Balancing vectorized query execution with bandwidth-optimized storage
Zukowski, M

Citation for published version (APA):
Chapter 5

Vectorized execution model

This chapter discusses in detail the vectorized execution model introduced in Chapter 4. First, Section 5.1 analyzes the properties of this model, comparing it to the previously proposed tuple-at-a-time and column-at-a-time models. Later, Section 5.2 discusses the implementation of data processing operations in this model, first identifying the requirements of efficient implementations, and then providing a set of implementation techniques. Additionally, Section 5.2.2 discusses different possible choices of data organization during processing. All these techniques are synthesized in the description of an example implementation of one of the crucial database operators: a hash join. A simple vectorized implementation is initially presented in Section 5.3 and Section 5.4 discusses a set of techniques improving its performance. Finally, to complete the chapter, Section 5.5 provides a set of vectorized implementations of other interesting processing tasks.

5.1 Properties of the vectorized execution model

The goal of researching a new execution model was to overcome the problems found in the previously proposed tuple-at-a-time and column-at-a-time models. This section analyzes the properties of the new model and compares them with the existing solutions.
5.1.1 Interpretation overhead

In the traditional Volcano model, the data is processed in a 'pull' fashion, where the consuming operators ask their children for the next tuple. As a result, at least one \texttt{next()} call is performed for every tuple in every operator. Also within a single relational operator, multiple functions are called. For example, a Project operator that computes a sum of two columns needs to call an addition primitive for each tuple it processes. Note that these calls cannot be inlined by the compiler, as they are query-specific, hence the cost of passing the parameters and modifying the program counter is always present. Also, the actual addresses of the functions to call need to be read from memory, making it hard for a CPU to speculate ahead of the call. Finally, complex operator logic is performed for every tuple, causing the interpretation overhead to dominate the overall execution time.

With vectorized processing, in both scenarios the function call can be amortized over a large set of tuples. Figure 5.1 presents the results for TPC-H query 1 (scale factor 1) executed on MonetDB/X100 running on a 2-GHz Athlon64. The first observation is that using the optimal vector size can give a performance improvement of as much as 30 times. The second observation is on the influence of optimization settings in the compiler: with optimization, both using the gcc and icc compilers, performance gains are much bigger for larger vector sizes. This is because the actual data processing code can be efficiently optimized by the
Section 5.1: Properties of the vectorized execution model

The compiler, unlike the branch-heavy control logic found in operators. This aspect is further analyzed in Section 5.2.1.3.

Comparing to the column-at-a-time model used in MonetDB, the vectorized model can result in a slightly higher overhead, as the interpretation occurs for every vector. Still, with vector size in range of hundreds of tuples, this overhead is so small that its impact on the overall performance is negligible, as seen with an almost flat line in Figure 5.1. This is confirmed by the left-most side of the Figure 5.2 which shows that for large vector sizes the number of the CPU instructions stays virtually constant. Note that MonetDB suffers from main-memory materialization overhead, which degrades its performance, as discussed in section 3.3.

5.1.2 Instruction cache

The impact of instruction-cache misses on the performance of the tuple-at-a-time model has been identified for both OLTP query loads [ADHW99, HA04, HSA05], as well for OLAP-like queries [ADHW99, ZR04]. They can constitute even 40% of the entire query execution time [HA04]. Techniques to reduce this overhead include grouping different queries performing the same operation [HA04, HSA05] and buffering tuples within a single query [ZR04]. The latter technique is slightly similar to vectorized processing, since it passes multiple tuples between the operators. However, it causes an additional data copying cost, and the data processing is still performed in a tuple-at-a-time fashion.

To demonstrate the impact of the instruction cache on vectorized processing, we analyze the performance of three queries: TPC-H Q1, and two variants of it, Q1’ and Q1” that use roughly 2- and 3- times more different primitives, increasing the amount of used instruction memory. In Figure 5.2 for all three queries we provide both the total number of executed instructions, as well as the number of L1 instruction cache misses. As the results show, for Q1 instruction misses are negligible. For Q1’, the number grows somewhat, but still is relatively low. With Q1”, the size of separate code paths finally exceeds the size of the instruction cache, and we see that although the code size increased over 3 times over Q1, the number of misses can be as much as 1000 times higher, even though the total number of instructions grew only two-fold. Luckily, even for this complex query, the number of instruction-misses decreases linearly with a growing vector size, and the instruction-cache-miss overhead can be alleviated.

Similarly to the interpretation overhead, the overhead of the instruction misses in the vectorized model can be slightly higher than in MonetDB. Still, since it is amortized among multiple tuples in a vector, it is typically negligible.
5.1.3 Processing unit size

Comparing to the tuple-at-a-time and column-at-a-time models, the vectorized model provides a granularity of operation that falls between these two extremes. As a result, there are situations in which some logic that is usually executed for every tuple, can be executed on a per-vector base. A simple example is data partitioning, when the result partition sizes are not known in advance. The code for dividing a vector of N tuples into P partitions using the hash values could be as follows:

```c
for (i = 0; i < N; i++) {
    group = hash_values[i] % P;
    *(part[group]++) = values[i];
    if (part[group] == part_end[group])
        overflow(group);
}
```

Note that the overflow check is necessary for each tuple if we do not know the partition sizes in advance. While this check is usually false, we can still remove it from the loop, by exploiting the fact that in most cases the buffers for the destination groups are much larger than the size of the vector. As a result, we can check if every group buffer still contains enough tuples before processing each vector.

```c
for (i = 0; i < P; i++)
    if (part[i] >= part_sentinel[i])
```
In this situation, we check for a buffer overflow not \( N \) times, but \( P \) times. It is also possible to perform such check every few vectors, to further reduce its cost. This solution requires some extra 'sentinel' space left in the buffer, but this waste should be marginal (e.g. 1024 elements out of 128 thousands). We compared both solutions implemented using optimization techniques described in Section 5.2.4 and the second version gave a ca.15% improvement of the partitioning step (using 64 partitions and 1024-tuple vectors). Note that this optimization cannot be applied by the compiler automatically, since it requires modifications to the underlying data organization.

Another case where vectors can be a useful extra computational unit are exception situations. An example is handling of an arithmetic overflow. Typically, an overflow is checked for each performed operation. However, on some architectures, it is possible to check if an overflow occurred over a large set of computations (e.g. by using summary overflow bit in PowerPC CPUs \cite{PMAJ01}).

A different vectorized solution to overflow checking is proposed in Section 5.5.1.

A natural intermediate processing unit can also be helpful for data routing in a query plan. For example, the \texttt{exchange} operator \cite{Gra90} can distribute tuples for parallel processing using vectors. Also, in dynamic query optimization, for example in Eddies \cite{AH00}, adapting the plan every vector, and not every tuple, is beneficial.

Removing logic from the per-tuple loop has an additional benefit – the resulting code is typically simpler, allowing better optimizations by a compiler and more efficient execution on a CPU.

### 5.1.4 Code efficiency

In the vectorized model, operator functionality is decomposed into small processing units that we call primitives. As hinted before, thanks to their high specialization they provide code that is easy to optimize for the compiler and efficiently executed on modern CPUs. As an example, let us analyze the following simple routine that adds two vectors of integers:

```c
void map_add_int_vec_int_vec(int *result, int *input1, int *input2, int n) {
    for (int i = 0; i < n; i++)
        result[i] = input1[i] + input2[i];
}
```
We can identify the following properties of this routine: it does not contain any control dependencies, hence does not suffer from branch prediction misses; it does not contain any data dependencies, hence there are no stalls in the processing pipeline; a simple loop allows easy unrolling, reducing the loop overhead; data access is direct, there is no overhead in attribute extraction; data access is fully sequential, hence does not suffer from random cache misses and hardware prefetching can be applied; performed operations are simple and allow easy SIMDization. The last property already provides a 2x-8x speedup for various operations on many common data types, and with growing widths of SIMD units (e.g. 256-bits in Intel AVX [Int08]) this performance benefit of this technique will increase.

The described routine is a perfect example of how efficient vectorization can be – on a Core2Duo machine it spends only 0.92 cycles per single iteration. Comparing to an interpreted tuple-at-a-time approach, the performance benefit can be even two orders of magnitude.

While providing primitives having all described properties for all types of operations found in databases is probably impossible, efficient solutions can be developed even for complex problems. In Section 5.2.4 we will discuss a set of techniques helpful in the implementation of such routines.

5.1.5 Block algorithms

Processing multiple tuples does not only allow efficiently compiled and executed code. It also enables applying algorithms that require a set of tuples to work. For example, in software data prefetching, two major approaches are used: pipelined prefetching and group prefetching [CAGM04]. In the tuple-at-a-time model, they both require tuple buffering, while being directly applicable in the vectorized model. On the other hand, in the column-at-a-time model, the effective block size (full column) is typically too large to exploit the benefit of the prefetching – the data prefetched at the beginning of the column will be most likely evicted at the end.

Another technique that requires multiple tuples is efficient computation of selection predicates [Ros02]. With a block of tuples for which the same predicate needs to be evaluated, different approaches (binary AND, logical AND or no-branch) are optimal. The choice of the used method can be performed using a cost model [Ros02] at query compilation time, but also dynamically during
the query execution – information about selectivity in the previous vector is typically a good indicator for the current one.

Finally, having an opportunity to work with multiple tuples allows various programming tricks. For example, during processing data after selection, where selection result is a Boolean bitmap, we can exploit the knowledge of high predicate selectivity, similarly as in [ZR02]. Then the bitmap consists mostly of zeros, and we can check multiple bits in one go, speculating that they are not set, and handle the non-zero cases in an extra step. Similar tricks can be used to detect a zero in the vector that is a parameter for the division operation, handle NULL values in mostly non-NULL data, and more.

5.1.6 Scalability

In the column-at-a-time model, every operator fully materializes its result, making it ineffective for queries requiring disk-based intermediate results. The Volcano model, thanks to its pipelined nature, can process datasets larger than available memory, using on-disk materialization only for blocking operators (sort, hash-join). This property is directly inherited by the vectorized model.

