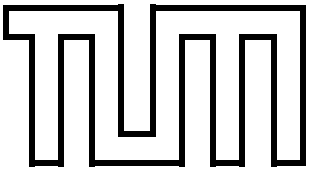


FAKULTÄT FÜR INFORMATIK

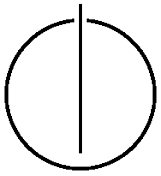
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

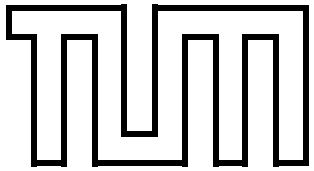
Master's Thesis in Computer Science

Adapting Main-Memory Databases to Modern Hardware Architectures

An Evaluation of Query Processing Using SIMD Instructions

Harald Lang





FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Computer Science

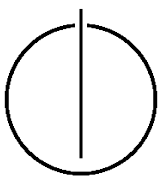
Adapting Main-Memory Databases to Modern Hardware Architectures

An Evaluation of Query Processing Using SIMD Instructions

Anpassung von Hauptspeicher-Datenbanken an moderne Rechnerarchitekturen

Eine Evaluation der Anfragebearbeitung unter Verwendung von SIMD Instruktionen

Author:	Harald Lang
Supervisor:	Prof. Alfons Kemper, Ph.D.
Advisor:	Prof. Dr. Peter Boncz, (CWI, Amsterdam)
	Viktor Leis, M.Sc.
Submission Date:	August 15, 2014



Ich versichere, dass ich diese Master's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, 15. August 2014 Harald Lang

Contents

List of Figures

List of Tables

Listings

1. Introduction

1.1. Motivation

In the last decade, database systems have been adapted to the continuously evolving hardware. Growing main-memory formed the basis of success for highly performant in-memory database systems and the increasing number of CPU cores led to very efficient approaches with respect to task-parallelism in query processing. With data stored in main-memory, the access latencies have been decreased by orders of magnitude. As a result, research has been focused on the efficient utilization of CPUs. Many concepts used in disk-based systems have been discarded as they have been become obsolete or too inefficient in their new environment. Resent research led to query execution strategies that turned the main-memory bandwidth into the new bottleneck.

However, many analytical workloads perform CPU-intensive tasks on large amounts of data. Typically, the data is aggregated to a small number of output tuples or even to a single scalar value. Many arithmetic operations and comparisons result in higher latencies and turns this kind of queries into a CPU-bound workload.

In this work we investigate on how CPU-intensive database queries can be accelerated using the *Single Instruction Multiple Data* (SIMD) capabilities of the latest x86 processors. With the Haswell architecture, Intel released the AVX-2 instruction set which provides a wide range of SIMD instructions for integer arithmetics. With AVX-2 an operation can be performed on up to thirty-two elements in parallel within one single CPU instruction. As database systems make much use of integer data types, the AVX-2 instruction set is a very promising subject of investigation in the context of database systems.

1.2. Related Work

Improving query processing performance using SIMD instructions has been investigated by many researchers. E.g. Willhalm et al. [22] proposed a table scan for column-oriented main-memory databases that operates on compressed data. Polychroniou et al. [17] used SIMD techniques to improve aggregations. Zukowski et al. [26] investigated the use of SIMD operations in a block-oriented query processing as it is used in the MonetDB database system [3] and in [25] Zukowski et al. presented a architecture-conscious hashing which significantly improved the efficiency of hash tables in terms of instructions per cycle (IPC). In [2], Balkesen et al. tuned different hash join implementations to the underlying hardware inter alia, by making use of SIMD. The HyPer database system [6] makes use of SIMD instructions to accelerate bulk-loading [13] and index lookups [9]. In the basic research of Zhou et al. [24] in 2002, the authors showed the potentials of SIMD in many different database operations. E.g. the evaluation of selection predicates, aggregation, index lookups, etc.

1.3. Research Questions

In contrast to previous works, this thesis does not focus on a single database operator, but instead it aims to efficiently employ SIMD instructions on larger parts of query execution plans. On this extended scope, performance related aspects are investigated that arise when SIMD is used across multiple operator boundaries.

In this work the following research questions will be answered:

1. What are the performance benefits for computationally intensive analytical queries?
2. Can queries be efficiently compiled into SIMD code?
3. What kind of operations/queries can not be efficiently processed with SIMD?
4. How does the AVX-2 instruction set compare to earlier SIMD instruction sets in terms of performance?

1.4. Scope

In this thesis we restrict our research to unary database operators, which we briefly introduce in chapter 2. We primarily focus on efficient arithmetics and aggregation; and on the evaluation of selection predicates in table scans. The scope of the considered operators is sufficient to execute *single-table queries*. Further, we only consider relations that consist of integer data. We do not investigate on binary operators like the join operator. Especially, the most important hash join operator is not well suitable for employing SIMD instructions, due to its random memory access patterns. However, the computation of hash values can be performed in parallel. Therefore, in chapter 6 we evaluate three prominent hash functions in SIMD.

1.5. Approach

To answer the research questions stated in section 1.3, we chose the following approach:

1. To determine the potentials of AVX-2 in terms of performance, we manually implement a CPU-intensive query in C++ using SIMD instructions. We then compare the runtime with a second hand-written non-SIMD implementation. The query and the data set for this experiment are taken from the TPC-H benchmark [19] which we will introduce in the following chapter. In preparation of this experiment, we establish a suitable in-memory representation of the data set that makes use of narrow data types, mostly 8-bit integers.
2. In the second stage, we implement a prototypal query compiler to answer the question, if automatically compiled queries can compete with handwritten queries in terms of performance. The prototype should be able to produce AVX-2 code, as well as SSE code for comparison.

1.6. Organization

The rest of this work is organized as follows. Chapter 2 describes the fundamentals of SIMD in x86 architectures. Further, it provides an overview of existing query execution

strategies and it introduces two queries that are used for benchmarking throughout the work.

Chapter 3 covers the first stage of the approach (given in section 1.5) where the potentials of AVX-2 are experimentally determined using a handwritten query. The subsequent chapters are part of the second stage, where we focus on code generation and the implementation details of the individual operators.

Chapter 4 introduces our query execution model that enables data-parallelism via SIMD instructions. In chapter 5 we shows how the evaluation of selection predicates is implemented using SIMD instructions. Further, it discusses some implications for the underlying storage. In chapter 6 we cover basic arithmetics and special math functions. As part of this chapter, we present efficient implementations for checked integer arithmetics. Type conversion in SIMD registers is discussed in chapter 7 and chapter 8 covers the group-by operator.

In chapter 9 we present the prototypal query compiler that has been developed as part of this work and the experimental results are presented in chapter 10.

2. Fundamentals

2.1. SIMD in x86 Architectures

In Flynn's taxonomy [4], a computer that is able to perform an operation on multiple data elements within a single instruction is classified as a *single instruction, multiple data* (SIMD) computer. Most of today's x86 processors fit into this class, as they provide supplementary instruction sets to speed up data-intensive computations. In x86 architectures, SIMD operations are performed in an additional set of registers, which are typically 128- or 256-bit wide. These registers can be filled with multiple data elements. A single SIMD instruction then performs the same operation on all element. Figure 2.1 shows a SIMD instruction that performs an arbitrary operation $\circ$ on eight elements in parallel. In general, the number of results are equal to the number of input elements.

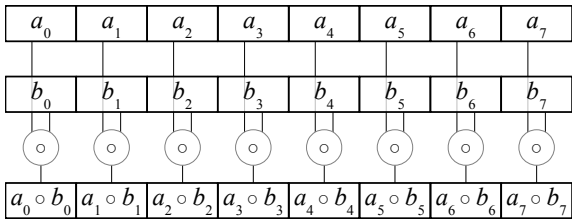


Figure 2.1.: A common SIMD operation.

SIMD instructions are often termed as *vector instructions*, because they process a fixed number of input elements at once. Code parts that make use of SIMD instructions are therefore also referred to as *vectorized code*. The expression in Figure 2.1 can therefore also be written as $\overrightarrow{a} \circ \overrightarrow{b}$.

In SIMD terminology, we often refer to the elements which are stored in a SIMD register as *packed elements*. The number of elements, that can be *packed* into a single register depends on bit-width of the element's data type. The bit-width of a SIMD register is either fixed to 128 or 256 bits. Therefore, the maximum number of packed elements of type T is $P = \frac{\text{bit-width(REG)}}{\text{bit-width}(T)}$.

The term *lane* refers to the to the bit-width and to position of packed elements. E.g. if a 256-bit SIMD instruction performs an operation "on 64-bit lanes", then the contents of the input registers are interpreted as four 64-bit values, and the i^{th} lane refers to data stored in bits $[64 \cdot (i+1) - 1; 64 \cdot i]$.

2.1.1. SIMD Instruction Sets

In this work we primarily focus on Intel's latest *Advanced Vector Extensions 2* (AVX-2), but we also operate with many different earlier instruction sets such as the *Streaming SIMD Extensions 2* (SSE2) and the *Supplemental Streaming SIMD Extensions 3* (SSSE3). Throughout this work, we only distinguish between SIMD instructions that operate on 256-bit registers, like AVX-2, and instructions that operate on 128-bit registers, like SSE2, SSSE3, etc. For simplicity, we use "AVX-2" or "SIMD-256" to refer to 256-bit instructions,

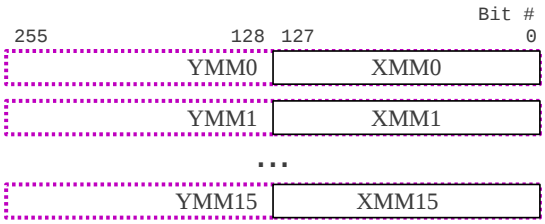


Figure 2.2.: SIMD register layout.

and "SSE" or "SIMD-128" (imprecisely) subsume all instructions that operate on 128-bit registers. For non-SIMD instructions, that operate on regular CPU registers, we use the term *scalar*- or "x86"-instructions.

2.1.2. Register Naming and Layout

In earlier processors, SIMD instruction where limited to eight 128-bit registers, which were named as XMM0 to XMM7. With the release of the *Advanced Vector Extension* (AVX) instruction set, Intel introduced 256-bit wide registers and doubled their number. The newly introduced registers are named as YMM0 to YMM15. XMM and YMM registers are physically the same, because the XMM names refer to the lower 128 bits of the corresponding YMM register, as illustrated in Figure 2.2. - Due to the fact, that XMM and YMM registers share the lower 128 bits, the YMM registers are also called *extended* SIMD registers and the term *extended packed elements* sometimes refer to their contents. - Instructions from earlier instruction sets can only access the lower 128 bits, but due to the new instruction encoding (VEX), legacy 128-bit instructions benefit from the eight additional XMM registers. Further, the VEX encoding provides a three-operand form for legacy instructions, which allows a compiler to produce more efficient code compared to the two-operand form, where the content of at least one input register is overridden by the results. To access the higher 128 bits of a YMM register with SSE instruction, we have to move the data into the lower bits beforehand. This is accomplished using the `vextracti128` instruction.

2.1.3. Intrinsics Functions

To make use of SIMD instructions in high-level languages like C and C++, Intel provides for almost all instructions corresponding *intrinsics functions*- and type-declarations in C. Throughout this work, we make heavy use of these intrinsics. Therefore, we introduce some basic type declarations and the general naming conventions.

The data types `__m128i` and `__m256i` are used to represent the content of SIMD registers filled with integer data. An instance of `__m256i` can either represent thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit integers. Typically, these types are only used to call intrinsics functions. A common code pattern is to reinterpret an array of integers as an array of `__m256i` before an intrinsics function is called.

The functions use the following naming convention: `<prefix>_<operation>_<type>`. The prefixes `_mm` and `_mm256` denote the bit-width of the SIMD instruction, whereas `_mm` refers to SIMD-128 and `_mm256` to SIMD-256, respectively. The middle part contains the name of the operation (mostly in short form) and last part of the function name denotes the type of the packed elements. For instance, the following function performs an addition on eight 32-bit integers:

```
__m256i _mm256_add_epi32 (__m256i a, __m256i b)
```

Where EPI stands for *extended packed* (signed) *integers*.

