University of Magdeburg

School of Computer Science

Master Thesis

Performance Impacts of Different Code Optimizations for the Radix Join

Author:

Vinod Chelladurai

January 20, 2015

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
M.Sc. David Broneske
Department of Technical and Business Information Systems

Chelladurai, Vinod:
Performance Impacts of Different Code Optimizations for the Radix Join
Master Thesis, University of Magdeburg, 2015.

Abstract

Due to recent advancements in hardware technology, main-memory database systems are gaining more importance. This is because of the increasing capacities of the random-access memory over the years, as a result of which the entire data can now be held and processed within the limits of the available main-memory. In order to achieve a maximum performance, database operators in these systems need to be heavily tuned to exploit the capabilities of the underlying hardware, more specifically the capabilities of modern CPUs. To this end, software code optimizations relative to such hardware capabilities can be applied over an existing database algorithm. In our work, we study the performance behaviour of a set of code optimizations for the database join operator. For this, we adopt a set of optimization techniques from the literature and analyze their impacts on radix hash join algorithm.

Acknowledgements

This thesis represents the end of my two years bonding with Otto-von-Guericke University Magdeburg for graduating with a Master degree. At this moment, I would like to express my gratitude to all the people who made it possible.

First and foremost, I would like to thank Professor Gunter Saake for giving me an opportunity to undertake this thesis work in his department. I would also like to thank for all his encouraging lectures on database topics as part of my degree that really motivated me to approach this work.

I would like to express my heartful gratitude to David Broneske for giving me a very big support throughout the course of this thesis. Without his assistance and supervision, it would have been much harder to accomplish this work. I would also like to appreciate his thoughtful comments and patience over my work that really helped me to get on the right direction.

Also, I would like to thank Christian Krätzer for his valuable time and consideration to be an external reviewer for this thesis.

Finally, I would like to acknowledge my family members and friends who were very supportive and tolerant during every tough situation I faced all through my career.

Contents

List of Figures

List of Tables

List of Code Listings

List of Algorithms

1. Introduction

Over the years, there has been a steady increase in the capacity of main-memory due to the increasing chip densities and the reduction in cost of semiconductor devices [KWLP12]. Such improvements in RAM have raised the importance of main-memory database systems since it is reasonable for several applications to store all their data and process them within the limits of the available main-memory [GMS92]. As a consequence, the database algorithms which were previously running on traditional database systems have to be adapted to meet the performance needs of a main-memory database [Bro13].

Due to the placement of data in a physical main-memory, the bottleneck of disk access and consequently the expensive I/O operations are omitted in a main-memory database system. As a consequence, database algorithms for main-memory environment are bound to the central processing unit (CPU) rather than to the I/O. Therefore, a main-memory database algorithm can reach an optimal performance by tuning them to the underlying hardware. More specifically, the algorithm has to exploit the ever-increasing processing capabilities of modern CPUs [BBS14]. Recent evolution in CPU architectures have introduced a number of hardware features such as increased number of CPU cores, wide vector registers with advanced instruction sets, increased cache-memory capacities and efficient pipelining capabilities [KKL$^+$09, RBZ13]. These CPU features usually vary across different machines and hence, it is difficult to predict the performance behavior of a database algorithm on a given machine. Thus, it is important to understand the performance of a main-memory database algorithm when run in systems with modern CPU capabilities.

From a software engineering point of view, a main-memory database algorithm can exploit these CPU capabilities through the application of code optimization techniques. There are several well known optimization techniques to tune an algorithm to the underlying hardware. Some examples are loop unrolling, loop fission, vectorization, parallelization and branch-free technique. However, an optimal performance is not guaranteed through the application of all available code optimization techniques since the advantage of an optimization technique for a database algorithm depends upon the given machine and the workload [BBHS14]. Hence, the important task is to find the set of code optimizations which yield the best performance for a given database algorithm.

Goal of Our Thesis

In this thesis work, we study the performance impact of a set of code optimization techniques and their possible combinations for a database join operator. Although, there are several techniques to implement a database join, we choose the radix join technique due to the following reasons:

1. Due to its high hardware-conscious property, the radix join is gaining more importance in the context of main-memory database join operations [BTAÖ13, Man02].
2. Recently published results argue that with increasing number of CPU cores in multi-core CPU architectures, a main-memory radix join algorithm would dominate several other hash join techniques. Further, it would also continue to provide advantages over the sort-merge join technique [BATz13, BTAÖ13, KKL$^+$09].

The main objective of the research in our thesis is driven by the following question:

> *What set of code optimizations reaches the optimal performance for the radix hash join technique?*

To meet this objective, we present the main contributions in this thesis work as below:

1. We analyze how the radix join algorithm exploits the parallelism capabilities *(data-level parallelism and thread-level parallelism)* offered by systems with multi-core architectures via two code optimization techniques – *vectorization* and *parallelization.*
2. We attempt to alleviate the problems faced by the radix join due to the presence of control hazards via the code optimization technique – *branch-free code* or *no-branching implementation.*
3. Once we implement the radix join optimized with the above three code optimization techniques, we identify the individual techniques that tend to provide reasonable performance over the scalar version of the radix join and then apply a combination of such code optimization techniques.
4. Following our implementation, we conduct a study on the performance behavior of the radix join optimized with each of the above techniques using two different machines and different set of workloads.

5. Finally, we present our results for all the optimized versions of the radix join on two separate machines and for different set of workloads. Based on our results, we draw a common conclusion for the radix join behavior and compare our results with the earlier published findings.

Structure of Our Thesis

Apart from this chapter, our thesis comprises seven further chapters. In Chapter 2, we present sufficient background knowledge required to understand the basics of main-memory database systems and their properties. Further, we also discuss two important main-memory join algorithms – sort-merge join and hash join and their trends with respect to improving hardware capabilities. Finally, we illustrate the importance of code optimization techniques and describe those techniques adopted as part of this thesis work in detail. In Chapter 3, we discuss the principle of radix join, their advantages over other hash join techniques and finally describe their working logic in detail.

In Chapter 4, we discuss the central part of this thesis, i.e., we explain our approaches of implementing the adopted code optimization techniques to the scalar version of the radix join. To this end, we explain how we evaluated them by describing our evaluation method in detail in Chapter 5 and in addition, we also present a brief summary of our evaluation results at the end of this chapter. In Chapter 6, we present sufficient information about the previous work in the literature on code optimization techniques that are related to both join as well as other database operators. We provide a summary of our results and attempt to illustrate its concurrence with the previous published results in Chapter 7 and finally in Chapter 8, we identify the steps that need to be done in future as a continuation of this thesis work.

2. Background

In this chapter, we present important background knowledge required for a good understanding of the main memory radix join and also various code optimizations for database operations. For this, we present an overview about main-memory database systems, their evolving importance in real-time applications and various characteristics of the same in contrary to a traditional database system in Section 2.1. In Section 2.2, we present a description of selected main-memory join algorithms in the literature and their behavior with respect to the underlying architecture. Finally, in Section 2.3, we describe several code optimizations adopted as part of this thesis work.

2.1 Main-Memory Database Systems

Main-memory database systems (MMDB), also known as in-memory database systems, are database systems where the entire database fits into the main-memory, i.e., permanent storage of data in the physical main-memory. This is in contrast to conventional databases where the data is disk resident and moved into the main-memory during data access on request from the Central Processing Unit (CPU) [GMS92]. Such systems have gained more importance over the years, because increasing chip densities and reduced cost of semiconductor devices makes it feasible to offer systems with gigabytes and terabytes of main-memory [KWLP12]. This is particularly attractive for a number of existing and emerging real-time applications whose database sizes are within the bounds of the main-memory capacity. Even for applications whose database sizes are larger than the available main-memory limit, the existing database can be divided into a number of logical databases based on certain types or classes of data and the data accessed with more frequency can be stored in the main-memory. Some of the examples for existing real-time MMDB designs are Main-Memory Recoverable Database with Stable Log (MARS), Hardware Logging (HALO), Silicon Database Machine (SiDBM),IBM's Office-By-Example (OBE), SAP HANA (High Performance Analytic Appliance) and Microsoft Hekaton [GMS92, Eic88, DFI^{+}13, BK^{+}14].

2.1.1 Distinction between In-Memory Database Systems and Disk-Resident Database Systems

With the storage of data in the physical main-memory, several distinctions exist in the design and the access pattern behavior between the in-memory database system and the disk-resident database system due to the varying properties of the main-memory and the magnetic disk. However, all such distinctions still ensure the ACID property compliance of a general database system. We explain some of the prominent distinctions in the following sections.

Design

The storage of data in a main-memory avoids placement of disks in the path of a query processing plan for database operations since the data can be accessed directly from the main-memory [GMS92]. Thus, the bottleneck of disk access is now omitted, which reduces the time required for data access by several orders of magnitude compared to a regular disk access in a disk-resident system (about 1:10^6 in favor of the main-memory) [VAD+11]. As a consequence, MMDBs favor high-speed access to stored data which result in an increased response time of database operations and a speedup in performance of up to significant levels can be realized [GMS92]. Also, main-memory databases follow a well-organized and optimized design pattern when compared to conventional databases. For example, due to the omission of expensive I/O operations on disk and flash memory, MMDBs run out of need for various mechanisms such as tertiary memory management and file management. However, mechanisms to ensure long-term accessibility such as recovery and backup in a MMDB are handled by the existing disk memory [Eic88].

Cache Memories and Index Structures

In spite of providing faster access to data compared to disk-resident systems, an in-memory database fully exploits the available cache memory to achieve a further improvement in data retrieval operations. The main goal in this context is that almost every data item needed for an entire transaction can be placed in a single cache memory or a hierarchy of cache memories of large size to improve the overall performance (cache-conscious operations). The disk-resident database, on the other hand, also takes advantage of very large cache memories but the transfer of data from the disk to the cache requires the computation of disk addresses since the index structures in existence are designed solely for the disk memory and not for the main-memory. However, in case of MMDBs, with the storage of database relations and their corresponding tuples directly in the main-memory, applications or software programs access the tuple attributes via the pointer to tuples. Index representation of this type is cheaper when compared to the existing disk indices and they also eliminate the need for index compression mechanisms and handling of fields of various lengths. Thus, the MMDB index structures are primarily concerned in reducing the computation time of overall database operations, at the same time utilizing a comparatively reduced memory space [LC86].

Buffer Management

A disk-resident database system requires the management of buffer policies. For example, when the CPU requests a data item, the disk address is first computed with the help of which the buffer manager checks whether the requested data already exists in the main-memory or need to be moved into the main-memory from the disk. However, in case of memory-resident systems, since the data is already available in the main-memory, the data can now be accessed efficiently using the memory address (index representation described in previous section), thus eliminating the need for the enforcement of buffer management [GMS92].

Volatility

In spite of offering various benefits over disk-resident database systems, MMDBs are prone to serious volatility problems and system errors due to the volatile nature of the main-memory. However, recent advancements in non-volatile main-memory devices such as NVRAM (non-volatile random-access memory) have been a significant factor for the feasibility of the usage of MMDBs in real world applications at a comparatively higher cost [LC86].

Similarly, several other distinctions exist in the context of concurrency control, recovery mechanisms, compression mechanisms, data clustering mechanisms and data representation with all of these distinctions primarily giving rise to an optimized approach for main-memory databases [GMS92]. Thus, memory resident databases play a vital role in applications where reduction in latency, performance improvement in query processing and utilization of minimal hardware resources are of significant importance.

2.1.2 Cache Sensitivity

As mentioned in Section 2.1.1, an in-memory database system fully takes advantage of high speed cache memories to enhance the overall performance of database query processing applications. With increasing processor speeds over the years, the access latency of the random-access memory (RAM) cannot cope with the improvements in CPU speed for several economical reasons. Thus, there is a big performance gap between operating speeds of the CPU and the memory access latency. To bridge this gap, cache memories are introduced. Cache Memories are small blocks of high-speed memories situated between the main-memory and the CPU that temporarily hold main-memory contents that are actively in use [Man02]. Based on the main-memory content they store, a cache memory can be distinguished into two different types - *data cache*, a read-write memory which caches main-memory data items and *instruction cache*, a read-only memory which caches active instructions in use. For any database operation, the content of a cache memory is first accessed for data/instructions. If the data request is served within the cache memory without the need for a main-memory access, it is called *cache hit*. In case only a portion of the required content is available in the cache due to cache memory overflow or non-availability of data/instructions, then the remainder of the content is accessed in a normal way from the main-memory and then moved into the cache. This phenomenon where the data request cannot be served with the available data in the cache memory is called *cache miss*. Both cache hit rate and cache miss rate have severe impact on the execution rate of main-memory algorithms. This is because for most of the database applications, the time required to access the cache memory contents is only about *10-15* percent of the total time required for a main-memory access and thus they improve the overall execution rate of the application programs [Smi82].

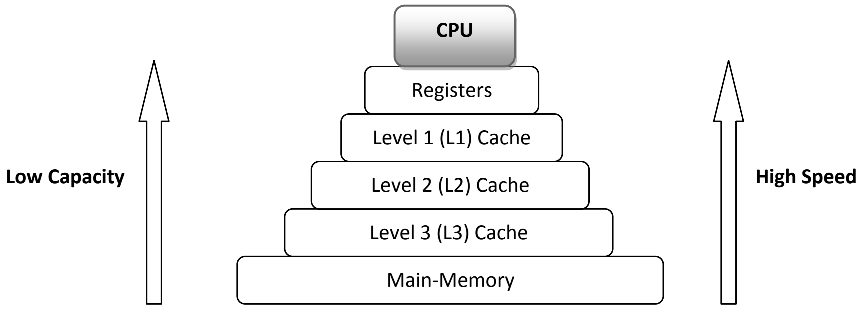

Figure 2.1: A simple representation of a three-level cache hierarchy

With recent advancements in hardware capabilities, cache memories are organized by means of a hierarchy between the CPU and the main-memory. With respect to MMDBs, careful usage of such a hierarchical cache memory helps improving the cost of a main-memory database operation. Manegold et al. proposed a memory access cost model for hardware consisting of N levels of caches used together with translation lookaside buffers (TLB) for demonstrating the reduction in cost of memory accesses for main-memory database algorithms [MBK02]. The most common hierarchical cache representation is a *two-level* cache (*L1* and *L2*), with data in *L1* cache can be accessed with a latency of *1-2* CPU cycles, allowing for access with the highest speed in cache hierarchy while data in *L2* cache can be accessed with a latency of *5-10* CPU cycles. The caching mechanism in this type of hierarchy is such that the *L2* cache usually backs up the copies of data in *L1*, i.e., whenever a *L1* cache content need to be replaced, it is first moved into the *L2* cache and then the new content is copied either from the main-memory or *L2* cache (if available). Rapid advancements in chip technology have introduced a new form of cache hierarchy which includes a third level of cache memory (*L3*) with a latency of *10-20* CPU cycles. All three levels of memory are integrated directly on the CPU die itself and their memory capacity varies proportionally with the increasing hierarchical levels. For example, Intel's Core i5-2500 supports a *256* Kb *L1* cache, a *1* Mb *L2* cache and a *6* Mb *L3* cache for each of its CPU core. In Figure 2.1, we show an example of a simple representation of a *three-level* cache hierarchy. It is clear that closer the cache memory to the processor in the hierarchy, higher the speed and lower the memory capacity [MBK02, BBS14, KW02].

In addition to the cache capacity, cache hit-miss ratios, and the cache-level hierarchy, Manegold et al. argue two other cache-specific hardware parameters that are sensitive to the overall performance of main-memory database operations [Man02]. They are as follows:

- **Cache-Lines**, also known as cache blocks, represent the internal organization of the cache memory and are the smallest unit of data transfer across different levels in a cache hierarchy, i.e., complete data inside the block is either read or written. For example, whenever a cache miss occurs in an *L1* cache due to memory overflow, an old line is replaced by means of a cache replacement strategy. Several cache replacement strategies are presented in [Zah07]. After this replacement, a new line is moved either from *L2* cache or main-memory with every bit stored in the line/block transferred in a parallel fashion. Typical size of cache-lines ranges from *16-128* bytes and are critical in determining the performance of a main-memory system [Man02].

- **Cache Associativity** is a cache memory metric that determines the cache-line to which the newly requested data would have been loaded. The simplest case is a cache memory with *directly mapped* associativity where each data in main-memory finds exactly one location in the cache memory. Though the lookup appears to be simple in this type of cache, it incurs a significant amount of conflict misses (cache misses that do not occur in caches with a different associativity). Contrary to this is a *fully associative* cache where a cache-line comprising the main-memory data can go to any location in the cache memory. Even though it overcomes the problem of conflict misses, the task of searching for cached data incurs a significant overhead compared to a directly mapped cache. Modern processors support the use of a *n-way* set associative cache which performs a lookup on *n* different locations for each data request [Man02].For example, Intel's Core i5-2500 and Xeon E5-2690 implements a *4-way* set associative caching mechanism.

To summarize, cache memories are of vital importance for enhancing the execution performance of any type of database systems. More specifically, metrics and characteristics such as cache capacity, cache hierarchy, hit–miss ratio, cache-line sizes and cache associativity are more sensitive for the overall performance of main-memory database operations.

2.1.3 Column-Oriented Storage Model

A typical relational database management system is organized in such a way that individual rows or tuples of a relation are stored in contiguous memory locations (row-oriented approach). An alternative way to this would be to re-organize the storage model based on a column-oriented fashion, i.e., database records are stored by means of individual columns. As mentioned in the previous section, cache capacity is one of several characteristics that influence the performance of a main-memory database operation. Allocation of records by means of column-oriented approach tends to improve the cache consciousness and therefore the overall execution rate [BKM08]. For instance, in a row-oriented storage model, executing an aggregation or projection operation involves moving more columns or attributes than needed from the main-memory into the cache whereas in a column-oriented approach, only those columns required for the operation are moved into the cache [Los09]. Thus, the probability of occurrence of cache misses due to cache capacity overflow is higher in the former than in the latter case. For our work, we adopt a column-oriented approach for the storage model. This will be discussed in more detail in Chapter 4.

The major disadvantage of a column-oriented storage model is the task of tuple reconstruction in case a number of additional attributes need to be gathered for a database operation. A tuple reconstruction is a simple join operation between the attributes in the result and the additionally needed attributes based on tuple identifiers (TIDs) or Object Identifiers (OIDs) [IKM09]. Although this task appears to be a performance overhead in column-oriented stores, a significant amount of benefit in execution performance is realized on the other hand because of the reduced size of the processed data. In addition to the cache-conscious benefits, the column-oriented storage model provides various other benefits as follows:

- Storage of records in a column-oriented fashion favors the application of compression mechanisms at an improved rate due to the similarity between contiguous data items. Moreover, it also favors the applicability of additional compression schemes that are not favored by a row-oriented storage model [Fer05].
- Since the execution of various database operations such as aggregation and projection processes only the required columns, a column-oriented model significantly avoids a large number of I/O burdens in case of real-time business intelligence queries [Los09, KSS12].
- A row-oriented model needs to maintain additional storage space for the implementation of index structures. On the contrary, most of the column stores allow the storage of data within the index itself, thus providing a more optimized storage model. In addition, such index structures improve the performance of query processing applications by providing a sophisticated approach for database operations within and across the attributes [Los09].

2.2 Main-Memory Joins

A relational join is a database binary operation that retrieves tuples from two different relations by using information common to both of them. For example, consider a relation containing details about all departments in an enterprise and another relation with information about employees of the same enterprise. The result of a typical join operation in this case would be an output relation describing various details about the departments with its corresponding employees. With the evolution of main-memory databases in recent years, Leonard D. Shapiro argue that main-memory availability of significant fraction of the memory required by one of the joining relations facilitates an effective implementation of join operation without the need of an external disk memory [Sha86]. In addition, careful exploitation of architectural features such as cache memories, translation lookaside buffers (TLB) and processor-memory bandwidth improves the access latency of performing a join operation in the available main-memory [KKL+09].

A join operation between two relations can be simply executed by searching data items that possess the same value for an attribute common to both relations. A brute-force approach in this case would be a nested loop join algorithm in which for every tuple in the outer relation, the whole inner relation is scanned for potential matches. In this way, the execution is repeated iteratively until all tuples in the outer relation are processed. However, nested loop joins are highly complex and do not provide an optimal solution for applications comprising of very large relations [Bro13].

In the following sections, in order to have a good insight on the working logic of main-memory joins, we focus on two important join algorithms which are based on sorting and hashing/clustering approaches - *Sort-Merge Join* and *Hash Join*.

2.2.1 Sort-Merge Join

As the name suggests, this type of join is carried out in two phases - *sort phase* and *merge phase*. In the sort phase, both input relations, R and S, are sorted on their respective join attributes. Then, in the merge phase, the sorted relations are sequentially scanned and whenever there is a match between their respective join attributes, tuples from both relations are inserted into the output relation. This implementation of merge phase assumes that the join attribute on the inner relation R is a primary key, i.e., it possesses only unique values for the attribute. However, in case of a non-primary key attribute, the merge phase needs to be implemented with an additional nested loop join algorithm for facilitating several passes over all tuples possessing the same join attribute value. The complexity of a typical sort-merge join is *O(n log n)* for each relation with the sort phase comprising more than *98%* of the overall execution time of the join algorithm [KKL+09, ME92].

Sort Phase

As explained in Section 2.1.2, an in-memory database has to fully exploit the use of shared cache memories so as to reduce the number of accesses to the main-memory. Hence, during the sort phase, the entire input relation is divided into a number of chunks such that for a cache memory of size C bytes and a data item of d bytes, the size of each chunk would be *C/2d* such that each chunk resides in the cache and is sorted separately to produce a list of sorted runs or data blocks. Each of the individual sorted runs is then merged with one another in a number of steps until a single sorted run is produced. In order to reduce the cache memory overflow during the merging of sorted runs, Chhugani et al. proposed a multi-way merging algorithm implemented with cache-conscious FIFO queues in which more than two runs are merged at once incrementally

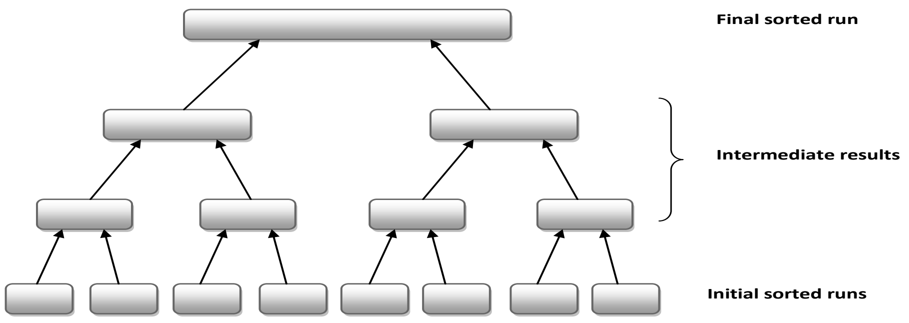

Figure 2.2: Multi-way merging implemented as multiple 2-way merging – adopted from [CNL+08]

to produce a final sorted run [CNL+08]. In Figure 2.2, we depict a simple multi-way merging algorithm implemented by means of multiple two way merging blocks. Each node maintains a FIFO queue and whenever there is a room for an entry in the parent node's queue, the algorithm checks for the number of elements in the corresponding child nodes' FIFO queues. If each of the child nodes contains more than one element, then the smallest of all these elements is moved into the parent's queue. This procedure is repeated recursively until both child queues becomes empty which means the parent node contains a sorted run of the merged child nodes [CNL+08, BATz13].

It should be noted that the algorithm discussed so far is based on the assumption that sorting is performed only on the tuple keys. However, in case of (key,rid) pairs, where *rid* refers to the address of the tuple containing the key, a straightforward approach is to simply extend the algorithm by treating the (key,rid) pairs as a single entity and comparing only the first m bits comprising the key of each entity [CNL+08].

Merge Phase

Algorithm 2.1: Sort-merge join – merge phase (equi-join)

```
Input 1 : Sorted inner relation R with join attribute $J_R$
Input 2 : Sorted outer relation S with join attribute $J_S$
Output : Relation $O$
$t_S$ : tuple of S
$t_R$ : tuple of R
for each $t_S \in$ S do
    while $t_R.J_R <= t_S.J_S$ do
        read next $t_R$ from R
        if $t_R.J_R = t_S.J_S$ then
            O = insert($t_R$, $t_S$)    /* join is executed */
        end
    end
end
```

Once the two input relations are sorted, a sequential scan is performed between join attributes of the sorted input relations and whenever there is a match satisfying the join condition, tuples from both relations are inserted into the output relation. If there is no match found, then the next tuple is read from the relation with the smaller join attribute value. We illustrate an overview of the merge phase for an equi-join in Algorithm 2.1 [ME92].

