps in the algorithm.

As shown in Figure 7(a), step 1 allocates an input buffer for the build relation and an output buffer for every intermediate partition. It uses horizontal filters. Each partition keeps a page counter for the outgoing pages. When the counter equals to K, the number of pages per sub-partition, a new filter is allocated from a memory pool and the counter is reset to 0. For every build tuple, the algorithm extracts the join attribute to compute a 32-bit hash code. It determines the partition number by using the hash code and copies the tuple (with projection if needed) to the output buffer. The algorithm also computes and sets the three bit positions of the current horizontal filter. For better cache performance, we employ group prefetching as described in [4]. The only difference is the addition of prefetching for the Bloom filter positions. Moreover, a tuple's hash code is stored in the page slot area as recommended in [4].[6]

As shown in Figure 7(b), Step 3 is similar to Step 1 with the following differences. First, the algorithm tests every probe tuple against the set of vertical filters in the tuple's partition. A tuple is dropped when all the resulting bits are 0. Second, positive results show which sub-partitions may contain matching tuples for the given probe tuple. The

[6] A build partition page slot consists of a 4B hash code and a 2B tuple offset. Every two slots are combined together to align the hash codes at 4B boundaries.

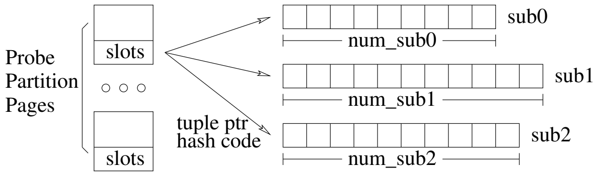


Figure 8: Extracting probe tuple information for every sub-partition using counting sort

sub-partition ID(s) is recorded in the slot area of the same output page containing the tuple.[7] In most cases, a single sub-partition ID is found. Note that slots may be of variable size now. This is not a problem since the probe slots will only be visited sequentially (in the counting sort step) in the join phase algorithm, as will be described in Section 5. Third, the number of probe tuples associated with each sub-partition is counted, which is used (in the counting sort step) in the join phase algorithm.

5 Cache-Stationary Join Phase

The join phase algorithm consists of the following steps:

1. Read build and probe partitions into main memory;
2. Extract per-sub-partition probe tuple pointers;
3. Join each pair of build and probe sub-partitions.

By using the sub-partition information collected in the partition and inspection phase, the algorithm achieves good cache performance without copying any tuples. The sub-partition information is stored in the order of probe tuples in the probe intermediate partitions. However, Step 3 visits all the probe tuples of a single sub-partition and then moves on to the next sub-partition. It requires the sub-partition information in the order of sub-partition IDs. Therefore, probe tuple sub-partition information has to be sorted before use. In the following, we first describe how Step 2 performs counting sort, then discuss the use of prefetching to improve performance and robustness in Step 3.

5.1 Counting Sort

The algorithm knows the number of sub-partitions and the number of probe tuples associated with each sub-partition; the latter is collected in the I/O partition phase. Therefore, we can use counting sort, which is a fast O(N) algorithm, for extracting probe tuple information (the probe tuple pointers and hash codes) for every sub-partition.

As shown in Figure 8, for every sub-partition, we allocate an array, whose size is equal to the number of probe tuples associated with the sub-partition. The algorithm visits the slot area of all the probe partition pages sequentially. For every slot, it computes the tuple address using the tuple offset. Then the algorithm copies the tuple address and the hash code to the destination array(s) that are specified by the sub-partition ID(s) recorded in the page slot. Assuming

[7] From high address to low address, a probe partition page slot consists of a 4B hash code, a 2B tuple offset, a 1B number of sub-partitions, a sequence of sub-partition IDs each taking 1B. We align slots on 4B boundaries and a slot takes 8B when there is a single sub-partition ID.

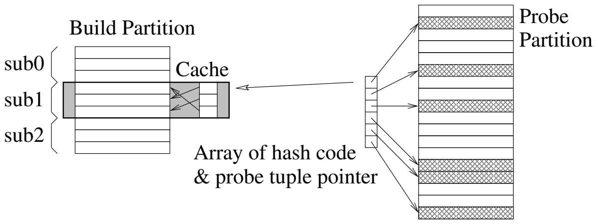


Figure 9: Joining a pair of build and probe sub-partitions

the build join attribute values are mostly unique, there is often a single sub-partition ID for a probe tuple, and the tuple address and hash code are only copied once. After processing all the probe page slots, the algorithm obtains an array of (tuple pointer, hash code) pairs for every sub-partition. Note that the tuples themselves are not visited nor copied in the counting sort.