A newly introduced aspect of the vectorized model is its scalability with respect to the complexity of the query and the cache size. With complex query plans that internally keep a large number of vectors, the vector size needs to be reduced to fit the data in the CPU cache, diminishing the benefits of the reduced interpretation overhead. As discussed in Section 4.2.2.2, depending on the query complexity, the vector size should be chosen such that all data fits in either L1 or L2 cache. Since the L2 caches of modern CPUs are in order of megabytes, vectors can be sufficiently large to remove the interpretation overhead (hundreds of tuples) and still fit in the cache even for queries with hundreds of vectors.

5.1.7 Query plan complexity and optimization

The query plans for the vectorized model usually match the plans of the tuple-at-a-time model: they are trees of N-ary operators working in a pipelined fashion. As a result, the vectorized model can benefit from decades of research on the traditional query plans optimization.

In the column-at-a-time model, query plans are significantly more complex, mostly due to the used binary algebra – for the same task, multiple per-column operations need to be performed. However, due to the materializing nature of this model, the query plans are closer to an imperative programming language,
and many optimizations from that area are additionally applicable. For example, common subexpression elimination is straightforward here, while being potentially non-trivial in the pipelined model [DSRS01].

5.1.8 Implementation complexity

Implementation of relational operators in the tuple-at-a-time model has been studied over the last 3 decades and is well understood. Still, typical solutions provide code that needs to be very generic, making the implementation often highly complex. In the column-at-a-time model, every operator both consumes and produces simple data arrays of known data types. This makes the implementation of most operators relatively straightforward.

The vector-at-a-time model brings a new challenge of decomposing the processing into independent vectorized primitives. While for some operators (e.g., projections) it is easy, for some it is significantly more challenging, as discussed later in this chapter. An interesting option in this model is that it is possible to emulate both tuple- and column-at-a-time models internally in the operators, allowing easy system prototyping.

5.1.9 Profiling and performance optimization

In the tuple-at-a-time model, the processing thread continuously switches between all operators inside the active plan segment, performing both control logic and the actual data processing. As a result, profiling the execution of individual processing steps is relatively complex: putting a time (or e.g., hardware event) counter for every step inside the operator is often too expensive, and sampling process activity or simulations can be imprecise. Even when the performance bottleneck is localized, improving the performance of that part is often hard, as the involved code is typically large and complex.

In the column-at-a-time model, profiling is straightforward – each operator is fully independent and hence it is trivial to measure its cost. This allows easy detection of bottleneck operators. Still, within the operator it is unclear how much time different operations take. For example, in a hash-join operator, the operator-level profiling does not provide information on the cost of the build- and probe-phase separately.

The vectorized model lands, again, in between. Since the profiling overhead is amortized among multiple tuples, it is possible to precisely measure the performance of every operator and every primitive. As a result, it is easy to spot fine-grain performance bottlenecks. Additionally, once a bottleneck is located,
### Section 5.1: Properties of the vectorized execution model

<table>
<thead>
<tr>
<th></th>
<th>Tuple</th>
<th>Column</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>query plans</td>
<td>simple</td>
<td>complex</td>
<td>simple</td>
</tr>
<tr>
<td>instruction cache utilization</td>
<td>poor</td>
<td>extremely good</td>
<td>very good</td>
</tr>
<tr>
<td>plan-data cache utilization</td>
<td>poor</td>
<td>extremely good</td>
<td>very good</td>
</tr>
<tr>
<td>function calls</td>
<td>complex</td>
<td>extremely few direct</td>
<td>direct</td>
</tr>
<tr>
<td>attribute access</td>
<td>interpretation</td>
<td>processing</td>
<td>processing</td>
</tr>
<tr>
<td>time mostly spent on ...</td>
<td>poor</td>
<td>good</td>
<td>very good</td>
</tr>
<tr>
<td>CPU utilization</td>
<td>limited</td>
<td>applicable easy</td>
<td>applicable</td>
</tr>
<tr>
<td>compiler optimizations</td>
<td>medium</td>
<td>easy</td>
<td>medium</td>
</tr>
<tr>
<td>implementation</td>
<td>hard</td>
<td>expensive</td>
<td>simple</td>
</tr>
<tr>
<td>profiling and optimization</td>
<td>very cheap</td>
<td>limited</td>
<td>cheap</td>
</tr>
<tr>
<td>materialization overhead</td>
<td>good</td>
<td>small</td>
<td>good</td>
</tr>
<tr>
<td>scalability</td>
<td>large</td>
<td></td>
<td>small</td>
</tr>
<tr>
<td>volume of accessed data</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Comparison of the N-ary tuple-at-a-time (Tuple), MonetDB column-at-a-time (Column) and vectorized in-cache (Vector) execution models

The code involved is small (typically a single primitive, often only a few lines), making it relatively easy to optimize. Finally, some dynamic optimizations, like choosing one of the possible implementations of the same primitive, are easy in this model.

#### 5.1.10 Comparison summary

To sum up the comparison of the execution models, Table 5.1 shows their properties in all discussed areas. Clearly, the vectorized model combines the best properties of the previous approaches. Still, the question remains, how to actually implement a full system based on the principles of this model. The following sections try to address this issue.

---

1Note that for very complex query plans, in the vectorized model either the vector size shrinks and the model starts to suffer from some of the “Tuple” problems, or the vector size exceeds the cache capacity, causing some of the “Column” inefficiencies.
5.2 Implementing the vectorized model

5.2.1 Efficient implementation requirements

Since the major part of time in the vectorized execution model tends to be spent in the data processing primitives, it is important to provide efficient implementation of these. For optimal performance, the vectorized primitives need to meet a set of requirements described in this section. While not every data processing operation can have all the described features, the following sections introduce a set of optimization techniques that make it possible for most operations to get many of these benefits.

5.2.1.1 Bulk processing

To achieve the computational efficiency described in Section 5.1.4, data processing primitives should follow the idea of *bulk processing* – performing the same operation for multiple tuples independently. To achieve this, the primitives need to meet some criteria that can be seen as task independence at various levels of processing:

**primitive independence** - the first step is to make the primitives process multiple data items in one function call, without the need to communicate with other primitives.

**operation independence** - if processing of one tuple is independent from other ones, the same computation can be in parallel executed for multiple tuples. This has benefits for super-scalar execution on modern CPUs, and provides SIMD opportunities.

**CPU instruction independence** - when processing a given tuple, it is important that separate CPU instructions performing the operation are independent. Otherwise, it is possible that execution hazards described in Section 2.1.5 cause “pipeline bubbles”, damaging the performance.

5.2.1.2 Data location and organization

The location of data that a primitive processes can have a significant impact on the execution performance, as demonstrated in Section 4.2.2. Even with fully sequential access, reading and writing data to main memory is significantly more expensive than performing the operation in the CPU cache. Therefore, it is crucial to minimize RAM accesses and focus on in-cache execution.
Another issue is data organization. While MonetDB/X100 uses column-based structures for passing the data between operators, for some tasks row-based organizations are beneficial, as presented in Section 5.2.2. As a result, for different cases, varying in the type of operation, but also in data properties, different data organizations should be used for optimal performance.

### 5.2.1.3 Compiler optimization amenability

Another important factor for a high-performance primitive is its amenability to compiler optimizations. As mentioned in Section 5.1.1, the computation-intensive primitives in MonetDB/X100 result in a larger higher benefit of compiler optimizations higher than in the interpretation-intensive code found in the traditional database engines.

For better analysis of this issue, Figure 5.3 demonstrates the performance of the query from Section 4.2.2 dissected into total time (left), `mul1` performance (middle) and `add1` performance (right). Three `icc` compiler optimization levels have been used: `-O0`, with optimizations disabled; `-O1`, with basic optimizations; `-O2`, with more optimizations, including exploiting SIMD instructions. First observation is that compiler optimization adds very little performance for small vector sizes – even with `-O2` the benefit is less than 50% improvement. The reason for this is that in this situation the execution time is dominated by
function calls, which are hard to optimize, as they cannot be inlined. For larger vector sizes, the time spent in data-intensive primitives is relatively longer, and, since these primitives are more amenable to the compiler optimizations, the optimization impact increases.

Detailed analysis of the per-primitive optimizations effect shows that for memory-intensive `mul1` primitive the use of SIMD instructions does not improve performance. This is caused by this primitive being memory-bound. On the other hand, for the cache-intensive `add1` primitive, SIMD instructions provide a significant performance improvement, especially visible when the data stays in the L1 cache – the per-tuple cost can be even below a single CPU cycle. As a result, with optimal vector sizes, the properly compiled code can be over 10 times faster. However, the code needs to provide a relatively simple access pattern to allow such level of improvement.

### 5.2.1.4 Conclusion

The results in this section show that single improvements provide only a limited benefit. With small vector sizes, the benefits of bulk-processing are minimal, also reducing the impact of compiler optimization. In-cache data placement does not result in an improvement if the executing code is unoptimized, as the data-access cost is not the dominating factor anymore. And finally, the benefit of the bulk processing is significantly smaller for the non-cached, unoptimized code. As a result, a combination of all discussed properties is required for highly efficient code.

### 5.2.2 Choosing the data organization models

As discussed in Section 4.2.1.1 MonetDB/X100 uses single-dimensional vectors for data exchange between the operators. This section demonstrates that this layout is beneficial for sequential data access, which is an approach typically used by operators to consume and produce (but not necessarily process) data. It also discusses a number of other scenarios, where, depending on the operation and data location, either DSM or NSM can be beneficial. Finally, we outline the possibility of combining both models during the execution of a single query for an additional performance improvement. For more details on the issues described in this section, the reader is referred to [ZNB08].
5.2.2.1 Block-data representation models

When discussing the performance between NSM and DSM, it is important to define the used implementation of both models. The internal structure of systems following the same general model can vary significantly, by using different approaches to variable-width datatype storage, NULLs, compression etc. Following the block-oriented processing model of MonetDB/X100, we focus on the representation of entire blocks of tuples.