In general, there is a one-to-one relationship between the SIMD instructions and their corresponding intrinsics function. In most cases, we refer to the function names instead to assembler mnemonics to improve readability.

2.2. Relational Operators

In this work, we focus on database queries of the following form:

```
SELECT aggr_1(expr_1), ... , aggr_n(expr_n)
FROM R
WHERE p_1 AND ... AND p_m
GROUP BY attr_1, ... , attr_k
```

where $\texttt{aggr}_i$ denote aggregation functions like `SUM`, `AVG` and `COUNT` and $\texttt{expr}_i$ stands for arbitrary arithmetic expressions. The `WHERE` and `GROUP BY` clauses are optional. The selection condition in the `WHERE` clause is restricted to conjunctive predicates, and the `FROM` clause is restricted to a single relation.

This kind of queries can be (logically) expressed in relational algebra using the following relational operators:

- *Projection*, which is defined as $\prod_A(e) = \{\circ_{a \in A}(a : x.a) \mid x \in e\}$, where e denotes an arbitrary relational expression, $\circ$ the concatenation operator and A a set of attributes produced by e. $x.a$ refers to the attribute a in tuple x (dot-notation).
- *Selection*, which is defined as $\sigma_p(e) = \{x \mid x \in e \wedge p(x)\}$, where p denotes the selection predicate which tuples must satisfy.
- *Map / function evaluation*, which is defined as $\chi_{a:f}(e) = \{x \circ (a : f(x)) \mid x \in e\}$, where $f(x)$ denotes an arbitrary arithmetic expression or function, and a the name of the newly computed attribute.
- *Group by / aggregation*, which is defined as $\Gamma_{A;a:f}(e) = \{x \circ (a : f(y)) \mid x \in \Pi_A(e) \wedge y = \{z \mid z \in e \wedge \forall a \in A : x.a = z.a\}\}$.

Further we make use of the following equivalence to rearrange relational expressions:

$$
\begin{aligned}
\sigma_{p_1 \wedge p_2}(e)) &\equiv \sigma_{p_1}(\sigma_{p_2}(e)) \\
\sigma_{p_1}(\sigma_{p_2}(e)) &\equiv \sigma_{p_2}(\sigma_{p_1}(e)) \\
\Gamma_{A;a:f(g(x))}(e) &\equiv \Gamma_{A;a:f(x')}(\chi_{x':g(x)}(e)) \\
\Gamma_{A;a:\mathrm{avg}(x))}(e) &\equiv \Pi_{A \setminus \{a',t\}}(\chi_{a:a'/t}(\Gamma_{A;\, a':\mathrm{sum}(x);\, t:\mathrm{count}(*))}(e)))
\end{aligned}
$$

In general, logical algebraic operator may have different physical implementations. An example from the list above is the group-by operator, which can be implemented i.a. based on *sorting* or *hash-partitioning*. During query-translation or -optimization, the database system selects a concrete implementation for each operator and so it transforms the logical algebraic expression into a *physical algebra* expression that is logically equivalent to the former one, but suitable for execution. Typically, physical operators have annotations

that refer to the implementation details. For instance, a selection operator that processes multiple elements in parallel using SIMD might be denoted as σ_p^{SIMD}.

A very simple but important physical operator, is the *table scan* operator. The table scan iterates over all tuples of some table and passes them to the subsequent operator. In classical relational algebra, the scan operator does not have a logical counterpart, because accessing data is inherently a physical operation. For example, the logical algebra expression

$$\Gamma_{\{\};s:\text{avg}(x+y)}(\sigma_{x<42}(R))$$

can be translated into

$$\Pi_s(\,\chi_{s:t2/t3}(\,\Gamma^S_{\{\};t2:\text{sum}(t1),t3:\text{count}(*)}(\,\chi_{t1:x+y}(\sigma_{x<42}(\text{tscan}(R))))))),$$

where Γ^S denotes a scalar aggregation.

2.3. Query Execution Models

Once a query is translated into a physical algebraic expression (aka. the *query execution plan*), it basically contains all necessary information to execute the query on the underlying database. However, existing database systems differ considerably in the way queries are executed. The execution model we are going to use with SIMD adapts concepts of different existing execution models. Therefore, we give a brief overview of these existing models that are related to the execution model we use in this work.

2.3.1. Iterator Model

The most popular execution model is the iterator model [10]. In the iterator model, all physical operators share a common interface, which basically consists of the three functions: open(), next() and close(). Initially, the open() function is called on the topmost operator. During the function call, the operator performs initializations like memory allocations, acquiring file handles, etc. Further, the open()-function call is propagated to all subsequent operators. After the initialization is complete, the next() function is called (again) on the topmost operator. The next() function either returns a single output tuple or a NULL value which marks the end of the query result. If a output tuple was produced, the database system repeatedly calls next() until NULL is returned. Thus, the system *iterates* over the query results. As with the open() function, each operator calls the next() function of its succeeding operators. Each (non-root) operator produces an intermediate tuple which is handed over the caller of next(). Immediately after all results have been produced, the close() function is called to release all acquired resources.

Even though, the iterator model is very easy to implement, it is also known for its inefficiencies, which are mainly caused by many virtual function calls and branch mispredictions. In disk-based database systems the iterator model was a great success, because query evaluation was mostly I/O bound and the mentioned inefficiencies did not affect the overall system performance. However, with the increasing bandwidth of storage systems and eventually with the arise of main-memory database systems the situation changed dramatically, as most queries are no longer I/O bound. Instead, pipeline stallings caused by branch misprediction and the overheads that come with function calls now have considerable performance impacts.

2.3.2. Block-Oriented Processing

To overcome these shortcomings, the iterator model was improved so that operators pass multiple tuples (*block*s of tuples) at once to their parent operator [15], in order to reduce the number of function calls. In the MonetDB/X100 system [3] these blocks (aka. vectors) of tuples are scaled to the CPUs cache size to additionally avoid costly materialization in main-memory. Nevertheless, the iterator based executions models follow an *interpretation* approach and the associated costs can only be reduced to some certain degree.

2.3.3. Query Compilation

Recent developments therefore favor compilation approaches, where a given query plan is entirely translated into executable code. The Holistic Integrated Query Engine [7] (HIQUE) for instance, translates the query plan into C++ code which is then compiled into a machine executable binary. Thereby, the system makes heavy use of templates, which are expanded during compilation. This approach eliminates the concept of iterators entirely and so its overheads. Further, the system benefits from static code optimizations (e.g. function inlining) performed by the underlying compiler. However, this approach requires that every query goes through a compilation process which in case of C++ is very heavy-weight and therefore time consuming (in the order of seconds). The HyPer system [6, 14] uses a more complex but light-weight compilation process which generates low-level intermediate code which is then compiled by the Low Level Virtual Machine (LLVM) framework [8] into native code. Using an intermediate representation that is closer to native code, reduces the compilation time from seconds to milliseconds.

2.3.4. Produce/Consume Model

Beside the reduced compilation time, HyPer further improved the overall query performance by introducing a *data centric* query execution model. Whereas in the aforementioned models tuples are passed or copied between the physical operators, the data centric approach aims to keep tuples in CPU registers as long as possible. In the ideal case, a tuple is loaded into registers only once and remains there until all operators are finished processing the current tuple. This approach greatly improves code locality and reduces materializations into slower cache memory.

In this model, the physical operator provide a lean interface consisting of two functions: produce() and consume(). In contrast to the iterator model, where function calls immediately compute result tuples, the compilation approach follows the paradigm of *generative programming*. Thus, function calls instead *emit code* which in the later execution phase computes the query results. This code generation process is triggered by calling the produce() function of the root operator of a given query plan. As the name suggests, the produce() function emits the code to produce an output tuple. Typically, only the leaf operators are able to directly produce tuples, whereas inner operators rely on the input from their child. Inner operator use the produce() function to make initializations (if necessary) and to direct the control flow to the produce() function of their child operators. Once an operator has emitted the code that produces an output tuple, the consume() function of the parent operator is called. The consume() function then emits code that processes the previously produced tuple. - We illustrate the code generation in simplified pseudo-code which is based on the following query plan and given implementations: $\prod_y(\sigma_{x=42}(\text{tscan}(R)))$

Table Scan:

```
function produce()
  emit "for each tuple t in R"
  emit " load t.x → a0"
  emit " load t.y → a1"
  parent.consume(a0, a1)
  emit "end for"
```

Selection:

```
function produce()
  child.produce()

function consume(ai)
  emit "if a0 == 42"
  parent.consume(ai)
  emit "end if"
```

Projection:

```
function produce()
  child.produce()

function consume(ai)
  emit "output a1"
```

The above query plan then results in

```
for each tuple t in R
  load t.x → a0
  load t.y → a1
  if a0 == 42
    output a1
  end if
end for
```

As shown in the example, the produce/consume model generates very concise intermediate code. Further, it shows that the code generation is not restricted to emit only assembly-like code. In principal, any imperative programming language can be used as intermediate representation.

2.4. Benchmark Setting

In Section 2.1 we mentioned, that the number of elements that can be packed together depends on their data type. The wider the data type of an element is, the smaller is the number of packed elements P. Theoretically, this number denotes the maximum *degree of parallelism* that can be achieved, and therefore it denotes the upper bound for possible *speedup*s. This implies, that database systems, that want to take maximum advantage from SIMD, needs to make use of the smallest possible data types in their storage system. Possibly some lightweight compression scheme should be used to better utilize SIMD capabilities, but this is out of the scope of this work. In this work we restrict ourselves to narrowing down the internally used data types to the smallest possible types, depending on the maximum attribute values.

For performance measurements we use parts of TPC-H benchmark [19]. In this particular case, we evaluate the system only with TPC-H Query 1 (Q1) and 6 (Q6). Both

queries operate on a single table, namely the `lineitem` table. In total, 6 attributes (out of 16) are accessed by Q1 and Q6. Thus we reduced schema and data set to these six attributes. To gain a higher degree of parallelism, we choose the least memory consuming data type for each attribute. For comparison we also compiled the schema with wider data types, mostly 64-bit integers. Table 2.1 shows the reduced schema and the two internal representations, which are named "Narrow" and "Wide". Note, that we store date values as Julian Day Numbers (JDN). For the Narrow representation we additionally change the point of reference to January 1, 1970. Then, a date d is encoded as: $\text{JDN}(d) - 2440587$. Because we use signed 16-bit integers, the uppermost representable date is $\text{JDN}(2^{15} - 1) + 2440587 = 2473354 =$ September 18, 2059.