We use cache prefetching to hide the cache miss latency of reading page slots and writing to the destination arrays. We keep a pointer to the next probe page and issue prefetches for the next page slot area while processing the slot area of the current page. Similarly, for every destination array, we keep a pointer to the next cache line starting address. We issue a prefetch instruction for the next cache line before we start using the current cache line in the array.

5.2 Exploiting Prefetching in the Join Step

For every pair of build and probe sub-partitions, the algorithm first constructs a hash table. (We assume the same hash table structure as in [4].) Since the hash codes are stored in the build page slot area, the algorithm does not need to access the actual build tuples for creating the hash table. However, we expect the build tuples to be used frequently during probing. Therefore, we issue prefetch instructions for all build tuples of the sub-partition to bring them into cache, as shown in Figure 9.

After building the hash table, the algorithm visits the array containing the probe tuple pointers and hash codes of the sub-partition, as shown in Figure 9. It probes the hash table with hash codes in the array. If a probe is successful, the algorithm visits the probe tuple and the build tuple to verify that their join attributes are actually equal. It produces an output tuple for a true match.

We issue prefetches for the probe tuples and for the array containing probe tuple information. Here, we use a special kind of prefetch instruction, non-temporal prefetches, which are supported by Intel Itanium 2 and Pentium 4 architectures [6, 8]. Non-temporal prefetches are used to read cache lines that do not have temporal locality; the cache lines are supposed to be used only once. Therefore, cache lines read by non-temporal prefetches ignore the LRU states in the cache, and they go to a particular location in the corresponding cache set, thus minimizing cache pollution by the prefetched line. Since we aim to keep the build sub-partition and the hash table in cache, minimizing the cache pollution caused by visiting other structures is exactly what we want.

To prefetch the array containing probe tuple pointers and probe hash codes, we keep a pointer p to the cache line in the array that are $dist$ lines ahead of the current cache line ($dist = 20$ in our implementation). Suppose there are m pairs of pointers and hash codes in every cache line. The algorithm first issues prefetches for the first $dist$ lines and then sets p to the beginning of $dist + 1$ line. Whenever the algorithm finishes processing m probe tuples, it issues a prefetch for the cache line pointed by p and increases p by a cache line. The algorithm checks p against the end of the array to stop prefetching. To prefetch for the probe tuples, we use a buffer to temporarily store the pairs of pointers pointing to the build and probe tuples that correspond to successful hash table probes. When this buffer is full, we visit these tuple pairs using software-pipelined prefetching.

Finally, we improve the robustness of our algorithm by issuing prefetches for the build tuples while prefetching for the probe tuples. In most cases, the build tuples are already in cache, and these prefetches do not have effects. However, if the build tuples are replaced somehow, the prefetches can bring the build tuples back into the cache quickly. We do not prefetch the hash table for the same purpose because it requires larger changes to the algorithm and therefore may incur significant run-time cost.

6 Experimental Results

In this section, we evaluate the CPU cache performance of our inspector joins against the cache prefetching and cache partitioning algorithms through detailed cycle-by-cycle simulations. Moreover, in Section 6.5, we exploit the inspection approach to detect situations where there are duplicate build keys or where relations are nearly sorted, and choose the best join phase algorithm.

6.1 Experimental Setup

Implementation Details. We implemented five hash join algorithms: group prefetching, software-pipelined prefetching, cache partitioning, enhanced cache partitioning with advanced prefetching support, and our inspector join algorithm. We store relations and intermediate partitions as disk files, and the join algorithms are implemented as stand-alone programs that read and write relations in disk files. We keep schemas and statistics in separate description files for simplicity. Statistics about the number of pages and the number of tuples are used to compute hash table sizes, numbers of partitions, and Bloom filter sizes.

Our cache prefetching implementations mainly follow the descriptions in [4]. Prefetch instructions are inserted into C++ source codes using gcc inline ASM macros. The only difference is that the algorithms utilize a single Bloom filter for removing probe tuples having no matches. We add prefetches for the Bloom filter to the group and software-pipelined prefetching algorithm in the I/O partition phase. In our experiments, we find that the performance results of the two prefetching algorithms are very similar. To simplify presentation, we only show the group prefetching curves, which are labeled as "*cache pref*".