**DSM representation.** Traditionally, the Decomposed Storage Model [CK85] proposed for each attribute column to hold two columns: a surrogate (or object-id) column and a value column. Modern column-based systems [BZN05, SAB+05] choose to avoid the former column, and use the natural order for the tuple reorganization purposes. As a result, the table representation is a set of simple value arrays, each containing consecutive values from a different attribute. This format is sometimes complicated e.g. by not storing NULL values and other forms of data compression [ZHNB06, AMF06], but we assume that on the query execution level data is normalized into a contiguous sequence of values. This results in the following simple code to access a specific value in a block:

```plaintext
value = attribute[position];
```

**NSM representation.** The exact tuple format in NSM can be highly complex, mostly due to storage considerations. For example, NULL values can be materialized or not, variable-width fields result in non-fixed attribute offsets, values can be stored explicitly or as references (e.g. dictionary compression or values from a hash table in a join result). Even fixed-width attributes can be stored using variable-width encoding, e.g. length encoding [WKHM00] or Microsoft’s Vardecimal Storage Format [AD07].

Most of the described techniques have a goal of reducing the size of a tuple, which is crucial for disk-based data storage. Unfortunately, in many cases, such tuples are carried through into the query executor, making the data access and manipulation complex and hence expensive. In traditional tuple-at-a-time processing, the cost of accessing a value can be acceptable compared to other overheads, but with block processing, handling complex tuple representations can consume the majority of time.

To analyze the potential of NSM performance, we define a simple structure for holding NSM data, which results in a very fast access to NSM attributes. Tuples in a block are stored contiguously one after another. As a result, tuple
offset in a block is a result of the multiplication of the tuple width and its
index. Attributes are stored ordered by their widths (wider first). Assuming
attributes with widths of power of 2, this makes every value naturally aligned
to its datatype within the tuple. Additionally, the tuple is aligned at the end to
make its width a multiple of the widest stored attribute. This allows accessing a
value of a given attribute at a given position in the table with this simple code:

```plaintext
value = attribute[position * attributeMultiplier];
```

**Direct vs. Indirect Storage.** Variable-width datatypes such as strings cannot
be stored directly in arrays. A solution is to represent them as memory
pointers into a heap. In MonetDB/X100, a tuple stream containing string val-
ues uses a list of heap buffers that contain concatenated, zero-separated strings.
As soon as the last string in a buffer has left the query processing pipeline, the
buffer can be reused.

Indirect storage can also be used to reduce value copying between the oper-
ators in a pipeline. For instance, in MonetDB/X100, the Select operator leaves
all tuple-blocks from the data source operator intact, but just attaches an array
of selected offsets, called the *selection vector*. All primitive functions support
this optional index array:

```plaintext
value = attribute[selection[position]];
```

Other copy-reduction mechanisms are possible. For example, MonetDB/X100
avoids copying result vectors altogether if an operator is known to leave them
unchanged (i.e. columns that just pass through a Project or the left side of an
N-1 Join).

Note that the use of index arrays (selection vectors) is not limited to the
Select operator. Other possibilities include e.g. not copying the build-relation
values in a HashJoin, but instead storing references to them. In principle, each
column could have a different (or no) selection vector. This brings multiple op-
timization opportunities and challenges. For example, a single primitive can be
implemented assuming fully independent selection vectors, or provide optimized
code for cases where some of the selection vectors are shared. This might pro-
vide extra performance, but can significantly increase code size and complexity.
For this reason, these optimizations are not yet exploited in MonetDB/X100:
all columns in a dataflow share the same selection vector information.
5.2.2.2 NSM and DSM in-memory performance

This section demonstrates how the choice of storage model influences the performance of a given operation. The experimental platform used in the microbenchmarks is a Core2 Quad Q6600 2.4GHz CPU with 8GB RAM running on Linux with kernel 2.6.23-15. The per-core cache sizes are: 16KB L1 I-cache, 16KB L1 D-cache and 4MB L2 cache (shared among 2 cores).

**Sequential data access.** Figure 5.4 present the results of the experiment in which a SUM aggregate of a 4-byte integer column is computed repeatedly in a loop over a fixed dataset. The size of the data differs, to simulate different block sizes, which allows identifying the impact of the interpretation overhead, as well as the location (cache, memory) in block-oriented processing. We used GCC, using standard (SISD) processing, and additionally ICC to generate SIMD-ized DSM code (NSM does not benefit from SIMD-ization since the values to operate on are not adjacent). In the NSM implementation, we use tuples consisting of a varying number of integers, represented with $NSM-x$.

To analyze the impact of the data organization on CPU efficiency, we look at the performance of $NSM-1$, which has exactly the same memory access pattern.
and requirements as the DSM implementations. The results show that DSM, thanks to a simpler access code, can provide a significant performance benefit, especially in the SIMD case.

The other aspect of this benchmark is the impact of the interpretation overhead and data location. While for small block sizes the performance is dominated by the function calls\(^2\), for larger sizes, when the data does not fit in the L1 cache anymore, the data location aspect becomes crucial.

Looking at the performance of wider NSM tuples, we see that the performance degrades with increasing tuple width. As long as the tuples are in L1, the performance of all widths is roughly equal. However, for NSM-16 and higher (64 byte tuples or longer) once the data shifts to L2, the impact is immediately visible. This is caused by the fact that only a single integer from the entire cache-line is used. For NSM-2 to NSM-8, the results show that the execution is limited by the L2 bandwidth: when a small fraction of a cache-line is used (e.g. NSM-8) the performance is worse than when more integers are touched (e.g. NSM-2). Similar behavior can be observed for the main-memory datasets.

We see that if access is purely sequential, DSM outperforms NSM for multiple reasons. First, the array-based structure allows simple value-access code. Second, individual primitive functions (e.g. SUM, ADD) use cache lines fully in DSM, and L2 bandwidth is enough to keep up. As mentioned before, during query processing, all tuple blocks used in a query plan should fit the CPU cache. If the target for this is L2, this means significantly larger block sizes than if it were L1, resulting in a reduced function call overhead. Finally, the difference in sequential processing between DSM and NSM can be huge if the operation is expressible in SIMD, especially when the blocks fit in L1, and is still significant when in L2.

**Random data access.** Figure 5.5 demonstrates an experiment investigating the random-access performance. An input table consists of a single key column and 4 data columns, contains 4M tuples, and is stored in DSM for efficient sequential access. The range of the key column differs from 1 to 4M. We perform an experiment equivalent to this SQL query:

```sql
SELECT SUM(data1), ..., SUM(dataN)
FROM TABLE GROUP BY key;
```

\(^2\)In a real DBMS the overhead of function calls and other interpretation is significantly larger [BZN05] — this was a hard-coded micro-benchmark.
To store the aggregate results, we use a simple array with the key column as a direct index into it. In DSM, the result table is just a collection of arrays, one for each data attribute. In NSM, it is a single array of a size equal to the number of tuples multiplied by 4 (the number of data attributes). In each iteration, all values from different data attributes are added to the respective aggregates, stored at the same index in the table.

The faster access code of the DSM version makes it slightly (up to 10%) faster than NSM as long as the aggregate table fits in the L1 cache. However, once the data expands into L2 or main-memory, the performance of DSM becomes significantly worse than that of NSM. This is caused by the fact that in DSM every memory access is expected to cause a cache-miss. In contrast, in NSM, it can be expected that a cache-line accessed during processing of one data column, will be accessed again with the next data column in the same block, as all the columns use the same key position.

Figure 5.5 also shows experiments that use software prefetching: we interspersed SUM computations with explicit prefetch instructions on the next tuple block. The end result is that prefetching does improve NSM performance when the aggregate table exceeds the CPU caches, however in contrast to [CAGM07], we could not obtain a straight performance line (i.e. hide all memory latency). In general, our experience with software prefetching indicates that it is hard to use, machine-dependent, and difficult to tune, which makes it hard to apply it in generic database code.

5.2.2.3 Choosing the data model

The results from the previous section suggest that DSM should be used for all sequentially-accessed data as well as for randomly-accessed data that fits in the L1 cache, and NSM should be used for randomly-accessed data that does not fit in L1. Other model-specific optimizations might influence the choice of the used data layout. For example, in [HNZB07, ZNB08] the authors demonstrate an NSM-based technique that uses SIMD instructions to perform aggregation of values from different columns at the same time. Row-storage has also been exploited in [JRSS08] to compute multiple predicates on different columns in parallel. These optimizations demonstrate that the choice of a particular data layout while enabling some optimizations, might make other ones impossible. This problem can be partially reduced by on-the-fly format conversion, implemented either as a side-effect of performing some operation (e.g. a SUM routine reading NSM and producing DSM), or as an explicit phase [ZNB08]. Still, this approach increases the complexity of the query plan significantly and incorpo-
rating it inside an operator pipeline is an interesting challenge.

In MonetDB/X100 DSM is currently used as the only data exchange format between the operators. This is motivated by the observation that operators typically consume and produce their outputs in a sequential manner. Internally, the operators have a flexibility to choose a storage model most fitting the needs of a used algorithm. Currently, it is typically DSM, but it is expected that the future versions of e.g. HashJoin operator will be able to work with both NSM- and DSM-based data structures.

5.2.3 Decomposing data processing

The core of the vectorized system architecture is the separation of the control logic performed by the operators and the raw data processing performed in primitives. As a result, a methodology to convert a traditional algorithm implementation into a vectorized form is necessary. This problem is close to query compilation for the binary algebra of MonetDB [BK99], but it is different in the following aspects: since it needs to be adapted to the pipelined model, it goes even deeper in the operator decomposition, and additionally needs to handle the N-ary nature of the operators. As a result, expressing complex relational operators in a vectorized model is a challenge in itself.

5.2.3.1 Multiple-attribute processing

One of the main benefits of vectorized processing is the high efficiency of the primitives. To achieve this efficiency, however, the primitives are allowed very little (or no) degree of freedom - a single routine can only perform one specific task on a defined set of input types. As a result, usually a primitive is applied to perform a given function on just one or two attributes. This is enough in many cases, e.g. in the Project operator, which only adds new columns without the need for manipulating existing ones. However, in many operators, e.g. in aggregation and joins, multiple attributes need to be handled.