Versioning with Architectural Changes

With the recent evolution of the underlying computer architecture, modern computers used today have come up with various architectural features such as multiple cores integrated on a single chip with provision for several threads on each core *(multi-threading)* and support for vectorization via *SIMD (Single Instruction Multiple Data)* instructions. These architectural features result in an improved execution performance via data- level parallelism and thread-level parallelism [KKL+09]. We explain these parallelisms briefly below:

- ***Thread-level parallelism*** is a form of parallelism achieved by running multiple threads concurrently such that each thread performs the execution of a part of the entire algorithm. In this way, each thread performs different tasks of the same algorithm (*task decomposition*) or split the same task among themselves within the algorithm context (*data decomposition*). Therefore, an algorithm is said to be complete only when all running threads are finished with their individual execution [AR06].

- ***Data-level parallelism***, in contrast to thread-level parallelism, is a form of parallelism in which a single instruction, which is previously applied on only one data item at a time, can be now be applied simultaneously on more than one data item. Instructions of this type, known as SIMD instructions, are capable of operating on *128*-bit or *256*-bit vectors simultaneously and are supported via cores with extended width registers called SIMD registers [Bik04].

We explain both these parallelisms in detail in Section 2.3. Taking advantage of these architectural changes has resulted in an improved version of the existing scalar implementation of the sort-merge join. Chhugani et al. proposed several versions of sort-merge join including SIMD, parallel and a combination of SIMD with parallel versions with all of these versions primarily outperforming the scalar version by a significant factor [CNL+08]. Also, both Chhugani et al. and Kim et al. project that with advent of wider SIMD architectures in the future (such as one operating on *512*-bit vectors simultaneously), sort-merge is likely to gain more advantage with respect to its execution performance [KKL+09, CNL+08].

2.2.2 Hash Join

The principle behind a hash join on two input relations, R and S, is to build a hash table *(build phase)* by scanning the join attribute (in most cases, the primary key) of the smaller relation R. After the hash table is built, the larger relation S is scanned sequentially *(probe phase)* and for each tuple in relation S, a lookup is performed on the hash table for finding matching keys. Finally, an entry is inserted into the output relation by appending tuples from both relations for each matching pair of keys. In this section, we explain two basic hash join implementations – Canonical Hash Join and Partitioned Hash Join.

Canonical Hash Join

The canonical hash join is the simplest of all hash join algorithms. In the build phase, a hash table is built by applying a user-defined hash function, *h(K)* on the inner relation R until the relation R is scanned completely [BTAÖ13]. In order to avoid collisions while performing lookup on the hash table, in most of the cases, the hash table is designed in such a way that its size is approximately two times larger than the cardinality of the inner relation [KKL+09].

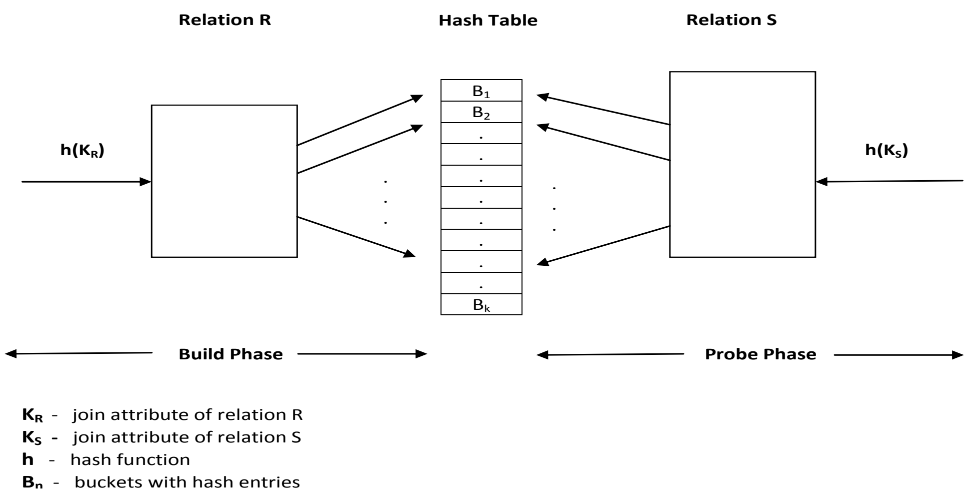
Relation R
Hash Table
Relation S
$h(K_R)$
$h(K_S)$
B_1
B_2
B_k
Build Phase
Probe Phase
K_R - join attribute of relation R
K_S - join attribute of relation S
h - hash function
B_n - buckets with hash entries

Figure 2.3: Canonical hash join – adopted from [BTAÖ13]

Then, in the probe phase, the same hash function, *h(K)* is applied on the outer relation S. The result of the hashing indicates the entry in the hash table built for R to which the keys or the join attribute values of S should be compared. Whenever there is a match during comparison, an entry consisting of tuples from R and S is inserted into the output relation. Assuming that the complexity of data accesses to the hash table is constant, the time complexity of a canonical hash join is around *O(R + S)* [BTAÖ13]. We depict a simple working logic of a canonical hash join in Figure 2.3.

Partitioned Hash Join

The main disadvantage of a canonical hash join is that when the size of the built hash table exceeds the size of the cache memory, it leads to a significant amount of cache misses. To overcome this problem, the partitioned hash join algorithm is proposed. The basic idea behind this algorithm is to build a number of hash tables such that each of their individual sizes fall within the limits of the available cache memory. Hence, cache misses that occur during the probe phase can be significantly reduced. To achieve this property, this algorithm introduces a new phase called *partition phase* in which both input relations are divided into a number of partitions. The assignment of tuples to a particular partition is determined by a user-defined hash function, $h_1(K)$. Hence, the input to the build phase would consist of partitions of the inner relation R instead of the entire relation. For each of these partitions, a hash table is built separately by applying a user-defined hash function $h_2(K)$. Thus, the resulting hash tables are small enough to avoid cache misses. It should be noted that the hash functions $h_1(K)$ and $h_2(K)$ are not equal.

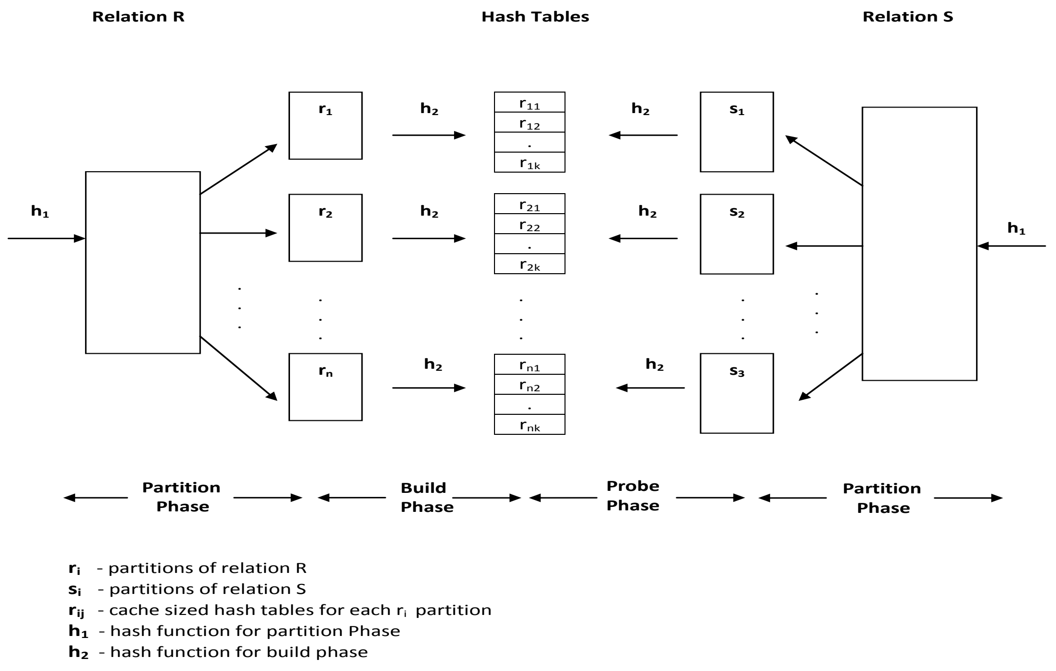
Relation R
Hash Tables
Relation S
h₁
r₁
r₂
rₙ
h₂
r₁₁
r₁₂
r₁ₖ
r₂₁
r₂₂
r₂ₖ
rₙ₁
rₙ₂
rₙₖ
s₁
s₂
s₃
Partition Phase
Build Phase
Probe Phase
Partition Phase
rᵢ - partitions of relation R
sᵢ - partitions of relation S
rᵢⱼ - cache sized hash tables for each rᵢ partition
h₁ - hash function for partition Phase
h₂ - hash function for build phase

Figure 2.4: Partitioned hash join – adopted from [BTAÖ13]

Finally, in the probe phase, the hash function $h_2(K)$ is applied separately on each partition of the relation S and probing process is carried out with their respective hash tables as in a canonical hash join. Even though the use of cache sized hash tables avoid cache misses during the build phase, the algorithm suffers from translation lookaside buffer (TLB) misses during the partition phase. This is because TLBs caches the virtual memory mapping for partitions residing on different pages and TLB misses are prone to occur if the number of partitions created exceeds the number of available TLB entries. To overcome this problem, the concept of radix partitioning of hash join is proposed in which the number of TLB entries serve as an upper bound for the number of created partitions. We explain the partitioned radix join in detail in Chapter 3. In Figure 2.4, we present a simple representation of a partitioned hash join [BTAÖ13].

Versioning with Architectural Changes

Similar to a sort-merge join, the architectural changes with data-level parallelism and thread-level parallelism can be exploited to extend the serial hash join implementations. With respect to improvement in performance, all hash joins significantly benefit through thread-level parallelism. However, there is no clear evidence for their improvement with data-level parallelism via SIMD instructions. Kim et al. argue that application of SIMD instructions together with their limitations in handling the scatter and gather operations limits the exploitation of data-level parallelism [KKL+09]. But, there is no evidence for the behaviour of hash joins when we limit applying SIMD only to functionalities not involving such gather and scatter operations. Furthermore, there is no clear distinction about the performance behaviour among different types of hash join with SIMD architectures. We will discuss scatter and gather operations and the restrictions of SIMD instructions with these operations in detail in Section 2.3 under limitations of vectorization.

2.3 Code Optimizations

As mentioned in the previous section, recent evolution in the computer architecture have introduced numerous features that apparently improve the processing capabilities of modern CPUs. Examples of such features include provision of CPU with multiple cores on a single chip with each core supporting one or more hardware threads and vectorization via *128*-bit SIMD registers. Furthermore, the inclusion of branch predictors in commodity CPUs have also become a trend to make their pipelining capabilities more efficient [KKL+09, RBZ13]. In order to take advantages of all these processor capabilities, a database algorithm needs to be explicitly hand-tuned to exploit them. However, the question of whether an algorithm really benefit from such explicit tuning is still an open research problem. Therefore, it is important to understand the behavior of the performance of an algorithm when run in hardware with modern CPU capabilities. Current approaches implement database operations in a high-level programming language and tune them using proposed code optimizations for the given processing capabilities. To automate this process, recent results of the software engineering community should be transferred to the database community in order to find the optimal set of code optimizations to be applied for the given workload and the machine [BBHS14].

Code optimization is the method of identifying pieces of code that require improvement and transforming them such that the resultant code achieves better performance with respect to time and space. The transformation in the code is usually made transparent to the outside world. Code optimization techniques are classified into two broad categories - *processor-independent* and *processor-dependent*. The former can be applied regardless of the underlying architecture while the latter mainly aims at the exploitation of the processor intrinsics. Heiko Fall et al. proposed several code optimizations that fall in both these categories [FM04]. As part of this thesis work, we adopt three important optimization techniques in the literature - *vectorization, parallelization* and *branch-free coding*. We explain them in detail in the following sections.

2.3.1 Vectorization

The term *vector* refers to a set of data items that are placed adjacent to one another. *Vectorization* is a process in which an algorithm is transformed from its original scalar form into a vector form such that the operation intended (which is previously applied only between two operands at a time) can now be simultaneously applied between several pairs of operands, i.e., application of a single instruction to multiple data items in the resultant vector. These instructions are called *SIMD (Single Instruction, Multiple Data)* instructions [Bik04]. CPU cores supporting SIMD capabilities (cores with extended width registers called SIMD registers) were originally built to improve the performance behaviour of multimedia applications that reiterate a particular operation on large arrays of numeric elements. Over the years, several efforts have been undertaken by numerous researchers to obtain significant benefits from the SIMD technology for database operations due to the nature of iterative processing of long sequence of records in database applications. Jingren Zhou and Kenneth A. Ross have already proved the benefits that various database operations like scan, aggregation and index designs could obtain from a SIMD technology [ZR02].

SIMD Instructions

A SIMD instruction can be viewed as a set of instructions packed together and operating on more than one element at the same time. Thus, they unroll a serial loop implementation to several depths allowing for an efficient execution via data-level parallelism [Bik04]. In Figure 2.5, we represent the parallel execution of an operation of the form A *op* B (where *op* is an operator) between two sets of items with each set packed together in two separate *128*-bit SIMD floating point registers. The registers A and B are aligned to *16*-byte boundaries such that they are loaded with four data items A_1, A_2, A_3, A_4 and B_1, B_2, B_3, B_4 respectively each of width *4* bytes. Now, the operation is executed simultaneously between each pair of operands *(A_1,B_1)*, *(A_2,B_2)* and so on. Finally, the result is stored in another *128*-bit register. The individual data items (for example, result of A_1 *op* B_1*)* can be extracted using various intrinsics relative to the SIMD architecture.

SSE Intrinsics

Modern CPU architectures support SIMD capabilities through a number of flavors. For example, on Intel cores, the benefits offered by SIMD instructions are realized by means of *SSE (Streaming SIMD Extensions), SSE2 (Streaming SIMD Extensions 2)* and *AVX (Advanced Vector Extensions)* instructions, whereas in case of AMD machines, they

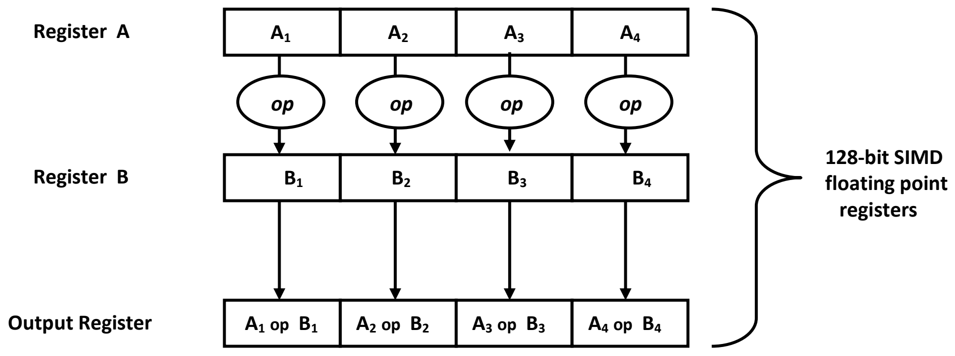

Figure 2.5: Data-level parallelism via SIMD instructions – adopted from [ZR02]

are supported via *3DNow!, Enhanced 3DNow!* and *3DNow! Professional* technologies [ZR02]. As part of this thesis work, we make use of Intel's SSE intrinsics for the application of SIMD code optimizations.

Streaming SIMD Extensions are coding extensions supported by Intel's C++ compilers for the provision of SIMD capabilities. Intel implemented them by extending their existing x86 architecture with eight registers labelled XMM0 through XMM7, each of width *128* bits. These SSE registers support instruction sets capable of operating only on single precision floating point numbers, i.e., they operate only on *32*-bit mode, for example, processing four *32*-bit integers simultaneously. Thus, for enabling operations in *64*-bit mode (such as double precision floating point numbers and *64*-bit integers), Intel extended the SSE instruction sets by further adding eight more registers labelled XMM8 through XMM15. The instruction sets supported by the resulting architecture are called SSE2 instruction sets which add additional instructions to the existing SSE instructions [Bik04]. The processing capability of SSE/SSE2 registers with respect to various data types can be understood with the help of Figure 2.6.

From Figure 2.6 on the facing page, it is clear that the total number of bits supported by any SSE/SSE2 register for processing is *128*. Hence, the input data that need to be processed by SSE registers must be aligned to *16*-byte boundaries. For the purpose of more efficient processing, Intel extended the width of SIMD registers from *128* bits to *256* bits which are realized via *AVX (Advanced Vector Extensions)* instruction sets in which functionalities for *32*-byte boundary alignments are implemented. In addition, there are several other Intel intrinsics that support SIMD instructions such as *SSE3, SSE4.1, SSE4.2* and *AVX2.* A detailed explanation of all these instruction sets can be found in [JKLM12] and [sim06]. The main objective of these special purpose intrinsics is to enable the programmers to make use of methods and variables available in C++ language to realize the SIMD capabilities without the need of explicit assembly language coding. Some compilers also allow *autovectorization*, i.e., a program can be converted

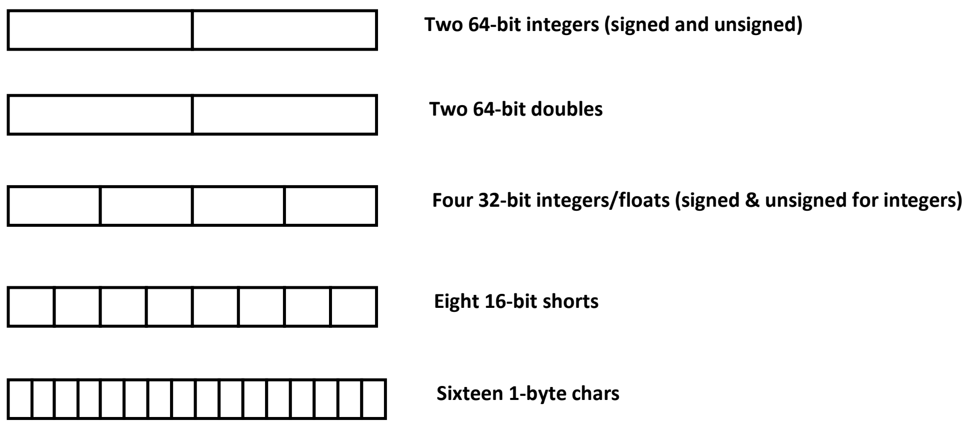
Two 64-bit integers (signed and unsigned)
Two 64-bit doubles
Four 32-bit integers/floats (signed & unsigned for integers)
Eight 16-bit shorts
Sixteen 1-byte chars

Figure 2.6: Processing capabilities of SSE/SSE2 registers for various data types – adopted from [Bik04]

from a serial representation into a vector representation without the programmer's intervention [Bik04].

Listing 2.1: Serial implementation of integer multiplication

```
void add()
{

// declaration and defintion of  arrays a[] and b[]
// declaration of output array c[]

for(int i=0;i<array_length;i++)
{
   c[i] = a[i] * b[i];
}

...
}
```

In Listing 2.2, we illustrate how a simple multiplication operation is optimized by the application of SSE intrinsics compared to it a sequential execution in Listing 2.1. Here, we use the SSE intrinsic `_mm_mul_epi32` (line *17* of Listing 2.2) to perform parallel multiplication between the *32*-bit integer arrays *a* and *b*. The contents of the two arrays are loaded into the *128*-bit registers *A* and *B* (lines *15* and *16* of Listing 2.2 on the next page) respectively in a series of iterations for the simultaneous application of multiplication operation on four pairs of data items as opposed to a sequential multiplication in Listing 2.1. Thus, the overall SIMD operation (first *for* loop in Listing 2.2 on the next page) can be viewed as a simple loop unrolling of depth *4* combined with the SIMD capability. Thus, the performance of the overall vector implementation is significantly improved compared to its scalar counterpart. It should be noted that only the *16*-byte aligned data would be processed in the first *for* loop. For this purpose, a variable `simd_length` is used to keep track of the *16*-byte alignment length of the contiguous memory locations. The remaining unaligned items are then processed in the second loop in a typical sequential fashion (lines *21-24*). However, the overhead imposed by the sequential execution of the second loop, load and store operations does not significantly affect the overall performance of a vectorized algorithm especially when the input array contains a large number of items for processing.

Listing 2.2: Vector implementation of integer multiplication

```
 1  void add()
 2  {
 3
 4  // declaration and defintion of  arrays a[] and b[]
 5  // declaration of output array c[]
 6
 7  __m128i A, B,C;
 8  __m128i* sse_array1 = reinterpret_cast<__m128i*>(a);
 9  __m128i* sse_array2 = reinterpret_cast<__m128i*>(b);
10  int simd_length     = sizeof(__m128i)/sizeof(int);
11
12  // for 16-byte aligned data
13  for(int i=0;i<simd_length;i++)
14  {
15     A = _mm_load_si128(&sse_array1[i]);
16     B = _mm_load_si128(&sse_array2[i]);
17     C = _mm_mul_epi32(A,B);
18     _mm_storeu_si128(&sse_array3[i], C);
19  }
20  // for loop to process remaining unaligned data
21  for(...)
22  ...
23
24  }
```

Limitations

Despite offering high degrees of data-level parallelism, SIMD instructions are prone to several limitations. Thus, while hand-tuning an algorithm to exploit the benefits of a SIMD architecture, a programmer should take care while applying SIMD optimization techniques so that the overall performance is not sacrificed by such limitations [ZR02]. Some of the significant limitations of SIMD instructions are as follows:

1. **Non-Contiguous Memory Access – Gather Operations**

   As mentioned previously, SIMD instructions operate on data items stored adjacent to one other, i.e., in contiguous memory locations with *16*-byte or *32*-byte boundary alignments (in case of SIMD registers with *128*-bit width or *256*-bit width respectively). Therefore, for operations involving read on data items stored in non-contiguous locations, SIMD functionality requires gathering elements from different locations which incurs a high performance overhead. Thus, an optimal solution in this case would be to revert back to the sequential processing of such operations [Bik04].

2. **Scatter Operations and Update Collisions**

   Similar to a contiguous memory load, the results of a single SIMD register can be stored only in contiguous memory locations (*16*-byte or *32*-byte boundary limits). Hence, SIMD cannot handle storage of data to non-contiguous memory locations, i.e., scattering of data to non-sequential memory addresses. Further, when the result of parallel writes of a single SIMD instruction map to the same location, it results in a collision overhead. Current implementations of SIMD architectures are not supported to handle such collisions and hence, similar to a gather operation, an efficient way of processing in both these cases is to adopt a sequential strategy [KKL+09].

3. **Branching Statements**

   SIMD does not favor operations involving branching statements. For control flow operations such as *if-else*, an alternative SIMD primitive can be implemented by means of *masking*, i.e., process of generating a vector containing *1's* or *0's* corresponding to true or false conditions [ZR02]. For example, in case of a database scan operation, this principle of masking can be used. However, the branching statements involved in detecting the mask value (*0* or *1*) to find the match for scan cannot be executed using SIMD capabilities and hence, this task has to be done in a serial fashion. At the same time, vector solutions of this type are not applicable to *if-else* conditions in all situations. Moreover, an optimal performance is not guaranteed in every situation where implementations of this type are used [KKL+09].

4. **Function Calls**

   SIMD functionalities do not support the vectorization of operations involving calls to pre-defined or user-defined functions. Exceptions to limitations of this type are mathematical functions supplied along with compiler intrinsics and user-defined inline functions [ZR02].

2.3.2 Parallelization

Parallelization is a form of computing in which a given task is split into a number of smaller tasks each of which is then executed concurrently. On single core architectures, parallelization is realized by means of *simultaneous multithreading (SMT)* in which the entire program's instructions are divided among all available threads in the core such that each thread runs concurrently sharing the program's resources and their execution is independent to one another. With the evolution of multi-core CPU architectures, the concept of SMT can be extended further to take advantages of all available threads in the entire hardware system.

Akhter and Roberts claim that the speedup achieved as a result of parallelizing a program can be measured by the below metric:

$$Speedup = \frac{Response\ time\ of\ the\ best\ serial\ program}{Response\ time\ of\ the\ equivalent\ parallel\ program} \quad [AR06]$$

Note that the parallelism discussed here is inter-instruction parallelism as opposed to the intra-instruction parallelism provided by SIMD instructions.

Multi-threading

A thread is the smallest unit of program execution which consists of a set of instructions related to one another. Every program contains at least one thread *(main thread)* during its initialization. The main thread in turn can create other threads or can simply execute the entire program on its own. From a hardware point of view, each thread has its own execution path and the operating system handles the task of software to hardware thread mapping. A multi-core CPU consists of several cores with each core provided with a separate hardware environment and support for one or more threads. However, it should be noted that if the system is not used in SMT mode (hyper-threading in case of Intel architectures) , each core can use only one thread for its execution despite offering support for many threads. Therefore, parallelization can be achieved by means of multi-threading in which the execution of the program's instructions is partitioned among multiple threads. In this case, each thread shares the entire resource allocated to the program (for example, memory) and execute independently in a concurrent fashion. Finally, the output can be obtained by combining the results of all threads. Thus, a program execution in a multi-threaded environment is said to be complete only when all the invoked threads complete their individual execution.