The two cache partitioning algorithms both use the group prefetching implementation for the I/O partition phase; they perform re-partition and join cache-sized sub-partitions in the join phase. The enhanced cache partitioning performs advanced prefetching similar to that of the inspector join for joining a pair of cache-sized sub-partitions. It also performs advanced prefetching to reduce the re-partition cost. This algorithm serves as a stronger competitor to our algorithm.

In every experiment, the number of I/O partitions generated is a multiple of the number of CPUs. Then the same join phase algorithm is run on every CPU to process different partitions in parallel. The partition phase algorithms take advantage of multiple CPUs by conceptually cutting input relations into equal-sized chunks and partitioning one chunk on every CPU. Every processor generates the same number of partition outputs. The i-th build partition will conceptually consist of the i-th build output generated by every processor. The probe partitions are generated similarly. Every CPU will build its own filter(s) based on the build tuples it sees. After partitioning the build relation, the generated filters are merged. For the single-filter scheme, all filters are OR-ed together to get a single filter. For the multi-filter scheme, different CPUs actually generate horizontal filters for different sub-partitions. Therefore, the algorithm can directly perform horizontal to vertical filter conversion. Then, the same filter(s) is shared across all the CPUs for testing probe tuples.

Experimental Design. We use a simple schema for all the relations: a tuple consists of a 4-byte randomly generated join key and a fixed-length payload. An output tuple contains all the fields of the matching build and probe tuples. In all the experiments except those in Section 6.5, a probe tuple can match zero or one build tuple, and a build tuple may match one or more probe tuples. We test the performance of our solution in various situations by varying the tuple size, the number of probe tuples matching a build tuple (which is the ratio between probe and build relation sizes), and the percentage of probe tuples that have matches. We vary the latter from 5% to 100% to model the effects of a selection on a build attribute different from the join attribute.

In all our experiments, we assume the available memory size for the join phase is 50MB and the cache size is 1MB, which follow the settings in [4]. Note that when multiple join instances are running on multiple processors, the actual memory allocated is 50MB multiplied by the number of instances. For example, in the case of 32 CPUs, the total memory used for the join phase is 1600MB. The Bloom filter false positive rate for the cache prefetching algorithm, and the two cache partitioning algorithms is set to be 0.05. The individual Bloom filter false positive rate for our inspector join algorithm is set to be 0.001.

Simulation Parameters. We evaluate the CPU cache performance (of user mode executions) of all the algorithms through detailed cycle-by-cycle simulations. We generate fully-functional executables with gcc and run the programs on a simulator that models a shared-bus SMP system with 1.5GHz processors. We vary the number of processors in our experiments. The memory hierarchy is based on the Itanium 2 processor [7]. However, the simulator only supports two levels of caches while Itanium 2 processor has three levels of caches. Therefore, we choose the size of L2 cache to be in between the sizes of the L2 and L3 caches on an Itanium2 machine. The latency is also chosen to be in between the actual L2 and L3 latencies. The Itanium 2 processor supports only software-simulated integer divide. We measured on a real machine that an integer divide takes about 50 cycles. The simulator does not drop a prefetch when miss handlers are all busy and/or if it incurs a DTLB miss. This models the behavior of $lfetch.fault$ instruction of Itanium 2 processor. Important simulator parameters are shown in Table 4.

Table 4: Simulation parameters

Processor pipeline parameters	
Clock Rate	1.5 GHz
Issue Width	4 insts/cycle
Functional Units	2 Integer, 1 integer divide, 2 Memory, 1 Branch, 2 FP
Reorder Buffer Size	128 insts
Integer Multiply/Divide	4/50 cycles
All Other Integer	1 cycle
Branch Prediction Scheme	gshare [14]