A typical approach to this problem is to separate the processing into two phases: one that computes some form of an index structure that is common for all attributes, and the second that uses this structure to perform some computation per attribute. For example, in hash-aggregation [ZHB06], first a position in the hash-table for each tuple is computed using all aggregate keys, and then each aggregate function is computed using it. A similar approach can be used in other operators: in hash-join and merge-join two aligned index-vectors are created,
defining matching pairs of tuples in the input relations; in radix-sort the bucket id is computed for each tuple and used to copy non-key attributes; etc.

5.2.3.2 Phase separation

Tuple-at-a-time implementations of most operators contain control logic that is hard or impossible to embed in a single efficient primitive. An example for aggregation using cuckoo-hashing has been presented in [ZHB06]. Here we will present a vectorization process for a different operator: Top-N. Let us take a look at the pseudocode for each tuple in a heap-based Top-N implementation:

```java
if (tuple.key > heap.minimum) {
    position = heap.insert(tuple.key);
    heap.values.copy(tuple, position);
}
```

Here, `heap` has a separate `values` section that contains tuple attributes not taking part in the actual heap processing. This code can be decomposed into two separate vectorized parts:

```java
selected = select_bigger(input[key], heap.minimum);
heap.process(selected)
```

The initial selection can easily be vectorized for multi-attribute keys. This approach can result in false-positives – tuples that will not enter the heap because a tuple earlier in the same vector increased the heap minimum. Still, in most cases, a large majority of tuples is filtered out with a highly efficient `select_bigger` function, making the cost of an additional check in the later phase negligible.

The next step is to decompose `heap.process` into separate primitives (ignoring false-positives for simplicity):

```java
positions = heap.insert(input[key], selected);
foreach attribute
    heap.values[attribute].copy(input[attribute], positions);
```

Here, `heap.insert` for each input tuple returns its position in the value area (freed by the expelled tuple), and the `copy()` routine copies all values for a given attribute into their positions in the `values` section.

5.2.3.3 Branch separation

The next issue in operator decomposition is handling situations where different processing steps are taken for each tuple. An example of such a situation is
hash-aggregation using a bucket-chained hash-table. Code for each tuple looks as follows:

```java
key = tuple.values[KEY];
hash = computeHash(key);
group = hash % num_groups;
idx = buckets[group];
while (idx != NULL) {
    if (key_values[idx] == key)
        break;
    idx = next[idx];
}
if (idx == NULL)
    idx = insert(group, key);
foreach aggregate
    compute_aggr(aggregate, idx, tuple);
```

The presented code is branch-intensive, making it hard for bulk processing. This problem has been identified in the context of software memory prefetching for hash-table processing, where authors annotate each tuple with a special state-identifier, and later combine stages at the same positions in different code-paths into a single stage, using tests on the tuple states to determine the actual code to execute [CAGM04]. A related technique allowing handling this issue is separating the input tuples that are at the same stage of processing into groups. Such a technique can be applied to our aggregation code, resulting in the following vectorized version:

```java
keys = input.columns[KEY];
hashes = map_hash(n, keys);
groups = map_modulo(n, hashes, num_groups);
dxs = map_fetch(n, groups, buckets);
searched = vec_sequence(n); // 0,1,...,n-1
misses = vec_empty();
found = vec_empty();
do {
    separate_misses(searched, misses, dxs);
    separate_found(searched, found, key_values, keys, dxs);
    follow_list(searched, next, dxs);
} while (searched.not_empty());
insert_misses(misses, dxs)
foreach aggregate
    compute_aggr(aggregate, dxs, input);
```

Here, `separate_misses()` extracts all tuples from `searched` for which `idxs` points to the end of the list, and saves them in `misses`. Then, `separate_found()` extracts all tuples, for which the bucket has been found (key matches). Finally, `follow_list()` updates bucket pointers in `idxs` with the next bucket in the linked list for all tuples that are neither determined as nulls nor found. This
Section 5.2: Implementing the vectorized model

process repeats while there is some tuple that needs to follow the list. Finally, all tuples in misses are inserted into the hash table, and their bucket indices are saved in idxs (we omit the details of this phase, but it needs to take care of duplicate keys in misses). Such code is beneficial for performance for two reasons: there are fewer but longer loops, and the loop code is simpler, allowing efficient execution. A vectorized hash-join implementation presented in Section 5.3 follows this approach, achieving performance comparable with a hand-written solution.

5.2.4 Primitive implementation

Once the data processing functionality is separated into primitives, the next task is to provide efficient implementations of these. In this section we analyze how to approach the problem of primitive implementation and discuss the programming techniques that allow development of CPU-friendly routines.

5.2.4.1 Primitive development and management

Due to the high primitive specialization along the data type, data representation and other dimensions, the number of different routines can be very high, making the manual implementation impractical. As a result a special language for describing primitives should be used. For example, in the MonetDB/X100 kernel, ca. 3000 lines of the Mx macro language are expanded into ca. 185,000 lines of C code implementing almost 3000 different functions.

The approach described in Section 4.2.1.3 results in a single function for each primitive signature. However, it is very well possible that on various hardware and compilation platforms different implementations of the same task can provide different performance, without a single multi-platform winner. For example, one CPU family can provide SIMD operations of some type, another can allow explicit memory prefetching, and yet another can have both capabilities – all these platforms might require different implementation approaches for optimal performance. This problem is addressed e.g. in an open-source LibOIL library [Lib] that provides multiple specialized implementations for a small set of typical data processing tasks, and at runtime determines which one to use based on CPU capabilities and micro-benchmarks. A vectorized DBMS can follow this approach to optimize performance of primitives that are execution bottlenecks. This idea can be extended even further to exploit dynamic data properties. For example, in the Select operator different approaches can be optimal depending
on the selectivity $\text{[Ros02]}$, and the runtime optimizer can dynamically choose the best primitive implementation.

### 5.2.4.2 Control dependencies

Deep execution pipelines in modern CPUs cause severe performance degradation in case of branch mispredictions. In the example code in Section 5.1.4, the only branch taken is the loop control. While this branch is easy to predict, hence relatively cheap, compilers usually further reduce its cost by applying loop unrolling.

Let us look at another routine that selects out the indices of tuples bigger than a given constant:

```c
for (i = 0, found = 0; i < n; i++)
    if (input[i] > val)
        result[found++] = i;
return found;
```

As analyzed in $\text{[Ros02]}$, such code is efficient only for very low or very high selectivities due to branch mispredictions. In this case, and in many others, the control dependency can be replaced with a data dependency, resulting in the following routine:

```c
for (i = 0, found = 0; i < n; i++) {
    result[found] = i;
    found += (input[i] > val);
}
return found;
```

While issuing more instructions, this approach does not have a hard-to-predict ‘if’, and results in a significant performance improvement, as discussed in $\text{[Ros02]}$ and confirmed in Figure 5.6. Another possible approach for complex, branch-heavy routines, is to separate tuples going into different code paths, as discussed in Section 5.2.3.3.

### 5.2.4.3 Data dependencies

Some of the control-dependency solutions involve replacing them with data-dependencies. Such dependencies can also be inherent to a data processing task. A typical case is aggregation – for example, a routine that increases the $\text{COUNT}$ values stored in $\text{result}$ looking at the group identifiers from $\text{groupids}$ looks like this:
Section 5.2: Implementing the vectorized model

Figure 5.6: Performance of control-dependency based and data-dependency based selection routines (Core 2 Duo)

```
void aggr_count_int_vec_int_vec(int *result, int *groupids, int n) {
    for (int i = 0; i < n; i++)
        result[groupids[i]] += 1;
}
```

In this code, each tuple depends on the previous one, causing data stalls in the CPU pipeline. One approach to reduce these stalls, is to use multiple copies of the `result` array, and make different tuples update different versions of it.

```
void aggr4_count_int_vec_int_vec(int **result, int *groupids, int n) {
    for (int i = 0; i < n; i += 4) {
        result[0][groupids[i+0]] += 1;
        result[1][groupids[i+1]] += 1;
        result[2][groupids[i+2]] += 1;
        result[3][groupids[i+3]] += 1;
    }
}
```

The latter solution, while minimizing data dependencies between iterations, increases the memory consumption for ‘result’ arrays by a factor 4. Still, if such extra cost is acceptable, this approach allows for a significant performance improvement. For example, on our Core2Duo test machine it improved the performance from already very good 2.76 cycles/tuple (with 256 groups) to 2.05 cycles/tuple. On some architectures this difference can be significantly larger.

Another solution to the data dependency problem is to combine multiple operations into one primitive. For example, in some scenarios, multiple aggregates are computed at one processing stage – such a situation occurs in TPC-H query 1 [HNZB07]. Then, it is possible to compute e.g. 4 aggregates in one primitive:

```
void multiaggr_sum_int_vec4_int_vec(int **result, int **values, int *groupids, int n) {
    for (int i = 0; i < n; i++) {
```
This solution, similarly to the previous routine, reduces the data dependencies and improves the performance. One of the major problems here is the use of the same data type for all 4 aggregations, which limits its applicability. Still, in scenarios like data mining, with queries often computing dozens of aggregates at once, this technique can be beneficial.

### 5.2.4.4 SIMDization

SIMD instructions allow processing multiple elements with one CPU instruction. Originally, they were designed to improve multimedia processing and scientific computing, but they have also been suggested for the databases [ZR02]. While having a large potential, SIMD instructions suffer from two limitations important for database processing. First, usually SIMD instructions can only operate on a set of values sharing the same data type, and the data types are usually limited to 32/64 bit integers and floats. Secondly, in most ISAs, SIMD write/load instructions usually do not have scatter/gather functionality, making them only useful for fully sequential data processing.