		Internal Data Types	
Column	TPC-H Spec.	Narrow	Wide
`l_quantity`	decimal	`int 8`	`int 64`
`l_extendedprice`	decimal	`int 32`	`int 64`
`l_discount`	decimal	`int 8`	`int 64`
`l_tax`	decimal	`int 8`	`int 64`
`l_returnflag`	fixed text, size 1	`int 8`	`int 8`
`l_linestatus`	fixed text, size 1	`int 8`	`int 8`
`l_shipdate`	date	`int 16`	`int 32`

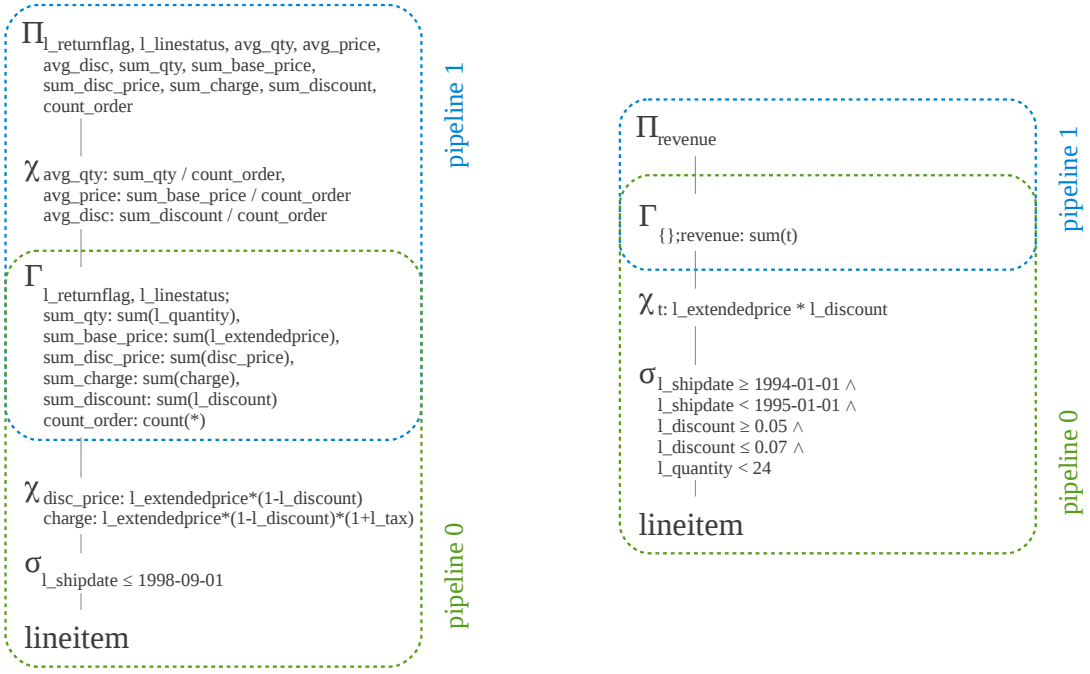
Table 2.1.: Reduced schema of the TPC-H lineitem table.

Figure 2.3 depicts the relational operator trees of Q1 and Q6 as they are used in our performance evaluations. The characteristics of Q1 are: (i) High selectivity, because more than 95% of the tuples *survive* the selection operator. Further, (ii) many arithmetic operations are performed. In total, seven additions and two multiplication are performed for each surviving tuple, until its materialization in the group-by operator. (iii) The tuples are reduced to only four output tuples during aggregation. Thus, the time spent in the subsequent operators is negligible small.

In contrast, Q6 only performs two arithmetic operations, one in the map operator and another one in aggregation. Important characteristics are: (i) No grouping during aggregation. The surviving tuples are reduced to a single scalar value. (ii) Many selection conditions. In total, five comparisons are performed on three different attributes. (iii) Low selectivity. Approximately 2% of the tuples survive the selection.

Both queries together, deliver meaningful results regarding the performance of SIMD arithmetics, selection and the two kind of aggregations. Throughout this work we use them as our running examples.

We ran all benchmarks single-threaded on a Intel Haswell (Core i5-4670T) machine clocked at 2.30 GHz with 16 GB of main-memory (DDR-1333). As C++ compiler we used LLVM/Clang++ version 3.4. As input we used the lineitem table with scale factor 10 (≈ 60 million tuples). If not stated otherwise, we make use of the narrow data types.



(a) Q1 (b) Q6

Figure 2.3.: Relational operator trees of TPC-H Queries 1 and 6.

3. Motivation for SIMD Processing

As mentioned in the introduction, we initially determined the potentials of the AVX-2 instruction set in terms of performance. For this, we manually implemented the first TPC-H query (Q1) in C++ and tuned it to the underlying hardware.

We basically implemented three different versions of Q1: A sequential version which processes a single tuple at a time, an AVX-2 version where the evaluation of arithmetic expressions as well as the aggregations are entirely performed in SIMD registers, and further a hand-tuned AVX-2 version where we additionally exploited the exact knowledge about the input data. In the latter case we stored the two aggregates sum(l_quantity) and count(*) in a single 64-bit lane. We exploited the fact, that both aggregates can be represented using 32 bits. This allowed us to update five aggregates in parallel within a single SIMD instruction. In all three implementations we make use of the narrow data set.

As figure 3.1 shows, the runtime of Q1 can be reduced by ~30% compared to the hand-written sequential version. However, this kind of optimization requires precise domain knowledge which is typically provided by min/max-indices. Therefore, in database systems, like HyPer, that do not maintain min/max-indices, we can not make use of these optimization techniques. Nevertheless, with the "clean" AVX-2 version we still achieve a 27% performance improvement which motivates further investigations.

The performance difference between HyPer and the handwritten sequential version is basically due to the compact data representation we use for our experiments.

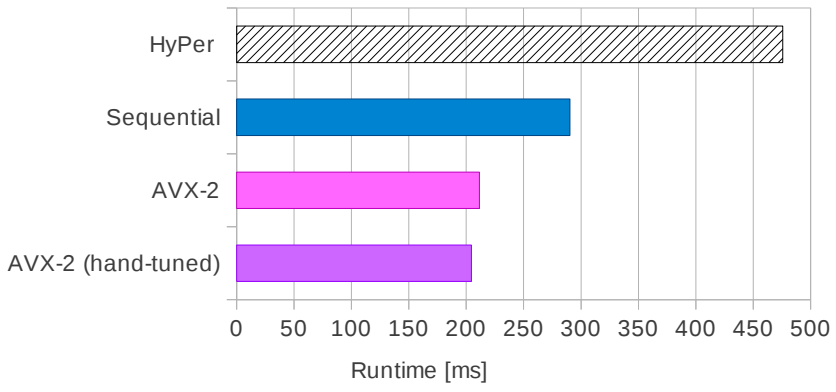


Figure 3.1.: Performance comparison of handwritten TPC-H Q1 implementations.

In the rest of this thesis we focus on automatically generating SIMD code and we describe the implementation details of the individual operators. We pursue the goal of developing a query compiler which generates code that is as efficient as our handwritten implementation. Because the generated code will be similar to the handwritten code in many aspects, we omit implementation details in this section. Instead, we briefly discuss the differences in the later evaluation chapter.

4. Block-Wise Query Execution

In this chapter we conceptionally introduce the execution model which we employ for SIMD enabled query evaluation. We adapt concepts from existing models which we briefly introduced in section 2.3. Due to the outstanding performance of the HyPer database system [6], we based our model on the produce/consume code generation model [14]. In its original form the produce/consume model is designed to process *tuple-at-a-time*, which is naturally not well suited for SIMD processing. Therefore we slightly extended the function behaviors of the physical operators. A producing operator is now intended to produce a *block of tuples* until the code of a consuming operator is executed, because processing multiple tuples at a time paves the way for employing SIMD instructions. We refer to this as *block-wise* processing. But in contrast to vector-wise processing which is used in MonetDB/X100, our approach differs in the following aspects: (i) Each block has a constant size and within a single query the size does not change, where size refers to the number of tuples. (ii) The total number of tuples in a block is quite small, because we aim to keep elements in CPU registers as long as possible. Thus, the block size is determined by the number of elements that fit into SIMD registers, and not by the CPU's cache size. Further, the block size must be chosen large enough, that it would be possible to apply wide SIMD instructions.

4.1. Code Generation

For the intermediate representation (IR) we use the C++ programming language, what is basically due to simplicity reasons. Thus, we also "adapt" long compilation times.

During the code generation process, each operator emits C++ code for its corresponding logical operation, where the generated code mostly consists of intrinsics function calls. The intrinsics can therefore be seen as the *basic building blocks* (or primitives) every query is made of. Therefore, a single (logical) operation issues at least one (machine) instruction. In our block-wise approach, each operation is always applied on a block-level. Thus, if the block size exceeds the number of elements that can be packed into a single register, then multiple instructions are performed. In that case the operator emits a small loop that iterates over the current block, as it is shown in the following listing.

```
int16_t result[32] __attribute__ ((aligned (32)));
__m256i* result0 = reinterpret_cast<__m256i*>(result);
__m256i* a0 = reinterpret_cast<__m256i*>(a);
__m256i* b0 = reinterpret_cast<__m256i*>(b);
for (size_t i = 0; i < 2; i++) {
  result0[i] = _mm256_add_epi16(a0[i], b0[i]);
}
```

The above listing contains some frequently recurring code patterns: (i) Intermediate results are stack-allocated arrays (from the perspective of C++). (ii) Function arguments are re-interpreted as generic SIMD data types to match the function signatures of intrinsics, and (iii) for-loops with simple bodies and a small amount of iterations.

It is noteworthy that the result of the operation is not necessarily materialized on stack memory and that the reinterpretation statements are eliminated during compilation. Further, the for-loops are unrolled by the compiler, thus the resulting native code does not contain conditional jumps. In experiments we observed that manual loop unrolling results in less efficient register allocation.

Careful handling is needed when intrinsics are used in combination with reinterpret casts, because load and store instructions require the data to be 16-byte aligned in case of SSE and 32-byte aligned in case of AVX-2. If SIMD data types like `__m128i` and `__m256i` are used explicitly, the compiler automatically takes care of a proper alignment. In case of reinterpretation of arrays as SIMD types, the corresponding array must be annotated with alignment information.

Most instructions exists in three different variants: As SIMD-256 (AVX-2), SIMD-128 (SSE) and as scalar instructions (x86). During code generation, each operator selects the widest possible instruction. If the block size is smaller than the width of the narrowest SIMD instruction, then the code generator is forced to select a scalar instruction. From the perspective of a query compiler, this can be seen as the *instruction selection* phase.

4.2. Degree of Parallelism

In earlier sections, we informally introduced the *degree of parallelism* metric (DoP). The DoP metric provides useful information about how well a given query can benefit from SIMD and gives a (rough) estimation about the speedup. In later chapters we show example queries, where employing SIMD has negative impact on query performance. Using the DoP metric, these cases can be easily identified and quantified. Therefore, we give the fundamental formal definition of DoP.

So far we used P to denote the number of elements that can be packed into a single register. On a single-instruction-level, P also denotes the degree of parallelism for a specific operation. At a coarse grained level a query basically consists of a (finite) sequence of operations $s := (o_1, \ldots, o_n)$, and each operation o has a corresponding instruction with a specific degree of parallelism P_o. Then, the DoP of a given sequence is defined as:

$$\mathrm{DoP}(o_1, \ldots, o_n) = \frac{\sum_{i=1}^{n} P_{o_i}}{n} = \overline{P_{o_i}} \quad \left[\frac{\text{elements}}{\text{instructions}}\right] \tag{4.1}$$

We assume that $\forall o_i(P_{o_i} \leq B)$, where B denotes the block size. By design, the DoP can not exceed the block size, because the instruction selection ensures, that the instruction width is less or equal to the block size. However, the DoP's upper bound is independent from the block size, because it is naturally bound by the instructions widths: $\mathrm{DoP} \leq \max(P_{o_i})$.

4.3. Block Size

Throughout this work we let B denote the block size. We showed that a block which is too small limits the maximum degree of parallelism that we can gain from SIMD. On the other hand, larger block sizes increase the register pressure and typically lead to spilling. In the later evaluation section we show how the block size influences the query runtime. So far, we only restrict the block size to be a power of two.

$$B = 2^b \quad , b \in \mathbb{N}$$

Due to the maximum number of elements that can be processed in parallel using SIMD is also a power of two, restricting the block size ensures that *all* elements of a block can be processed exclusively using SIMD instructions.

$$B \bmod P_o \equiv 0 \quad \text{if } B \geq P_o$$

For instance, in database queries that access attributes which are stored as 8-bit integers, the max. degree of parallelism with AVX-2 is 32. Thus, we have to set the block size at least to 32 to fully utilize SIMD instructions.

5. Evaluation of Selection Predicates

In this chapter we focus on the *select* operator. Typically, selection is performed at the very beginning of query evaluation. Query optimizers perform logical transformations on the operator tree to *push selections* down as much as possible. In most cases, the selection is then performed immediately after the table scan. This simple optimization, which can be found in almost any text book, aims to reduce the cardinality of the intermediate results and therefore the computational costs of the subsequent operators.