In order to achieve true parallelism, the number of threads set to use by the programmer should not exceed the total number of available threads in the system to reduce *context switching*, a process in which the control of a program's execution is transferred from one thread to another thread during which the state of the thread should be saved for it to be restored later. An algorithm tuned to exploit the thread-level parallelism on one machine is always not portable to other machines since the number of cores and consequently the total number of threads vary across several machines [AR06]. For example, the Intel Core 2 Quad Q9550 CPU contains four cores with each core supporting a single thread whereas the Intel Xeon E5 -2690 CPU contains a total of eight cores with two threads per core [BBS14].

An algorithm in a multi-threaded environment may be decomposed into one of the two forms as explained below:

- ***Task decomposition*** - A process in which distinct modules of a program that do not have a sequential flow are assigned to multiple threads and are then executed concurrently. To have a good benefit from this approach, a programmer should take care in order to ensure that the execution of individual modules does not conflict with each other and do not share same resources [AR06].

- ***Data decomposition*** – A process in which a single task or module is split among multiple threads for a better performance, thus reducing the overall problem size. In this way, the amount of work done for a given time period is more compared to a sequential execution of the same module. However, a performance barrier exists in this type of multi-threading when several threads access the same block of data contained in the module [AR06].

For using parallelization techniques in our work, we adopt the principle of data decomposition since we have a sequential flow between distinct modules used in our implementation. Thus, we apply parallelization techniques to each of our individual modules. We explain this in detail in Chapter 4.

Limitations

Parallelizing a program using multi-threading is not guaranteed to provide performance benefits over a sequential execution in all situations. Some of the challenges faced by a multi-threaded program are as follows:

1. **False Sharing**

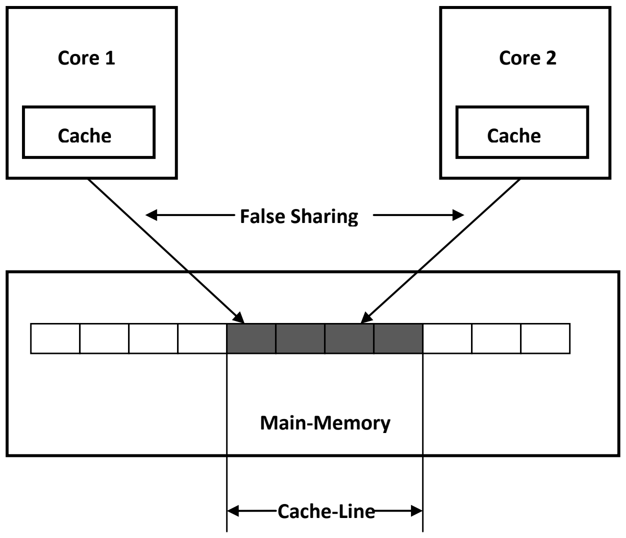

Figure 2.7: False sharing

Consider a scenario in a dual-core system in which each core is provided with one thread. Now, if each thread attempts to make an access to two separate data items that are stored in the same cache-line of the underlying memory for an update operation, then the memory system marks this cache-line as invalid.

This phenomenon is known as *false sharing*. Therefore, any attempt to make an update on data items belonging to this cache-line should take place directly in the memory to maintain cache coherency. False sharing degrades the performance to a significant extent even though threads accessing such data items do not perform the actual update [AR06]. We depict this phenomenon in Figure 2.7 on the preceding page.

2. **Race Conditions and Cache-Line Bouncing**

   Race condition is a situation that occurs when multiple threads attempt accessing the same data item. Therefore, such items should be protected by a locking mechanism, thereby preventing concurrent accesses. This is because if threads accessing the same item perform a write operation, it leads to unexpected results at the end of the program execution. Thus, activities of this kind need to be synchronized such that one thread waits until the other thread completes its work on that item and explicitly releases the lock. Consequently, when a lock is released and the data item under protection is accessed by a thread belonging to another separate core, then the cache-line corresponding to this item and the lock has to be transferred to the cache memory of that core. This is called *cache-line bouncing*. Both race conditions and cache-line bouncing limits the benefits offered by multi-threading [AR06, Kle09].

3. **Load Imbalance**

   High degree of parallelism can be achieved only when there is an equal amount of load on all threads being invoked for a task. Load imbalance causes serious degradation to the performance since the overall response time of the task largely depends upon the thread allocated with the maximum load. For example, when an activity is synchronized among two threads T_1 and T_2, then the thread T_1 with the largest load causes another thread T_2 to wait for a long time for the resource being held by T_1. Thus, T_2 cannot proceed in executing its other activities.

4. **Deadlock**

   Deadlock is a typical multi-threading barrier that occurs when a thread is made to wait indefinitely for a resource that will never be released. For example, assume two threads T_1 and T_2. Now, T_1 is waiting for a resource X_1 being held by T_2. However, T_2 can release this resource only by getting access to another resource X_2 which in turn is held by T_1. Thus, none of them can proceed with their execution, thereby leading to an indefinite deadlocked situation.

2.3.3 Branch-Free Code Technique

Before exploring into the branch-free code optimization technique, we begin with the pipelining of CPUs and then explain control hazards and branch prediction techniques in the following paragraphs to illustrate the need for such optimization techniques.

CPU Pipelining

Pipelining in CPU is a technique in which an instruction execution can be viewed as pipelining of several stages with each stage executed in one CPU Cycle (two times the frequency of a master clock). In this way, multiple instructions may be filled into the pipeline leading to a simultaneous execution of all such instructions [Hei11]. We depict the architecture of a *classic RISC (reduced instruction set computer)* pipeline consisting of a five-stage CPU pipelining mechanism in Figure 2.8. We describe each stage of the CPU pipelining as below.

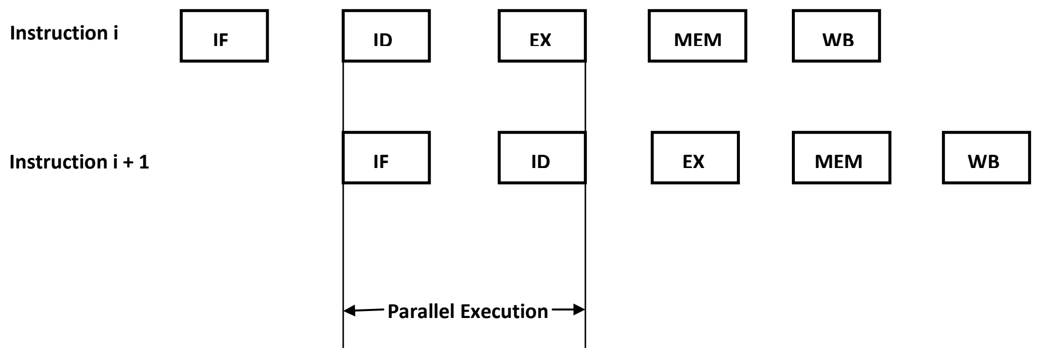

Figure 2.8: Five-stage CPU pipelining of depth 2 (RISC Pipeline)

- ***Instruction Fetch (IF)*** fetches the instruction into the instruction cache and increases the program counter (PC) by one. At this stage, the translation look aside buffer (TLB) performs the instruction memory mapping from virtual to physical memory addresses [Hei11].
- ***Instruction Decode (ID)*** decodes the fetched instruction and reads the operands required for the execution from respective registers [Hei11].
- ***Execution (EX)*** is a stage in which the arithmetic logic unit (ALU) calculates the memory address required for various register-to-register and register-to-immediate operations (For example, *load* and *store* operations). In case of branching instructions, the ALU executes the actual branch to determine its output condition [Hei11].
- ***Memory (MEM)*** is a stage in which the actual loading and storing of data across various registers are performed. However, this stage does not perform any operation that alters the actual register data [Hei11].
- ***Write Back (WB)*** is the final stage of the CPU pipelining in which the result of the instruction execution is written back to respective registers. In case of a branch instruction, this stage appears to remain idle since instructions of this type do not produce any output to be written to the memory [Hei11].

Control Hazards

Control Hazard is a common type of problem that occurs in a pipelined CPU architecture when the instruction to be pipelined is a branch instruction (more specifically conditional branches such as *if-else* statements). Instructions of this type usually can have more than one output. However, the CPU cannot detect the output instruction in advance, i.e., at the time of fetching the main branch instruction. As a result, control hazards force re-fetching of instructions based on the result of the branch instruction. We illustrate this situation with a simple example for a conditional branch instruction in Figure 2.9. Here, we assume that the instruction *i* is a conditional branch instruction which is detected as a branch during the instruction decode stage (ID) and decoded to true or false by the ALU during the execution stage (EX). At this point, the CPU fetches the target instruction determined by the branch result and hence, the instruction *i+1*, the immediate instruction after *i*, is stalled for a total of three cycles.

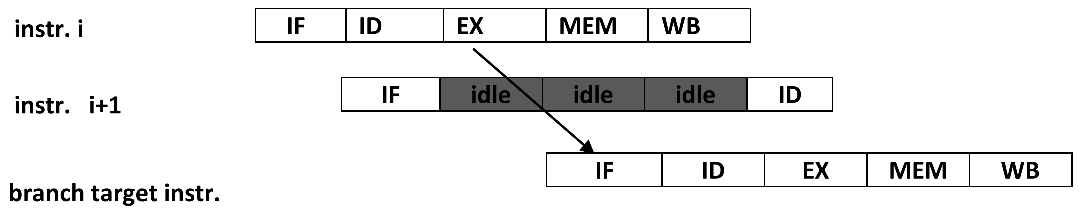

Figure 2.9: Control hazards

Branch Prediction and Predication - Branch-Free Code

In order to overcome the problems faced due to control hazards, modern CPU architectures are implemented with built-in branch predictors. These predictors maintain a history of branching instructions based on data distribution in the past and hence attempt to predict the outcome of a branch. Thus, these architectures decide whether to include or omit the depending instruction of a branch output based on their prediction. However, such branch predictions made by modern CPUs are not guaranteed to be accurate at all times. Ross claims that for a given period of time, if an instruction is included only *50%* of the whole time period based on branch prediction, then the same percentage of mispredictions are likely to occur [Ros04]. Branch mispredictions are very costly, since every time when a prediction fails, an instruction in need has to be re-fetched after flushing the pipeline, thereby substantially reducing the instruction throughput [BBS14]. An optimized solution in this case would be to convert the conditional branch statements of a program into data dependencies, i.e., removal of branches by means of *branch-free code technique.* This procedure is known as *predication* [Ros04]. In Listing 2.3 on the facing page, we show an example of the predication mechanism for a simple database scan operation.

Listing 2.3: Branching code vs. branch-free code

```

// a[]       - input array
// res[]     - result array
// comp_val - value for comparison
// pos       - position of result array

// Branching Implementation
for (int i = 0; i < total_num_tuples; i++)
if (a[i] < comp_val)
res[pos++]=i;

// Branch-Free Implementation
for (int i = 0; i < total_num_tuples; i++)
res[pos] = i;
pos += (a[i] < comp_val);
```

The major disadvantage of a branch-free code optimization technique is that it does not provide an efficient solution over its branching instruction counterpart in all situations due to the execution of a large number of instructions in the corresponding branch-free code [BBS14]. Thus, Kenneeth A. Ross suspects that branch prediction failures might pose substantial performance problems for any CPU architecture over the next few generation [Ros04].

2.4 Summary

To summarize, in this chapter, we first presented a brief introduction about main-memory database systems and their distinctions from a regular disk-resident database system. Then, we explained various properties of main-memory database systems including their storage and processing models. Further, we also presented a detailed working principle of two important in-memory join algorithms in the literature - *sort-merge join* and *hash join*, along with their recent trends with modern architectural changes. Finally, we gave a brief explanation about the importance of code optimization techniques, followed by a detailed explanation of those techniques adopted as part of our thesis work.

3. Radix Join

Over the last few years, two diverging approaches have been proposed in the context of main-memory hash join algorithms. According to these two approaches, hash join algorithms can be grouped into two broad categories – *hardware-conscious joins* (e.g., partitioned hash join) and *hardware-oblivious joins* (e.g., canonical hash join). The former assumes that tailoring the algorithm carefully to hardware parameters of the underlying architecture (e.g., cache memory capacities, TLB entries, memory bandwidth) results in optimal performance of the algorithm. Contrary to this, the latter assumes that such tailoring requirements are not needed due to the capability of the modern hardware in hiding latencies caused by cache and TLB misses without affecting the performance. Teubner et al. confirm that for increasing sizes of the joining relations, hardware-conscious join algorithms results in better performance than hardware-oblivious join algorithms [BTAÖ13].

In this chapter, we explain radix join, another type of hardware-conscious joins, on which we study the performance of code optimization techniques discussed in the previous chapter. For this, we start with the need for the radix join in Section 3.1, followed by its working principle in Section 3.2.

3.1 Disadvantages of Canonical and Partitioned Hash Joins

The hash join algorithms that we discussed in the previous chapter, canonical hash join and partitioned hash join, suffers from various disadvantages as mentioned below:

- As mentioned in the previous chapter, a canonical hash join executes a join algorithm in two phases – *build* and *probe*. The algorithm creates a single hash table for the entire inner relation in the build phase, to which the individual tuples of the outer relation are then scanned and probed for joining partners in the probe phase. The main disadvantage of this approach is that it leads to a large number of random-memory accesses during probing. Further, when the size of the hash table being created exceeds the cache memory capacity, the total number of cache misses increases significantly affecting the overall execution performance of the algorithm [BTAÖ13].

- A partitioned hash join, in analogous to the canonical hash join, avoids the cache miss problem by introducing an extra phase called *partition* phase. In this phase, the algorithm partitions both relations into a number of clusters or partitions with their sizes falling within the capacity of the cache memory. Once all partitions are created, build and probe operations are executed as before for each pair of partitions. Thus, individual hash tables created for each partition satisfy the cache memory limits. However, the algorithm is prone to serious TLB misses if the number of partitions created exceeds the total number of available TLB entries. Therefore, for every partition whose mapping history is not already contained in the TLB buffer, a separate TLB caching mechanism need to be performed [BTAÖ13].

To overcome both these problems in a single join algorithm, radix join is proposed.

3.2 Principle of Radix Join

A radix join is a special type of partitioned hash join in which the join algorithm is self-tuned in such a way that the number of partitions created does not exceed the total number of TLB entries. Thus, the number of available TLB entries serve as an upper bound for the total fan-out of partitions for the algorithm. Further, the algorithm also reduces the number of random-memory accesses [BTAÖ13]. Similar to a partitioned hash join, a radix join algorithm is also executed in three phases - *partition phase, build phase and probe phase.* In Figure 3.1 on the next page, we depict an overview of a two pass radix join. We describe the working logic of the entire radix join in the following subsections. For our convenience, we refer build and probe phases together as *join phase* in the remainder of this thesis.

3.2.1 Partition Phase

The basic idea behind radix join is to perform a different kind of partitioning called *radix partitioning* or *radix clustering.* In this method, the algorithm uses a pre-defined number of bits called *radix bits* for partitioning the entire input relation in a sequence of multiple passes P_i. The task of assigning tuples to individual partitions in each pass is based on the use of a hash function $h_{1,i}(K)$ which determines the partition number to which an individual tuple should be stored. For each pass P_i, the hash function $h_{1,i}(K)$ uses a different set of radix bits and generates equivalent hash values for all the incoming relation tuples (more specifically, the keys). These hash values indicate the partition numbers relative to relation tuples.

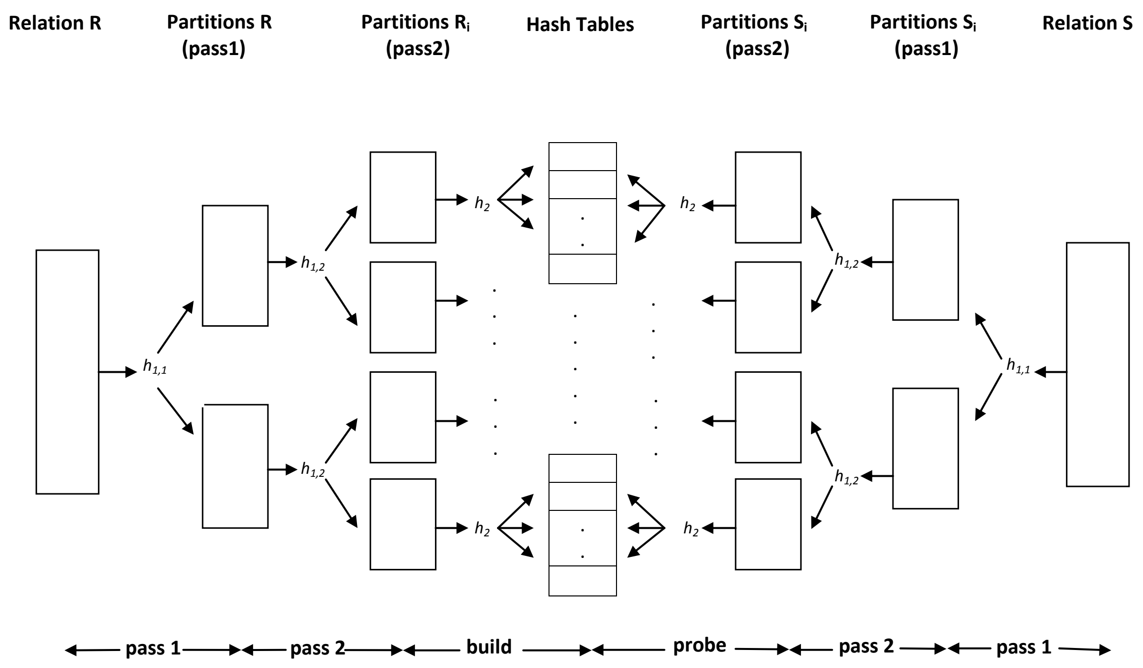

Figure 3.1: Two-pass radix join – adopted from [BTAÖ13]

Let B denote the total number of radix bits used for the entire partitioning phase. Then, the hash function $h_{1,i}(K)$ in each pass P_i uses B_i bits starting from the left such that $\sum B_i = B$. The number of partitions obtained from a cluster C at the end of each pass is given by $N_i(\mathrm{C}) = 2^{B_i}$. Therefore, the total number of partitions at the end of the entire partition phase of a radix join will be 2^B.

In Figure 3.2 on the following page, we show an example of a radix clustering for a simple two-pass radix join. Here, we set the number of radix bits to 3. Thus, the total number of partitions created from applying the algorithm over the relation R (which is assumed to be a single cluster or partition itself) at the end of the first pass with the hash function $h_{1,1}(K)$ using the leftmost one bit will be *2*. The second pass then recursively applies the algorithm to each partition created during the first pass such that the hash function $h_{1,2}(K)$ uses the remaining two bits of the total radix bits. Therefore, the total number of partitions created at the end of the second pass and consequently the partitioning phase will be *8*. Thus, the relation tuples will be re-ordered during every pass. An important property of the radix join is that if the number of passes is set to one, then the algorithm behaves like a simple partitioned hash join. For a typical main-memory database join operation, a two-pass radix join will be sufficient [BTAÖ13, Man02].

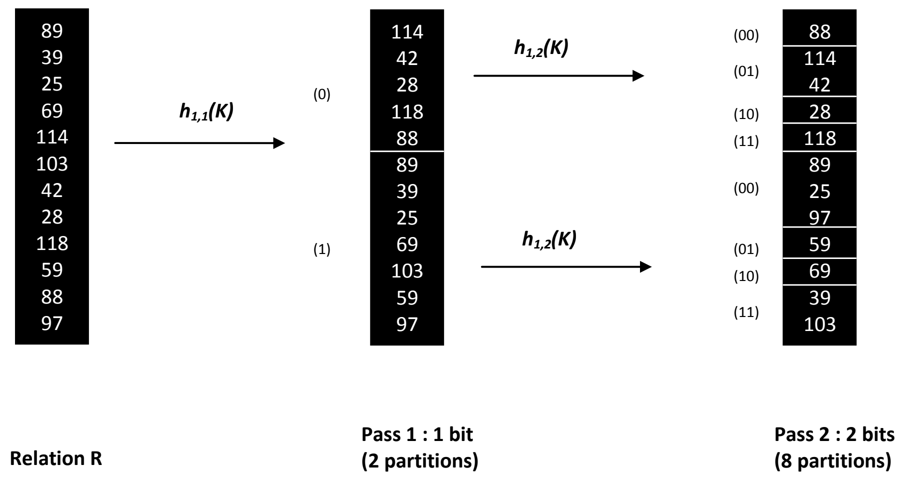

Figure 3.2: Two-pass radix partitioning (using 3 bits)

Manegold et al. indicate three parameters that determine the optimal performance of a radix join [Man02]. They are as follows:

1. Number of passes used (P).
2. Total number of bits (B) used for the entire partitioning phase.
3. Number of bits (B_i) considered for each pass P_i.

The above parameters should be carefully tuned to the underlying hardware such that the number of partitions (2^B) is smaller than the total number of available TLB entries and cache-lines. In this way, a radix join can avoid both cache and TLB misses, at the same time reduces the number of random accesses to the main-memory. Teubner et al. argue that using two passes with *14* or *11* bits for Intel and AMD architectures during the entire partitioning phase refines the partition size in each pass such that each resulting partition fits well in the cache and reduces the overall memory costs involved [BTAÖ13]. Further, even distribution of radix bits B_i in each pass reduces the maximum number of bits used per pass which in turn keeps the number of partitions to be filled during each pass within the TLB entry limits [Man02].

Another major advantage of the radix join is that the algorithm does not require the need of strict boundaries placed between each partition. This is because, the relation is ordered on radix-bits such that each key will have the same B lowest bits for its hash value, and therefore each partition appears consecutively with a cardinality of $|R|/2^B$. Thus, a radix join avoids the overhead of using additional data structures for implementing partition boundaries [Man02]. This can also be seen in Figure 3.2, where each partition appears one below the other without any visible partition boundaries.

3.2.2 Join Phase (*Build and Probe*)

Once partitions for input relations are created, a canonical hash join algorithm comprising of both build and probe phases is applied individually between each pair of partitions R_i and S_i (corresponding to two input relations R and S respectively). For executing the join phase, Manegold et al. proposed the method of *classical bucket-chaining algorithm*, while Kim et al. proposed *re-ordered histogram-based hash table algorithm* [KKL$^+$09, Man02]. For our work, we use the bucket-chaining algorithm since the former provides an optimized implementation of the join phase compared to the latter and even over the latter algorithm optimized with SIMD instructions.

Classical Bucket-Chaining Algorithm

The principle behind a classical bucket-chaining algorithm, in the context of radix join, is to maintain a data structure called *buckets* for implementing the hash table for every partition R_i during the build phase. Then, a user-defined hash function $h_2(K)$ is applied on keys contained in R_i to produce an integer hash value. This value indicates to which bucket the key should be placed. A bucket collision is said to occur when more than one key corresponds to the same bucket. In this case, the colliding keys are inserted into another data structure called *next* and all these colliding keys are chained together by means of a linked list [KKL$^+$09, BTAÖ13, ZHB06]. A good hash function is one that results in a uniform distribution of hash values. However, implementing such function is not an easy task in all situations. A traditional method of implementing the hash function is to use an integer modulo operation on the incoming keys to obtain the hash value. We show a simple example of a bucket-chained hash table representation in Figure 3.3, where we use the hash function *key mod 10.* It can be seen that collision occurs in the 0th bucket since keys *120*, *140* and *190* represent the same hash value. Hence, keys *120* and *190* are inserted into the *next* data structure and all these colliding keys are chained together by a linked list connected to this bucket.

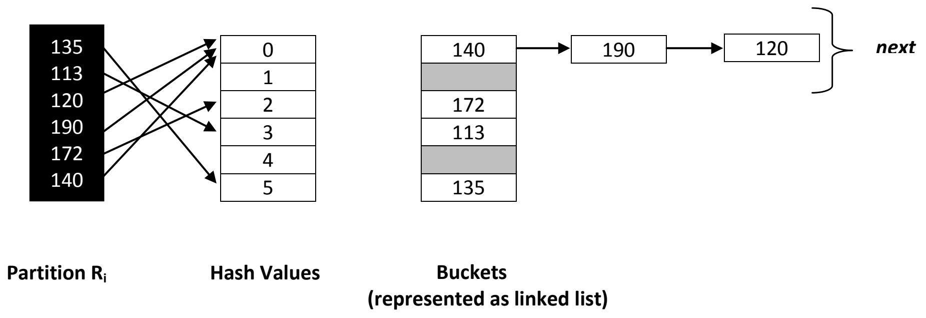

Figure 3.3: Classical bucket-chained hash table

Once the hash table is built, probing is done by applying the same hash function $h_2(K)$ on keys contained in partitions S_i. The result of the hash function now indicates the bucket number to which this key should be compared. Thus, in this case, the key should be probed with all the individual entries chained together by the linked list referenced by the bucket for finding a matching join partner [Man02]. Thus, in the example shown in Figure 3.3 on the previous page, any key in S_i with a hash value *0* should be probed with *120*, *140* and *190* for a possible join partner.