Overcoming the problem of datatypes is sometimes possible by casting a column to a different datatype (e.g. a character into an integer, or a float into a double). As for the strict sequential data locality, one of the solutions is to use an alternative data representation. In the previous example we used data storage known in the SIMD world as Structure-of-Arrays (SOA). It is possible to further extend it to use 'Array-of-Structures' (AOS), as presented in Figure 5.7. Note a parallel between SOA-AOS and DSM-NSM. AOS can be seen as a subset of NSM that holds data tightly packed for efficient SIMD processing. This approach has been previously presented in the context of database processing on the Cell processor [HNZB07]. As a result, our multi-aggregation code from the previous section becomes:

```c
void multiaggr_sum_int4_vec_int_vec(int4 *result,
      int4 *values, int *groupids, int n) {
  for (int i = 0; i < n; i++)
    result[groupids[i]] = SIMD_add(result[groupids[i]], values[i]);
}
```
5.3 Case study: Hash-Join

This section demonstrates how presented techniques can be used to implement a vectorized version of a hash-join, one of the most important database algorithms. Initially, we present a relatively straightforward hash-join implementation – the next section will introduce a set of additional optimizations.

5.3.1 Problem definition

Hash-join is one of the physical implementations of the relational equi-join operator, which is a specialization of the generic join operator. Formally, any join between relations $R$ and $S$ can be represented as: $R \bowtie_\phi S = \sigma_\phi (R \times S)$. Here, $\phi$ is a join condition, for equi-join represented as: $\phi \equiv (rkey_1 = skey_1 \land \ldots \land rkey_n = skey_n)$, where $rkey_i$ and $skey_i$ are the key attributes from $R$ and $S$ respectively. The most often used version of an equi-join is an $N$-1 join, where keys in $S$ are unique, and for every tuple in $R$ there is exactly 1 matching tuple in $S$. We will assume this type of join in the remainder of this section.

5.3.2 Standard implementation

In the hash-join, first a build relation $S$ is used to construct a hash-table containing all the tuples from $S$ indexed on the key of $S$. In the second phase, the key of every tuple from the probe partition $R$ is looked up in that hash-table, and the result tuples are constructed. The following code performs the described process for a simple case, where the input relations build and probe, each with three attributes are joined, where the first two attributes (0 and 1) constitute the key. The data is stored as simple arrays, and a new, two-column result relation is produced containing only the values of the non-key attribute from
Figure 5.8: Simple bucket-chained hash table, using modulo 5 as a hash function

both inputs. We use a simple bucket-chained hash-table, presented in Figure 5.8. Here, the next array represents the linked list of all tuples falling into a given bucket, with a value 0 reserved for the end of the list.

```
// Build a hash table from "build"
for (i = 0; i < build.size; i++) {
    bucket = rehash(hash(build.values[0][i]), build.values[1][i]) & mask;
    hashTable.values[0][i + 1] = build.values[0][i];
    hashTable.values[1][i + 1] = build.values[1][i];
    hashTable.values[2][i + 1] = build.values[2][i];
    hashTable.next[i + 1] = hashTable.first[bucket];
    hashTable.first[bucket] = i + 1;
}

// Probe the "probe" relation against the hash table
for (i = 0; i < probe.size; i++) {
    bucket = rehash(hash(probe.values[0][i]), probe.values[1][i]) & mask;
    current = hashTable.bucket[bucket];
    while (hashTable.values[0][current] != probe.values[0][i] || // assume eventual hit
        hashTable.values[1][current] != probe.values[1][i]) {
        current = hashTable.next[current];
    }
    result.values[0][i] = probe.values[2][i];
    result.values[1][i] = hashTable.values[2][current];
}
```

Note that this is a hard-coded implementation for double-key, single-value relations with known attribute data types. A real system needs to be able to perform a join on any combination of relations, including multi-key attributes with different data types, as well as different numbers of attributes. Clearly, even using macro expansions, providing the hard-coded version for all the input combinations is impossible. The following section will demonstrate how, looking at this algorithm, a generic high-performance vectorized operator can be realized.
5.3.3 Vectorized implementation

The vectorized implementation of the hash-join should be able to consume entire vectors with tuples and process them following the principles discussed in Section 5.2.1. The implementation in MonetDB/X100 provides most of the desired properties, and is based on the following observations:

- During the build phase, the processing for different tuples in a vector is not fully independent. If multiple keys fall into the same bucket, they need to be processed one after another. This can cause some data dependency, but it is not possible to avoid it with this hash table organization.

- During the probe phase, processing of different tuples is fully independent, thanks to the assumption of the $N$-$I$ join: each probe tuple generates exactly one result tuple, hence the location of each result tuple is known.

- Finding the position in the hash table is a one-time investment for every tuple, during both build and probe phases. Once done, it allows quick insertion or lookup of multiple attributes.

- Following the linked list in the inner loop during the probe phase might take different number of steps for different tuples. Also, it introduces data and control dependencies, which are bad for modern CPUs, and makes it impossible for this code to overlap the cache misses that might occur during the linked list traversal.

5.3.3.1 Build phase

The vectorized implementation of the build phase follows closely the hard-coded version presented above, but uses vectors of size $n$ as input and allows arbitrary column combinations in the input. The simplified code is as follows:

```c
// Input: build relation with N attributes and K keys
// 1. Compute the bucket number for each tuple, store in bucketV
for (i = 0; i < K; i++)
    hash[i](hashValueV, build.keys[i], n); // type-specific hash() / rehash()
    modulo(bucketV, hashValueV, numBuckets, n);
// 2. Prepare hash table organization, compute each tuple position in groupIdV
hashTableInsert(groupIdV, hashTable, bucketV, n)
// 3. Insert all the attributes
for (i = 0; i < N; i++)
    spread[i](hashTable.values[i], groupIdV, build.values[i], n);
```
The first task during the build phase is to find the bucket number for each build tuple. To support processing of arbitrary number and combination of key attributes, this phase is decomposed into a set of steps, as follows:

- Compute the `hashValueV` vector using a `hash*` (e.g. `hash_slng`) primitive computing a type-specific hash-function, using the first key column as a parameter.

- Adjust the `hashValueV` vector by applying a type-specific `rehash*` primitive that combines an existing hash value with a hash value for the second key column. Repeat for the remaining key columns.

- Compute the `bucketV` vector containing the bucket number for each tuple using a modulo (or and) primitive.

The resulting `bucketV` is the vectorized equivalent of the `bucket` variable in the previous section. Having this, it is possible to apply the insertion process to all tuples. In step 2 in the algorithm, the hash-table organization is prepared by adjusting the `first` and `next` arrays:

```c
hashTableInsert(groupIdV, hashTable, bucketV, n) {
    for (i = 0; i < n; i++) {
        groupIdV[i] = hashTable.count++;
        hashTable.next[groupIdV[i]] = hashTable.first[bucketV[i]];
        hashTable.first[bucketV[i]] = groupIdV[i];
    }
}
```

At the same time, the `groupIdV` vector is computed, holding for each input tuple its position in the hash table. In step 3, all the input attributes are inserted into the matching positions in the hash table with type specific `spread` functions:

```c
spread(hashTableValues, groupIdV, inputValues, n) {
    for (i = 0; i < n; i++)
        hashTableValues[groupIdV[i]] = inputValues[i];
}
```

### 5.3.3.2 Probe phase

The probe phase has two problems making it especially challenging. First, during the linked list traversal, equality comparisons can be arbitrarily complex, depending on the key structure. Secondly, the linked list traversal seems to require a per-tuple loop that would internally need to perform this complicated equality check.
In the MonetDB/X100 implementation of this phase we exploit the fact that while the inner loop length for different tuples can significantly differ, the number of steps is limited, and most tuples need to check only one or two elements in the hash table. This allows us to modify the way the linked list is traversed for all the tuples. We first find the first element in the list for every tuple. Then, we compare if these elements match our probe keys. For tuples that have a value difference, we find the next element in the list and repeat the process.

```c
// Input: probe relation with M attributes and K keys, hash-table containing
// N build attributes
// 1. Compute the bucket number for each probe tuple.
// ... Construct bucketV in the same way as in the build phase ...
// 2. Find the positions in the hash table
// 2a. First, find the first element in the linked list for every tuple,
// put it in groupIdV, and also initialize toCheckV with the full
// sequence of input indices (0..n-1).
lookupInitial(groupIdV, toCheckV, bucketV, n);
m = n;
while (m > 0) {
    // 2b. At this stage, toCheckV contains m positions of the input tuples for
    // which the key comparison needs to be performed. For each tuple
    // groupIdV contains the currently analyzed offset in the hash table.
    // We perform a multi-column value check using type-specific
    // check() / recheck() primitives, producing differsV.
    for (i = 0; i < K; i++)
        check[i](differsV, toCheckV, groupIdV, hashTable.values[i], probe.keys[i], m);
    // 2c. Now, differsV contains 1 for tuples that differ on at least one key,
    // select these out as these need to be further processed
    m = selectMisses(toCheckV, differsV, m);
    // 2d. For the differing tuples, find the next offset in the hash table,
    // put it in groupIdV
    findNext(toCheckV, hashTable.next, groupIdV, m);
}
// 3. Now, groupIdV for every probe tuple contains the offset of the matching
// tuple in the hash table. Use it to project attributes from the hash table.
// (the probe attributes are just propagated)
for (i = 0; i < N; i++)
gather[i] (result.values[M + i], groupIdV, hashTable.values[i], n);
```

### 5.3.3.3 Performance

We have experimentally analyzed the performance of the presented algorithm by comparing it with the hard-coded routines presented in the previous section. The performance of the vectorized implementation is tested with 2 vector sizes: 1 tuple, which simulates tuple-at-a-time approach, and 1024 tuples. Two 2- and 3-attribute relations were used, with 1- and 2-attribute keys, respectively. The probe relation always contains 4M tuples, all having a matching key in the build
Figure 5.9: Comparison of a hard-coded hash-join implementation with the generic vectorized implementation in MonetDB/X100 (Core 2 Quad, 2.4GHz)

relation. The build relation, and hence the hash table, contains from 16 to 4M tuples with unique keys.