In general, the evaluation of selection predicates is a quite simple task as it (in most cases) consists of the evaluation of binary relational operators like $<, =, >$, etc. However, the use of SIMD instructions has significant implications for query execution.

Note, that we do not consider the more complex set oriented predicate evaluation like $a \overset{?}{\in} \{\dots\}$. However, short *IN-lists* like `... WHERE attribute IN (value1, value2, ...)` can be compiled down to a sequence of equality checks.

The rest of this chapter is structured as follows: Sections 5.1 and 5.2 give an overview about the available instructions in AVX-2 and show how selection predicates are evaluated. Sections 5.3 to 5.6 discuss the implications of using SIMD instructions.

5.1. Available Compare Instructions

The AVX-2 instruction set provides an *equal* and a *greater than* comparison on signed integers of 8, 16, 32 and 64 bit width. Further it provides the logical operations `AND`, `OR` and `AND-NOT`. Based on these instructions, the query compiler can translate selection predicates of the forms $attribute_1 \circ attribute_2$ and $attribute \circ constant$, where $\circ \in \{=, !=, <, \leq, \geq, >\}$, by applying the following logical transformations:

Rule	Condition		Transformation
1)	$a \leq b$	$\mapsto$	$a = b$ `AND` $b > a$
2)	$a < b$	$\mapsto$	$b > a$
3)	$a \geq b$	$\mapsto$	$a = b$ `AND` $a > b$
4)	$a != b$	$\mapsto$	$a = b$ `AND-NOT true`

Where a and b are either attributes or constants. In the most cases the $\leq$ and $\geq$ conditions can be transformed into $>$ comparisons which reduces the number of instructions per comparison:

5)	$a \leq b$	$\mapsto$	$a < b + 1$, if $b < \max(\text{dom}_{\text{int}}(b))$
6)	$a \geq b$	$\mapsto$	$a > b - 1$, if $b > \min(\text{dom}_{\text{int}}(b))$

Where $\text{dom}_{\text{int}}(A)$ denotes the domain of the *internal data type* of an attribute A. Rules 5) and 6) reduce the number of instructions from 3 to 1 if either a or b is constant. Otherwise, the addition/subtraction is performed at runtime instead of compile time, which results in 2 instructions per comparison.

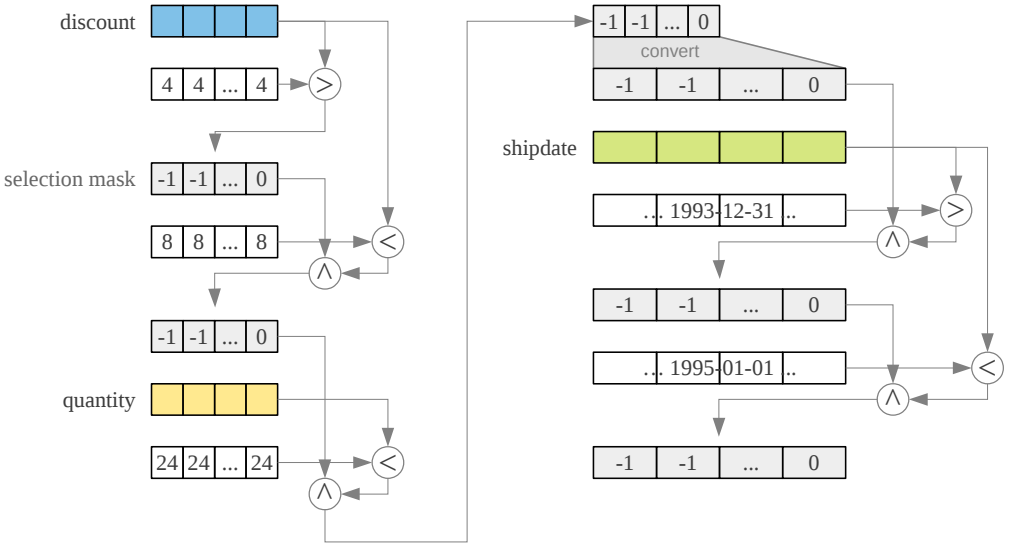


Figure 5.1.: Evaluation of selection predicates in Q6 using SIMD instructions.

5.2. Selection Mask

A SIMD comparison is performed on multiple packed elements P. Again, P denotes the number of packed elements within a single SIMD register. When performing P comparisons in parallel the result is a *bit mask* instead of a single boolean value. The resulting bit mask consists of P packed elements where either all bits of each element are set to 1 or 0. The bits of a packed element are set to 1, if the comparison evaluated to `true`. Otherwise, all bits are set to 0's. We refer to this bit mask as the *selection mask*. Multiple conjunctive or disjunctive predicates are evaluated directly on the selection mask using the bitwise `AND` and `OR` instructions as depicted in Figure 5.1.

5.3. Type Conversions

As described in the previous section, the selection mask has the same number of packed elements as the attribute that is part of the selection predicate. In general, the query compiler has to handle predicate evaluations on multiple attributes of different types, which causes type conversions of the selection mask. By default, the query compiler creates and initializes a selection mask of the same type as the attribute of the first selection predicate. For each of the following selections the mask is either converted into a wider or a narrower data type whereby a conversion results (physically) in a newly created selection mask. Note that down-casting the selection mask to a narrower type does not result in a loss of information, because the selection mask only consists of boolean values represented as 0 and -1. However, narrowing the selection mask can be prevented if the predicates are ordered by the bit-width of their attribute types.

5.4. Branches

In contrast to tuple-at-a-time processing, where each selection predicate can be translated into a conditional jumps, SIMD selection differs in the following points. First, checking

the branch condition is slightly more expensive. Second, the branch can only be taken if *all* conditions that have been evaluated in parallel evaluated to `false`.

At this point, an important design decision has been made, regarding the case where not all tuples in the current block satisfy the selection predicate. Basically, there are two options to handle the presence of *non-qualifying tuples* in a block. (i) Reducing the block size down to the number of qualifying tuples, and only pass the qualifying tuples to the subsequent operator, or (ii) leave non-qualifying tuples in place and make the subsequent operators aware of them.

For our evaluation, we took the second option, because reducing the block size would cause the following additional operations:

1. Determining the positions of qualifying tuples.

2. Shuffling tuples so that they are stored contiguously.

Especially shuffling causes significant costs, because *all* columns need to be rearranged. Further, a varying block size prevents loop unrolling in the subsequent operators.

Leaving non-qualifying tuples in place naturally does not cause any costs in the selection operator, but subsequent operators may have additional efforts, checking the selection mask. However, making operators aware of non-qualifying opens up the opportunity to compile queries without any conditional jumps, which is particularly interesting for high selectivity queries like Q1.

Branch Condition In principle, if at least one single tuple qualifies, then the control flow must continue with the code generated by the subsequent operator. With SIMD, this check can be efficiently implemented by performing a `movemask` instruction on the selection mask. The `movemask` instruction interprets the content of a SIMD register as packed 8-bit elements. It creates a mask from the most significant bit of each element and stores the result in a (scalar) integer value. The result mask is zero, if all conditions evaluated to `false`. If the result mask is not equal to zero, then at least one tuple qualifies.

The following listing shows the complete implementation with block size set to 32 and a selection mask represented as 16-bit integers:

```
int branch = 0;
__m256i* sel = reinterpret_cast<__m256i*>(selection_mask);
for (size_t i = 0; i < 2; i++) {
    branch |= _mm256_movemask_epi8(sel[i])
}
if (branch == 0) continue;
```

Predicate Selectivity Typically, multiple selection predicates are evaluated in the order of their *selectivity*, which is defined as $sel_p = \frac{|\sigma_p(R)|}{|R|}$. Predicates with lower selectivity are evaluated first to reduce the number of intermediate tuples and therefore the costs for further comparisons.

In general, this also applies for the block-wise processing, but the selectivity of a predicate has lower impacts, because a branch can only be taken if the predicate disqualifies all tuples in a block. The probability that a block contains no qualifying tuple highly depends on the block size. With an increasing block size, it is more likely that a block contains qualifying tuples and the control flow continues with the evaluation of the remaining selection predicates.

Further, reordering the predicates may also incur additional costs. If the selection predicate which is most selective refers to an attribute of a wider data type as the others, the selection mask needs to be down-casted subsequently.

For instance in Q6 the most selective predicate is the range predicate on `l_shipdate`: $p_1 =$ 1994-01-01 $\leq$ `l_shipdate` $<$ 1995-01-01 with a selectivity of $sel_{p_1} = 0.15$. The bit-width of `l_shipdate` is 16 whereas the other predicates refer to 8-bit attributes. Even though the predicate selects only 15% of the tuples, the number of blocks is reduced by only 19%. Thus, for 81% of the blocks, a down-cast of the selection mask is performed which results in barely measurable improvements (less than 0.05%). Eventually, the costs for checking the branch condition and down-casting the selection mask canceled out any performance gains. In total, 40% of all blocks survive the selection. Thus, the 2% qualifying tuples are distributed among 40% of the blocks. Introducing a branch after the selection would eliminate the costs of the remaining operators for 60% of the blocks. However, the runtime profile of Q6 is dominated by selection and costs for branching do not amortize the costs of the remaining operators $\Gamma^S_{\{\};\text{revenue:sum}(t)}(\chi_{t:\text{l\_extendedprice*l\_discount}}(\ldots))$. Introducing a branch results in a 8% performance decrease.

5.5. Qualifying Blocks

As shown by example of Q6, the decisive parameters whether to introduce a branch or to run a query branch-free basically are the possible cost savings through the subsequent operators and the additional costs incurred by the branch itself. In the block-wise processing parallel operators incur costs on a per-block basis. Thus, independently from the number of qualifying tuples in a block, the costs remain constant. If the query contains a branch then these costs are incurred for all blocks that contain at least one qualifying tuple. We refer to these blocks as *qualifying blocks*. As mentioned before, the impact of a branch is lower compared to a traditional sequential execution because the branch can only be taken if the selection predicate disqualifies all tuples in the current block, which becomes more unlikely the greater the block size is. In general the number of qualifying blocks is an unknown quantity. In the best case scenario the total number of qualifying tuples k are distributed among $\left\lceil \frac{k}{B} \right\rceil$ blocks. In the worst case, the k tuples are distributed among as many blocks as possible. Then the number of qualifying blocks N_b is:

$$\text{Worst case: } N_b = \begin{cases} k & \text{if } k \leq \frac{|R|}{B} \\ \frac{|R|}{B} & \text{otherwise} \end{cases} \tag{5.1}$$

If we assume, that every possible set of k tuples is selected with the same probability $\frac{1}{\binom{|R|}{k}}$, we can apply Yao's formula [23] to estimate the number of qualifying blocks. For simplicity we assume $|R|$ is a multiple of B which implies that every block of size B also contains B tuples. In our model, all tuples are stored contiguously in memory and only the last block may not be entirely filled. Therefore, the assumption leads to an error of $\frac{|R| \bmod B}{\left\lfloor \frac{N}{B} \right\rfloor}$ which is negligible for large relations. Further let $m = \frac{|R|}{B}$ denote the total number of blocks and $N = |R|$ the relations cardinality[1]. Then the number of qualifying blocks is

$$\overline{\mathcal{Y}}^{N,m}_B(k) = m \cdot \mathcal{Y}^N_B(k) \tag{5.2}$$

where $\mathcal{Y}^N_B(k)$ is the probability that a block contains at *least one* of the k selected tuples

[1] the notation of Yao's formula is adapted from [12]

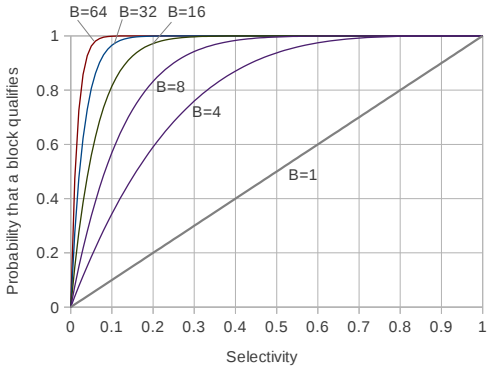


Figure 5.2.: Probability that a block qualifies with varying selectivities and block-sizes.

$$\mathcal{Y}_B^N(k) = \begin{cases} 1-p & k \leq N-B \\ 1 & k > N-B \end{cases}$$

and p is the probability that a block contains *none* of the selected tuples

$$p = \frac{\binom{N-B}{k}}{\binom{N}{k}}.$$

Due to the binomial coefficients the computation of p is very costly. We therefore make use of the following approximation that has been developed by Waters [21]:

$$p \approx \left(1 - \frac{k}{N}\right)^B \tag{5.3}$$

With $sel = \frac{k}{N}$, $m = \frac{N}{B}$ and $N = |R|$ we get the estimated number of qualifying blocks as a function in sel:

$$N_b^B(sel) = \frac{|R|}{B} \cdot \left(1 - (1 - sel)^B\right) \tag{5.4}$$

Figure 5.2 shows the probability that a block qualifies depending on the selectivity. Because the probability function grows exponentially with B as exponent it is very sensitive to the block size. E.g. for $B = 32$, approximately 99% of the blocks are scanned if $sel = 0.13$. In contrast for $B = 4$, 99% are scanned if $sel \approx 0.7$.