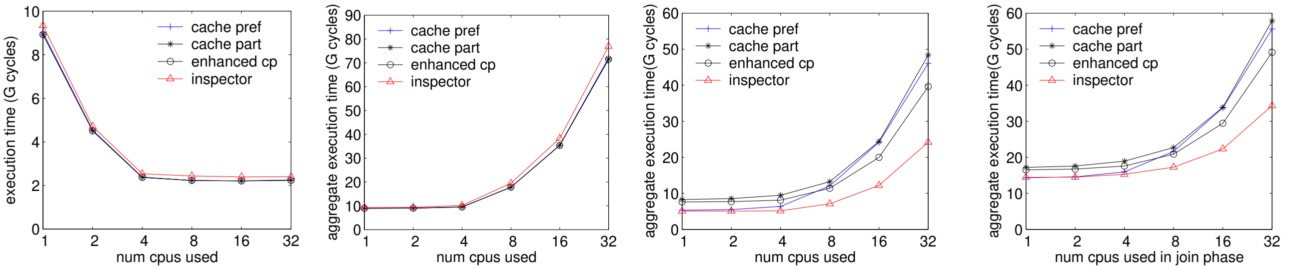
Memory parameters	
Line Size	64 bytes
Primary Instruction Cache	16 KB, 4-way set-assoc.
Primary Data Cache	16 KB, 4-way set-assoc.
Miss Handlers	32 for data, 2 for inst.
DTLB	128 entries, fully-assoc.
DTLB Miss Handlers	1
Page Size	8 KB
Unified Secondary Cache	1 MB, 8-way set-assoc.
Primary-to-Secondary Miss Latency	10 cycles (plus any delays caused by contention)
DTLB Miss Latency	30 cycles (plus any delays caused by contention)
Primary-to-Memory Miss Latency	250 cycles (plus any delays caused by contention)
Main Memory Bandwidth	1 access per 15 cycles

6.2 Varying the Number of CPUs

Figure 10 compares the performance of the algorithms while varying the number of CPUs. The legend labels and the corresponding algorithms are as follows: *cache pref* (group prefetching), *cache part* (cache partitioning), *enhanced cp* (cache partitioning enhanced with advanced prefetching support), *inspector* (inspector join). The experiments join a 500MB build relation and a 2GB probe relation. The tuple size is 100B. 50% of the probe tuples have no matches and every build tuple matches 2 probe tuples.

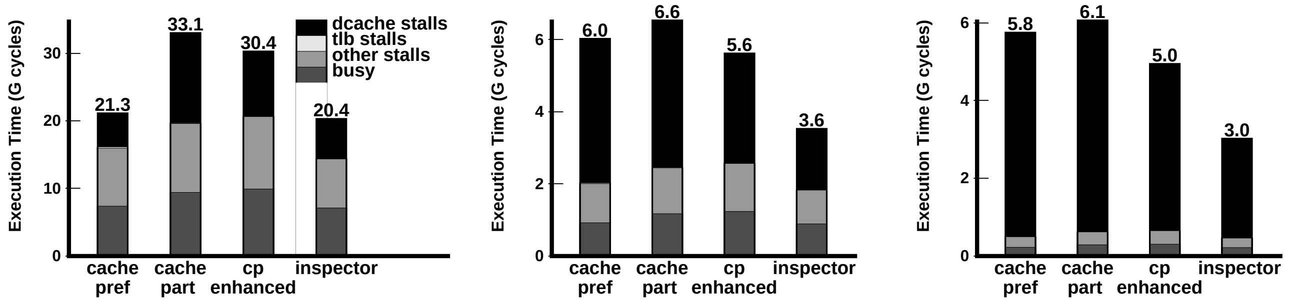
Figure 10(a) shows the partition phase performance, and Figure 10(b) shows the aggregate performance on all CPUs used in the partition phase. We see that the partition phase curves are very similar. Compared to the other schemes, the inspector join incurs a slight overhead. (The ratio between the partition phase execution times of the best algorithm and the inspector join is 0.88-0.94.) This is mainly because of the computational cost of converting horizontal filters into vertical filters and testing a set of filters. The most costly operation is extracting the bit positions of 1's from a bit vector in both conversion and filter testing. This overhead will become less significant as processors are getting faster. As shown in Figure 10(a), all the curves become flat after the 4-CPU case. Therefore, all the following experiments use up to 4 CPUs in the partition phase.