As Figure 5.9 shows, for cache-resident hash tables the performance of the generic MonetDB/X100 version is only ca. 2 times slower than hard-coded, specialized implementation. Surprisingly, once the hash table does not fit in the cache anymore, MonetDB/X100 implementation is faster than the hard-coded one. This is caused by the fact that all the operations in the vectorized version are independent, allowing e.g. overlapping of main-memory accesses. In the hard-coded version, control- and data-dependencies do not allow it, making the impact of cache-misses higher. The tuple-at-a-time implementation suffers from significant interpretation overhead, but is also less sensitive to the hash-table size. As a result, while the vectorized version provides a 30-times improvement for cache-resident data, this improvement goes down to factor 7 on memory-resident data. This demonstrates the importance of combining CPU-efficient vectorized execution with cache-optimized data placement, discussed in the next section.

5.4 Optimizing Hash-Join

The vectorized hash-join implementation demonstrated in the previous section achieves high in-cache efficiency, but suffers from significant performance degra-
Section 5.4: Optimizing Hash-Join

dation when working on RAM-resident data, caused by random memory accesses related to the linked list traversal. This problem can be reduced by using hashing methods that do not need a linked list, for example cuckoo hashing [PR04], as discussed in [ZHB06]. Still, even with this improvement the overhead of cache-misses can dominate the cost of per-tuple processing. Two main techniques were previously proposed to address this problem.

The first technique, proposed by Chen et al., uses explicit memory prefetching instructions inside the hash lookup routine [CAGM04]. This transforms hash-lookup throughput from a memory latency-limited into a memory band-width-limited workload, which can strongly improve overall hash-join performance. Our CPU-optimized hashing, however, has become too fast for memory bandwidth. Optimized cuckoo-hashing implementation from [ZHB06] spends only 7 CPU cycles per lookup and touches at least two cache lines. On a 1.3GHz CPU this implies bandwidth usage of 24GB/s, which exceeds the available RAM bandwidth. For that reason, we employ the second technique, based on hashtable partitioning. This idea was originally introduced for I/O based hashing in Grace Join [FKT86] and Hybrid Hash Join [DKO+84] algorithms. More recently, with Radix-Cluster [MBK02], this work has been extended to hash-partitioning into the CPU cache.

The problem with these partitioned hashing techniques is that all the data needs to be first fully partitioned, and only then processed [Gra93]. This works fine in the disk-based scenario, as the temporary space for the partitions is usually considered unlimited. Main memory capacity, however, cannot be assumed to be unlimited, meaning that if the data does not fit in RAM during partitioning, it has to be saved to disk. Since using the disk when optimizing for in-cache processing is reasonable only in extreme scenarios, we propose a new hash partitioning algorithm that, while providing in-cache processing, prevents spilling data to disk.

5.4.1 Best-Effort Partitioning

Best-effort partitioning (BEP) is a technique that interleaves partitioning with execution of hash-based query processing operators without using I/O. The key idea is that if the available partition memory is filled, data from one of the partitions is passed on to the processing operator (aggregation, join), freeing space for more input tuples. In contrast to conventional partitioning, BEP is a pipelinable operator that merely reorders the tuples in a stream so that many consecutive tuples come from the same partition. Operators that use BEP, like Partitioned Hash Join and Partitioned Hash Aggregation, create a separate
InitBuffers(numBuffers)
while tuple = GetNextTuple(child)
    p = Hash(tuple) mod numPartitions
    if MemoryExhausted(p)
        if NoMoreBuffers()
            maxp = ChooseLargestPartition()
            ProcessPartition(maxp)
            FreeBuffers(maxp)
            AddBuffer(p)
    Insert(p, tuple)
for p in 0..numPartitions - 1
    ProcessPartition(p)
    FreeBuffers(p)

Figure 5.10: Best-Effort Partitioning (BEP) algorithm

hash table per partition, and detect which hash table should be used at a given moment looking at the input tuples. When one of the hash tables is active, the operations on it are performed for many consecutive tuples, hence the cost of loading the hash-table into the cache is amortized among them.

Interestingly, the consuming operator can still benefit from BEP even with a single hash table, because of improved temporal locality of accesses. Still, the benefit will be significantly smaller, as memory related to the current partition is not “dense”, and some space in fetched cache lines might be wasted by data of the other partitions.

An algorithm from Figure 5.10 presents an implementation where each partition consists of multiple buffers. When no more buffers are available, we choose the biggest partition to be processed, for two reasons. Firstly, it frees most space for the incoming tuples. Secondly, with more tuples passed for processing, the time of loading the hash-table is better amortized due to increased cache-reuse.

5.4.2 Partitioning and cache associativity

The main-memory performance of data partitioning algorithms, with respect to the number of partitions, number of attributes, sizes of the CPU cache and TLB has been studied in [MBK02] and [MBNK04]. However, to our knowledge, one other important property of modern cache memories has been ignored so far: cache associativity. As discussed in Section 2.2.2, cache memories typically are not fully associative, but rather N-way associative. As a result, for different
addresses with the same bits used to determine the set id there are only $N$ possible locations in the cache. For example, Figure 5.11 presents a 2-way associative 64KB cache with 64-byte cache lines – there are 512 sets, determined by bits 6..14 (mask 0x7fc0 of the memory address, and 2 cache-lines in each set.

This limitation on the number of possible locations in the cache can significantly influence the partitioning performance. This can be demonstrated by the analysis of this simple partitioning function:

```c
for (i = 0; i < n; i++) {
    partno = HASH_TYPE(src[i]) & PARTITION_MASK;
    dst[partno][counts[partno]++] = src[i];
}
```

It is a common situation that the addresses of the dst buffers are aligned to the page size. As a result, using the cache from Figure 5.11 and a page size of 8KB, all these addresses will map onto only 4 separate cache addresses, each holding 2 cache-lines. This means that if we partition into more than 8 buffers, there is a high probability that, when we refer to a buffer that has been recently used, the cache-line with its data has already been replaced, possibly causing a cache-miss. Since the partitioning phase is usually performed using hash-values, data is roughly uniformly distributed among partitions. As a result, this cache associativity thrashing may continue during the entire execution of this primitive. Since the previous experiments with Radix-Cluster [MBK02] were primarily performed on a computer architecture where high fan-out partitioning deteriorated due to slow (software) TLB miss handling, these issues had previously not been detected.
Figure 5.12: Impact of number of partitions and buffer allocation method on partitioning performance on various hardware architectures

A simple solution for the cache associativity problem is to shift each buffer address with a different multiple of a cache line size, such that all map to different cache offsets. Figure 5.12 presents the performance of the partitioning phase with both aligned and non-aligned buffers on Pentium Xeon and Itanium2 CPUs. As the number of partitions grows, the performance of aligned buffers goes down, quickly approaching the cost of random-memory access per each tuple. The non-aligned case, on the other hand, manages to achieve speed comparable to simple memory-copying even for 256 partitions. When more partitions are needed, it is possible to use a multi-pass partitioning algorithm [MBK02]. BEP can be easily extended to handle such a situation.

5.4.3 BEP performance

Performance of hash processing with best-effort partitioning is influenced by a number of factors presented in Table 5.2. The first group, data and query properties define the number of tuples stored in a hash table and their width, determining a size of the hash table. The second group, partitioning settings, determine the size of per-partition hash tables. Finally, the hardware factors influence the recommended size of the small hash tables, hence the partitioning fan-out. Moreover, cache and memory latencies influence the desirable cache-reuse factor, which determines the amortized cost of data access.
Section 5.4: Optimizing Hash-Join

Figure 5.13: Aggregation performance with varying number of partitions and distinct keys (20M tuples)

Figure 5.14: Execution profiling with varying number of partitions and distinct keys (20M tuples)
Table 5.2: Best-Effort Partitioning parameters

<table>
<thead>
<tr>
<th>Description</th>
<th>Symbol</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of distinct values</td>
<td>$D$</td>
<td>1 M</td>
</tr>
<tr>
<td>Number of tuples</td>
<td>$T$</td>
<td>20 M</td>
</tr>
<tr>
<td>Input width</td>
<td>$\hat{i}$</td>
<td>4 B</td>
</tr>
<tr>
<td>Hash-table: data width</td>
<td>$\hat{h}_d$</td>
<td>4 B</td>
</tr>
<tr>
<td>Hash-table: buckets width</td>
<td>$\hat{h}_b$</td>
<td>8 B</td>
</tr>
<tr>
<td>Hash-table: per-key memory =</td>
<td>$\hat{h}_w$</td>
<td>20 B</td>
</tr>
<tr>
<td>$\hat{h}_d + 2 \cdot \hat{h}_b$ (Cuckoo, 50% fill ratio)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash-table: size =</td>
<td>$</td>
<td>H</td>
</tr>
</tbody>
</table>

**BEP settings**

| Available buffer memory               | $|M|$ | 30 MB |
| Number of partitions                  | $P$  | 16     |
| Partition: size = $\frac{|M|}{P}$     | $|M_p|$ | 1.875 MB |
| Partition: tuples buffered = $\frac{|M_p|}{P}$ | $T_p$ | 480 K |
| Partition: hash-table size = $\frac{|H|}{P}$ | $|H_p|$ | 1.25 MB |
| Number of per-lookup random accesses (Cuckoo) | $a$ | 4 |

**Hardware properties (Example = Itanium2)**

| Cache size                           | $|C|$ | 3 MB |
| Cache line width                     | $\hat{C}$ | 128 B |
| Cache latency                        | $l_C$ | 14 cycles |
| Main-memory latency                  | $l_M$ | 201 cycles |

We now discuss in detail one particular scenario of using BEP for partitioned hash aggregation. This setting is later used in experiments on our Itanium2 machine. The relevant hardware and algorithm parameters are listed in Table 5.2 which in its rightmost column also contains the specific hardware characteristics of Itanium2. Note that Itanium2 has a large and fast L3 cache, which is the optimization target (in case of Pentium4, it is best to optimize for L2).