In case of Q6, the 2% selected tuples that survive the selection are distributed among 40% of the blocks. An estimation based on Yao's formula results in 48%. However, the basic assumption that every k-set is selected with the same probability does not generally match reality (that well). More precise estimations require detailed knowledge about the distribution of the selected tuples, but this is out of scope of this work.

The costs incurred by a branch are also hard to quantify. Typically, checking the branch condition adds constant costs: 1 vector instruction and 1 scalar comparison. However, the performance impact of the conditional jump instruction depends on it's predictability and therefore also on the distribution of the qualifying tuples. - In the later chapter 8 we show the impact of branch mispredictions in the context of grouping and aggregation. - The decision whether or not to introduce a branch in some sequence of operations $(o_1, \ldots, o_n)$ depends on the (estimated) number of qualifying blocks N_b and the costs incurred by the branch C_{Branch} itself. In general, the query runtime can be reduced if the following holds:

$$\frac{|R|}{B} \cdot C_{Block}(o_1, \ldots, o_n) > \frac{|R|}{B} \cdot (C_{Block}(o_1, \ldots, o_k) + C_{Branch}) + N_b^B(sel) \cdot C_{Block}(o_{k+1}, \ldots, o_n)$$

where C_{Block} denotes the costs for a sequence of operations in block-wise processing.

5.6. Implementation Issues

As mentioned before, most AVX-2 instructions require the data to be 32-byte aligned. Especially during selection, data is directly accessed on the global memory (the heap). From the C++ perspective a vector or an array which is allocated on the heap is accessed directly using an intrinsics function. The corresponding memory region is *re-interpreted* as an array of type `__m256i`, which enables the use of SIMD intrinsics without copying the data. This implies that the underlying storage has to guarantee that all column-arrays are at least 32-byte aligned.

Further, the allocated memory region must be a multiple of the amount of data that is read at once. During a table scan, the columns are read in chunks of at least 256 bits (or 32 bytes). Depending on the query plan, the optimizer might decide to load larger chunks during one iteration. The decisive parameter is the *block size* B, which specifies how many tuples are processed in one loop iteration. Depending on the data type T of the attribute and the cardinality of the relation N, the size of the allocated memory region must be $\left\lceil \frac{N}{B} \right\rceil \cdot \text{sizeof}(T) \cdot B$. This can be easily implemented by using STL vectors with an initial size of B_{max}, where B_{max} denotes the maximum block size of the database system. Note, that $32 \leq B_{max}$ and the block size is a power of two. The lower bound of 32 guarantees that the initial vector size of the narrowest data type is equal the size of a YMM-register, and because the STL vector doubles in size every time it gets full, the size of the allocated memory is always a multiple of $B_{max} \cdot \text{sizeof}(T)$ and therefore of $B \cdot \text{sizeof}(T)$.

Although the system can guarantee that memory reads happen within certain boundaries, it cannot guarantee that the number of records stored in a table is also a multiple of the block size. This circumstance leads to a special case that needs to be handled in the last iteration of the table scan. If the current read position i plus the block size B exceeds the table's cardinality N, then the current block contains random data from uninitialized memory. There are basically two options to handle this case. First, the last block of tuples is processed sequentially one tuple at a time or second, the query compiler manipulates the selection mask within the last iteration to mark uninitialized memory regions a non-qualifying tuple. The later approach requires that a conditional branch is introduced in the table scan loop.

In contrast, processing the last block sequentially does not require a conditional branch within the table scan loop, but it introduces a second code path that processes the remaining tuples. However, because the query compiler adapts the *produce/consume model*, the query compiler has to compile the query plan twice, once for SIMD and once for the non-parallel version for the last table scan iteration. This implies that every operator needs to be able to produce both types of code which approximately doubles code size and the compilation time. Possibly the development time doubles as well.

The query compiler that has been developed as part of this work prefers the second alternative, which is much easier to implement as it simply sets the corresponding entries in the selection mask to 0, beginning from index $N \bmod B$ to $B - 1$. As mentioned before, this introduces an additional comparison for each block of tuples. The branch itself is

well predictable by the CPU as it is taken only once during query processing. Therefore, the system trades reduced development and compile time with slightly increased runtime. Experiments with the TPC-H Query 1 and 6 have shown, that the impact is ~1% for Q1 and ~0.05% for Q6.

This approach does not cause any additional overhead or special cases in the subsequent operators of the query plan, as all other operators are already aware of the presence of non-qualifying tuples. However, the notable aspect here is, that the selection mask needs to be initialized (once) even if the query doesn't contain any `WHERE` clause. Therefore, masking random data as non-qualifying tuples is the table scan operator's responsibility.

6. Arithmetics and Special Math Functions

In analytical workloads, arithmetics play an important and performance critical role. Arithmetics appear in many different operators, e.g. in the `GROUP BY` operator, the `MAP` operator and as well as in the `SELECT` operator.

A very popular example for a computationally intensive query is Q1 from the TPC-H benchmark. If we compile the query with a data centric approach as proposed by Neumann [14], then approximately 38% of the runtime is spent in arithmetic operations. Whereas 10% is spent in the map operator and the remaining 28% in the aggregation functions of the group by operator.

In this section we take a closer look at arithmetics and special math functions in general and we show how basic arithmetic operations are efficiently implemented using SIMD instructions. We primarily focus on the AVX-2 instruction set, but we also present performance numbers with the older SSE instruction sets for comparison. At some points we give an outlook (or estimations) for the upcoming AVX-512 instructions. Because arithmetics appear in different database operators, this chapter is intended to cover the very basics, whereas later chapters show how the different high-level operators can benefit from SIMD arithmetics.

This chapter is structured as follows: Section 6.1 gives an overview about the available SIMD instructions. In section 6.2 we briefly quantify the importance of the individual operations based on their frequency of occurrence within the TPC-H benchmark. Section 6.3 covers the specifics of integer multiplication and in section 6.4 we show how overflow handling is implemented using SIMD instructions.

Experimental Evaluation Throughout the chapter we present performance numbers that have been collected in micro-benchmarks. The experiments are performed as follows: First, three small arrays a, b and c of size 5 KiB each are allocated, where a and b are initialized with random values. The two arrays a and b are used as input, whereas c is used as an output buffer. We iterate over the input elements, apply the arithmetic function under test $\circ$ and store the results in $c[i] \leftarrow a[i] \circ b[i]$. We deliberately use small arrays that easily fit into the CPU's L1 cache to minimize memory bandwidth and latency impacts.

We also take into account that modern Intel CPUs can perform up to four integer operations in parallel. Therefore, to fully utilize all available ALUs, we additionally unrolled all loops by a factor of two and four, so that two or four results are computed within one loop iteration. The experiment is repeated 200k times and as a result, we take the time of that run that performs best.

The Clang++ compiler front-end is parameterized with `-O3` and `-march=core-avx2` to produce code that is optimized for performance. Further we used `-fno-vectorize` to prevent the compiler from automatically producing SIMD code.

6.1. Available Instructions

The AVX-2 instruction set provides almost all basic arithmetic operations for the most common data types. For the implementation of database systems, in particular integer

		Operation						
		+	-	*	/	min	max	abs
Data type	`int 8`	✓	✓	–	–	✓	✓	✓
	`int 16`	✓	✓	✓	–	✓	✓	✓
	`int 32`	✓	✓	✓	–	✓	✓	✓
	`int 64`	✓	✓	–	–	–[2]	–[2]	–[2]
	`unsigned int 8`	✓[1]	✓[1]	–	–	✓	✓	
	`unsigned int 16`	✓[1]	✓[1]	–	–	✓	✓	
	`unsigned int 32`	–	–	✓	–	✓	✓	
	`unsigned int 64`	–	–	–	–	–[2]	–[2]	

[1] = saturation arithmetic, [2] = available in the upcoming AVX-512 instruction set

Table 6.1.: Available arithmetic operations in AVX-2.

operations are very important as many high level SQL data types are represented internally as integers.

Table 6.1 gives an overview about the availability of SIMD arithmetics and special math functions. It is noteworthy that not for all scalar operations a vectorized counterpart exists. Especially, the integer division is missing for all integer types and for signed 64-bit integers, only the addition and subtraction operations are implemented in hardware. Further, the availability of vector operations is not symmetric w. r. t. signed and unsigned integer types. Although, some of the missing operations will be available in the upcoming CPU generation with the AVX-512 instruction set, most of the operations that are yet implemented in AVX-2 will not be implemented on the larger 512-bit registers. Therefore, AVX-512 can be seen as a supplemental instruction set.

In the following we only consider signed arithmetics, as most of the signed arithmetics operations are directly available in hardware.

6.2. Relevance of Arithmetic Operations

As block-wise query processing primarily targets OLAP workloads we use the TPC-H benchmark as a reference to roughly quantify the importance of individual arithmetic operations. Among all 22 TPC-H queries the arithmetic operations are made of 61% additions/subtractions (including aggregations), 26% multiplications, 10% division and 3% min/max selection (Table 6.2), whereas 20% of the multiplications and 100% of the divisions are performed *after* an aggregation. Arithmetic operations that are performed after an aggregation are not considered as performance critical, because the number of aggregated tuples is much smaller than the cardinality of the scanned table. Therefore, the absence of integer division is not crucial for implementing the map operator. In general, the query compiler handles missing vector operation by falling back to the corresponding scalar operation.

Switching between vector- and scalar-operations in general causes significant performance impacts (which we show in the later section 7.4) and should therefore be avoided whenever possible. In contrast to the integer division, additions and multiplications are performed in heavily frequented query pipelines. For instance in Q1, eight additions and two multiplications are performed on each tuple, thus they play an important role regarding the overall query runtime.

		Operation				
		+/-	*	/	min	max
Query	1	7	2	3		
	2				1	
	3	2	1			
	4	1				
	5	2	1			
	6	1	1			
	7	2	1			
	8	3	1	1		
	9	3	2			
	10	2	1			
	11	3	4			
	12	2				
	13	2				
	14	3	2	1		
	15	2	1			1
	16	1				
	17	2	1	2		
	18	2				
	19	2	1			
	20	1	1			
	21	1				
	22	3		1		
	$\sum$	**47**	**20**	**8**	**1**	**1**

Table 6.2.: Number of arithmetic operations in the TPC-H benchmark (including aggregations).