Figure 10(c) shows the total aggregate times of all CPUs for the join phase. The *cache pref* and *cache part* curves are the same as in Figure 1(b). Our inspector join is the



(a) Partition phase performance
(b) Partition phase aggregate performance
(c) Join phase aggregate performance
(d) Total aggregate performance (up to 4 CPUs in partition phase)

Figure 10: Varying the number of CPUs used



(a) num CPUs used in join phase = 1
(b) num CPUs used in join phase = 8
(c) num CPUs used in join phase = 32

Figure 11: Join phase CPU time breakdowns for CPU 0

best. Because of the memory bandwidth sharing, the cache prefetching curve degrades significantly when there are 8 or more CPUs. Since our algorithm combines locality optimizations and cache prefetching, it is less sensitive to bandwidth contention. Compared to the cache prefetching algorithms, our inspector join algorithm achieves 1.7-2.1X speedups when 8 or more CPUs are used.

As shown in Figure 10(c), the two cache partition algorithms are worse than the cache prefetching algorithm and our inspector join when there are less than 4 CPUs. This is mainly because of the large re-partition overhead, which consists of more than 36% of their join phase execution times. The enhanced algorithm is always better than the original algorithm, which verifies the effectiveness of the applied prefetching techniques. As the number of CPUs increase, the enhanced algorithm becomes significantly better than the cache prefetching algorithms because it utilizes cache-sized sub-partitions to reduce the number of cache misses. However, it still degrades quickly beyond 4 CPUs. This is mainly because the re-partition step is quite sensitive to memory bandwidth sharing. Compared to the enhanced cache partitioning algorithm, our inspector join achieves 1.6-2.0X speedups with 1-32 CPUs.

Figure 10(d) shows the aggregate performance of both phases using up to 4 CPUs in the partition phase. When there are 8 or more CPUs, inspector join achieves 1.3-1.7X speedups over the cache prefetching algorithm and the enhanced cache partitioning algorithm.[8]

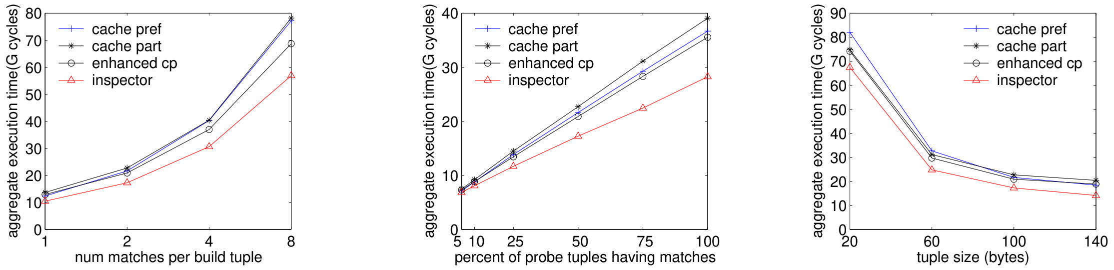
Figure 11 shows the CPU time breakdowns for the join phase of the algorithms. The breakdowns are for the tasks running on CPU 0 in the system. The Y axis shows the execution time. Every bar is broken down into four categories: CPU busy time, stalls due to data cache misses (including the effect of L2 misses), stalls due to DTLB misses, and other resource stalls. Comparing Figure 11(a) and (b), we can see that the fractions of data cache stalls for the three left bars increase dramatically. This clearly shows the impact of memory bandwidth sharing on the performance. In contrast, our cache-stationary algorithm achieves quite good cache performance. At 32 CPUs, cache stalls dominate all bars, as shown in Figure 11(c). Even in this case, our algorithm is better than the other algorithms.

6.3 Varying Other Parameters

Figure 12 shows the benefits of our inspector join algorithm varying the number of probe tuples matching a build tuple (which is the ratio between probe and build relation sizes), the percentage of probe tuples that have matches, and the tuple size. All the experiments use 8 CPUs in the join phase. The three figures share a common set of experiments, which correspond to the 8-CPU points in Figure 10.

Figure 10(a) varies the number of matches per build tuple from 1 to 8 (while keeping the build relation size fixed).

[8] The above figures omit the GRACE hash join curves for clarity. Compared to the GRACE hash join algorithm, our inspector join achieves 1.5-4.1X speedups for the join phase and 1.7-2.9X speedups for the entire hash join with 1-32 CPUs. The speedups at 32 CPUs are 1.5X and 1.7X, respectively.