**Example Scenario.** Assume we need to find 1M unique values in a 20M single-attribute, 4-byte long tuples using 50MB of RAM on our Itanium2 machine with a 3MB L3 cache with 128-byte cache-lines. A hash table with a load factor of 0.5 occupies 20MB using optimized single-column Cuckoo Hashing \cite{ZHB06}: 16MB for the bucket array and 4MB for the values. Using 16 partitions will divide
Section 5.4: Optimizing Hash-Join

Figure 5.15: Impact of available buffer space (20M tuples, 1M unique values)

Experiments. Figure 5.13 compares in a micro-benchmark naive (non-partitioned) and best-effort partitioning hash aggregation, in a "SELECT DISTINCT key FROM table" query on a 20M 4-byte wide tuples table, with a varying number of distinct keys. When this number is small, the hash table fits in the CPU cache, hence the partitioning only slows down execution. When the number of keys grows, the hash table exceeds the cache size, and best-effort partitioned execution quickly becomes fastest. Figure 5.14 shows a performance break-down into partitioning cost, hash table maintenance (lookup and inserts) and hash function computation. With more partitions, the data locality improves, making the hash table maintenance faster. On the other hand, more partitions result in a slower partitioning phase. Finally, we see that with partitioned execution the cost of the hash-function is two times higher, as it is computed both in...
partitioning and lookup phases. Depending on the cost of computing this function (especially when it is computed over multiple attributes), it can be more beneficial to store it during partitioning and reuse it during lookup.

The performance of partitioned execution depends highly on the cache-reuse ratio during one processing phase, which in turn depends on the amount of buffer space. As Figure 5.13 shows, with an increasing number of buffered tuples, performance improves since more tuples hit the same cache line. If the number of partitions is big enough to make the hash table fit in the cache, adding more partitions does not change performance given the same buffer space. Finally, we see that the performance curve quickly flattens, showing that the performance can be close to optimal with significantly lower memory consumption. In this case, processing time with a buffer space of only 2M tuples is the same as with 20M tuples (which is equivalent to full partitioning). We see this reduced RAM requirement as the main advantage of best-effort partitioning.

**Cost Model.** We now formulate a cost model to answer the question “what is the amount of buffer memory that should be given to BEP to achieve (near) optimal performance?”

The cost of the amortized average data access cost during hash-table lookup depends on the cache-reuse factor:

\[
access\_cost = l_C + \frac{l_M}{\text{reuse\_factor}}
\]

The cache-reuse factor is the expected amount of times a cache line is read while looking up in the hash table all tuples from a partition. It can be computed looking at the query, partitioning and hardware properties from Table 5.2:

\[
\text{reuse\_factor} = \frac{T_p \cdot a \cdot \hat{C}}{|H_p|} = \frac{|M| \cdot a \cdot \hat{C}}{\hat{i} \cdot D \cdot \hat{h_w}}
\]

A good target for the cache-reuse factor is to aim for an amortized RAM latency close to the cache performance, for example 25% higher:

\[
\frac{l_M}{\text{reuse\_factor}} = \frac{l_C}{4}
\]

This, in turn, allows us to compute the required amount of memory BEP needs:

\[
|M| = \frac{l_M \cdot 4 \cdot \hat{i} \cdot D \cdot \hat{h_w}}{l_C \cdot a \cdot \hat{C}}
\]
In the case of our Itanium2 experiments we arrive at:

\[ |M| = \frac{201 \cdot 4 \cdot 4 \cdot 1M \cdot 20}{14 \cdot 4 \cdot 128} = 9,409,096 \text{ B} = 2,352,274 \text{ tuples} \]

and in case of Pentium 4:

\[ |M| = \frac{370 \cdot 4 \cdot 4 \cdot 1M \cdot 20}{24 \cdot 4 \cdot 128} = 10,103,464 \text{ B} = 2,525,866 \text{ tuples} \]

This prediction is confirmed in Figure 5.15 where a buffer of 2M tuples results in the optimal performance.

As a final observation, it is striking that the amount of partitions does not play a role in the formula. The cost model does assume, though, that the hash table fits in the CPU cache. This once again is confirmed in Figure 5.15 which shows that once partitions are small enough for them to fit in the CPU cache, performance does not change. Note that on Pentium4, the 16 partition line is in the middle, because at that setting the hash-tables \((20\text{MB}/16 = 1.25\text{MB})\) are just a bit too large to fit L2, but average latency has gone down with respect to pure random access.

### 5.4.4 BEP discussion

Best-effort partitioning can be easily applied to various relational operations. In aggregation, the ProcessPartition() function simply incrementally updates the current aggregate results. In joins and set-operations, the regular partitioning can first be used for the build relation, and then BEP can be applied for the probe relation. This allows, for example, cache-friendly joining of two relations if only one of them fits in main memory. This can be further extended to multi-way joins using hash teams [GBC98].

The flexibility of BEP memory requirements is useful in a scenario where the memory available for the operator changes during its execution. If the memory manager provides BEP with extra memory, it can be simply utilized as additional buffer space. If, on the other hand, available memory is reduced, BEP only needs to pass some of the partitions to the processing operator and free the buffers they occupied.

The ideas behind BEP can be applied in a scenario with more storage levels, e.g. in a setup with a fast flash drive and a slow magnetic disk. If the hash-table does not fit in main memory, and the partitioned data is too large to fit on a flash drive, BEP can be used to buffer the data on a flash device and periodically process memory-size hash tables, possibly again using BEP to make
it cache-friendly. This scenario raises the question whether it is possible to build a *cache-oblivious* data structure \cite{FLPR99} with properties similar to those of BEP.

BEP is related to a few other processing techniques besides vanilla data partitioning. *Early aggregation* \cite{Lar97} allows computing aggregated results for part of the data and later join combine them. In parallel *local-global aggregation* \cite{Gra93}, tuples can be distributed using hash-partitioning among multiple nodes. If the combined memory of these nodes is enough to keep the whole hash table, I/O-based partitioning is not necessary. In *hybrid hashing* \cite{DKO+84}, the effort is made to keep as much data in memory as possible, spilling only some of the partitions to disk. While there are clearly similarities between BEP and these techniques, BEP provides a unique combination of features: (i) it allows efficient processing if the data does not fit in the first-level storage (cache), (ii) it optimizes data partitioning for a limited second-level storage (main memory), (iii) it allows a non-blocking partitioning phase, and, finally, (iv) it can be easily combined with dynamic memory adjustments.

### 5.5 Extending the vectorized world

One of the concerns related to vectorized processing is that originally it has been limited to pure numeric processing, ignoring many issues crucial to database performance, but often neglected in research. In this section we discuss how vectorized processing can be applied in some of these areas.

#### 5.5.1 Overflow checking

Arithmetic overflows are rarely analyzed in the database literature, but they are a necessity in a production-quality system. While CPUs do detect overflows, many programming languages (e.g. C++) do not provide mechanisms to check for them. As mentioned in Section 5.1.3 on some platforms, a special *summary overflow* processor flag can be checked to detect an overflow over a large set of computations. Still, mainstream Intel and AMD CPUs do not have such capabilities, and software solutions need to be applied to this problem. One of the approaches is to cast a given datatype into a larger one, and check if the result of the arithmetic operation fits into the smaller datatype range. A simple overflow-handling addition primitive for unsigned integers could then look like this.
int map_add_int_vec_int_vec(uint *result, uint *input1, uint *input2, int n) {
    for (int i = 0; i < n; i++) {
        ulong l1 = input1[i];
        ulong l2 = input2[i];
        ulong res = l1 + l2;
        if (res > 0xFFFFFFFFUL)
            return STATUS_ERROR;
        result[i] = (uint)res;
    }
    return STATUS_OK
}

Note again that the overflow check is performed for every tuple. An optimized
vectorized version could look like this:

int map_add_int_vec_int_vec(int *result, uint *input1, uint *input2, int n) {
    ulong tmp = 0;
    for (int i = 0; i < n; i++) {
        ulong l1 = input1[i];
        ulong l2 = input2[i];
        ulong res = l1 + l2;
        tmp |= res;
        result[i] = (int)res;
    }
    if (tmp > 0xFFFFFFFFUL)
        return STATUS_ERROR;
    return STATUS_OK;
}

While the check in the first version can be perfectly predicted, it still causes
some overhead. As a result, removing it in the second version gives us a 25%
boost on our test Core2Duo machine.

5.5.2 NULL handling

NULL handling is another often ignored issue in performance-focused research.
There are different options for NULL representation in data, including a reserved
NULL value, a list of positions with (or without) NULL values and a bitmap with
a bit set for every NULL value. In this section we demonstrate an example where
vectorization improves the NULL handling with the bitmap representation.

Figure 5.16 demonstrates the performance of different possible implementa-
tions of a NULL-handling integer addition primitive. Both inputs use a data
representation with a “value” vector holding all the values (including unde-
defined values for NULLs), and a “bitmap” vector holding the NULL bitmap.
The first, “iterative” version, for every tuple checks if the proper bit in either
input bitmap is set. If so, it sets the bit in the destination NULL bitmap, and
if not, performs the actual addition. The “every-8” and “every-4” versions performs a trick mentioned in Section 5.1.5: they check 8 or 4 bits at once in both inputs, and if either is non-zero, they perform the slow “iterative” code. This can be beneficial with a very small number of NULL values. Finally, the “full-computation” version exploits the fact that in many cases performing the computation for the NULL values does not cause an error, as long as the result value is marked as NULL. First, it creates a destination bitmap by simply binary-ORing the source bitmaps. Then, it performs the addition for all tuples. Both loops can be very efficiently optimized by the compiler, resulting in a 3 to 8 times faster implementation. Clearly, this aggressive approach is not applicable in every case – for example, dividing by a NULL tuple that has a zero in the value vector can cause an error. Also, when the performed computation is expensive, the extra performed operations can outweigh the benefit of removing the comparison. Still, it is a nice example of the improvement possible with vectorized processing.

5.5.3 String processing

An obvious application of vectorized processing for strings is in areas where there exist some (possibly approximate) fixed-width string representations, for example hash values or dictionary keys for strings from limited domains. In
such a case, for example in a string equality-search, vectorized processing can be used to perform efficient filtering-out of the non-matching strings using integer processing, possibly followed by the expensive full-string comparison for a subset of tuples.