6.3. Vectorized Multiplication

A common vector operation takes two input vectors $\overrightarrow{a}$, $\overrightarrow{b}$ performs an operation $\circ$ and produces an output vector of the same size as the input vectors. An exception is the vector multiplication which has some specialties: 1.) The available multiplication instructions for the different data types are inconsistently implemented in hardware. 2.) In general, the bit-width of the products is twice the bit-width of the arguments. 3.) AVX-2 (as well as SSE) only implement the 16- and 32-bit multiplication. Therefore, instructions that operate on 8- and 64-bit integers are not directly available in hardware. The following list, taken from the Intel Intrinsics Guide [5], shows the available instructions and short descriptions:

- `__m256i _mm256_mul_epi32 (__m256i a, __m256i b)`
  "Multiply the low 32-bit integers from each packed 64-bit element in `a` *and* `b`*, and store the signed 64-bit results in* `dst`*."* [1]
- `__m256i _mm256_mulhi_epi16 (__m256i a, __m256i b)`
  "Multiply the packed 16-bit integers in `a` *and* `b`*, producing intermediate 32-bit integers, and store the high 16 bits of the intermediate integers in* `dst`*."*
- `__m256i _mm256_mulhi_epu16 (__m256i a, __m256i b)`
  "Multiply the packed unsigned 16-bit integers in `a` *and* `b`*, producing intermediate 32-bit integers, and store the high 16 bits of the intermediate integers in* `dst`*."*
- `__m256i _mm256_mullo_epi16 (__m256i a, __m256i b)`
  "Multiply the packed 16-bit integers in `a` *and* `b`*, producing intermediate 32-bit integers, and store the low 16 bits of the intermediate integers in* `dst`*."*
- `__m256i _mm256_mullo_epi32 (__m256i a, __m256i b)`
  "Multiply the packed 32-bit integers in `a` *and* `b`*, producing intermediate 64-bit integers, and store the low 32 bits of the intermediate integers in* `dst`*."*

As the list shows, all multiplications double the bit-width of the argument, but only the `_mm256_mul_epi32` instruction also returns the result with a doubled bit-width. Whereas all other operations just return either the higher or the lower bits of the result value.

For our prototypal query compiler we implemented two different kind of SIMD multiplications. (i) The *type-preserving* multiplication, that produce output vectors of the same type as the input vectors, similar to the vector addition/subtraction. (ii) The *type-extending* multiplication, where the bit-width of the output is doubled, like in `_mm256_mul_epi32`. Because, the product of two n-bit integers can always be represented as a $2n$-bit integer, the type-extending implementations cannot cause integer overflows. - Note, that the handling of overflows is discussed in detail in section 6.4.

6.3.1. Type-Extending Multiplication

First, we focus on the implementations that double the bit-widths. Because the `_mm256_mul_epi32` intrinsics function serves as a role model for the following implementations, we first explain how the function internally works.

The function expects the 32-bit arguments to be stored in 64-bit lanes, which means that each 32-bit input element occupies 64-bit of the YMM-register. The instruction treads the lower 32-bits of every 64-bit lane as a input, whereas the higher 32-bits are

[1] *"*`dst`*" refers to the return value of the intrinsics function.*

ignored. In other words, the multiplication expects spaces of 32-bit between the input elements. In database systems that store attributes in column-oriented fashion, this has the implication that columns cannot directly serve as an input for the multiplication instruction. Instead, an additional *shuffle* or *convert* instruction is necessary beforehand. In AVX-2 the most efficient way to do this, is to employ the `_mm256_cvtepi32_epi64` (`vpmovsxdq`) instruction that converts four 32-bit integer stored in a XMM register into four 64-bit integers, as sketched in figure 6.1.

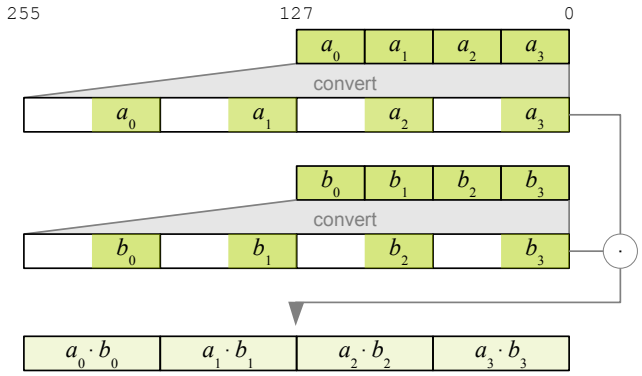


Figure 6.1.: Type-extending SIMD multiplication.

Multiplying two input vectors with 4 packed elements each, results in (at least) 3 vector instructions. The involved type conversion instruction takes a 128-bit register as input and writes its output to a 256-bit register. A multiplication of four elements may also result in 4 vector instructions, if the input elements are stored in the higher 128 bits of an YMM register. In that case an additional *extract* instruction is issued to move the contents into the lower 128 bits of an YMM register.

Based on the available instructions (listed on page 30) we implemented the 8- and 16-bit multiplication the same way.

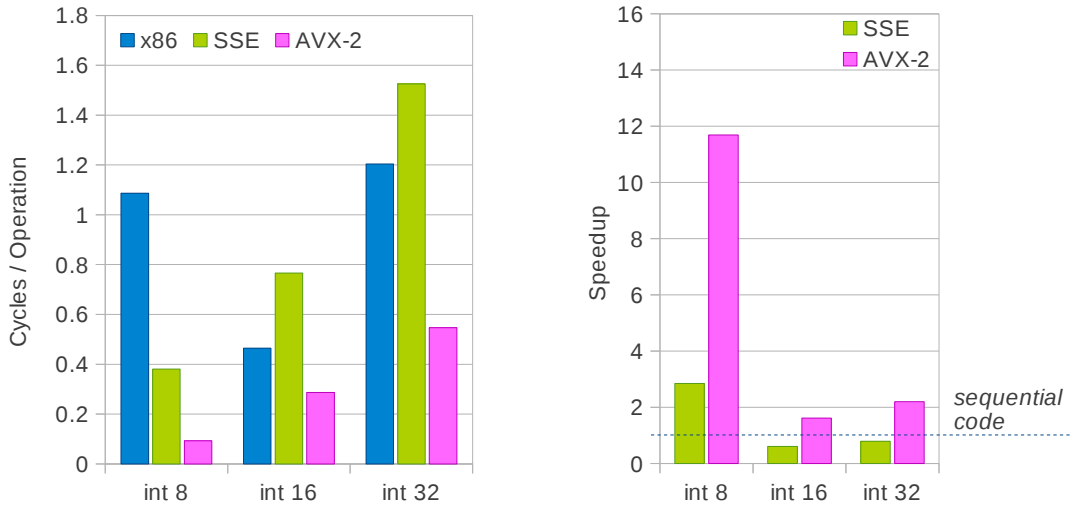
```
void mul16_epi8_to_epi16_avx2(__m128i* a, __m128i* b, __m256i* dst) {
    __m256i a16;
    __m256i b16;
    convert16_epi8_to_epi16_avx2(a, &a16);
    convert16_epi8_to_epi16_avx2(b, &b16);
    dst[0] = _mm256_mullo_epi16(a16, b16);
}
```

```
void mul8_epi16_to_epi32_avx2(__m128i* a, __m128i* b, __m256i* dst) {
    __m256i a32;
    __m256i b32;
    convert8_epi16_to_epi32_avx2(a, &a32);
    convert8_epi16_to_epi32_avx2(b, &b32);
    dst[0] = _mm256_mullo_epi32(a32, b32);
}
```

A type-extending 64-bit multiplication has not been implemented, because the developed prototype does not support 128-bit integers. Nevertheless, we implemented a *type-preserving* 64-bit multiplication that supports overflow detection (see section 6.3.2).

Figure 6.2 shows a performance comparison of the individual implementations with varying data types. Especially the speedups are much lower as one might expect. For

example, in AVX-2 we can perform four 32-bit integer multiplication at once, but because we also have to take the type-conversion into account, we only achieve a speedup of ~2. In SSE, using 128-bit registers, only two operations can be performed at once, here the costs for type-conversion do not amortize. Only the 8-bit multiplication benefits from SSE instructions. In general, a type-extending multiplication reduces the degree of parallelism. For example, the 32-bit multiplication processes four elements in parallel within three instructions, thus the DoP($*_{i32\to i64}$)=$1.\overline{3}$.



(a) Performance in [Cycles / Multiplication]

(b) Speedup of SIMD implementations compared to sequential code.

Figure 6.2.: Performance comparison of type-extending multiplication with varying data types.

6.3.2. Type-Preserving Multiplication

In this section we describe the type-preserving implementations. For the 16- and 32-bit multiplication, we can directly use the corresponding intrinsics functions, as shown below. Only the 8- and 64-bit multiplication need to be implemented manually.

Operation		Intrinsics Function
$*_{i16}$	$\longleftrightarrow$	`_mm256_mullo_epi16`
$*_{i32}$	$\longleftrightarrow$	`_mm256_mullo_epi32`

8-Bit Integer Multiplication The (probably) most efficient way to implement a 8-bit multiplication in AVX-2 is to use the `_mm256_mullo_epi16` function as sketched in figure 6.3. Both input vectors are first type-converted into 16-bit integers and after the multiplication has performed, the lower 8-bits of each 16-bit element are shuffled into upper/lower 64-bits of the 256-bit register. Finally, the contents of the upper and lower 128 bit are combined using a bitwise `OR`. This intermediate step is necessary, because we can only shuffle bits within 128-bit boundaries. For bit-shuffling, we employ the `_mm256_shuffle_epi8` (`vpshufb`) instruction. In total, 5 C++ function calls are required to multiply both vectors, which are translated into 6 native instructions. Therefore, a single multiplication of two elements is performed within 0.375 instructions, in average. The additional instruction is caused by the fact, that we interpret a YMM-register as two XMM-registers. The `OR`

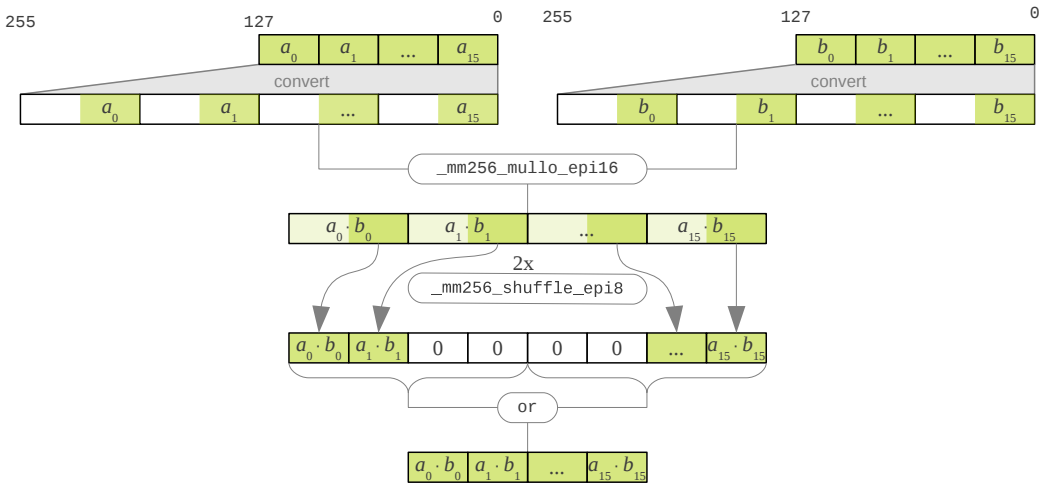


Figure 6.3.: SIMD multiplication of 8-bit integers.