(a) Varying the number of matches per build tuple

(b) Varying the percentage of probe tuples having matches

(c) Varying the tuple size

Figure 12: Aggregate total performance varying three parameters when 8 CPUs are used in the join phase

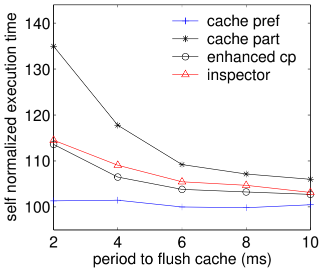
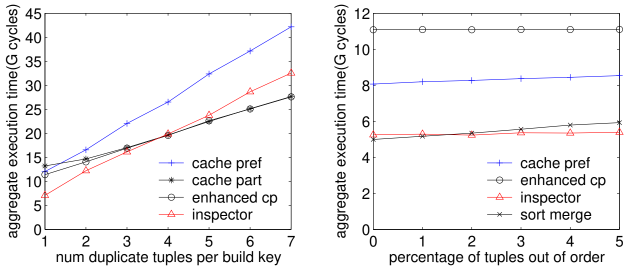


Figure 13: Robustness against cache interference (*self-normalized* join phase performance, num CPUs used=1)



(a) Duplicate build keys

(b) Nearly sorted relations

Figure 14: Exploiting the inspection mechanism

Figure 10(b) varies the percentage of probe tuples having matches (while keeping the probe relation size fixed). Figure 10(c) varies the tuple size from 20B to 140B (while keeping the build relation size fixed). Note that the number of tuples decreases as the size of the tuple increases. Therefore, all the curves have the downward shape. Note that in the 20B experiments, a cache line of 64B contains multiple probe tuples. Since the cache-stationary join visits probe tuples non-sequentially, it may incur multiple cache misses for every cache line in the probe partition. However, our inspector join with cache-stationary join phase is still the best even for the 20B experiments.

In all the experiments, we can see that our inspector join algorithm is the best. For all the experiments except the 5% points in Figure 10(b)[9], our inspector join achieves 1.1-1.4X speedups compared to the cache prefetching algorithm and the enhanced cache partitioning algorithm.

6.4 Robustness of the Algorithms

Figure 13 shows the performance degradation of all the algorithms when the cache is periodically flushed, which is the worst case interference. We vary the period to flush the cache from 2 ms to 10 ms, and report the execution times self normalized to the no flush case. That is, "100" corresponds to the join phase execution time when there is no cache flush.

The cache prefetching algorithm sees at most 3% performance degradation because of cache flushes. It is very robust because it does not assume that any large data structures stay in the cache. In contrast, the original cache partitioning algorithm assumes the exclusive use of the cache, and suffers from a 6-35% performance degradation for the cache flushes. Like the original cache partitioning, our inspector join algorithm and the enhanced cache partitioning algorithm both try to keep a build sub-partition and its hash table in the cache. To improve robustness, both of the algorithms perform prefetching for build tuples.[10] As shown in Figure 13, this technique effectively reduces the performance degradation to 2-15%, which is a 2X improvement compared to the original cache partitioning.

6.5 Choosing the Best Join Phase Algorithm

By default, our inspector join algorithm uses the cache-stationary join phase. However, our inspection approach can detect situations where cache-stationary join phase is not the best algorithm and choose a different one. Figure 14(a) varies the number of duplicate build tuples per build join attribute value. The duplicate-free points correspond to the 8-CPU points in Figure 10(c). We see that the default cache-stationary join phase of the inspector algorithm is still the best until 3 duplicates per key. However, when there are 4 duplicates per key, enhanced cache partitioning gets better. The prefetching algorithm needs to visit

[9] When there are only 5% probe tuples having matches, the aggregate join phase execution time only consists of 10-24% of the total aggregate execution time. Therefore, the difference is small between all the algorithms optimizing the join phase performance.

[10] As we do not prefetch for the hash table, we expect to pay higher cost than pure prefetching schemes when the cache is flushed. Prefetching for the hash table is much more complicated than prefetching only for build tuples, and may incur more run-time overhead for normal execution.

multiple build tuples for every probe tuple in the duplicate key cases. Since the visits are all cache misses, the performance of the prefetching algorithm suffers significantly. As discussed in Section 3.4, our inspection approach detects duplicate keys by counting the number of sub-partitions matching every probe tuple. When a probe tuple on average matches 4 or more sub-partitions, it chooses enhanced cache partitioning as the join phase algorithm. Thus the actual inspector join performance tracks the best of the *inspector* and *enhanced cp* curves in the figure.