Improved Hash Funcition

Manegold et al. argue that use of hash functions for $h_2(K)$ as mentioned above are very expensive since they cost around *40-80* CPU cycles on modern architectures [Man02]. Hence, they adopted an alternative method in which the number of hash buckets (N) is set to a power of *2*, i.e., $N = 2^i$. Thus, they replaced the costly modulo operation by a cheap bitwise AND operation between the key and *N-1*. Their results confirm that the newly implemented hash function improves the overall performance of the algorithm by four times. The hash function can be represented as below:

$$h_2(K) = (Key\ AND\ N\text{-}1) \qquad \text{[Man02]}$$

With this hash function $h_2(K)$, the process of probing can be done in a similar fashion as mentioned earlier.

Improvements to Bucket-Chaining Algorithm

Teubner et al. replaced the concept of linked list representation by means of array position indexes such that both the bucket and the *next* data structure are implemented by means of plain *C* arrays [BTAÖ13]. We represent the above example using this method in Figure 3.4.

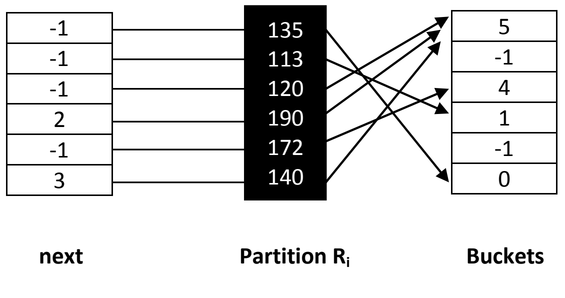

Figure 3.4: Improved bucket-chained hash table – adopted from [BTAÖ13]

In this method, all the individual entries in the bucket and the *next* arrays are initialized to *-1*. An entry in the bucket consists of the index of the tuple key instead of the actual key value. In case of a bucket collision, the current bucket value is inserted into the *next* array of the key that causes collision and the bucket entry is updated with the index of this key. Thus, in the above example, the bucket number *0* is updated with the index of its most recent key *140* and the *next* array of this key is updated with the previous value in this bucket, i.e., 3, the index of the key *190*. Therefore, to perform a probe, the partition keys with index values corresponding to the bucket and the *next* entries are looked up recursively as long as the value in the *next* array is greater than -1. For instance, in Figure 3.5, we show the steps required to perform a probe operation for the key *120*. For sake of simplicity, we omit the mapping between the partition R_i and its corresponding bucket entries. The steps for probing are explained in detail as below:

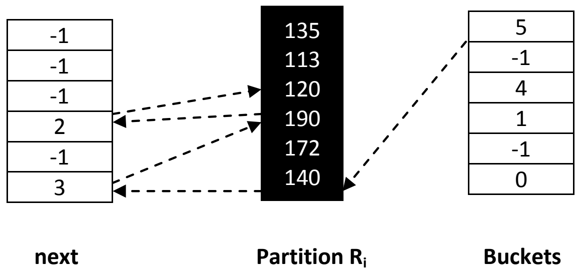

Figure 3.5: Improved bucket-chained hash table – lookup for 120

1. The hash value of the key *120* is *0*. A lookup on this bucket gives the index of R_i (5) for starting the process of probing.

2. The R_i value corresponding to the index *5* is not equal to *120* and at the same time, its corresponding *next* value is greater than *-1*, which in this case is *3*. Thus, another lookup is performed on R_i corresponding to the index *3* which again gives a non-match.

3. Finally, a lookup is performed on the *next* value of R_i with index *3*, the output of which is *2*. The R_i value relative to the index *2* is *120* and hence, a match is obtained. Further lookups are stopped since the *next* value corresponding to the most recent index (2) is *-1*.

Teubner et al. argue that, for any in-memory hash join operation, performing a probe using this method provides an efficient performance over the previous method [BTAÖ13]. For our implementation of the radix join, we use this method for performing the probe along with the build in the join phase.

4. Implementation of Optimization Techniques

We now turn our attention to the actual implementation of various code optimization techniques that we discussed in the previous chapter (*vectorization, parallelization and branch-free coding*) to the scalar radix join algorithm. In addition, we also aim at implementing a combination of these individual optimization strategies that are found to provide improvement to the scalar version. Hence, we can identify all possible variants within the context of these optimization strategies that have a positive influence for a radix join. For a standard serial radix join, several versions are proposed in the literature [BLP11, Man02, BTAÖ13]. We adopt one of these versions with certain changes as a starting point for the application of our optimization techniques.

In this chapter, we begin with the explanation of a set of preliminaries adopted for our implementation in Section 4.1. Then, in Section 4.2, we present an overview of our scalar version that we further use for our work. Finally, in Section 4.3 and Section 4.4, we explain our approach of applying the optimization techniques (both individual and combination) to each phase of the radix join.

4.1 Preliminaries

Framework

For our experiments, we use the *Column-oriented GPU-accelerated DBMS (CoGaDB)* for implementing the optimization techniques. CoGaDB is a main-memory database management system developed at the University of Magdeburg. The main purpose of this database system is to provide a hybrid environment (CPU/GPU platforms) with database operators needed to achieve optimal performance during query processing. The database uses column-oriented approach for its data storage and operator-at-a-time strategy as its data processing model. The system is implemented using C++ and

Cuda C languages. Hence, we use C++ language for our implementation. Further, we also use various features (for example, smart pointers for the storage of join results) offered by this system for our work. A detailed explanation of CoGaDB can be found at the corresponding website [1].

Experimental Setup

For the application of our optimization techniques, we consider columns or attributes with *32*-bit integer data items. CoGaDB stores these columns using simple *C arrays* instead of STL containers or structures for performance reasons. As mentioned already, we use C++ language for our implementation and GNU C++ compiler *4.6.4* for compiling our implemented radix join variants. Once the join processing is over, we store the join results in a *Binary Association Table (BAT)*. For this, we use the abstract pointer, *PositionListPairPtr*, offered by CoGaDB. Here, we extract and store Row-IDs (RIDs) of matching tuples from join columns of both input relations using which the subsequent tuple reconstruction can be performed. However, efficient tuple reconstruction is not part of our work and, thus, not further discussed here. Regarding the type of the join, we consider a simple equi-join (join condition with predicates of the form θ=A) for all our radix join variants.

4.2 Serial Radix Join

For applying all our optimization techniques, we adopt the serial implementation of the radix join proposed by Teubner et al. [BTAÖ13]. Their implementation is publicly available at their website [2]. However, we make certain changes to the existing code for our work as explained below:

- The versions of the scalar radix join proposed by Teubner et al., Blanas et al., Kim et al. and Manegold et al. use *64*-bit wide integer tuples of the form (*key,value*) stored together in structures [BLP11, Man02, KKL$^+$09, Man02]. Since our framework (CoGaDB) is based on column-oriented storage of data items, we consider only the storage of *32*-bit integer keys. This allows the storage of keys in contiguous memory locations which provide several benefits as mentioned in Chapter 2 under *column-oriented storage model*. Further, storage of keys in this fashion proved to be advantageous for the application of our vectorization techniques. We explain them later in this chapter.

- All versions of the scalar radix join that we mentioned above do not consider the storage of join results. Instead, their implementations are based on counting the number of matches obtained as a result of performing the join between two input relations. However, in our implementation, we store the position of matching tuples of both input relations from join operation. For this, we extract the RIDs

[1]http://wwwiti.cs.uni-magdeburg.de/iti_db/research/gpu/cogadb
[2]http://www.systems.ethz.ch/projects/paralleljoins

of the original input keys (before partitioning) at run-time and store them along with their respective keys in separate *plain C arrays*.

We now present an overview of the partition phase and the join phase of the serial radix join implementation that we adopted from Teubner et al. [BTAÖ13] in the following sub-sections.

Partition Phase

As mentioned in Chapter 3, using two passes for a radix join will be sufficient for a typical main-memory database to create cache-sized partitions for probing. Further, use of *14* or *11* bits is sufficient to significantly avoid a large number of cache and TLB misses in Intel and AMD architectures. For our implementation, we set the number of radix bits to *14*. However, a potential misconfiguration of radix bits between these two options would not have a major impact on the performance of the algorithm [BTAÖ13]. We mentioned in Chapter 3 that with B bits, the total number of partitions or clusters obtained at the end of the entire partitioning phase will be 2^B. Thus, in our case, we have *16384* partitions (2^{14}). Teubner et al. confirm that by performing a radix partitioning with this configuration, we significantly reduce the number of cache misses as well as TLB misses compared to the implementation of Blanas et al. with *2048* partitions created with a single pass [BTAÖ13, BLP11]. Note that the cost of the partition phase increases with increasing number of partitions. However, it significantly reduces the size of all partitions which allows the join operation to be performed even faster with partitions confirming to cache memory limits. Further, using *14* or *11* bits does not have a major problem on TLB configuration requirements (store and load TLB misses) in modern architectures supporting a TLB capacity of *512* page entries (at *L3* level).

Listing 4.1: Partition phase – scalar implementation

```

//Preliminary Step : Determine the number of partitions

// tpp         - number of tuples per partition
// dest[]      - Memory allocated for storage of partitions (keys)
// dest_pos[] - Memory allocated for storage of partitions (positions)

//Determine the number of tuples in each partition
for(unsigned int i=0; i<num_tuples; i++)
{
        partition_number = radix_hashing(key);
        tpp[partition_number]++;
}


//Allocate memory for each partition based on the count determined above

//Store the keys in their respective partitions
for( unsigned int i=0; i<num_tuples; i++)
{
        partition_number   = radix_hashing(key);

        *dest[partition_number] = *key;
        ++dest[partition_number];

        *dest_pos[partition_number] = *position;
        ++dest_pos[partition_number];

}
```

We show an overview of the serial radix join implementation in Listing 4.1. For partitioning the input relation, we use the concept of radix partitioning, i.e., in each pass, we use the leftmost *7* bits for determining the number of partitions and our hash function uses the same number of bits for assigning the individual tuples (*32*-bit keys) to a particular partition. Once we set the number of partitions using this configuration, we perform the partitioning phase in three steps as below:

1. Determine the number of keys in each partition - computation of local histogram for each partition (prefix-sum) (lines *9-13*).
2. Allocate the memory for each partition based on the number of keys determined in the previous step (line *16*).
3. Store the actual keys to their respective partitions (lines *19-29*).

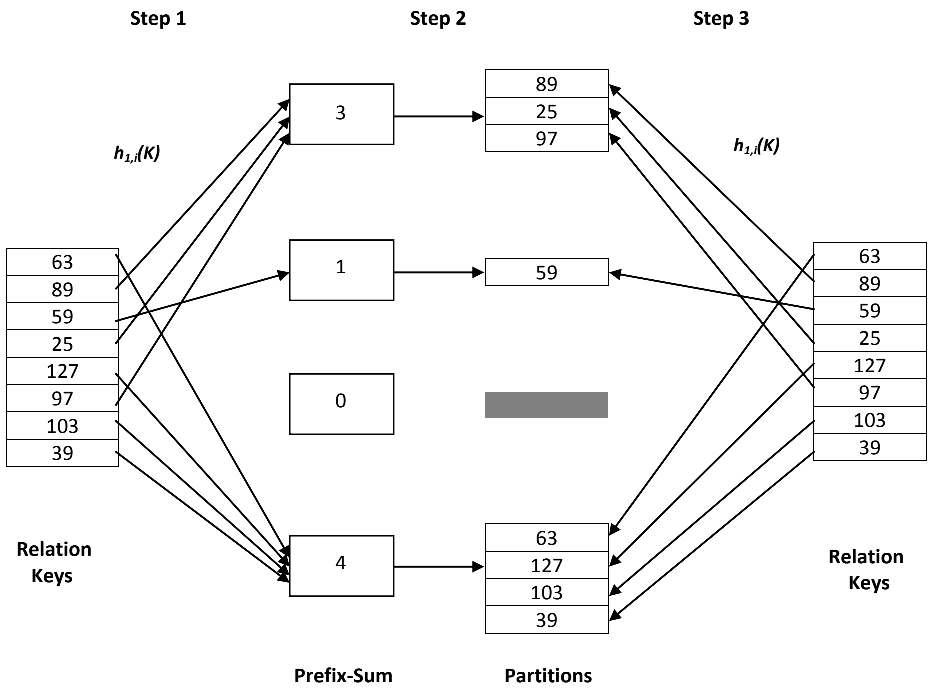

Figure 4.1: Serial radix partitioning

In each pass P_i, we apply the radix hash function $h_{1,i}(K)$ on the incoming keys of the input relation. The output of this hash function determines the partition number to which the key should be placed. Hence, we update the tuple counts of partitions relative to the partition number. Once we determine the tuple counts for all partitions, we allocate the memory for each partition based on these counts, followed by which we store the actual keys. In Figure 4.1, we show a simple example for the whole idea behind serial radix partitioning. In this example, the relation keys *89, 25* and *97* belong to partition *0* (determined by $h_{1,i}(K)$). Using the prefix-sum technique, the tuple count of this partition is updated to *3* based on which the memory is allocated to store the corresponding keys. Finally, all the three keys are stored in partition *0*. The partition *2* does not contain any keys and hence, no memory space is reserved for it (denoted in grey in Figure 4.1 on the preceding page).

Note that along with the actual keys, we perform the partitioning even for their respective positions (lines *26* and *27*), i.e., we store even the relation RIDs of the tuple keys in a separate set of partitions using the partition number obtained as a result of performing the radix partitioning over their respective keys. Therefore, the partition number and the partition RID (position of a key in a partition) will be the same for keys and their respective positions. Thus, during the join phase, we do not build a separate hash table for partitions filled only with relation RIDs and they can be extracted by simply retrieving the partition RIDs of their respective keys using entries in the hash table. Hence, with extremely smaller sized partitions for keys and their respective positions, we do not conflict the cache miss avoidance property of the radix join.

Join Phase (*Build and Probe*)

The implementation of Teubner et al. performs build and probe together for each set of partition pairs R_i and S_i (corresponding to two input relations R and S) [BTAÖ13]. Thus, once all partitions are created for R and S, we check the number of tuple keys in both partitions belonging to each pair and then send them to the join phase only if their respective tuple counts are greater than zero. Thus, we reduce the total number of unnecessary probing. This principle is slightly different from the implementation of Blanas et al. where the build is first performed for all R_i (smaller partitions) and then probing is carried out separately for each set of partition pairs. This increases the number of memory trips since the partition R_i need to be accessed from the main-memory during both build and probe phases [BLP11, BTAÖ13]. Note that while sending each partition pair to the join phase, we send partitions filled with relation RIDs along with partitions filled with keys.

Listing 4.2: Join phase – scalar implementation

```

//R_partition, S_partition          - partitions with keys
//R_pos_partition, S_pos_partition  - partitions with relation RIDs
//R_tids, S_tids                    - arrays to store positions of matching keys(PositionListPairPtr)

//build
for(unsigned int i=0; i<num_R_partition;)
{
        bucket_number          = Hash(R_partition[i]);
        next[i]                = bucket[bucket_number]; //for referencing keys
        bucket[bucket_number]  = ++i;
}

//probe
for(unsigned int i=0; i<num_S_partition; i++)
{
        bucket_number = Hash(S_partition[i]);
        for(...) // probe all R keys with same bucket number
        {
          // id - partition RID referenced by the bucket number
          if(R_partition[id] == S_partition[i])
          {
                // we store the RIDs of matching keys
                R_tids[pos] = R_pos_partiton[id];
                S_tids[pos] = S_pos_partition[i];
                pos++;
          }
        }
}
```

The implementation of Teubner et al. consists of both classical bucket chaining join algorithm and re-ordererd hash table algorithm for carrying out the join phase [BTAÖ13]. As mentioned in Chapter 3, we adopt the classical bucket chaining algorithm for our work. We show this implementation in Listing 4.2 on the preceding page. We explain the build and probe of the join phase one by one below:

- **Build** - For performing the build, Teubner et al. first created the hash table for R_i. For this, they adopted the strategy proposed by Manegold et al. [Man02, BTAÖ13]. Accordingly, the number of buckets in the hash table is set to a power of *2 ($N=2^i$)* and the hash value for an individual key is determined using the hash function based on a bitwise AND operation between the key and *N-1* instead of a modulo hash function. We explained this hash function in detail in Chapter 3. Further, they replaced the concept of pointers by plain arrays with position indexes for referencing tuples of R_i in order to achieve an efficient main-memory algorithm design (lines *7-12*). Hence, the hash table is filled with partition RIDs of keys belonging to R_i.

- **Probe** - Once the hash table is built, the same hash function is applied over keys of S_i partitions to determine the hash value or the bucket number for probing. Then, keys of S_i are compared with keys of R_i referenced by the bucket number for determining a match (lines *15-21*). As we discussed earlier, at this step, the implementation of Teubner et al. store only the number of matches [BTAÖ13]. However, for our implementation, we retrieve the original positions (relation RIDs) of matching keys and store them in separate arrays of type *PositionListPairPtr*, an abstract smart pointer offered by CoGaDB (lines *24* and *25*). Using this array, the tuple re-construction can be performed for retrieving the entire tuple corresponding to joined attribute values.

Now, we have our serial radix join with certain changes from Teubner's implementation to meet our needs. Using this implementation, we apply our optimization techniques for studying their performance behavior for an in-memory radix join technique.

4.3 Individual Optimizations

In this section, we explain how we apply the discussed optimization techniques individually to each phase of the radix join. Accordingly, we first explain our approach to the partition phase in Section 4.3.1, followed by the join phase in Section 4.3.2.

4.3.1 Partition Phase

For optimizing the whole partition phase, we apply our optimization techniques to steps *1* (local histogram) and *3* (storage of partition tuples) mentioned in Section 4.2. We do not change any settings in parameters used for a serial radix join that configure the partitioning stage such as the number of passes or the fraction of radix bits used in each pass. Moreover, our implementation of the serial radix join does not contain any conditional branches in the partition phase. Therefore, we implement branch-free code optimization technique only for the join phase. Thus, in this section, we discuss vectorization and parallelization strategies for the partitioning stage in the following subsections.

4.3.1.1 Vectorization

Listing 4.3: Partition phase – vectorized implementation

```
1
2  //Determine the number of tuples/keys in each partition
3  for(unsigned int i=0; i<simd_num_tuples; i++)
4  {
5          __m128i partition_number = simd_radix_hashing(simd_keys);// 4 partition numbers
6          tpp[((int*)&partition_number)[0]]++;
7          tpp[((int*)&partition_number)[1]]++;
8          .
9          .
10 }
11
12 //serial processing for remaining keys that do not fit in one complete SIMD register
13
14 //Allocate memory for each partition based on the count determined above
15
16 /Store the keys in their respective partitions
17 for( unsigned int i=0; i<simd_num_tuples; i++ )
18 {
19         __m128i partition_number = simd_radix_hashing(simd_keys);
20
21         // unroll for all SIMD keys
22         *dest[((int*)&partition_number)[0]] = *simd_keys[0];
23         ++dest[((int*)&partition_number)[0]];
24
25         *dest_pos[((int*)&partition_number)[0]] = *simd_positions[0];
26         ++dest_pos[((int*)&partition_number)[0]];
27
28         *dest[((int*)&partition_number)[1]] = *simd_keys[1];
29         ++dest[((int*)&partition_number)[1]];
30
31         *dest_pos[((int*)&partition_number)[1]] = *simd_positions[1];
32         ++dest_pos[((int*)&partition_number)[1]];
33
34         .
35         .
36         .
37 }
38
39 // Remaining keys processed in a serial fashion
```

As mentioned in Chapter 2, we apply SIMD techniques (*data-level parallelism*) via C++ SSE instruction sets for implementing our vectorized radix join algorithms. To benefit from these SIMD techniques, a programmer must take care with explicit handling so that the overall performance of the resulting algorithm is not sacrificed [Bik04]. For our approach, we use this idea for writing the vectorized partition code. Therefore, we apply vectorization only to code areas where SIMD instructions give an advantage over its serial counterpart. We sketch an overview of the code (integer implementation) for this approach in Listing 4.3.

As stated earlier, we perform the partitioning in three steps. As a preliminary step, we first determine the number of partitions in each pass and then perform the steps required for partitioning the input relations. Accordingly, we fill the SIMD register with relation keys and then apply the hash function for radix partitioning (line *5*). Since we implement our optimization algorithms for *32*-bit integers, we use *128*-bit SIMD registers that can hold four *32*-bit (*4* bytes) keys at a time. Note that our SIMD implementation is slightly different from the implementation proposed by Kim et al. for sort-merge join and hash join where only two tuples can be processed simultaneously because of the *64*-bit *(key,value)* tuple structure [KKL+09]. Our serial radix hash function for computing the partition number consists of two different operations - Bitwise *AND* followed by a logical right shift operation between the input key and the fraction of radix bits used in each pass for partitioning. To convert this into an equivalent vector form, we use the intrinsics ***_mm_and_si128*** and ***_mm_srli_epi32*** that determines the partition number for four different tuples/keys at the same time. However, the process of updating the partition tuple counts (prefix-sum) cannot be done here in a SIMD fashion. We know that any single SIMD operation on *32*-bit integers can be viewed as a simple loop unrolling of depth *4* combined with SIMD capability. Thus, for our implementation, once we compute the partition numbers in a SIMD fashion, we unroll the further processing inside the respective *for loops* until we reach a depth of *4*, i.e., we serially update the partition tuple counts for four different keys on which the SIMD radix hash partitioning is applied (lines *6-9*).

Once we determine the partition tuple counts, we need to allocate memory for all partitions and then store the corresponding keys and positions (RIDS) in these memory locations. However, this step cannot be processed using SIMD instructions since different partitions reside in different memory locations and hence, SIMD keys (four *32*-bit keys) and their corresponding SIMD positions (four *32*-bit relation RIDs) need to be scattered to non-contiguous memory addresses. But, we know that SIMD architectures do not support scatter storage of data and therefore, we perform this step in a serial manner. Thus, as before, we use SIMD instructions only to compute partition numbers for four different keys simultaneously, followed by which we perform the remaining steps in a scalar fashion. In lines *21-36*, we show this step for storing four different keys and their corresponding RIDs to their respective partitions.

Thus, in our vectorization approach, we combine the use of SIMD instructions along with a loop unrolling of depth *4* (for *32*-bit integers). Therefore, we avoid the limitations suffered by a general hash join because of SIMD scatter operations as mentioned by Kim et al. [KKL+09].

4.3.1.2 Parallelization

We now explain how we apply thread-level parallelization concepts to the partition phase of the radix join. In this section, we give an overview about the support provided by C++ for multithreading, followed by our approach to apply them for partitioning the input relations.

Multi-threading in C++

The concept of multi-threading is supported in C++ using the standard library *thread.h*. However, not all versions of C++ enable the use of this library since it is implemented only for C++ 11 and higher standards. For our work, we make use of the threading functionalities using boost libraries which is portable in lower C++ standards. A detailed explanation of C++ boost libraries for multi-threading can be found in [Wil07].

Shared Partitioning

As mentioned in Chapter 2, we use the principle of data decomposition for parallelizing the radix join, i.e., we parallelize the partition phase and the join phase separately since both these phases have a sequential flow between them. We set the number of threads equal to the number of CPU cores, i.e., for each available CPU core, we use at most one thread so that each thread divides the whole partitioning task among them. This is because using more than one thread per core *(Simultaneous multi-threading)* degrades the performance especially in the SMT region due to the sharing of physical resources such as caches(*L1*) and TLBs among threads respective to each core [BATz13].