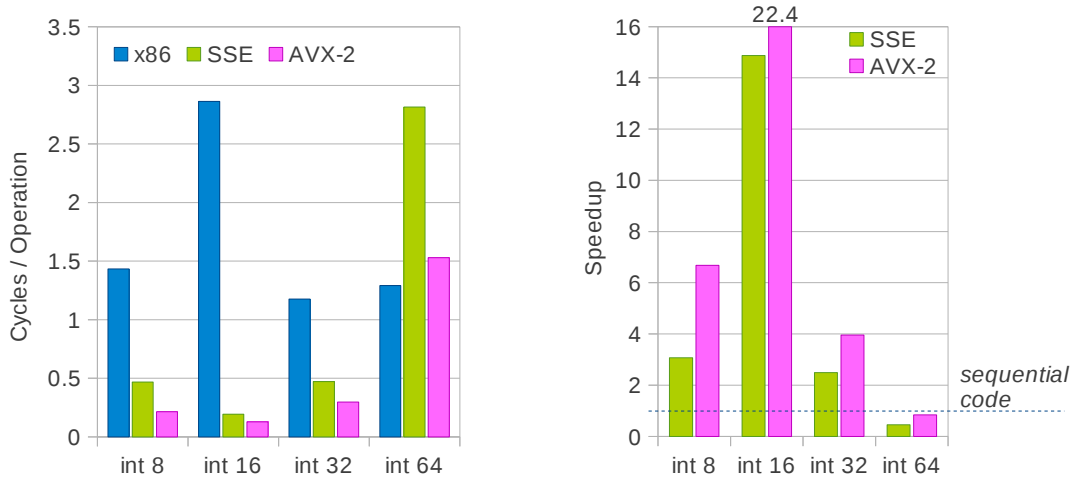
instruction cannot access the higher 128-bit of a YMM-register, therefore an additional `_mm256_extracti128_si256` (`vextracti128`) instruction is issued to move the content into a regular XMM-register. In this case, we can avoid the *extraction* if we only employ AVX-2 instructions. The alternative implementation is described in appendix A. Unfortunately, the *AVX-2-only* version is ~14% slower. The performance decrease is caused by the `_mm256_permute2x128_si256` (`vperm2i128`) instruction, which has very high latencies.

64-Bit Integer Multiplication A 64-bit multiplication is composed of multiple 32-bit multiplications as shown in listing 6.1. The implementation has little room for optimizations. We only exploit the fact, that the `_mm256_mul_epu32` ignores the higher 32 bits of the input elements, therefore we do not have to mask out the higher bits. In total, the multiplication of four 64-bit integer results in 9 vector instructions and therefore in 2.25 instructions per operation.

```
void mul4_epi64_avx2(__m256i* a, __m256i* b, __m256i* dst) {
    const __m256i hi_a = _mm256_srli_epi64(a[0], 32);
    const __m256i hi_b = _mm256_srli_epi64(b[0], 32);
    const __m256i t1 = _mm256_mul_epu32(a[0], hi_b);
    const __m256i t2 = _mm256_mul_epu32(a[0], b[0]);
    const __m256i t3 = _mm256_mul_epu32(hi_a, b[0]);
    const __m256i t4 = _mm256_add_epi64(_mm256_slli_epi64(t3, 32), t2);
    dst[0] = _mm256_add_epi64(_mm256_slli_epi64(t1, 32), t4);
}
```

Listing 6.1: SIMD multiplication of 64-bit integers.

The performance comparison is shown in figure 6.4. As expected, the 64-bit multiplication does not benefit from SIMD instructions, whereas the 16- and 32-bit multiplications are two times faster compared to the type-extending implementations.



(a) Performance in [Cycles / Multiplication]

(b) Speedup of SIMD implementations compared to sequential code.

Figure 6.4.: Performance comparison of type-preserving multiplication with varying data types.

6.4. Overflow Handling

For all database queries that contain integer arithmetics the database system has to be aware of possible integer overflows. Basically, there a two different strategies to handle overflows. First, the database system evaluates queries in a way that *prevents* overflows. Second, the system explicitly *checks* if an overflow occurred right after an arithmetic operation has been performed. Both strategies are not mutual exclusive and can therefore be used within a single query.

6.4.1. Preventing Overflows

A simple way to prevent overflows is to convert the operands into a wider type before the arithmetic operation is performed. The SIMD multiplication instruction is a suitable example as it prevents overflows by doubling the bit-width of the data type. More evolved implementations make use of meta data, e. g. min/max values, to choose the smallest data type that is guaranteed to represent the result correctly. For systems that dynamically create code at runtime, an overflow prevention strategy has great potential in terms of performance, because overflow handling can be shifted from runtime to compile-time. Therefore, the number of instructions issued by mathematical operations can at least be halved compared to the overflow detection strategy, which incurs at least one additional instruction to check the CPU's status register. However, there are cases where overflow prevention also has negative impact on runtime. For example, if the system determines (based on the meta data) that an overflow *might* occur, it casts the operands into larger types, even if the overflow is very unlikely or actually never happens. The casting operation itself and as well as the usage of larger data types incur additional runtime costs, especially if the type's bit-width exceeds the machine's register bit-width. Therefore, this can be considered as a pessimistic approach.

Basically, expressions of the form $attribute \circ constant$ are considered harmless, because the target type can be determined precisely based on the given min/max-intervals. In

contrast, for expressions of the form $attribute_1 \circ attribute_2$ the database system can not determine the smallest target type exactly, as illustrated in the following example.

Given a relation $\mathcal{R} := \{[A, B]\}$ where $\forall a \in A : a \in [0, 100]$ and $\forall b \in B : b \in [0, 100]$

R	
A	B
0	100
42	35
100	13
47	0

and the expression $e := a + b$. If we compute the range of e based on the ranges of a and b, we get $e \in [0, 200]$. Because a (signed) 8-bit integer cannot represent the value 200, we have to type-convert into 16-bit integers, although the concrete values of e never exceed the domain of an 8-bit integer. This is not a big issue with 8-bit integers, but it becomes costly if we unnecessarily cast from 32-bit to 64-bit integers or to even larger types.

6.4.2. Detecting Overflows

Modern x86 CPUs provide a special status register that helps detecting integer overflows. Even though high level languages like C++ do not support overflow detection as part of the language specification, compilers like LLVM/Clang++ provide special *built-in functions*[2] for checked arithmetic operations. For example the built-in function

```
bool __built-in_saddll_overflow(long long x, long long y, long long *sum);
```

performs a checked 64-bit integer addition. The returned boolean signals, if an overflow occurred. Unfortunately, these functions are only available for scalar operations. If we employ SIMD instructions, overflow detection needs to be performed manually, without hardware support.

In [20], Warren presents efficient implementations for almost all scalar arithmetic operations. In the following we show the overflow detection implementations for addition and multiplication and re-implement them using SIMD instructions:

Addition An integer overflow caused by addition (or subtraction) can be easily detected using the bitwise operators `XOR` and `AND` as follows:

```
result = a + b;
status = (result ^ a) & (result ^ b);
```

If the most significant bit in `status` is set, an overflow occurred. Otherwise the msb is 0. The manual check requires three additional instructions plus an additional instruction to extract the overflow bit, which can be done for example by right-shifting `status >> ( size_of(status)* 8 - 1)`.

The following listing shows the complete function for adding 16-bit integers. In addition to the two operands, the functions expects a third (reference) argument, where the overflow flag is stored. The function is designed to preserve already set `overflow` flags, thus multiple arithmetic operations can be performed and subsequently, the overflow flag is checked only once.

[2] *http://clang.llvm.org/docs/LanguageExtensions.html#checked-arithmetic-builtins*

```
int16_t add_i16(int16_t a, int16_t b, int64_t& overflow) {
    const int16_t result = a + b;
    const int64_t local_overflow = ((result ^ a) & (result ^ b)) & 0x8000;
    overflow |= local_overflow;
    return result;
}
```

Listing 6.2: Checked (scalar) integer addition without hardware support.

The implementation of a SIMD version is very straightforward, as for all required operations a corresponding AVX-2 instruction exists. Only the update of the `overflow` variable differs from the scalar version. Here, we employ the `movemask` instruction to collect the most significant bits of each element. Because the `movemask` instructions interprets the content of `local_overflow` as 8-bit integers, we additionally have to mask out some bits if the addition is performed on larger types. The listing 6.3 shows a 16-bit integer addition, therefore we have to mask out every second bit starting at position 0, as illustrated in Figure 6.5 on the next page. The gabs between the individual bits can be ignored, because for overflow detection, we only need to know if *any* bit is set.

```
void add16_epi16_avx2(
           __m256i* a, __m256i* b, __m256i* dst, int64_t& overflow) {
    const __m256i result = _mm256_add_epi16(a[0], b[0]);
    const __m256i a_xor_result = _mm256_xor_si256(a[0], result);
    const __m256i b_xor_result = _mm256_xor_si256(b[0], result);
    const __m256i local_overflow =
                      _mm256_and_si256(a_xor_result, b_xor_result);
    overflow |= _mm256_movemask_epi8(local_overflow) & 0xaaaaaaaa;
    dst[0] = result;
}
```

Listing 6.3: Checked integer addition in AVX-2.

In total, a checked addition requires 5 vector instructions + 2 scalar instructions. Listing 6.4 shows a slightly improved implementation which is based on *saturation arithmetic*. With saturation arithmetics no overflows (or underflows) occur. If the result exceeds the range of the data type, the result is either set to the maximum possible value, in case of overflows, or to the minimum possible value, in case of underflows. For example, if the range is $[-128, 127]$, then $120 +_{\text{sat}} 10 = 127$ or $0 -_{\text{sat}} 200 = -128$. An operation is considered as overflowed, if the result value is either the min. or the max. possible value. This leads to a little inaccuracy at the boundaries, because in the given implementation it is undecidable whether the operation really overflowed or it just results in one of the two extreme values. E.g. the operation $120 +_{\text{sat}} 7$ erroneously reports an overflow.

```
void add16_epi16_avx2_saturation(
           __m256i* a, __m256i* b, __m256i* dst, int64_t& overflow) {
    const __m256i result = _mm256_adds_epi16(a[0], b[0]);
    const __m256i is_min = _mm256_cmpeq_epi16(result, MIN_EPI16_VEC);
    const __m256i is_max = _mm256_cmpeq_epi16(result, MAX_EPI16_VEC);
    const __m256i is_min_or_max = _mm256_or_si256(is_min, is_max);
    overflow |= _mm256_movemask_epi8(is_min_or_max);
    dst[0] = result;
}
```

Listing 6.4: Checked integer addition in AVX-2 using saturation arithmetic.

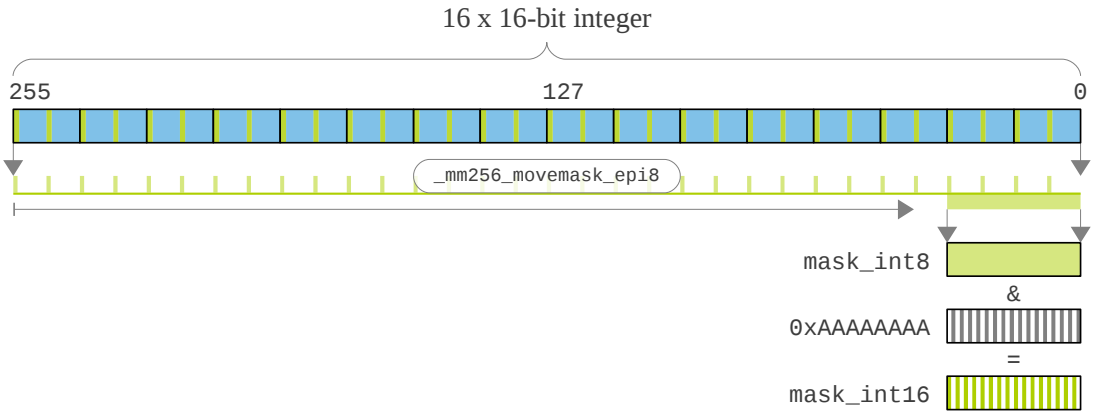


Figure 6.5.: Extraction of the most significant bits of packed 16-bit integers using the `_mm256_movemask_epi8` (`vpmovmskb`) instruction.

Compared to the first implementation, we only save one scalar operation. Micro-benchmarks have shown a little performance improvement of ~1%, if we use saturation arithmetics. Further, saturation arithmetics are only available for 8- and 16-integer addition/subtraction.