Figure 14(b) shows the performance when the source relations are nearly sorted. We vary the percentage of tuples out of order from 0% (fully sorted) to 5%. For the fully sorted case, we sort the input relations for the 8-CPU points in Figure 10(c). Then we randomly choose 1%–5% of tuples and randomly change their locations to generate the other cases.

As shown in Figure 14(b), the sort-merge algorithm performs the best as expected for the fully sorted case. However, to our surprise, the inspector join performs equally well. The reason is that for the fully sorted case, the build tuples in a build sub-partition are sorted, and the corresponding probe tuples located sequentially in the probe partition. Therefore, the cache-stationary join phase essentially visits both the build and the probe partitions sequentially. Since the hash table is kept in cache, its cache behavior is the same as the sort merge join, which only merges two in-order inputs. When more and more tuples are out of order, the sort merge join becomes worse than the inspector algorithm because of the increasing cost of sorting out-of-order tuples, while the inspector algorithm pays only a slight overhead to visit some probe tuples non-sequentially.

7 Conclusions

In this paper, we have proposed and evaluated *inspector joins*, which exploit the fact that during the I/O partitioning phase of a hash-based join, we have an almost free opportunity to inspect the actual properties of the data that will be revisited later during the join phase. We use this "inspection" information in two ways. First, we use this information to accelerate a new type of cache-optimized join phase algorithm. The cache-stationary join phase algorithm is especially useful when the join is run in parallel on a multiprocessor, since it consumes less of the precious main memory bandwidth than existing state-of-the-art schemes. Second, information obtained through inspection can be used to choose a join phase algorithm that is best suited to the data. For example, inspector joins can choose enhanced cache partitioning as the join phase algorithm when a probe tuple on average matches 4 or more sub-partitions.

Our experimental results demonstrate that inspector joins offer speedups of 1.1–1.4X over the best existing cache-friendly hash join algorithm when run on 8, 16, or 32 processors, with the advantage growing with the number of processors. We also observe that inspector joins are robust with respect to various properties of the data (e.g., tuple size, fraction of tuples with matches, etc.). Thus, inspector joins are well-suited for modern multi-processor database servers.

References

[1] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In *Proceedings of the 1986 SIGMOD Conference*, pages 61–71, May 1986.

[2] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of ACM*, 13(7):422–426, 1970.

[3] P. A. Boncz, S. Manegold, and M. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of the 25th VLDB*, pages 54–65, Sept. 1999.

[4] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving Hash Join Performance through Prefetching. In *Proceedings of the 20th ICDE*, pages 116–127, March 2004.

[5] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[6] Intel Corp. *IA-32 Intel Architecture Software Developer's Manual, Volumn 2B:Instruction Set Reference N-Z.* Order Number: 253667.

[7] Intel Corp. *Intel Itanium 2 Processor Reference Manual For Software Development and Optimization.* Order Number: 251110-003.

[8] Intel Corp. *Intel Itanium Architecture Software Developer's Manual*. Order Number: 245317-004.

[9] N. Kabra and D. J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *Proceedings of the 1998 SIGMOD Conference*, pages 106–117, June 1998.

[10] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of Hash to Data Base Machine and its Architecture. *New Generation Computing*, 1(1):63–74, 1983.

[11] B. Lindsay. Hash Joins in DB2 UDB: the Inside Story. *Carnegie Mellon DB Seminar*, March 2002.

[12] S. Manegold, P. A. Boncz, and M. L. Kersten. What Happens During a Join? Dissecting CPU and Memory Optimization Effects. In *Proceedings of the 26th VLDB*, pages 339–350, Sept. 2000.

[13] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh. Robust Query Processing through Progressive Optimization. In *Proceedings of the 2004 SIGMOD Conference*, pages 659–670, June 2004.

[14] S. McFarling. Combining Branch Predictors. Technical Report WRL Technical Note TN-36, Digital Equipment Corporation, June 1993.

[15] T. K. Sellis. Multiple-Query Optimization. *ACM TODS*, 13(1):23–52, 1988.

[16] L. D. Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM TODS*, 11(3):239–264, 1986.

[17] A. Shatdal, C. Kant, and J. F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of the 20th VLDB*, pages 510–521, Sept. 1994.

[18] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *Proceedings of the 27th VLDB*, pages 19–28, Sept. 2001.

[19] P. Valduriez. Join Indices. *ACM TODS*, 12(2):218–246, 1987.