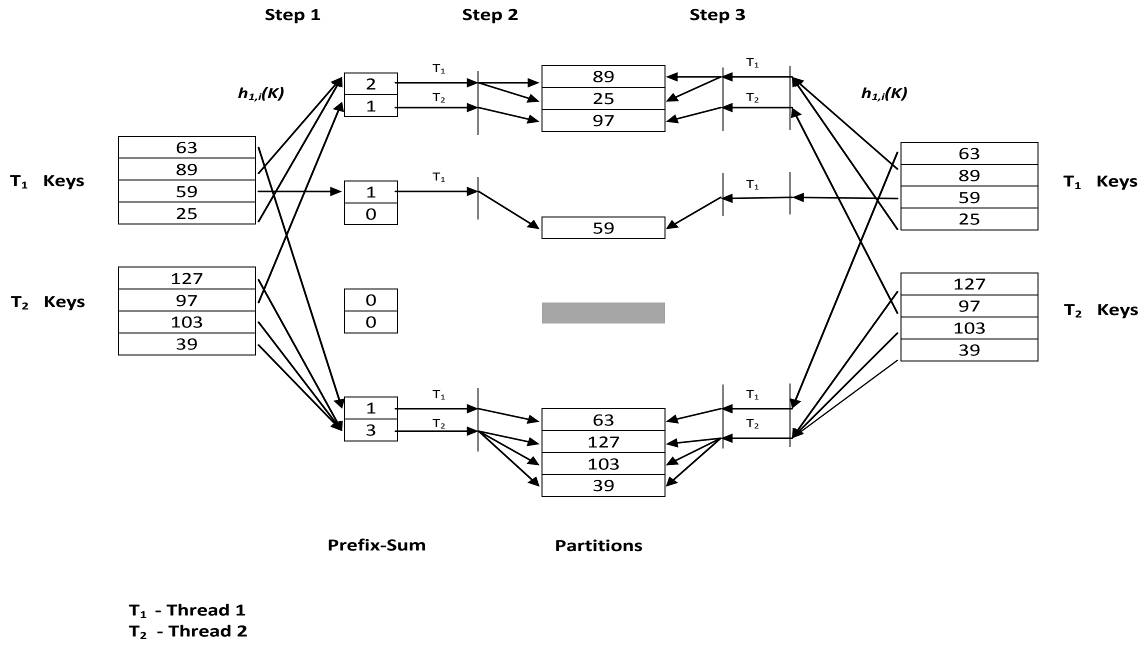

Figure 4.2: Shared partitioning using 2 CPU core threads

For partitioning the input relation in each pass, we use the concept of *shared partitioning* proposed by Teubner at al., i.e., each thread works on a different subset or sub-relation of the entire relation and then update its own partition tuple count, i.e., we perform a thread-relative prefix-sum. Thus, in every partition, we determine the number of tuple keys corresponding to each worker thread. Then, using these partition tuple counts, we reserve a range of memory for each thread in every available partition and then allow all worker threads to write their keys in their own reserved partition memory in a parallel fashion. In this way, we avoid the contention for memory resources and allow worker threads to perform their tasks without any need for synchronization [BTAÖ13]. In Figure 4.2, we show a simple example of shared partitioning using two threads. In this example, the key *63* is allocated to thread *1*, whereas keys *127, 103* and *39* are allocated to thread *2*. However, all these keys belong to partition *3*. Hence, the respective partition tuple counts for each thread in this partition is updated using the prefix-sum technique in a parallel fashion. Thus, the counts of thread *1* and thread *2* in this case are *1* and *3* respectively. Based on these counts, a separate memory is allocated for each of these two threads in partition *3*. Finally, each thread writes their respective keys to their corresponding memory in this partition again in a parallel fashion.

In Listing 4.4, we show our implementation for parallelized radix partitioning. Lines *4-8* shows how each thread updates the number of tuples relative to them in a particular partition. Once we perform this step in parallel, we allocate the memory for each thread in every available partition as mentioned earlier, following which we set each thread to perform the task of storing keys and positions (RIDs) in parallel without synchronizing them (lines *15-25*). Note that we use two dimensional arrays for partition tuple counts and partition memories in contrast to one dimensional arrays used in a serial version.

Listing 4.4: Partition phase – parallel implementation

```
1
2
3  //each thread in parallel -   determine the number of partition tuples/keys
4  for(unsigned int i=0; i<thread_num_tuples; i++)
5  {
6          partition_number = radix_hashing(key);
7          tpp[thread_id][partition_number]++;
8  }
9
10
11 //In main program - reserve thread-based partition memory based on above count
12
13
14 //each thread in parallel - storage of keys and positions
15 for( unsigned int i=0; i<thread_num_tuples; i++)
16 {
17         partition_number   = radix_hashing(key);
18
19         *dest[thread_id][partition_number] = *key;
20         ++dest[thread_id][partition_number];
21
22         *dest_pos[thread_id][partition_number] = *position;
23         ++dest_pos[thread_id][partition_number];
24
25 }
```

4.3.2 Join Phase

The output of the partitioning phase produces a set partition pairs equal to 2^B (*16384* in our case). For each partition pair, we check whether the number of tuples (keys) in both partitions exceeds zero before sending them to the join phase. As mentioned already, we perform the build and the probe together in join phase instead of having separate phases for each of the individual partition pairs. The time required for building hash tables for each partition R_i of the smaller relation is very less compared to the overall join phase. Hence, we are mainly concerned in applying our code optimization techniques to the areas where we perform probing. We discuss our method of optimizing the join phase in the following three subsections.

4.3.2.1 Vectorization

The probing between a partition pair R_i and S_i corresponding to two input relations R and S can be viewed as a simple scan of each key in S_i over all keys in R_i with the same hash value. In Listing 4.2 on page 41, we show how we perform this scan using two *for loops*. For applying our vectorization techniques to the join phase, we identified two approaches for handling them. We explain them below.

Approach 1

For our first vectorized approach of the join phase, we use SIMD instructions only for the second *for loop* in Listing 4.2 on page 41, i.e., for each key in S_i, we compute the hash value (bucket number) and then perform the probing with all R_i keys referenced by the corresponding bucket. For this, we pack all such R_i keys that fit into a single SIMD register and then execute the probing. We show this approach in Listing 4.5.

Listing 4.5: Join phase – vectorized implementation (first approach)

```
 1
 2  //probe
 3  for(unsigned int i=0; i<num_S_partition; i++)
 4  {
 5          __m128i comp_val = _mm_set1_epi32(S_partition[i]);
 6          bucket_number    = Hash(S_partition[i]);
 7
 8          //simd_bucket_length = simd length of R keys with same bucket number
 9          for(j=0; j<simd_bucket_length; j++)
10          {
11                  __m128i rkey     = _mm_set_epi32(...); //pack 4 keys at a time
12                  __m128i rkey_pos = _mm_set_epi32(...); //pack their positions
13                          mask     = SIMD_COMP(rkey,comp_val);
14
15                  for(k=0; j<sizeof(__m128i)/sizeof(int); k++)
16                  {
17                          if((mask >> k) & 1)
18                          {
19                            // we store the RIDs of matching keys
20                            R_tids[pos] = R_pos_partiton[[((int*)&rkey_pos)[k]]];
21                            S_tids[pos] = S_pos_partition[i];
22                            pos++;
23                          }
24                  }
25          }
26
27          // for remaining keys with the same bucket number - serial processing
28  }
```

Thus, in Listing 4.5 (starting from line *11*), we pack all R_i keys with the same hash value into a single SIMD register using the SSE intrinsic *_mm_set_epi32*. In addition, we also pack their respective relation RIDs into another SIMD register. Once the elements are packed, we perform a SIMD comparison, i.e., comparison of four R_i keys with a single S_i key and then retrieve a bit mask for each of these comparisons. Using this bit mask, we then evaluate the individual mask value for each of the R_i keys which determines whether there is a match with the main S_i key from the first *for loop*. Whenever there is a match, we store the corresponding relation RID of the R_i key and that of the main S_i key. We perform this approach for all R_i keys with the same hash value until we reach a multiple of 4 and then perform a serial processing for all the remaining keys that do not fill a complete SIMD register (line *27*).

The main disadvantage of this approach is the packing of R_i keys into a single SIMD register. This is because all R_i keys referenced by the same hash value are not guaranteed to reside in contiguous memory locations due to the nature of bucket-chained hash table. Hence, we need to gather all such keys into a single SIMD register. As mentioned in Chapter 2, SIMD implementations do not support such gather operations and therefore, are prone to serious performance degradation problems. Also, the mask evaluation performed for SIMD probing is very expensive which further worsens the overall performance of the probing. Moreover, during our testing process, we found this approach to produce poor results when compared to the approach that we will dicuss in the next subsection. Due to these reasons, we do not use this approach as our final implementation for the vectorized join phase.

Approach 2

To overcome the disadvantages presented above, we adopt a strategy similar to the vectorized partitioning. We show this approach in Listing 4.6. Thus, in this approach, we apply SIMD instructions only to the first *for loop*, i.e., we use SSE intrinsics to determine the hash value or the bucket number for four different S_i keys at the same time (line *6*). Then, for each S_i key in the SIMD register, we perform the scan in a serial fashion by unrolling them until we reach a depth of *4* (line *8-40*). By doing this, we avoid both gather operations as well as expensive mask evaluation in the first approach. Hence, we use this approach as our final vectorized implementation for the join phase.

Listing 4.6: Join phase – vectorized implementation (second approach)

```
 1
 2  //probe
 3  for(unsigned int i=0; i<simd_length; i++)
 4  {
 5
 6          __m128i bucket_num = SIMD_HASH(simd_skeys);
 7
 8          for(...) // probe all R keys with hash value bucket_num[0]
 9          {
10                  // id - partition RID referenced by the bucket_num[0]
11                  if(R_partition[id] == ((int*)&simd_skeys)[0])
12                  {
13                    // we store the RIDs of matching keys
14                    R_tids[pos] = R_pos_partiton[id];
15                    S_tids[pos] = ((int*)&simd_spos)[0];
16                    pos++;
17                  }
18          }
19
20          for(...) // probe all R keys with hash value bucket_num[1]
21          {
22
23                  // id - partition RID referenced by the bucket_num[1]
24                  if(R_partition[id] == ((int*)&simd_skeys)[1])
25                  {
26                    // we store the RIDs of matching keys
27                    R_tids[pos] = R_pos_partiton[id];
28                    S_tids[pos] = ((int*)&simd_spos)[1];
29                    pos++;
30                  }
31          }
32
33          .
34          .
35          .
36
37  }
38
39  // for remaining keys of S in a serial fashion
```

4.3.2.2 Parallelization

Similar to a vectorized join phase, we adopt two different approaches for parallelizing a scalar join phase of a radix join algorithm. In the first approach, we use the idea of parallelizing the individual join phases of every partition pair, i.e., in each join phase, the partition S_i is divided among worker threads so that each thread perform the process of probing between the hash table built for R_i and the portions of S_i allocated to them. In the second approach, we use the idea of parallelizing the overall join phase, i.e., we divide the total number of available partition pairs among worker threads so that each thread is provided with their own partition pairs with which they perform independent join phases. Thus, the former approach parallelizes each join phase at the granularity of individual partitions, while the latter approach uses parallelization at the granularity of the total number of partitions, i.e., table granularity. We explain the mechanism of each approach below in detail.

Approach 1

In the first approach, we use the idea of applying parallelization to the individual join phase, i.e., join phase of every R_i and S_i pair. We depict this approach in Listing 4.7. Thus, once the hash table is built for R_i, we divide the entire S_i among the available worker threads, i.e., we assign each thread a different subset of S_i and then perform the probing separately in a parallel fashion (lines *5-8*). Once all worker threads are done with their individual probing, we need to combine the probe results (matching RIDs) relative to them. For this, we use the prefix sum technique and determine the respective memory locations where each thread writes its result in the result array (lines *11-14*) . Finally, we copy the results of individual threads to these memory addresses in a parallel fashion (lines *16-21*).

Listing 4.7: Join phase – parallel implementation (first approach)

```
1
2  //R_results , S_results − thread−relative probe results
3  //result_size         − probe result size for each thread
4
5  for(unsigned int i=0; i<num_threads; i++)
6  {
7          do parallel : serial_probe(S_partition ,R_partition);
8  }
9
10 // prefix_sum[0] = loc ; //loc −last index in result array obtained from previous join phase
11 for(unsigned int j=0; j<num_threads + 1; j++)
12 {
13          prefix_sum[j] = prefix_sum[j −1] + result_size[j −1];
14 }
15
16 for(unsigned int i=0; i<num_threads; i++)
17 {
18         do parallel :
19         memcpy(&R_tids[prefix_sum[i]],&R_results[], result_size[i]*sizeof(TID));
20         memcpy(&S_tids[prefix_sum[i]],&S_results[], result_size[i]*sizeof(TID));
21 }
```

The main disadvantage of this method is the sharing of hash tables built for R_i partitions since each thread requires the entire hash table to find matches for their assigned S_i portions. Thus, the cache memories and consequently the TLBs relative to the hash table need to be shared among all threads. We already discussed the disadvantages of shared cache memories such as race conditions and cache-line bouncing in Chapter 2. Further, the number of tuples in each partition S_i is very small compared to the total number of tuples in the entire relation S. Therefore, in every join phase, each thread performs probing only for a limited number of tuples. Broneske et al. mentioned that assigning threads to small jobs of this kind degrades the benefits of parallelization [BBS14].

Approach 2

We know that all partition pairs obtained as a result of radix partitioning are independent, i.e., we perform the build and probe individually for each partition pair instead of the overall relation. Thus, in this method, we use the idea of parallelizing the overall join phase instead of individual joins, i.e., we perform a serial joining of all partition pairs in a parallel fashion. We sketch an overall idea of this approach in Figure 4.3.

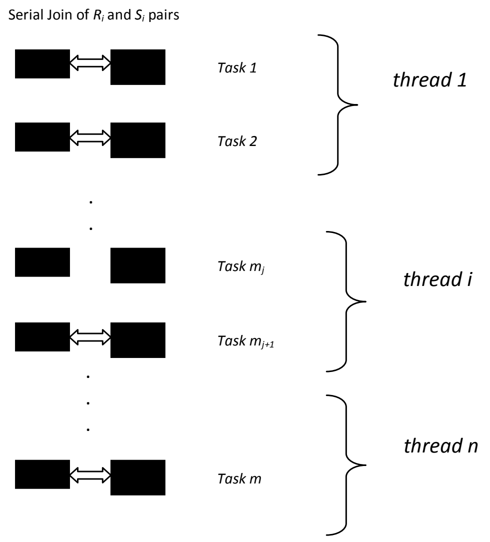

Figure 4.3: Parallel join via task creation with n CPU core threads

Listing 4.8: Join phase – parallel implementation (second approach)

```

//R_results, S_results  - thread-relative probe results
//result_size           - probe result size for each thread

//After radix partitioning is completed
if(R_partition_count >0 && S_partition_count >0)
{
        task(R_partition, S_partition); //insert the partitions into task
        task_count++;
}


//evenly allocate tasks among threads (task_count/num_threads)


for(unsigned int i=0; i<num_threads; i++)
{
        //each thread performs individual join phase for all partition pairs in its task
        do parallel : serial_bucket_chain(task);
}

// prefix_sum[0] = 0 ;
for(unsigned int j=0; j<num_threads + 1; j++)
{
        prefix_sum[j] = prefix_sum[j-1] + result_size[j-1];
}

for(unsigned int i=0; i<num_threads; i++)
{
        do parallel :
        memcpy(&R_tids[prefix_sum[i]],&R_results[], result_size[i]*sizeof(TID));
        memcpy(&S_tids[prefix_sum[i]],&S_results[], result_size[i]*sizeof(TID));
}
```

In Listing 4.8 on the preceding page, we show an overview of this approach. Thus, in this approach, we create a set of tasks, i.e., if we have m partition pairs where the tuple count of both partitions in each pair exceeds zero, then we create m tasks with each task holding a separate partition pair (lines *6-9*). We then divide these m tasks among the available worker threads. In this way, each thread t_j performs a serial join phase (build and probe) for every partition pair contained in the set of tasks assigned to them (lines *16-20*). Thus, each thread builds a separate hash table for probing each partition pair R_i and S_i. Therefore, we avoid the problem of shared cache memories and TLBs faced in the previous approach. Further, the load (number of tuples for probe) allocated to each thread is very high compared to the previous approach. Once all worker threads complete their individual serial join, we combine their probe results using a prefix sum technique as before (lines *22-33*).

4.3.2.3 Branch-Free Code Implementation

Listing 4.9: Join phase – branch-free implementation

```
 1
 2  //probe
 3  for(unsigned int i=0; i <num_S_partition; i++)
 4  {
 5          bucket_number = Hash(S_partition[i]);
 6          for(...) // probe all R keys with same bucket number
 7          {
 8                  // id - partition RID referenced by the bucket number
 9
10                  // we store the RIDs of matching keys
11                  R_tids[pos] = R_pos_partiton[id];
12                  S_tids[pos] = S_pos_partition[i];
13                  pos +=(R_partition[id] == S_partition[i]);
14          }
15  }
```

We know that in each join phase, we perform the build and probe for any partition pair R_i and S_i in a sequential manner. Accordingly, once the hash table is built for R_i, we start the probing operation between R_i and S_i sent to the join phase. Such probing is done by calculating the hash value for each tuple in S_i and then performing a scan over all keys with the same hash value in the hash table built for R_i. Thus, in our radix join algorithm, we have a conditional branch instruction at the point where the join performs this scan for the join predicate (*equality*). For converting this conditional branch implementation (lines *21-27* in Listing 4.2 on page 41) into a branch-free implementation, we adopt the strategy proposed by Broneske et al. for a simple database scan [BBS14]. Thus, we remove the serial join with conditional branch probing to a branch-free implementation at the cost of executing a comparatively large number of instructions as in Listing 4.9 (lines *6-14*). We already mentioned the advantages of these branch-less implementations in Chapter 2.

4.4 Combination of Code Optimizations

Our experimental evaluation, which we will discuss in the next chapter, confirm that in most of the cases, the branch-free implementation for the join phase does not provide an optimal solution over the join phase of the serial radix join. On the contrary, the parallel version offer significant benefits compared to all the other versions. Therefore, while applying the combination of optimization techniques to the serial radix join in our work, we consider only the combination of vectorization and paralelliztion techniques. We explain our approach of applying them to individual phases of the radix join as below.

4.4.1 Partition Phase – Parallelization + Vectorization

The approach that we use for partitioning the input relations is straightforward, i.e., similar to our approach used in parallel partitioning, we set one thread per core for parallelizing the whole phase. Each thread in turn uses SIMD instructions for performing the computation of partition histograms, followed by the storage of partition tuples. Further, we use two-dimensional arrays for both partition tuple counts as well as partition memory allocations.

4.4.2 Join Phase – Parallelization + Vectorization

We mentioned earlier in our parallel join phase (second approach) that we set each thread to perform the serial join for every partition pair allocated to them in a parallel fashion. Thus, for our combination of parallelization and vectorization techniques, we set each thread to perform the vectorized join simultaneously. Once all worker threads complete their probe activities, we store their individual results using the same prefix sum technique as in a parallel join phase. For the vectorized join phase, we use our second approach discussed under Section 4.3.2.1 on page 46.

4.5 Summary

To summarize, in this chapter, we first presented the framework and the data structures that we used for implementing all our code optimization techniques. Then, we explained how we modified the serial radix join adopted from Teubner et al. [BTAÖ13] to process only *32*-bit key values instead of *64*-bit tuples to suit our column-oriented database (CoGaDB). Further, we also explained how we stored the individual join results instead of observing the number of matches obtained from the join. Once we finalized our serial radix join version, we explained all our approaches of handling the discussed code optimization techniques for this version. We also showed several approaches of handling them and also mentioned the advantages and disadvantages for various approaches. Finally, we discussed the combination of these optimization techniques that we handled for the scalar radix join.

In the next chapter, we will discuss how we evaluated our implementation techniques and also present the results of our evaluation and analysis.

5. Evaluation

In the previous chapter, we presented all our radix join variants and explained our approaches to implement each of them. In this chapter, we show how we evaluated each of those variants using a common experimental validation. This chapter is divided into four sections. In Section 5.1, we present our setup for evaluating our radix join variants followed by a preliminary evaluation of the serial radix join adopted from Teubner et al. [BTAÖ13] in Section 5.2. In Section 5.3, we project the results expected for the performance behavior of each optimized variant relative to the scalar radix join. The predictions that we present in this section are based on our literature survey.

In Section 5.4, we discuss the actual behavior of all our variants with respect to the workload and the performance metric that we used for our evaluation. Finally, in Section 5.5, we give a brief summary of our observation about the performance behavior of all our optimized radix join variants.

5.1 Evaluation Setup

For comparing the performance behavior of the optimized radix join implementations, we use workload comprising two different variables (datasets and selectivity ratios) that are more relevant for in-memory database processing. Further, our workload mimic the ones that were already used in the literature for analyzing the performance of several hash join variants. For testing all our radix join variants, we use two different machines with enough main-memory to hold and process the given workload. On each machine, we evaluate the performance of our radix join variants by recording their time taken to process a given input workload in milliseconds *(ms)*. We explain our evaluation setup in detail as below.

Dataset

The first variable of our workload consists of several datasets as shown in Table 5.1 on the following page. All our datasets consist of in-memory database columns with varying cardinalities in the order of MBs *(megabytes)*. As mentioned in the previous chapter, we consider only tuple keys for our join processing of input relations and these keys are represented using *32*-bit integer attributes. The relation R (smaller relation) consists of only unique keys whereas the relation S (bigger relation) consists of a uniform distribution of keys contained in R.

S.No	**Absolute Cardinality Dataset (set 1)**	**Relative Cardinality Dataset (set 2)**
1	8MB:128MB	64MB:128MB 128MB:128MB
2	16MB:256MB	64MB:256MB 128MB:256MB 256MB:256MB
3	32MB:512MB	128MB:512MB 256MB:512MB 512MB:512MB

Table 5.1: Workload – datasets

Our datasets can be classified into two broad categories - *absolute cardinality dataset* and *relative cardinality dataset*, according to those used by Blanas et al. and Kim et al. [KKL+09, BLP11]. In the first category, we use three different datasets whose relation sizes are fixed and the difference in cardinalities of input relations R and S is significant with a ratio of *1:16*. Accordingly, the datasets belonging to this category have a cardinality of *8MB:128MB*, *16MB:256MB* and *32MB:512MB* for relations R and S respectively. We use the formula *1MB = 1,048,576* bytes for filling our in-memory keys relative to these cardinalities. We refer to this category of datasets as set *1* in the remainder of this chapter.

In the second category, for each size of the relation S used in set *1 (128MB, 256MB and 512MB)*, we increase the size of the relation R in a linear fashion. This dataset helps us to study the performance of our optimized variants with respect to scalability when the relative size of the relation R is increased. As we alter the relative size of the relation R, we also adjust the distribution of the key values in relation S. We refer to this category of datasets as set *2* in the remainder of this chapter.

Selectivity

The second variable of our workload is the join selectivity factor. We know that the join selectivity between two input relations R and S is given by $card(R \bowtie S)/card(R * S)$, where $card(R \bowtie S)$ refers to the number of joining tuples and $card(R * S)$ refers to the number of tuples obtained as a result of cross multiplication between R and S. However, for our experiments, we use a selectivity factor that is more relevant in the context of *online analytical processing (OLAP)* as proposed by Blanas et al. and Kim et al. [KKL+09, BLP11]. Accordingly, the selectivity factor here refers to the fraction of tuples in S that matches with those in R. For each dataset in our workload, we test our radix join variants for different join selectivities in the range of *0-100%*. We vary each selectivity ratio in steps of *10%*.

Thus, we have a perfect referential integrity constraint between R and S only for a full selectivity *(100%)*, where each tuple in S is guaranteed to find exactly one join partner in R. For selectivity factors less than *100%*, we still maintain a uniform distribution of the key values in S. For example, for a selectivity of *50%*, we fill half of the relation S with a uniform distribution of first *50%* of the key values contained in R, whereas for the remaining *50%* of S, we uniformly distribute them with keys outside the range of the key values contained in R. Thus, for selectivities less than *100%*, we select only the same fraction from R since we want to maintain a reasonable uniform distribution when the size of R approaches that of S for our datasets in set *2*. We perform the selectivity test for every dataset shown in Table 5.1 on the preceding page.

Machines

We test our implementations on two different machines, with the first machine being an Intel Core i5-2500 with a Sandy Bridge architecture and the second machine being an Intel Xeon E5-2609 v2 with an Ivy Bridge architecture. We refer to these two machines as machine *1* and machine *2* throughout this chapter. Both machines are installed with Ubuntu *10.04.3* operating system and on each machine, we use GNU C++ compiler *4.6.4* for compiling our radix join variants. The machine *1* is provided with a single socket with four cores, whereas the machine *2* is provided with two sockets, with each socket supporting four cores. Furthermore, each core on both machines is provided with a single thread and thus, we have four threads on machine *1* and eight threads on machine *2*. Both machines support a three level hierarchical cache memory with shared *L2* and *L3* caches. A detailed description about the cache memory capacities of these two machines can be found in Table 5.2.