Even when working on real strings, it turns out there are cases when vectorization can help. Figure 5.17 presents an experiment in which we perform string-equality search, using a 20-byte long key, and a collection in which half of the non-matching strings share some prefix with the key. We use a standard zero-terminated C string representation. We compare three types of implementations: one based on the standard `strcmp()` function, one in which the `strcmp()` functionality is inlined, and a vectorized one. In the vectorized implementation, instead of comparing the entire strings one after another, we first select out strings with a matching first character, then from these we select strings with a matching second character, and so on. Additionally, for the inlined and vectorized versions, we applied an additional optimization (“-opt4”), in which we compare 4 bytes in one step as integers (except for the last few bytes of the pattern). This technique is safe, as long as it is known that the 3 bytes after each string are safe to be addressed by the user process, which can be guaranteed by the memory allocation and buffer management facilities.

Figure 5.17 demonstrates that the standard `strcmp()` function is quite efficient and can beat the inlined version. Still, it loses with the vectorized solution for selectivities ranging from 0% to 25%. The main reason of this difference comes from the fact that the vectorized version performs fewer but longer loops.
over the data, reducing the loop management overhead. If the 4-byte comparison trick is applied, the vectorized version becomes a clear winner, providing as much as 2- and 4- times improvement over optimized inlined and default \texttt{strcmp()} implementation, respectively.

While not conclusive, this simple experiment shows that vectorization can be efficiently applied to some string processing problems. We believe that this research area, while relatively unexplored, has a potential for more significant improvements.

5.5.4 Binary search

Finding an element in an ordered sequence is a problem present in many data processing tasks. In databases, it occurs e.g. when searching for an element in a sorted dictionary, finding the next pointer during the B-tree traversal, performing merge join and more. Typically, binary search is used to implement this task. It is a very well researched problem, often being used as an example of algorithm design and analysis [Ben99]. Let us define this problem as finding an element \texttt{key} in an ordered array \texttt{data} of size \texttt{N}, returning the value \texttt{p} equal to a position in \texttt{data} of a found element, or \texttt{-1} if the element does not occur in the array. We extend this problem to a vectorized case, when we need to perform multiple binary searches for all values in a \texttt{keys} arrays of size \texttt{M}, and store the positions in the \texttt{result} array. This “bulk” implementation based on the binary search version from [Ben99] looks as follows:

```c
// BINARY SEARCH - SIMPLE
for (i = 0; i < M; i++) {
    key = keys[i];
    l = -1;
    u = N;
    while (l + 1 != u) {
        m = (l + u) / 2;
        if (data[m] < key)
            l = m;
        else
            u = m;
    }
    result[i] = u;
}
```

One possible optimization of this \texttt{SIMPLE} algorithm, discussed in [Ben99], exploits the fact that if the array size is a power of 2, the code can be simpler, as division by two never results in rounding errors. We used this technique to
implement a solution for a simplified binary search problem, when the key is guaranteed to be in the data:

```
// BINARY SEARCH - IMPROVED
powerOfTwo = pow(2, floor(log2(N - 1))); // biggest power of 2 smaller than N
splitIndex = N - powerOfTwo; // used to divide a problem into two problems
splitValue = data[splitIndex]; // with sizes guaranteed to be powers of 2
for (i = 0; i < M; i++) {
    key = keys[i];
    if (key >= splitValue)
        p = splitIndex;
    else
        p = 0;
    for (j = powerOfTwo / 2; j >= 1; j = j / 2)
        if (key >= data[p + j])
            p = p + j;
    result[i] = p;
}
```

Figure 5.18 presents results of an experiment, in which we perform a search of 2M keys in a data array of unique values, guaranteed to include the keys. The size of data increases from 3 to ca. 43 millions, with step 3. As Figure 5.18 shows, the IMPROVED version provides a significant boost, mostly related to its simpler code. Still, this boost is only visible when data fits in the CPU cache - once the lookups start to cause main memory accesses, the time gets dominated by the cache misses.
Looking at the code in the IMPROVED version, we can observe that the iterations in the inner loop are not independent. This is caused by the nature of the binary tree traversal – to find the next node, the previous node needs to be fully compared with. As a result, the code cannot fully exploit superscalar CPU features and cache misses cannot be overlapped between iterations. To improve this situation, we can exploit the fact that the inner loops are independent for different searched keys (values of $i$). Also, the value that is added or not to the current position $p$ at a given level ($j$) is the same for different keys. This leads to the following, VECTORIZED implementation of our problem:

```c
// BINARY SEARCH - VECTORIZED
VSIZE=256;  // vector size
// Perform computation vector by vector
for (processed = 0; processed < M; processed += VSIZE) {
    int *result_vector = result + processed;
    int *keys_vector = keys + processed;
    // First, for each vector, perform the first phase of the search
    for (i = 0; i < VSIZE; i++)
        result_vector[i] = splitIndex * (keys_vector[i] >= splitValue);
    // Then, for each search phase, perform it for all the elements in a vector
    for (j = powerOfTwo / 2; j >= 1; j = j / 2) {
        int *data_shifted = data + j;
        for (i = 0; i < VSIZE; i++)
            if (keys_vector[i] >= data_shifted[result_vector[i]])
                result_vector[i] += j;
    }
}
```

This version follows the idea of phase separation from Section 5.2.3.2. The state of each element in a given phase is saved in `result_vector`. As a result, each iteration in the inner-most loop is fully independent. Also, some optimizations become possible, e.g. introducing the `data_shifted` variable, which allows to remove the addition of $p$ during each comparison. While this version achieves a relatively poor performance, as seen in Figure 5.18 it is much more resistant to the data not fitting in the CPU cache. This is because the cache misses occurring in the lookup phase are independent, and can be overlapped by the memory controller. This motivates further optimizations of this routine. One problem found in this routine is the `if` statement in the inner-most loops. This control dependency can be easily converted into data dependency with this simple code:

```c
// BINARY SEARCH - VECTORIZED-NO-IF
...
    for (i = 0; i < VSIZE; i++)
        result_vector[i] += (keys_vector[i] >= data_shifted[result_vector[i]]) * j;
...
This version significantly improves the performance, by matching the IMPROVED version for in-cache data ranges and being more resistant to cache misses for larger data ranges.

The final optimization comes from the observation that since all the operations for different keys are independent, SIMD instructions can be used to further improve the performance. This SIMD-ized version of the binary search is as follows (using Intel ICC SIMD intrinsics [Int07c]):

```c
// BINARY SEARCH - VECTORIZED-SIMD
VSIZE=256; // vector size
for (processed = 0; processed < M; processed += VSIZE) {
  int *result_vector = result + processed;
  int *keys_vector = keys + processed;
  __m128i xm_idxvec = _mm_set_epi32(splitIndex, splitIndex, splitIndex, splitIndex);
  __m128i xm_valvec = _mm_set_epi32(splitValue, splitValue, splitValue, splitValue);
  // Prepare the first index in the search
  for (i = 0; i < VSIZE; i += 4) {
    __m128i xm_vals = _mm_load_si128((__m128i*)(keys_vector + i));
    xm_vals = _mm_andnot_si128(_mm_cmplt_epi32(xm_vals, xm_valvec), xm_idxvec);
    _mm_store_si128((__m128i*)(result_vector + i), xm_vals);
  }
  // Then, for each search phase, perform it for all the elements in a vector
  for (j = powerOfTwo / 2; j >= 1; j /= 2) {
    int *data_shifted = data + j;
    __m128i xm_jvec = _mm_set_epi32(j, j, j, j);
    for (i = 0; i < VSIZE; i += 4) {
      __m128i xm_idxvec = _mm_load_si128((__m128i*)(result_vector + i));
      int cmpval0 = data_shifted[result_vector[i + 0]];
      int cmpval1 = data_shifted[result_vector[i + 1]];
      int cmpval2 = data_shifted[result_vector[i + 2]];
      int cmpval3 = data_shifted[result_vector[i + 3]];
      __m128i xm_valvec = _mm_load_si128((__m128i*)(keys_vector + i));
      __m128i xm_cmpvalvec = _mm_set_epi32(cmpval3, cmpval2, cmpval1, cmpval0);
      __m128i xm_idxvec = _mm_add_epi32(xm_idxvec, _mm_andnot_si128(_mm_cmplt_epi32(xm_valvec, xm_cmpvalvec), xm_jvec));
      _mm_store_si128((__m128i*)(result_vector + i), xm_idxvec);
    }
  }
}
```

This VECTORIZED-SIMD version, while more complicated, allows to significantly reduce the computation costs by executing the same operations for multiple keys (4 in this case). Note that this implementation cannot be fully SIMD-ized due to ISA limitations. For example, Intel SSE instructions do not have single “gather” memory access instructions, allowing filling in a single SIMD register with data from multiple memory locations – this leads to explicit creation of
the \texttt{xm\_cmpvalvec} variable. Even with this non optimal implementation, Figure 5.18 shows that \texttt{VECTORIZED-SIMD} beats all other implementations by a large margin, especially when the data is in the CPU cache. Keep in mind that the non-vectorized versions cannot be SIMD-ized in a similar manner, since the computations there are not independent.

This section demonstrates again that vectorized processing, when applied, allows many optimization techniques impossible in classical approaches. We demonstrated how \textit{phase-separation} can be used to provide independent operations, allowing better overlapping of cache misses. Then, by removing the \textit{control dependency} we improved the performance on superscalar CPUs. Finally, the introduction of SIMD instructions allowed to amortize the computation cost among multiple elements. This gave a 2-3 times improvement over an optimized non-vectorized version, both for in-cache and in-memory data.

5.6 Conclusions

This chapter demonstrates that the vectorized execution model, proposed in Chapter 4 has numerous advantages over both the traditionally applied tuple-at-a-time model and the column-at-a-time model of MonetDB. However, the new approach results in new challenges, especially in the area of expressing relational operators in a vectorized way. Implementation techniques proposed in this chapter can make this process easier, allowing generic implementations of different data processing tasks that are often approaching the performance of hard-coded solutions. This high performance is achieved not only by removing the interpretation overheads found in an iterative approach, but also thanks to the ability to exploit various features of modern CPUs: superscalar processing, SIMD instructions and cache memories.