Specifications	**Machine 1**	**Machine 2**
Architecture	Sandy Bridge	Ivy Bridge
CPU	Intel Core i5-2500	Intel Xeon E5-2609 v2
CPU Frequency	3.3 GHz	2.5 GHz
Number of sockets	1	2
Number of cores per socket	4	4
Number of thread per core	1	1
L1-Cache	256 Kb	256 Kb
L2-Cache (Shared)	1 Mb	1 Mb
L3-Cache (Shared)	6 Mb	10 Mb
Cache-Line Size	64 bytes	64 bytes
TLB capacity	64 entries	64 entries

Table 5.2: Specifications for used machines

Performance Metric

We evaluate the performance of each radix join variant by using the metric - *response time in milliseconds (ms).* For more clarity, we measure three different response times as below:

- Time taken to complete the partition phase *(in milliseconds).*
- Time taken to complete the join phase *(in milliseconds).*
- Time taken to complete the overall radix join processing *(in milliseconds).*

By doing this, for each dataset and for different selectivity factors, we can easily analyze the individual phase that has the major impact on the behavior of the overall join processing of each radix join variant.

5.2 Preliminary Evaluation for Serial Radix Join

Before evaluating our optimized radix join variants, we first check whether the original scalar implementation that we adopted from Teubner et al. performs according to the previous work of Teubner et al. and Blanas et al. [BTAÖ13, BLP11]. We present our observation for each phase in the following subsections. For each implementation (serial and other variants), we repeated our testing for *50* times and discarded the slowest and the fastest *5* results according to the time taken for the overall radix join processing. On both machines (*1* and *2*), we set the number of radix bits to *14*.

Partition Phase

We present our evaluation results for the serial partitioning corresponding to the dataset *8MB : 128MB* in Figure 5.1 on the next page. As can be seen in this figure, the partitioning behavior varies across both machines. We explain them one by one below.

Machine 1 - On machine *1*, the partition phase nearly exhibits a constant performance across the entire range of selectivity factors *0-100%* with a very little variation. This is because, regardless of the selectivity factor, the radix partitioning is performed for the entire amount of input keys contained in input relations R and S. Thus, the variations observed in the partitioning behavior occur due to the nature of the data distribution accompanied by random memory accesses [KKL+09, BLP11]. It should be noted that, in addition to the tuple keys, we also store their respective Row-IDs (RIDs) in separate set of partitions. Therefore, we expect these variations to become worse especially for extremely high cardinalities of input relations.

Machine 2 - On machine *2*, we observe a different kind of partitioning behavior especially at the selectivity factors - *0%*, *50%* and *100%*, i.e., the variations in the partitioning behavior corresponding to these selectivities gradually became worse, whereas for

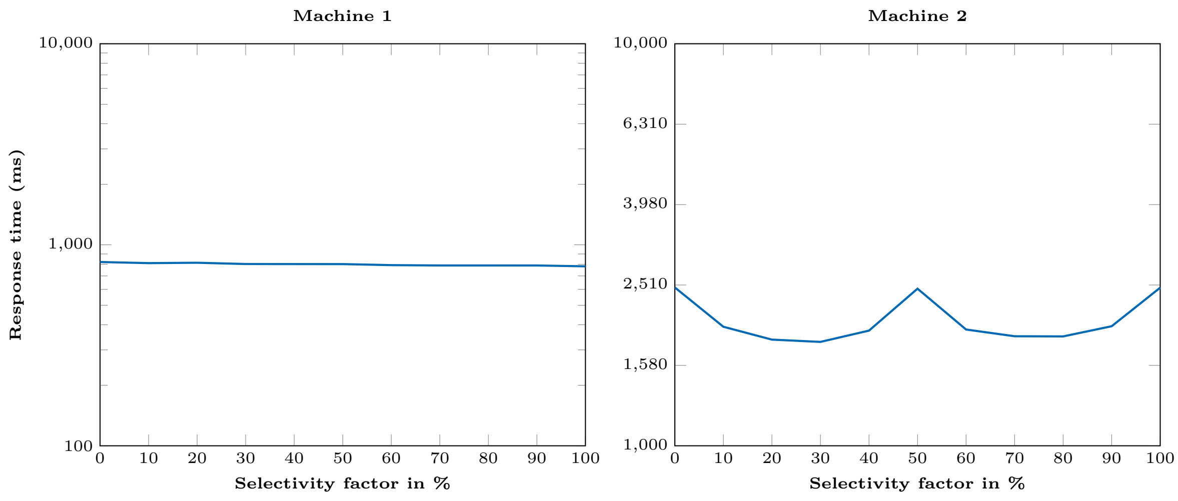

Figure 5.1: Partition phase – serial radix join (for dataset - 8MB : 128MB)

other selectivity factors, it exhibits a little variation. We notice the same result for all our datasets in both set *1* and set *2*. This is because the random memory access pattern normally shows different trends across different architectures according to the amount of TLB and cache misses encountered during partitioning [BLP11, Man02, KKL+09]. Accordingly, we found that the cache miss quantities for these selectivity factors *(0%, 50% and 100%)* are higher compared to other join selectivities. For instance, for the dataset *8MB : 128MB*, the number of *L3* cache misses for these selectivity ratios are in the order *1.8 ** 10^7 - *2.2 ** 10^7, whereas for other selctivity factors, it is always in the order *1.6 ** 10^7. However, for machine *1* with a smaller sized *L3* cache memory, the quantity of cache misses are stable across all selectivity ratios and are always in the order *2.2 ** 10^7. Also, it can be noted that the machine *2* generally consumes more time to complete the partitioning than the machine *1* due to machine *2*'s lower CPU clock frequency compared to machine *1*.

Reasons for Cache Miss Variations across Machine 1 and Machine 2

We explain the reason for such varying and stabilized cache misses for machines *1* and *2* respectively as below.

Our analysis reveal that, for machine *2*, at the selectivity factors - *0%*, *50%* and *100%*, increased cache misses occur due to the nature of the data distribution corresponding to these selectivity factors for the relation *S*. This means depending on the data distribution, certain partitions are updated with very less frequencies. For example, for a selectivity of *100%*, if a partition P_t is updated for a key value at a certain time period, the frequency for its successive updates is very low such that it goes out of the *L3* cache memory since we need to pull other tuple keys (along with their RIDs) and their corresponding partitions into this cache memory for performing their own partition updates. Therefore, the number of *L3* cache misses for partitions of type P_t are very high compared to other partitions that are updated with comparatively high frequencies.

However, for selectivities less than *100%* such as *90%* where we replace the last *10%* of S with keys that are not in R, we do not encounter the same kind of problem and hence the total cache miss quantities are reduced. We observe the same behavior for the selectivity factor of *0%*, where we entirely fill S with uniform keys that are not in R. Thus, we can conclude that we encounter such problems because of the nature of our data distribution corresponding to the last *10%* of S. Also, we notice that all the partition sizes for these two selectivity factors matches exactly with one another which further confirms our analysis. Similarly, for the selectivity factor of *50%*, we encounter a similar kind of problem due to the data distribution nature between the uniformly distributed keys of R contained in *40-50%* of S and the other set of keys (out of range of R) contained in *50-60%* of S.

However, we do not encounter these type of varying cache misses at different selectivity factors for machine *1*. This is because, the size of their *L3* cache memory is comparatively smaller than that of machine *2* and therefore, in addition to the high cache miss quantities corresponding to low frequency partitions of type P_t and cache misses corresponding to the actual relation keys, their cache misses are comparatively higher even for other partitions that are updated more frequently. Thus, the total number of *L3* cache misses are higher and more stabilized on machine *1* than on machine *2*.

We notice that the partitioning behavior on machine *2* concur with the argument of Blanas et al. for a hash join, i.e., on some multi-core architectures, when only a few number of cores (one core for a serial implementation) is set to perform the whole hash join processing, keeping the remaining available cores idle, the algorithm behaves in a different manner than the expected performance [BLP11]. Accordingly, on this machine, we observe that when we use all available cores for our parallelization techniques, the variations in the partitioning behavior at different selectivity factors are comparatively reduced and especially for very high cardinalities of R and S, they nearly achieve a constant performance similar to machine *1*. Moreover, for very high table sizes of R and S, both machines nearly consumed the same time for the serial partitioning, i.e., the performance of a serial radix partitioning on machine *2* approaches the performance on machine *1* for increasing cardinalities of R and S. We will discuss this in detail in Section 5.4.1 and Section 5.4.2.

Join Phase

We present our evaluation results for the overall join phase on machines *1* and *2* corresponding to the dataset *8MB : 128MB* in Figure 5.2 on the facing page. As seen in this figure, both machines exhibit nearly the same behavior pattern across the entire range of join selectivity. Similar to the partitioning phase, we observe that the machine *2* generally consumes more time to complete the join phase compared to the machine *1*. We explain the performance in build and probe phases separately as below.

Build - Similar to the partitioning phase, the time required to complete the building of hash tables for all R_i partitions during the join phase is generally independent of the selectivity factor. This is because, during each individual build, we store only the array position indices of the input keys of R_i to their respective buckets and the cost for this process generally depends upon the number of tuples in such partitions [BTAÖ13]. Since we have uniformly distributed primary keys in R, the size of every R_i partition is equal and thus, the time consumed for all individual build phases remains nearly the same. Generally, the time for every build is very less when compared to the individual probing of partition pairs [BTAÖ13, BLP11] .

Probe - The probe is the only phase that is significantly affected by the join selectivity factor. In general, for in-memory hash join operations, the time required to complete the probing across all partition pairs increases with increasing selectivity factors. This is because, as the join selectivity increases, the number of instructions to be executed also increases [BLP11]. Moreover, it should be noted that the probing between any partition pair R_i and S_i can be viewed as a simple scan of each key in S_i over all keys stored in R_i buckets with the same hash value. However, for each key in S_i, at most only one match is guaranteed from R_i (due to the storage of primary keys inR_i). Hence, the prediction expected from a branching instruction in the probe phase in most of the cases will always be a non-match. For lower selectivity factors, such predictions perform well since we have very few matches (join partners) whereas for higher selectivity factors, it does not give reasonable solutions. Therefore, as we increase the selectivity factor, the prediction accuracy generally degrades which in turn increases the response times. On many multi-core architectures, a strong linear correlation exists in the behavior of the overall performance of the join phase [KKL+09, BLP11].

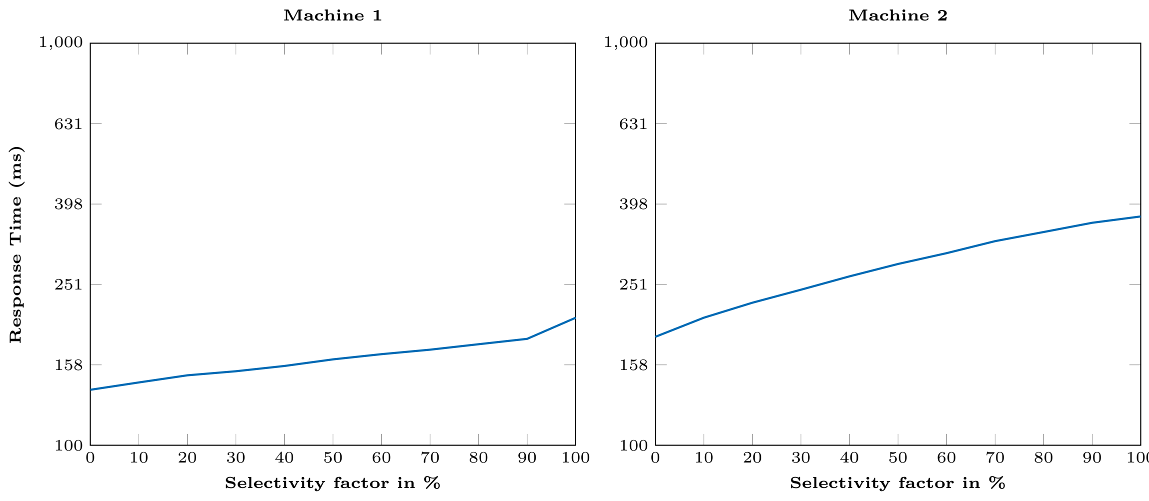

Figure 5.2: Join phase – serial radix join (for dataset - 8MB : 128MB)

As we discussed in Chapter 3 and Chapter 4, as the partition count increases, their respective sizes are refined such that the resulting partitions fit well in the cache memory. In spite of storing both keys and their corresponding RIDs in our implementations, even for our largest cardinality set *(512MB : 512MB)*, all partitions *(16384)* produced with *14* radix bits fit very well in cache memories (data caches) of the used machines. Thus, we expect our results of the serial join phase to nearly correlate with the results in the literature for all our datasets mentioned in Table 5.1 on page 54.

Overall Performance

As we observe from Figure 5.1 on page 57 and Figure 5.2 on the previous page, it is clear that the partitioning phase plays a dominant role in the overall time required to complete the radix join processing. From the evaluation of our scalar radix join implementation, we observe that this phase comprises approximately *75-90%* of the total time across various selectivity factors. Therefore, we can conclude that the overall performance of the radix join processing strongly depends upon the time spent on the partitioning of input relations. We show our results of the overall response time consumed by both machines for the dataset *8MB : 128MB* in Figure 5.3.

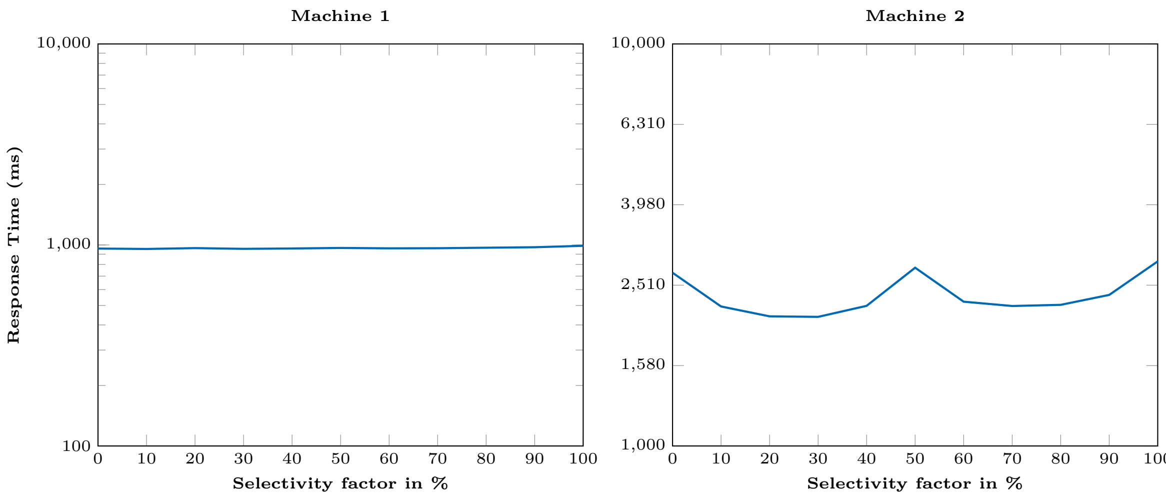

Figure 5.3: Overall performance – serial radix join (for dataset - 8MB : 128MB)

In this section, we evaluated the scalar implementation of the radix join. Since our evaluation results for the scalar radix join concur with those results presented in literatures [KKL$^+$09, BLP11], we now have a good starting point for further evaluating our optimized radix join variants and comparing their performance behavior with that of the scalar radix join in both the used machines.

5.3 Projected Results for Optimized Variants

Before presenting the actual evaluation results for our optimized radix join variants, we first present our expectation about their performance behavior in relation to the serial radix join in the following subsections. We also strengthen our expectations with facts from various literatures.

Vectorization

Kim et al. argue that the use of scatter and gather operations in the join phase limits the exploitation of SIMD architectures for any general hash join technique [KKL+09]. We already discussed in the previous chapter that we avoided such operations for our vectorized variants. Instead, we combined loop unrolling of depth *4* (for *32*-bit integers) along with SIMD instructions in both the partition phase and the join phase. However, we still expect these variants not to completely outperform the serial version in all situations due to a large number of conflicting instruction overheads that occur with such combinations.

Parallelization

As discussed in the previous chapter, for our parallel radix join implementation, we followed the approach of using at most only one thread per core. Thus, we avoided the use of *simultaneous multi-threading (SMT)* and therefore, we do no encounter the problems that arise from sharing of hardware resources among threads belonging to the same core. Further, for any hash join technique, a significant amount of scalability benefits are realized using such implementations [KKL+09, BATz13]. Therefore, we expect our parallel implementation to perform extremely well when compared to the scalar version for all our used datasets.

Branch-Free Code Technique

There is not enough evidence in the literature about the behavior of radix join techniques when optimized with branch-free implementations. As we mentioned in the previous section, once we have all partition pairs for input relations R and S, the probing between each partition pair R_i and S_i can be viewed as a simple scan of each key in S_i over all keys with the same hash value in the hash table built for R_i. It can be seen that even for a 100% join selectivity, every key in S_i is guaranteed to find only one match in R_i since the R_i is filled only with primary keys. Therefore, we can conclude that the scan selectivity of each key in S_i will always be *0/n* or *1/n*, where '*n*' is the total number of R_i keys with the same hash value that are chained together. Thus, the scan selectivity of each key in S_i will always be around 0%. Broneske et al. confirmed that for a simple scan operation, the branch-free implementation performs worse than the serial scan when the branch probability is very low [BBS14]. Thus, we expect that for any join selectivity factor, the branch-free implementation does not give an optimal performance for the join phase. Since we use the same serial partitioning for the branch-free technique, we expect the scalar radix join to completely outperform the branch-free radix join in all situations due to the worse performance that would arise from a branch-free probing.

5.4 Evaluation of Optimized Variants

Now we discuss the actual performance behavior of all our optimized radix join variants in comparison to the serial radix join. For the sake of simplicity, in the remainder of this chapter, we use the keyword *serial variant* to refer to the scalar radix join. For radix join implementations optimized with vectorization, parallelization and branch-free techniques, we denote them using keywords *vectorized variant*, *parallel variant* and *branch-free variant* respectively. Further, we use the keyword *P+V variant* to refer to the implementation optimized with a combination of both vectorization and parallelization techniques.

This section is further divided into three subsections. In Section 5.4.1 and Section 5.4.2, we discuss the performance of each optimized variant with respect to partition and join phases. Based on our discussions in these two sections, we summarize the overall performance of each optimized variant in relation to the serial variant in Section 5.4.3.

5.4.1 Partition Phase

In this section, we present our evaluation results for the partition phase of each radix join variant on both machine *1* and machine *2*. For each machine, we discuss the performance of each optimized variant with respect to all datasets used for our workload (set *1* and set *2*). Since we use the standard serial partitioning for the branch-free variant, we discuss only the partitioning behavior for vectorized, parallel and P+V variants in relation to the serial variant.

Machine 1

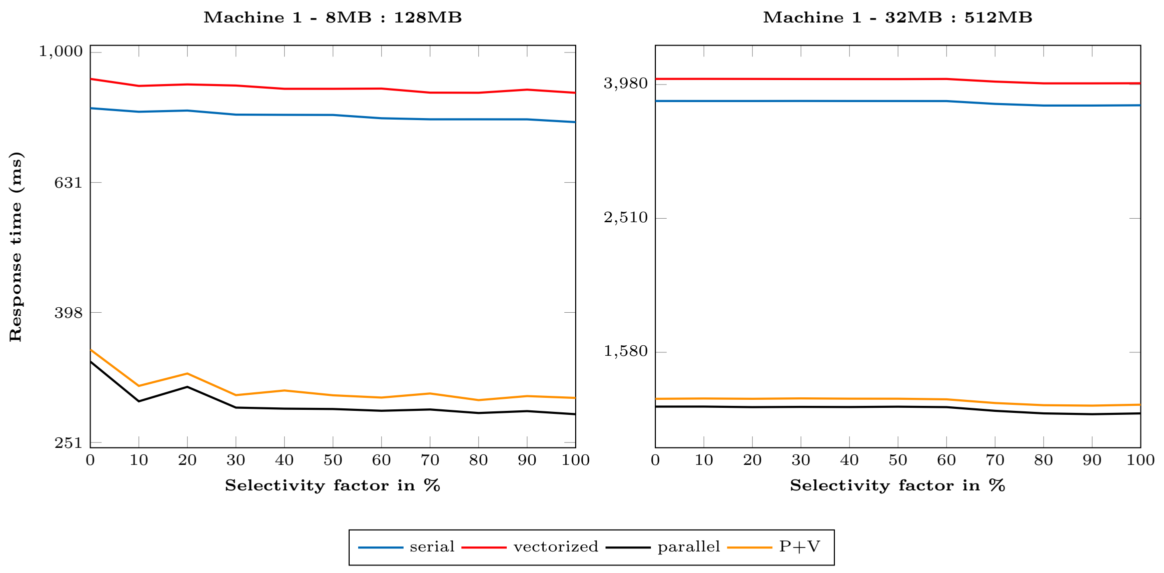

Figure 5.4: Partition phase – machine 1 (for datasets in set 1)

Set *1* - In Figure 5.4, we present our evaluation results for the partitioning behavior of all radix join variants on machine *1*. These results correspond to our evaluation for the absolute cardinality dataset (set *1*). From this figure, it is clear that both parallel and P+V variants perform significantly better than the serial variant, whereas the vectorized

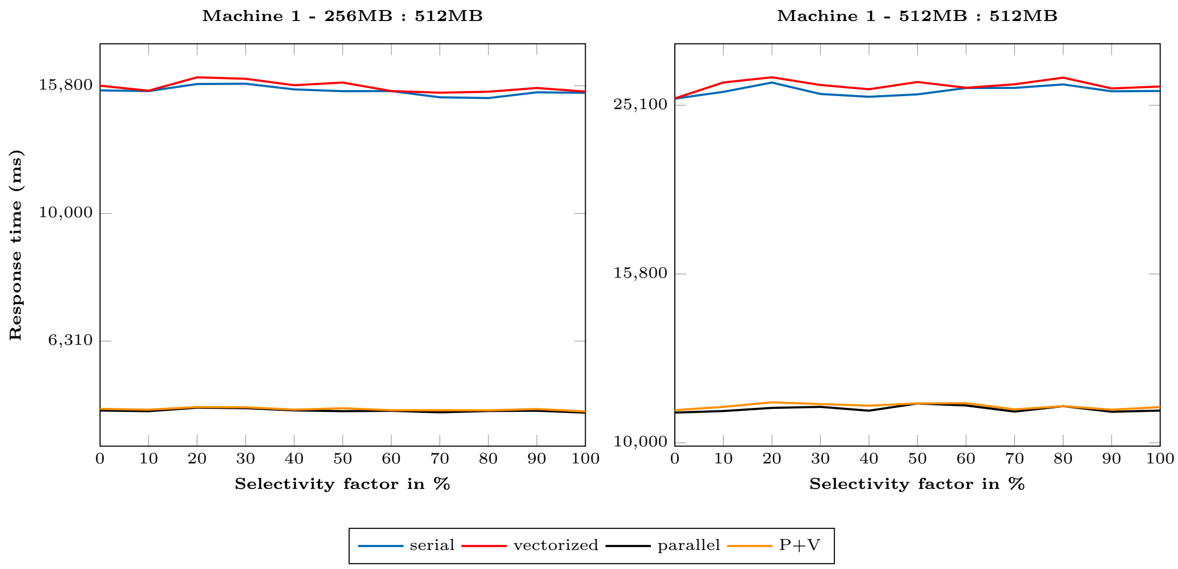

Figure 5.5: Partition phase – machine 1 (for datasets in set 2)

variant offers no improvement over the serial partitioning. Among parallel and P+V variants, we found that the parallel partitioning offers a slight advantage over the P+V partitioning over the entire range of selectivity factors. We observe the same behavior for all our datasets in set *1*.

Set *2* - In Figure 5.5, we present our results for the partitioning behavior on machine *1* for the relative cardinality dataset (set *2*), where we increase only the size of the relation *R* for each size of *S* used in set *1*. We present our results only for datasets - *256MB : 512MB* and *512MB : 512MB*, since we observe almost the same result as the size of *R* approaches towards its respective size of *S*. The results indicate that there is no change in the partitioning behavior, i.e., the optimized variants exhibit a behavior similar to their performance for set *1*, with the parallel partitioning once again providing the best performance over the serial partitioning.

Machine 2

Set *1* - On this machine, the results are nearly similar to machine *1* for all datasets belonging to set *1* with the only notable difference coming in the performance behavior between parallel and P+V variants. Here, only for our largest dataset *32MB : 512MB*, we observe that the P+V variant slightly outperforms the parallel variant at some points (corresponding to the selectivity factors *50%* and *60%*). We present our evaluation results for datasets - *8MB : 128MB* and *32MB : 512MB* in Figure 5.6 on the following page. It can be noticed that the variations observed in a serial partitioning are comparatively reduced when we use all available cores for our parallelization techniques.

Set *2* - Figure 5.7 on page 65 shows our evaluation results for datasets in set *2* on machine *2*. From this figure, we observe that the parallel variant which is sometimes

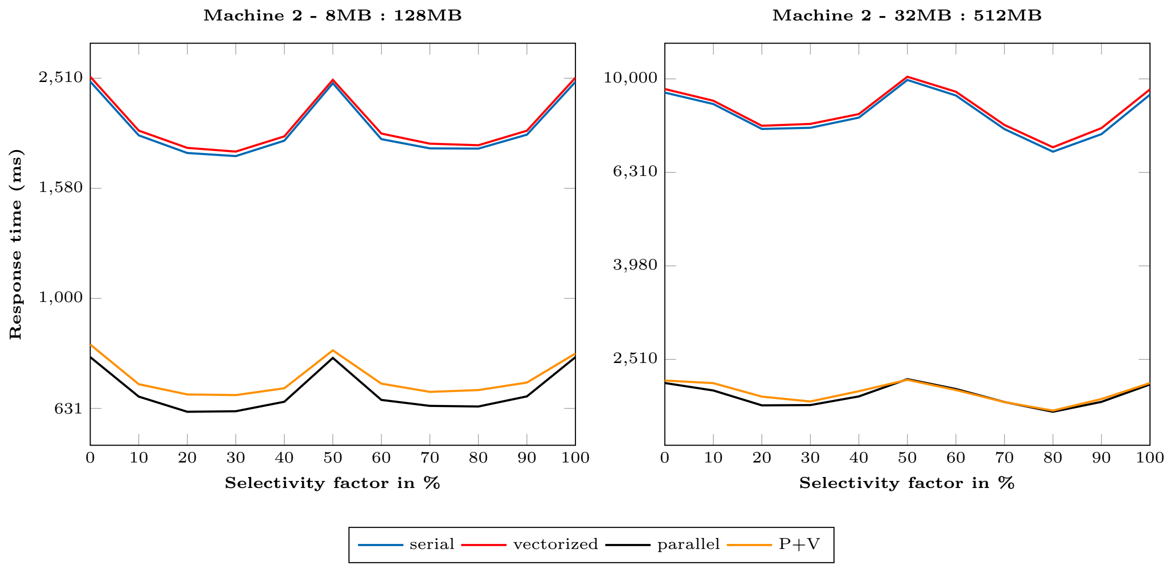

Figure 5.6: Partition phase – machine 2 (for datasets in set 1)

outperformed by the P+V variant (for the dataset *32MB : 512MB*) improves as we increase the relative size of *R*. Thus, similar to their performance on machine *1*, the parallel variant offers a comparatively better partitioning performance than the P+V variant. Moreover, the variations observed in the serial partitioning are even more reduced when parallelized and both parallel and P+V variants exhibit nearly a constant partitioning behavior across the entire range of selectivity factors. Also, we notice that for our datasets *256MB : 512MB* and *512MB : 512MB*, the serial variant on this machine consumed nearly the same time as on machine *1*. Similarly, the parallel variant and consequently the P+V variant on this machine (with *8* cores) consumed less time than on machine *1* (with *4* cores). Thus, we can conclude that the performance of serial and parallel radix join variants on machine *2* matches with their performance on machine *1* at increasing cardinality rate of relations *R* and *S*. In addition, we observe that compared to the serial variant, the performance of the vectorized variant on this machine gets worse than on machine *1* for increasing sizes of *R*.

To summarize the partitioning behavior of each radix join variant, we conclude that the parallel variant offers the best overall performance over the serial partitioning, whereas the vectorized variant offers the worst behavior among all the implemented radix join variants. We already mentioned in Section 5.2, that the partition phase has the major impact on the overall response time of a serial radix join implementation. From our observation, we found that this is applicable even for their respective optimized variants although the amount of time that a partition phase comprises for a serial radix join *(75-90%)* is not exactly the same for their respective optimized variants (especially parallel variant and P+V variant). Therefore, we expect those variants that perform well in the partition phase tend to become the best overall radix join variant.

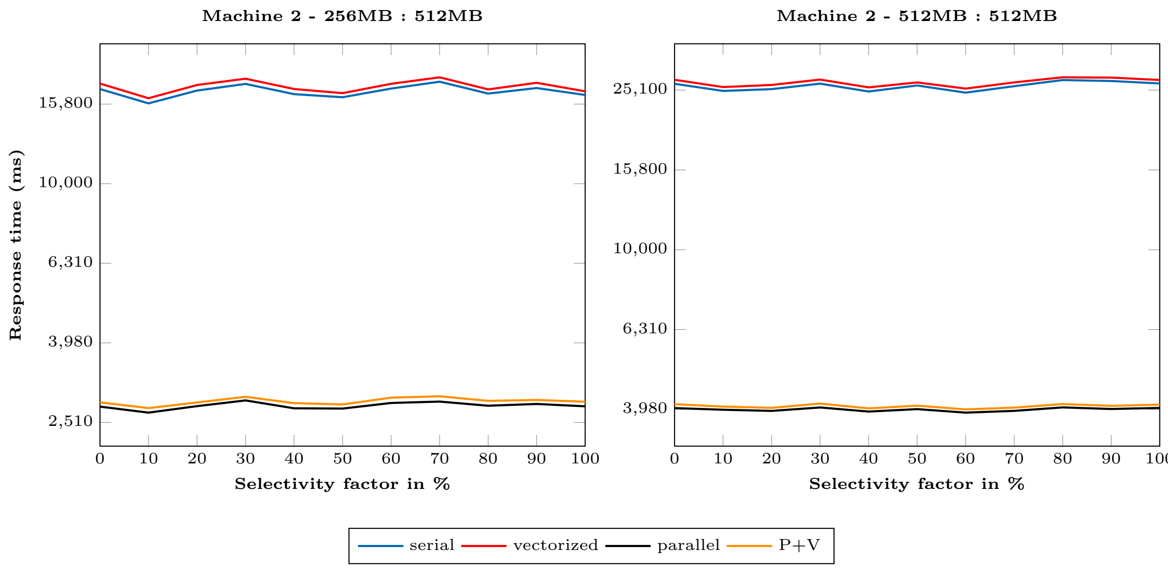

Figure 5.7: Partition phase – machine 2 (for datasets in set 2)

5.4.2 Join Phase

In this section, we discuss the performance behavior of each radix join variant in the join phase (build and probe). Since the probe is the only phase that is significantly affected by the join selectivity, for the sake of more clarity, we explain the behavior of each optimized radix join variant in relation to the serial variant separately respective to each machine in the following subsections. At the end of this section, we will discuss how the join phase performance of each optimized variant on both machines nearly correlate with each other for increasing sizes of R and S.

Machine 1

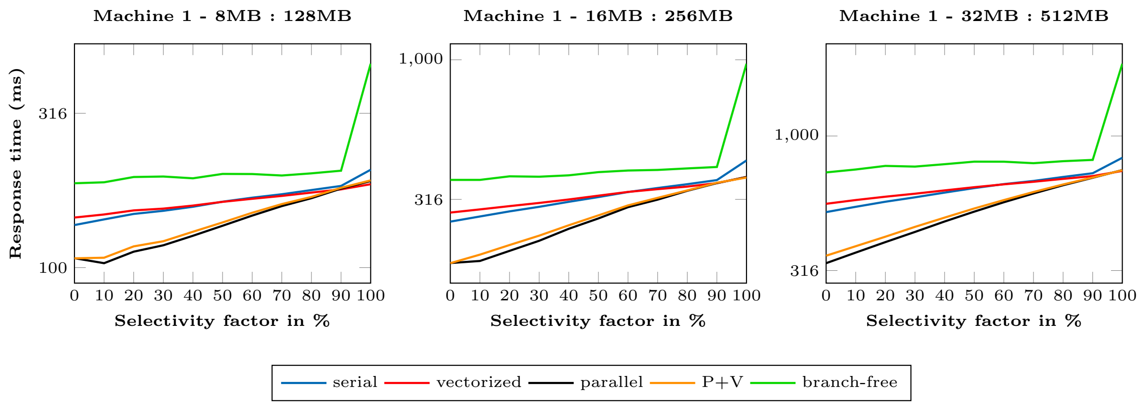

Figure 5.8: Join phase – machine 1 (for datasets in set 1)

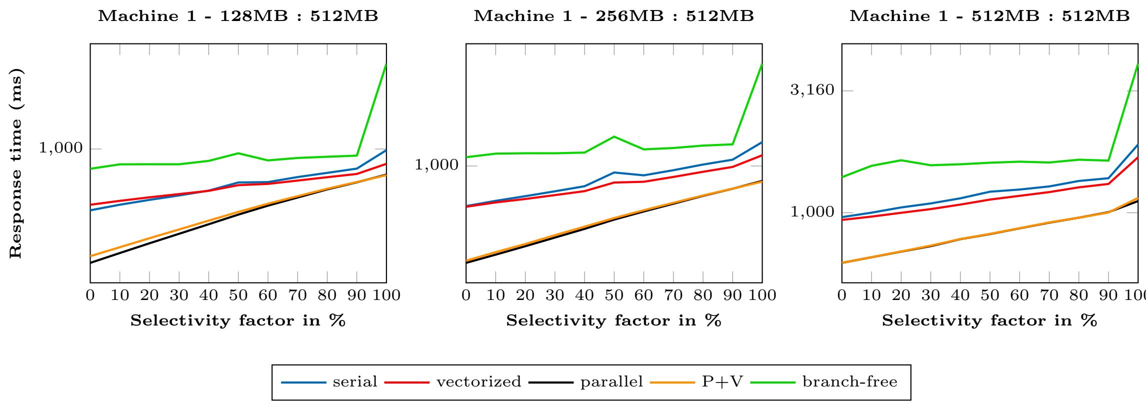

Figure 5.9: Join phase – machine 1 (for datasets in set 2)

In Figure 5.8 and Figure 5.9, we present the results for all radix join variants in the join phase. We summarize the performance of each optimized variant as below.

Parallel and P+V Variants - Similar to the partition phase, the join phase of both parallel and P+V variants outperform the serial variant for all our datasets in set *1*. Further, it can be seen from Figure 5.8 that, as we increase the cardinalities of both relations R and S in set *1*, the difference in response times between the serial variant and that of parallel and P+V variants also increases. Thus, from Figure 5.9, we can observe that when we increase the relative size of R in set *2*, both parallel and P+V variants become extremely efficient among all the other optimized variants. Among parallel and P+V variants, we observe that for both set *1* and set *2*, the parallel variant provides the best performance except at higher selectivity factors such as *90%* and *100%* since the P+V variant offers a slight advantage at these selectivities.

Vectorized Variant - From Figure 5.8, we can see that for all our datasets in set *1*, the vectorized variant offers advantage over the serial variant at selectivity factors higher than *60%*. Further, it also provides a slight advantage over both parallel and P+V variants at higher selectivities factors such as *90%* and *100%*. But, as we increase the relative size of R in set *2*, it cannot compete with the extremely efficient performance of parallel and P+V join variants at any selectivity factor for datasets belonging to this set. However, its advantage over the serial variant continues to improve and for datasets in set *2*, it outperforms the serial variant even at lower selectivity factors. For example, from Figure 5.9, we can see that for the dataset *128MB : 512MB*, the vectorized variant provides a better join phase performance at every join selectivity higher than *40%*, whereas for datasets *256MB : 512MB* and *512MB : 512MB*, the vectorized variant overtakes the serial variant at all selectivity factors.

Branch-Free Variant - From Figure 5.8 and Figure 5.9, we can observe that the branch-free variant produces the worst join phase behavior among all the other radix join variants. Especially, at *100%* join selectivity, its performance degrades drastically (extremely high jump from *90%* to *100%*). This is because, at this selectivity, every key in S has exactly one matching key for join in R. Thus, the branch prediction at this selectivity is always easier and therefore, its corresponding branch-free instructions suffer a major performance depreciation.

Machine 2

Like machine *1*, we discuss the join phase behavior respective to each radix join variant on machine *2* as below. In Figure 5.10 and Figure 5.11, we present our join phase evaluation results for both set *1* and set *2* on this machine.

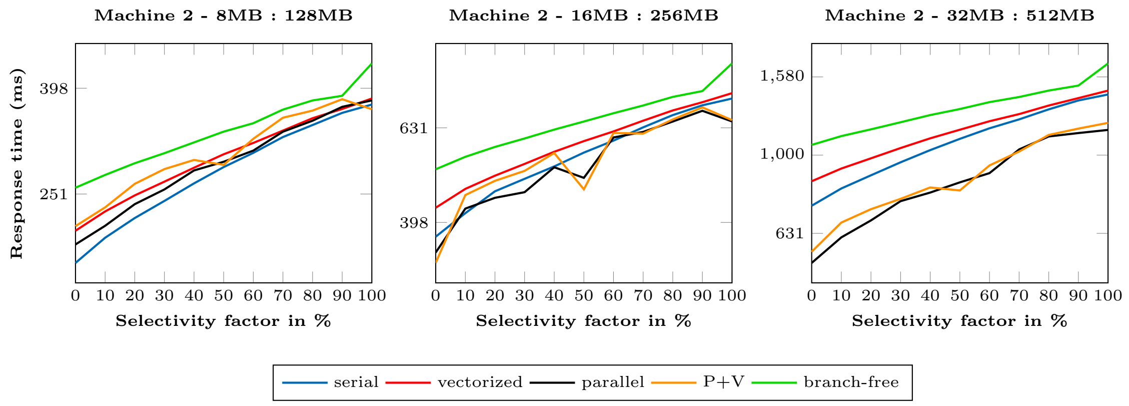

Figure 5.10: Join phase – machine 2 (for datasets in set 1)

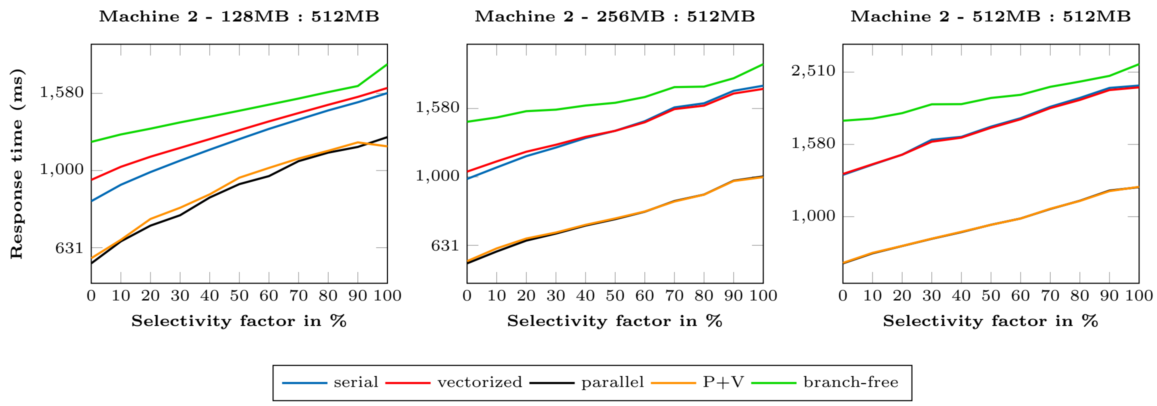

Figure 5.11: Join phase – machine 2 (for datasets in set 2)

Parallel and P+V Variants - From Figure 5.10, we can see that for datasets in set *1*, initially (for dataset *8MB : 128MB*) both parallel and P+V variants perform poorly against the serial variant. But, as we increase the cardinality of both relations R and S, their performance gradually increases and for our largest dataset in set *1* (*32MB : 512MB*), they completely outperform the serial variant. Thus, we expect both these variants to provide a similar performance like their behavior on machine *1* when we increase the relative sizes of R in set *2*. Accordingly, from Figure 5.11, we can observe that both parallel and P+V variants become extremely efficient for all datasets in set *2*. Among parallel and P+V variants, unlike the results observed for machine *1*, the P+V variant provides advantage over the parallel variant at *50%* selectivity for all datasets in set *1*. However, for set *2*, the performance of the P+V variant gradually decreases over the parallel variant as we increase the size of R and similar to machine *1*, the P+V variant offers advantage over the parallel variant only at higher selectivity factors (*90%* and *100%*).

Vectorized Variant - Unlike machine *1*, for all datasets in set *1*, the vectorized variant does not offer any benefits over the serial variant. However, we can see from Figure 5.10 that its performance came closer to the serial variant at higher selectivity factors. Accordingly, as we increase the size of R in set *2*, the vectorized variant starts improving and for our largest dataset (*512MB : 512MB*), it provides benefits over the serial variant at selectivity factors higher than *20%*, which nearly mimics their results observed for machine *1*.

Branch-Free Variant - From Figure 5.10 and Figure 5.11, we can see that similar to their behavior on machine *1*, the branch-free variant does provide any benefits over the other radix join variants. However, the degradation that occur at *100%* join selectivity is very less compared to their corresponding degradation on machine *1*.

From above discussions, it is clear that the join phase performance of parallel, P+V and vectorized variants on machine *2* closely matches with that of their performance on machine *1* for increasing sizes of R and S. Thus, we expect the join phase performance on both machines to nearly correlate with each other for input relations with cardinalities higher than the ones used in our workload.

5.4.3 Overall Performance

Despite the varying behaviour in the join phase, from our evaluation, we observe that the overall performance of each radix join variant is influenced by their partitioning behaviour. Therefore, as we expected earlier, the optimized variant that offers the best partitioning performance over the serial variant achieves the best overall performance for our datasets in both set *1* and set 2. In Figure 5.12 on the facing page, we present our evaluation results corresponding to the overall performance of each radix join variant for our largest dataset - *512MB : 512MB*. We summarize our observation as below:

- Due to their dominating partitioning behavior on both machines, parallel and P+V variants provide the best overall performance among all the other radix join variants for all our used datasets. Further, the variations observed in a serial radix join over the entire range of selectivity factors due to nature of the data distribution in the partition phase are reduced via both these parallelization techniques.
- Among parallel and P+V variants, we observe that the parallel variant offers the best overall performance over the P+V variant despite the parallel variant's

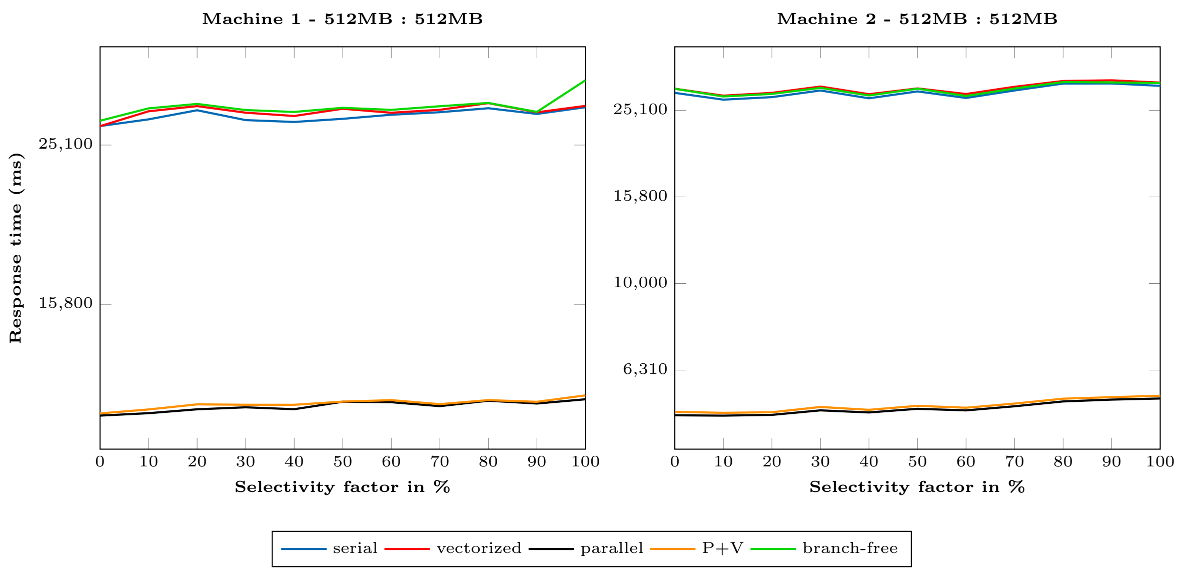

Figure 5.12: Overall performance of all radix join variants

performance depreciation against the P+V variant in the join phase at higher selectivity factors (*90%* and *100%*). This is because of the comparatively higher partitioning advantage that the parallel variant offers over the P+V variant at such selectivity factors.

- On both machines, we find that both vectorized and branch-free variants do not offer major advantages over the serial variant. This is because of the major partitioning performance degradation that the vectorized variant suffers against the serial variant. Similary, the branch-free variant's worse join phase behaviour reduces its overall performance significantly in contrast to the serial variant.

- Among vectorized and branch-free variants, we observe variations in their performance across the two used machines, i.e., on machine *1*, for datasets in set *1*, the vectorized variant with the worst partitioning behavior offers the worst overall performance except at *100%* selectivity, since the branch-free variant gets worse due its worse degradation in the join phase at this selectivity. However, as we increase the relative sizes of R, the vectorized variant gradually realize improvements over the branch-free variant at different selectivity factors and for our largest dataset (*512MB : 512MB*), it completely overtakes the branch-free variant. This is because of the advantage that the vectorized join phase offers over serial and branch-free join phases for increasing sizes of R and S on machine *1*.

- However, on machine *2*, the vectorized variant continues to degrade with increasing sizes of R and S and for our largest dataset (*512MB : 512MB*), it offers the worst overall performance. This is because the performance of the vectorized join phase on machine *2* in contrast to serial and branch-free join phases for increasing sizes of R is significantly less compared to its join phase performance benefits realized on machine *1*. Further, as explained in Section 5.4.1, the vectorized partitioning worsens for increasing cardinalities of R and S on machine *2* and hence, the overall radix join performance of the vectorized variant degrades drastically in contrast to all the other radix join variants.

From our discussions above, it is clear that compared to the impact of different cardinality sets, the join selectivity factor plays only a minor role in deciding the overall performance of each radix join variant. Teubner et al. argue that the input relation sizes (both absolute and relative) has a major impact on the overall radix join performance [BATz13]. We observe that this argument fits well even for their respective optimized variants. Especially, during the join phase, we observe that the performance of each radix join variant is influenced by varying cardinality sizes and on both machines, the performance of each radix join variant nearly mimics each other for increasing sizes of R and S.

5.5 Summary of our Evaluation

Based on our evaluation results for each optimized radix join variant, we summarize the performance of our adopted optimization techniques as below.

Vectorization

As mentioned in Section 5.3, the use of a large number of conflicting instructions (SIMD instructions and loop unrolling of depth *4*) over two passes of a radix partitioning add significant overhead to the partition phase. Hence, the vectorized partitioning proves to be non-advantageous for radix partitioning. However, in the join phase, less number of such conflicting instructions provides advantage on machine *1* for all datasets used in our workload. Still, on machine *2*, we expect such a constant improvement only for relations with extremely high cardinalities of R and S. Hence, we conclude that using vectorization for achieving optimization is not a suitable solution for the radix join technique. Therefore, as Kim et al. proposed, unless SIMD architectures provide support for scatter and gather operations, such architectures cannot be exploited fully by the radix join technique [KKL$^+$09]. Thus, the radix join cannot benefit significantly from vectorization.

Parallelization

Our evaluation results strictly confirm that in all cases, the scalar radix join benefits greatly when exploited with all available CPU cores. It should be remembered that for the parallel implementation of the radix join, we set only one thread per core. Thus, we avoid *simultaneous multi-threading (SMT)* and consequently the sharing of cache memories among threads belonging to the same core. Further, we do not encounter the problems of load imbalance among worker threads during the parallel partitioning since we evenly allocate the input relations among all threads to perform their individual partitioning. Even in the join phase, the problems of load imbalance are negligible due to the nature of uniformly distributed keys, i.e., the size of all partition pairs contained in the task set allocated to each worker thread are nearly equal because of the uniform distribution of keys.

Our parallelization advantages over the serial radix join for uniformly distributed keys also confirm the results of Kim et al. and Teubner et al. [KKL+09, BATz13] for a general hash join technique. Further, the variations observed due to the use of a single CPU core in a serial partitioning over the entire range of selectivity factors are reduced when executed with all available CPU cores for parallelization. Especially, for increasing sizes of R and S, we observe nearly a constant performance in parallel partitioning. Therefore, we propose parallelization as the best optimization method for the radix join technique.

Branch-Free Code Technique

As we projected earlier, our evaluation results clearly prove that in all cases, the branch-free code technique was not a suitable optimization method for the radix join technique. We already mentioned the reasons for such degrading performance in Section 5.3. Especially at a join selectivity of *100%* where the branch prediction is always easier, the branch-free implementation produces the worst behavior even over other poorly performing variants such as vectorization.

Parallelization Trends for Radix Join

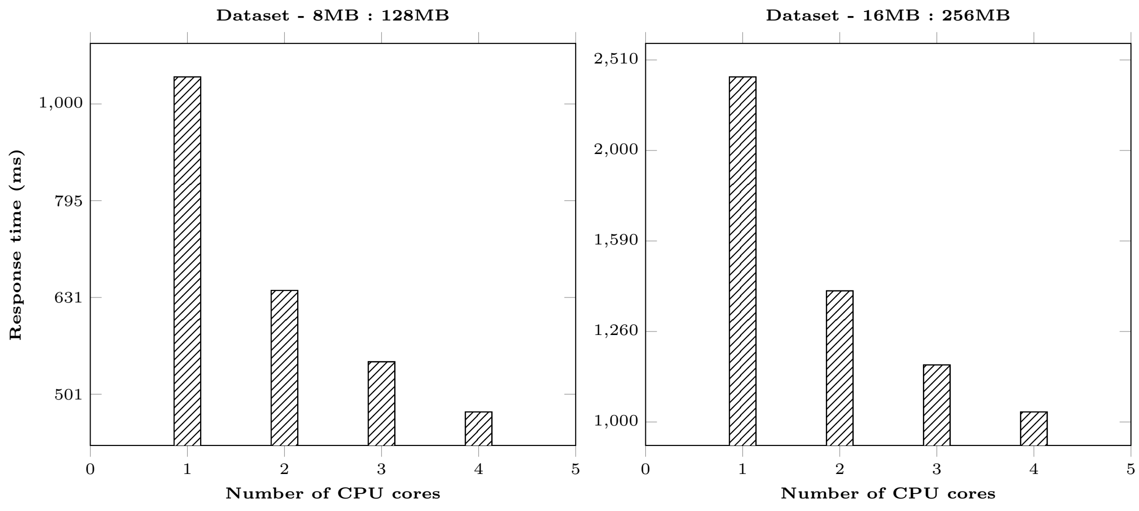

Figure 5.13: Parallelization trends for radix join (for datasets in set 1)

Since we found parallelization as the most favorable optimization technique for the radix join, we now see how it improves the scalability of the algorithm, i.e., we check how the performance improves as we increase the number of CPU cores. For this, we fix the selectivity factor at *100%* and test the parallel implementation for datasets in

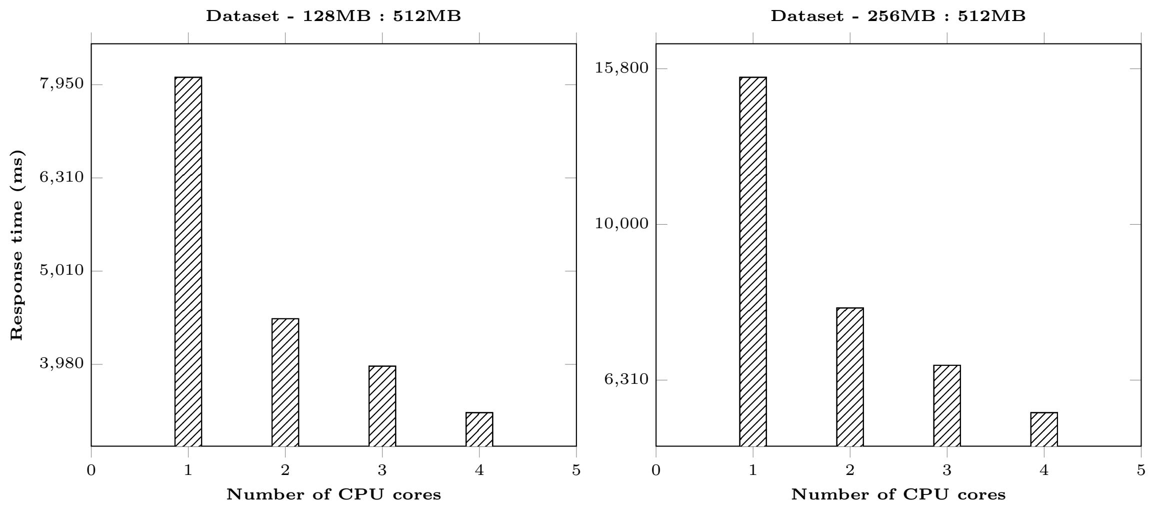

Figure 5.14: Parallelization trends for radix join (for datasets in set 2)

both set *1* and set *2*. In Figure 5.13 on the previous page and Figure 5.14, we present our evaluation results for machine *1*. As earlier, we set one thread per core such that we increase the number of threads in the range *1-4*.

From Figure 5.13 and Figure 5.14, it is clear that when we set more than one core to perform the radix join technique, we observe a significant improvement for parallelization. Further, between cores *2-4*, we nearly achieve a linear reduction in response times, i.e., for any core $n>1$, its response time is directly proportional to the response time consumed by the core *n+1* with a scalability factor of around *1.15*. We observe similar kind of scalability advantages for all our datasets in both set *1* and *2*.

Therefore, we conclude that parallelization will favor the radix join technique as long as the number of CPU cores is increased in future CPU architectures. Still, this technique need to be re-visited again to check its performance validity when exploited with simultaneous multi-threading. Further, in addition to providing benefits over other radix join variants, the parallel radix join would continue to remain the superior join over several other join techniques such as no partitioning hash join, partitioned hash join and sort-merge join. We will discuss this in detail in the next chapter.

6. Related Work

Code optimization techniques, in the field of database management systems, is an important concept that is visited often in order to understand the performance of various database operations in systems equipped with modern architectural capabilities. As part of our thesis, we studied the behavior of a database join operation implemented with certain code optimization techniques. Similarly, in the past, several researches have been carried out for studying the performance benefits of other database operations using various code optimization techniques. The content that we present in this chapter are based on surveys carried out to adopt ideas for using in our work. In general, the information that we present in this chapter belong to one of the following two categories:

1. Code optimizations in the context of various database operations
2. Code optimizations *(vectorization and parallelization)* in the context of join techniques

6.1 Code Optimizations for Database Operations

Broneske et al. studied the performance behavior of a database scan operation with respect to four different code optimization techniques – *loop unrolling, branch-free code technique, vectorization and parallelization* [BBS14]. In addition, they also implemented the combination of each of these optimization techniques for the scan operator. Based on their findings, they concluded that the performance of most of these optimized scan variants vary across different selectivity factors and different machines, i.e., there is no single scan variant that tends to become the best optimized variant in all situations. Hence, they proposed the method of cost model learning to select the best variant at run-time, i.e., use of a query engine which retains only the best optimal scan variant for future by learning the cost model of each variant depending upon the machine and the current workload.

Raducanu et al. proposed a framework called *microadaptivity* to overcome the problems associated with developing cost models for optimized variants [RBZ13]. This framework dynamically selects the best optimized variant for a particular database operator at run-time based on the historical performance of each variant under various environmental features such as used machines, workload and data distributions. To expose its validity, they tested the framework for different database operators such as scan, projection, join and aggregation optimized with loop unrolling, loop fission, branch-free code technique and vectorization. For every database operation, their respective optimized variants are implemented using vectorized processing, i.e., execution of all operator variants while processing a single attribute. However, their framework does not reveal the performance of the optimization techniques when combined with each other.

Vectorization and Parallelization Techniques for Database Operations

In order to gain benefits from modern CPUs that provide high degree of parallelism (both data-level parallelism and thread-level parallelism), several attempts have been made to find their advantages for database algorithms. Zhou et al. studied the performance of various database operations such as scan, aggregation, indexing and nested loop join when accelerated with modern SIMD architectures [ZR02]. More specifically, they attempted to apply SIMD techniques by carefully tuning them to the inner loop code present in the algorithms relative to these database operations. Similarly, Polychroniou et al. tested the aggregation operations with careful tuning of SIMD techniques to boost their performance [PR13]. Both these authors argue that explicit tuning of their adopted database operations to the underlying SIMD architectures result in providing performance benefits over the existing scalar operations. However, they do not study the advantages of SIMD techniques with respect to other code optimization techniques.

Chhugani et al. improved the scalability of a database merge sort by combining vectorization and parallelization techniques via SIMD instructions and multi-threading respectively [CNL^{+}08]. They realized that such combination of optimizations along with multi-way merging and cache blocking techniques improve the performance of a merge sort operation on a *4*-core processor with a SIMD width of up to *64* bits. Further, they also predict that their results are applicable on any multi-core processors with more than *32* cores.

6.2 Vectorization and Parallelization Techniques for Joins

Over the years, sort-merge join and hash join have been visited several times to find the better of these two techniques when exploited with multi-core architectures with modern CPU capabilities. To analyze such a performance behavior between these two join implementations, Kim et al. implemented both parallelized and vectorized versions of the partitioned hash join and the sort-merge join [KKL$^+$09]. Similarly, Teubner et al. implemented the parallel radix join and compared them with the sort-merge join optimized with a combination of both vectorization (via SIMD instructions) and parallelization techniques. [BATz13]. Further, they also compared the performance of the parallel radix join with the no partitioning hash join [BTAÖ13].

The experimental analysis of the optimized hash join variants over the scalar hash join with respect to different machines match with our findings for the radix join, i.e., Kim et al. concluded that SIMD techniques do not favor the hash join due to the limitations of scatter and gather operations, whereas the parallelization offer good scalability advantages particularly when the workload contains uniformly distributed data [KKL$^+$09]. However, they compared both these versions only to the scalar hash join and the sort-merge join. Further, comparison of these hash join versions with branch-free technique or their combinations are not part of their scope since their main goal was to study the exploitation of different types of join algorithms with modern CPU architectures.

Similarly, Teubner et al. argue that the parallel radix join performs significantly better than the no partitioning hash join [BTAÖ13]. Further, it also offers better performance over the sort-merge join improved with both vectorization and parallelization techniques [BATz13]. Blanas et al. revealed the performance benefits of the parallel radix join while comparing them with the no partitioning hash join and the partitioned hash join [BLP11]. Again, we noticed that parallelization techniques for the radix join offer significant benefits over its scalar implementation. In addition, the parallel radix join was also found to be the best hash join technique among all the other hash joins. Further, our study implies that more than selectivity factors, workload with varying cardinalities (both relative and absolute) reveal more insights into the behavior of the optimized variants for both hash join and sort-merge join.

7. Conclusion

In this thesis, we implemented the code optimization techniques to a scalar radix join for studying their performance impacts with respect to modern capabilities of the underlying CPU architectures. Based on our discussion in previous chapters, we summarize the previous published findings in the literature related to our work and compare the concurrence of our evaluation results with them as below:

- Most of the previous published results confirm that the hash join and consequently the radix join, unlike the sort-merge join, cannot benefit significantly from vectorization via increasing SIMD capabilities. This is because the current SIMD architectures do not support the storage of data to non-contiguous memory locations *(scatter operation)* required for radix partitioning. Further, while performing the probe between partition pairs, they suffer from major performance depreciation due to the loading of data from non-contiguous memory addresses *(gather operation)*. Due to these limitations, the radix join cannot fully exploit the SIMD architectures and hence need to be reverted back to its original scalar version. In our vectorized implementation, we replaced the scatter and gather operations by loop unrolling technique and hence, we explicitly tuned the algorithm such that it is not affected by such SIMD limitations. However, we still observed that the improvement in the resulting implementation is not enough to provide benefits over the original implementation of the scalar radix join. Therefore, we argue that unless SIMD architectures in modern processors come up with efficient hardware support for scatter and gather operations, the radix join cannot realize improvements from vectorization techniques via SIMD instructions.

- Current implementations of the radix join are expected to benefit significantly from any multi-core CPU architecture as long as the number of CPU cores is increased. Further, omitting *simultaneous multi-threading* (SMT) does not cause any performance depreciation for increasing sizes of the workload. Our evaluation results presented in Chapter 5 concurred with the previous published results, i.e., we observed a significant reduction in response times as we increased the number of cores where we set one thread per core for parallelization. Further, our parallel radix join implemented with all available CPU cores provided the best overall performance even among the other optimized radix join variants. Therefore, we argue that the radix join technique would continue to benefit greatly from parallelization techniques via multi-threading.

- Our implementation of the parallel radix join which is optimized further with vectorization techniques provided almost the same benefits as its parallel counterpart over the scalar radix join. However, it does not provide advantage over the actual parallel version due to the limitations of SIMD instructions that we mentioned earlier.

- In literature, there is little work considering the branch-free code technique for optimizing the radix join. However, we projected that, in the context of in-memory join operations, the branch-free code technique would not prove to be advantageous for the radix join. This is because, replacing a branch with non-branching instructions require execution of comparatively more number of instructions. Such instructions tend to provide benefits only when the outcome of a branching statement is hard to predict. But, in case of a radix join probing, such outcomes are easier to predict depending upon varying selectivity factors. We saw in Chapter 5 that our evaluation results matched with our prediction. Thus, we argue that the branch-free code technique does not help a main-memory radix join technique in any situation.

Finally, to conclude our whole work, we argue that the parallel radix join has been found to provide the best optimal performance over its scalar version and hence, we propose the parallelization technique as the best optimization technique for the scalar radix join. Further, as mentioned in the previous chapter, the parallel radix join would also remain the best in-memory hash join technique among both hardware-conscious and hardware-oblivious hash join techniques.

8. Future Work

In this thesis, we discussed the performance behavior of several code optimization techniques for the radix join. Even though we demonstrated the impacts of each optimization technique, we identify certain points that need to be addressed in the future to further confirm our analysis. Thus, in this chapter, we present some of the open problems that could be undertaken to extend our evaluation results.

Performance with Skewed Data Distribution

Our evaluation results for the performance of each radix join variant are based on input relations consisting of uniformly distributed data. Thus, one possible direction to extend our work would be to analyze the behavior of each variant with respect to skewed data distribution. Even though we expect the vectorized and branch-free variants to nearly produce the same performance for such data distributions, we expect the parallel variant to suffer performance depreciation due to load imbalance that might occur in the join phase. Especially, when skewed distributions such as Zipf data distributions are used, the probability of each tuple falling into the same partition are likely to increase in the worst case and hence, the scalability advantages offered by the parallelization technique are likely to get compromised significantly because of a major load imbalance during probing. Therefore, it is necessary to check whether such degradation affects their performance against other radix join variants, especially the variant that is implemented with both vectorization and parallelization techniques.

Performance with Re-Ordered Histogram-Based Hash Table Implementation

In Chapter 3, we mentioned two approaches for executing the join phase of a radix join - *classical bucket-chaining algorithm* and *re-ordered histogram-based hash table algorithm.*

However, for our work, we used only the bucket-chaining algorithm. Therefore, another possible way to extend our work would be to analyze the performance of the vectorized variant over the serial variant implemented with the re-ordered histogram-based hash table algorithm. This is because with this algorithm, during the *build* phase, the input keys in the inner partition (R_i) are re-ordered such that keys possessing the same hash value are stored in contiguous memory locations. Therefore, we expect the *probe* phase to benefit significantly via vectorization techniques since the gather operations which previously limited the exploitation of SIMD architectures due to the nature of bucket-chaining algorithm can now be performed efficiently. Even though such advantages that are likely to realize in the overall join phase are independent of the partitioning phase, we can still expect a reasonable performance over the serial radix join by limiting the application of vectorization techniques only to the join phase, i.e., serial partitioning combined with a vectorized join. Similarly, we can expect the same kind of behavior among the parallel variant and the variant implemented with both vectorization and parallelization techniques.

Performance with Simultaneous Multi-Threading

Throughout our work, for our used multi-core architectures, we set only one thread per core for carrying out the join operation. Thus, for the parallel radix join, we need to confirm their performance advantages when implemented on multi-architectures supporting more than one hardware thread per core. With this, we can analyze their behavior with respect to the exploitation of *simultaneous multi-threading* capabilities offered by such architectures. Even though we observed reasonable scalability advantages for our parallel variant, we expect such benefits to degrade due to simultaneous multi-threading. This is because, with simultaneous multi-threading, hardware resources (such as cache memories) are always shared among multiple threads belonging to the same core and hence, cache-conscious algorithms always suffer a major performance depreciation due to simultaneous multi-threading.

Numerous researches in this context have already been analyzed for studying the behavior of the parallel radix join with respect to other hash join techniques. However, there is not enough evidence about their behavior relative to other radix join variants while exploiting the simultaneous multi-threading capability of multi-core CPUs. Thus, it is important to analyze whether performance depreciation that is prone to occur due to sharing of hardware resources would affect their overall performance against other radix join variants.

Bibliography

[AR06] Shameen Akhter and Jason Roberts. *Multi-Core Programming*, volume 33. Intel press Hillsboro, 2006. (cited on Page 13, 22, 23, and 24)

[BATz13] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multicore, MainMemory Joins: Sort vs. Hash Revisited. In *Proceedings of the 40th International Conference on Very Large Data Bases (VLDB)*, pages 85–96, Hangzhou, China, September 2013. VLDB Endowment. (cited on Page 2, 12, 45, 61, 70, 71, and 75)

[BBHS14] David Broneske, Sebastian Breß, Max Heimel, and Gunter Saake. Toward Hardware-Sensitive Database Operations. In *Proceedings of the 17th International Conference on Extending Database Technology (EDBT)*, pages 229–234, Athens, Greece, March 2014. OpenProceedings.org. (cited on Page 2 and 16)

[BBS14] David Broneske, Sebastian Breß, and Gunter Saake. Database Scan Variants on Modern CPUs: A Performance Study. In *Proceedings of the 2nd International Workshop on In Memory Data Management and Analytics*, pages 1–15, Hangzhou, China, September 2014. Springer. (cited on Page 1, 8, 22, 26, 27, 49, 51, 61, and 73)

[Bik04] Aart JC Bik. Software Vectorization Handbook : Applying Intel Multimedia Extensions for Maximum Performance. May 2004. (cited on Page xi, 13, 17, 18, 19, 21, and 43)

[BK$^+$14] Martin Bachmaier, Ilya Krutov, et al. *In-Memory Computing with SAP HANA on IBM eX5 and X6 Systems.* IBM Redbooks, 2014. (cited on Page 5)

[BKM08] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the Memory Wall in MonetDB. *Communications of the ACM*, 51(12):77–85, December 2008. (cited on Page 10)

[BLP11] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 37–48, Athens, Greece, June 2011. ACM. (cited on Page 37, 38, 39, 41, 54, 56, 57, 58, 59, 60, and 75)

[Bro13] David Broneske. On the Impact of Hardware on Relational Join Processing. Master thesis, University of Magdeburg, Germany, August 2013. (cited on Page 1 and 11)

[BTAÖ13] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *Proceedings of the 29th IEEE International Conference on Data Engineering*, pages 362–373, Brisbane, Australia, April 2013. IEEE Computer Society. (cited on Page xi, 2, 14, 15, 16, 29, 30, 31, 32, 33, 34, 35, 37, 38, 39, 41, 42, 45, 52, 53, 56, 59, and 75)

[CNL+08] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture. *Proceedings of the VLDB Endowment*, 1(2):1313–1324, August 2008. (cited on Page xi, 12, 13, and 74)

[DFI+13] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1243–1254, New York, USA, June 2013. ACM. (cited on Page 5)

[Eic88] Margaret H. Eich. Mars: The Design of a Main Memory Database Machine. In *Database Machines and Knowledge Base Machines*, pages 325–338. Springer, 1988. (cited on Page 5 and 6)

[Fer05] Miguel C. Ferreira. Compression and Query Execution within Column Oriented Databases. Master thesis, Massachusetts Institute of Technology, USA, June 2005. (cited on Page 10)

[FM04] Heiko Falk and Peter Marwedel. *Source Code Optimization Techniques for Data Flow Dominated Embedded Software*. Kluwer Academic, 2004. (cited on Page 17)

[GMS92] Hector Garcia-Molina and Kenneth Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, December 1992. (cited on Page 1, 5, 6, and 7)

[Hei11] Joe Heinrich. *MIPS R4000 Microprocessor User's Manual*. MIPS Technologies, Inc., 2011. (cited on Page 25)

[IKM09] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Self-organizing Tuple Reconstruction in Column-stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 297–308, City of Providence, Rhode Island, June 2009. ACM. (cited on Page 10)

[JKLM12] Hwancheol Jeong, Sunghoon Kim, Weonjong Lee, and Seok-Ho Myung. Performance of SSE and AVX Instruction Sets. In *Proceedings of the 30th International Symposium on LATTICE Field Theory*, pages 1–7, Cairns, Australia, June 2012. (cited on Page 18)

[KKL$^+$09] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *Proceedings of the VLDB Endowment*, 2(2):1378–1389, August 2009. (cited on Page 1, 2, 11, 13, 14, 16, 21, 33, 38, 44, 54, 56, 57, 59, 60, 61, 70, 71, and 75)

[Kle09] Andi Kleen. Linux Multi-Core Scalability. In *Proceedings of the 16th International Linux System Technology Conference*, Dresden, Germany, October 2009. Intel Press. (cited on Page 24)

[KSS12] Veit Köppen, Gunter Saake, and Kai-Uwe Sattler. *Data Warehouse Technologien.* MITP, 2012. (cited on Page 10)

[KW02] Markus Kowarschik and Christian Weiß. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In *Algorithms for Memory Hierarchies, Advanced Lectures*, volume 2625 of *Lecture Notes in Computer Science*, pages 213–232, Saarland, Germany, March 2002. Springer. (cited on Page 8)

[KWLP12] Jens Krueger, Johannes Wust, Martin Linkhorst, and Hasso Plattner. Leveraging Compression in In-Memory Databases. In *The 4th International Conference on Advances in Databases, Knowledge, and Data Applications*, pages 147–153, Saint Gilles, Reunion, February 2012. IARIA. (cited on Page 1 and 5)

[LC86] Tobin J. Lehman and Michael J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 294–303, Kyoto, Japan, August 1986. Morgan Kaufmann. (cited on Page 6 and 7)

[Los09] David Loshin. Gaining the Performance Edge Using a Column-Oriented Database Management System. *White Paper*, 2009. (cited on Page 10)

[Man02] Stefan Manegold. *Understanding, Modeling, and Improving Main-Memory Database Performance.* PhD thesis, University of Amsterdam, The Netherlands, December 2002. (cited on Page 2, 7, 9, 31, 32, 33, 34, 37, 38, 42, and 57)

[MBK02] Stefan Manegold, Peter Boncz, and Martin L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 191–202, Hong Kong, China, August 2002. VLDB Endowment. (cited on Page 8)

[ME92] Priti Mishra and Margaret H. Eich. Join Processing in Relational Databases. *ACM Computing Surveys*, 24(1):63–113, March 1992. (cited on Page 11 and 13)

[PR13] Orestis Polychroniou and Kenneth A. Ross. High Throughput Heavy Hitter Aggregation for Modern SIMD Processors. In *Proceedings of the 9th International Workshop on Data Management on New Hardware*, pages 6:1–6:6, New York, USA, June 2013. ACM. (cited on Page 74)

[RBZ13] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. Micro Adaptivity in Vectorwise. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1231–1242, New York, USA, June 2013. ACM. (cited on Page 1, 16, and 74)

[Ros04] Kenneth A. Ross. Selection Conditions in Main Memory. *ACM Transactions on Database Systems*, 29(1):132–161, March 2004. (cited on Page 26 and 27)

[Sha86] Leonard D. Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM Transactions on Database Systems*, 11(3):239–264, August 1986. (cited on Page 11)

[sim06] *Intel® SSE4 Programming Reference*. Intel Press, 2006. (cited on Page 18)

[Smi82] Alan Jay Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982. (cited on Page 8)

[VAD+11] Berthold Vöcking, Helmut Alt, Martin Dietzfelbinger, Rüdiger Reischuk, Christian Scheideler, Heribert Vollmer, and Dorothea Wagner. *Algorithms Unplugged*. Springer, 2011. (cited on Page 6)

[Wil07] Anthony Williams. *Thread*, 2007. (cited on Page 44)

[Zah07] Mohamed Zahran. Cache Replacement Policy Revisited. In *The Annual Workshop on Duplicating, Deconstructing, and Debunking held in conjunction with the International Symposium on Computer Architecture*, New York, USA, June 2007. (cited on Page 9)

[ZHB06] Marcin Zukowski, Sándor Héman, and Peter Boncz. Architecture-Conscious Hashing. In *Proceedings of the 2nd International Workshop on Data Management on New Hardware*, pages 42–48, Chicago, Illinois, USA, June 2006. ACM. (cited on Page 33)

[ZR02] Jingren Zhou and Kenneth A. Ross. Implementing Database Operations Using SIMD Instructions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 145–156, Madison, WI, USA, June 2002. ACM. (cited on Page xi, 17, 18, 20, 21, and 74)

Declaration of Authorship

I hereby certify that this thesis has been composed by me and is based on my own work, unless stated otherwise. No other person's work has been used without due acknowledgement in this thesis. All references and verbatim extracts have been quoted, and all sources of information, including graphs and data sets, have been specifically acknowledged.

Magdeburg; January 20, 2015