Figure 6.6 on page 39 shows the performance of arithmetic addition with different integer types. It compares the scalar- with the vector-operations as well as the checked with the unchecked version. For comparison, we also implemented the operations using the older SSE instruction sets. We observed a 49% performance decrease[3] with the 256-bit AVX-2 instructions, whereas the hardware supported overflow detection with scalar operations causes as 40% performance decrease. In SSE, manual overflow detection adds 55% overhead, but in all cases, SIMD processing amortizes the additional costs and improves the computational performance, as shown in the table 6.3. In average, AVX-2 is 1.6 times faster then SSE.

	SSE		AVX-2	
	unchecked	checked	unchecked	checked
`int 8`	15.0	29.1	22.7	43.4
`int 16`	7.6	7.2	11.6	10.7
`int 32`	3.0	2.6	4.3	4.1
`int 64`	1.6	1.4	2.4	2.3

Table 6.3.: Speedup of vectorized addition (with and without overflow checks).

Multiplication As mentioned before, the product of two n-bit integers can always be represented as a $2n$-bit integer. To implement a manual overflow detection, as described in [20], we need to have access to the higher n bits of the product. If we assume that both operands have the same bit-width n, we can detect overflows as follows:

$$\mathrm{hi}(a \cdot_n b) \neq (\mathrm{lo}(a \cdot_n b) \ggg (n-1)) \tag{6.1}$$

[3] Median over all four data types.

Where hi and lo let us access the higher and lower n bits of the product and $\ggg$ denotes the signed right shift. Informally speaking, the upper halve must only contain sign bits and these sign bits need to be equal to sign bit in the lower halve. Otherwise, an overflow occurred.

Basically, the implementations with overflow detection are based on the type-extending multiplications as they give us access to the higher bits of the product. However, due to the fact, that the *signed right shift* instruction is not available for all integer types, we had to have find alternative implementations. The following list describes the most efficient implementations that have been found on our Haswell machine:

- *8-bit*: The 8-bit integer multiplication differs slightly from the above equation, because there is no *signed right shift* on 8-bit integers. Instead, the implementation does a "double shift" on 16-bit elements. The algorithm is illustrated in figure 6.7a on page 40. For the multiplication itself, we use the alternative type-preserving ("AVX-2 only") implementation which is shown in appendix A.1. The aforementioned latencies caused by the permutation instruction are hidden by the overflow detection code, which makes the AVX-2-only version the best performing implementation for 8-bit integer multiplication.
- *16-bit*: For multiplying 16-bit integer, we make use of the `_mm256_mulhi_epi16` and the `_mm256_mullo_epi16` instructions. Basically, the multiplication is performed twice, but it avoids the more costly type conversions, which makes the 16-bit multiplication the most efficient implementation among the others in terms of [cycles/operation].
- *32-bit*: The overflow checks in 32-bit multiplications is the most similar to the above equation. Here, we produce some *don't care* (DC) values that have to be masked out later on, but micro-benchmarks have shown, that this approach performs better compared to alternative implementations that rely on shuffling to avoid these DCs. The implementation is sketched in figure 6.7b.
- *64-bit*: The implementation for multiplying 64-bit integers is based on the type-preserving implementation (as shown in the previous section 6.3.2) and the overflow detection is similar to those of the 32-bit multiplication.

The checked 64-bit integer multiplication is the only arithmetic operation, that does not benefit from AVX-2 instructions. Compared to the sequential code, the SIMD code is 40% slower. Therefore, in queries that make use of checked arithmetics, the query compiler should fall back to sequential processing. Switching between scalar- and vector-operations within a single query incurs an additional overhead caused by moving data from vector registers into regular registers and vice versa. These additional costs are investigated in section 7.4. Note, that even though the unchecked 64-bit multiplication is 20% slower, it is still reasonable to use the AVX-2 version to avoid costly data movements between the different registers.

Figure 6.6 shows a performance comparison among all implementations in [cycles/operation]. The red bars denote the overhead caused by overflow detection.

6.5. Min/Max

As the table 6.1 (on page 28) shows, the vector extensions also lacks of min/max implementation for 64-bit integers. For completeness, we show how these functions are implemented in SIMD.

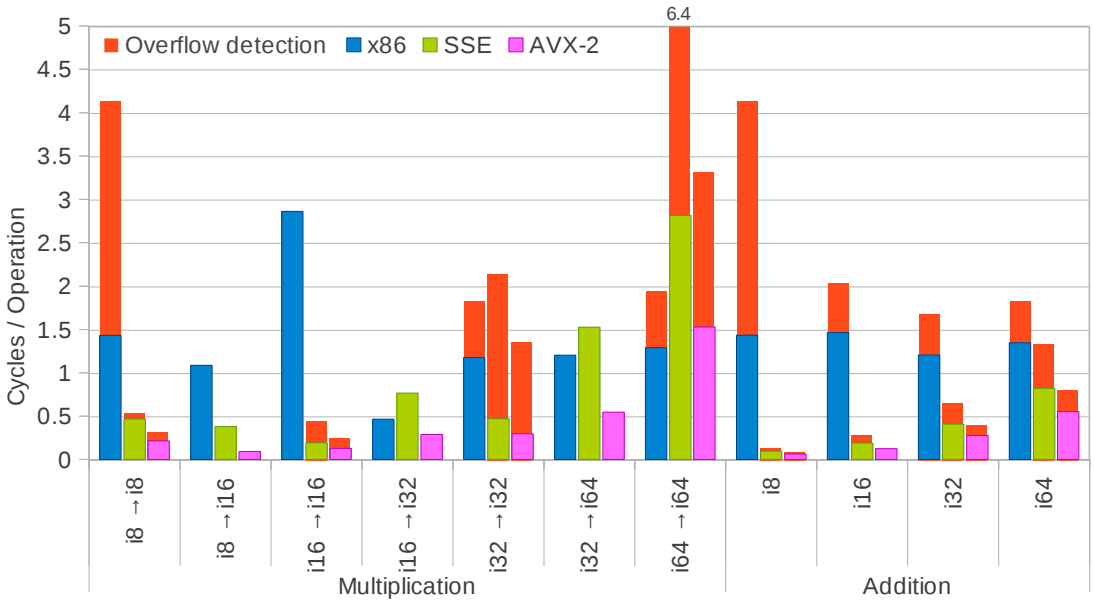


Figure 6.6.: Performance impact of overflow detection on arithmetic operations.

	SSE			AVX-2		
	unchecked	checked	type-ext.	unchecked	checked	type-ext.
`int 8`	3.1	7.7	2.9	6.7	13.0	11.7
`int 16`	14.9	5.5	! 0.6	22.4	10.0	1.6
`int 32`	2.5	! 0.9	! 0.8	4.0	1.3	2.2
`int 64`	! 0.5	! 0.3	n/a	! 0.8	! 0.6	n/a

Table 6.4.: Speedup of vectorized multiplication (with and without overflow checks).

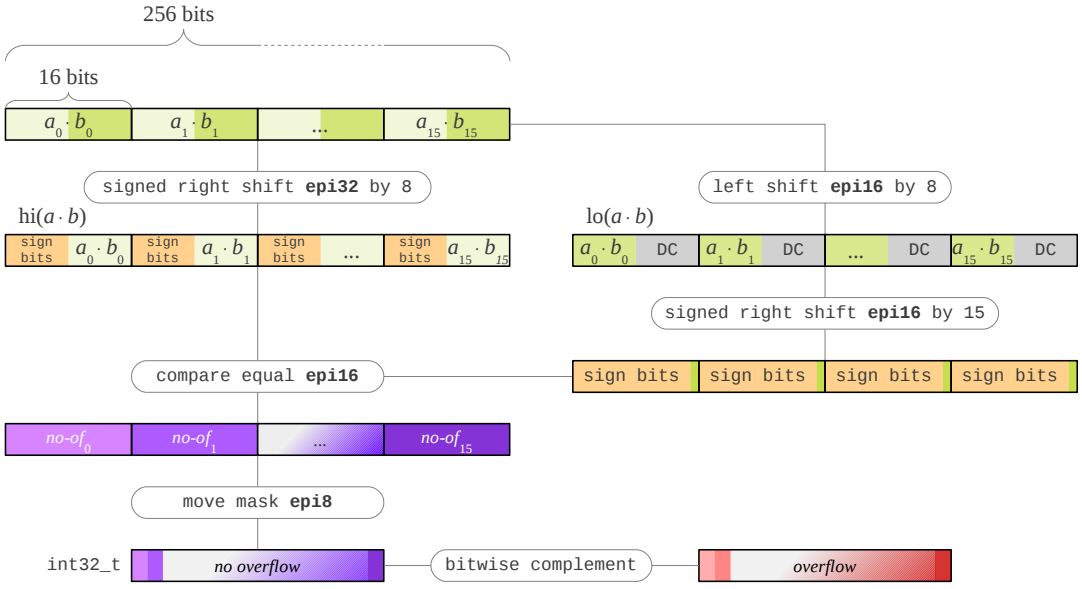
Computing the min or max basically takes two instructions, whereas the first is a less-than or a greater-than comparison and the second is a blend instruction that copies elements from two source registers into a third destination register using a control mask. As control mask we can directly use the bit mask that is produced by the comparison instruction (see listing 6.5).

6.6. Hashing

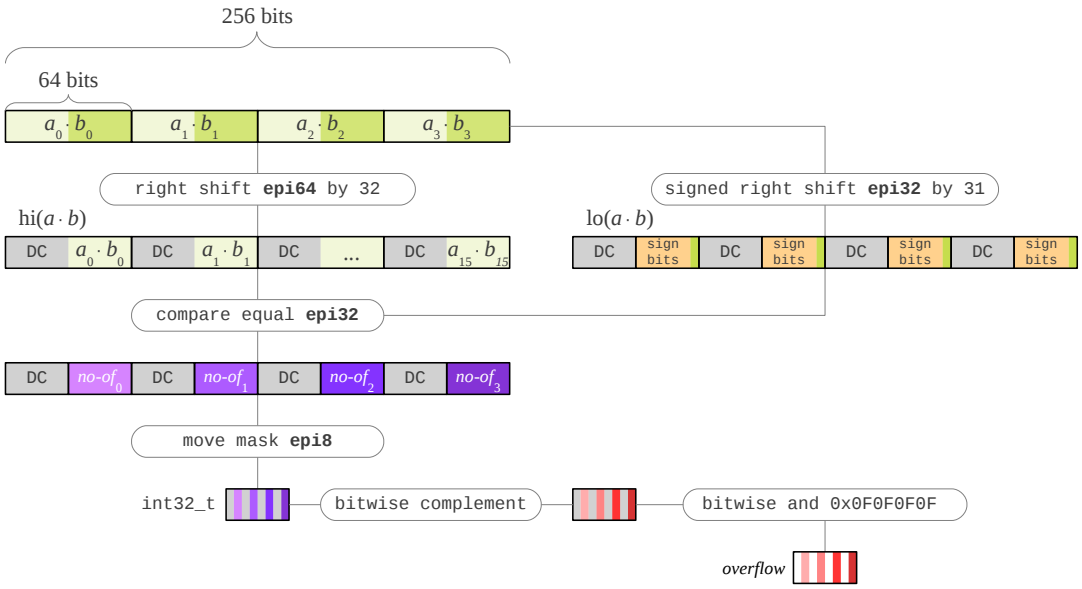
Beside the basic arithmetic operations, we also evaluated three popular hash functions in SIMD. Even though we do not investigate hash tables in this work, the computation of hash values can be pushed to the vectorized code part of a query and can therefore be

```
__m256i _mm256_min_epi64(__m256i a, __m256i b) {
    __m256i mask = _mm256_cmpgt_epi64(a, b);
    __m256i result = _mm256_blendv_epi8(a, b, mask);
    return result;
}
```

Listing 6.5: Selecting the minimum using the `_mm256_blendv_epi8` (`vpblendvb`) instruction.



(a) Overflow detection in 8-bit integer SIMD multiplication. - Extension of figure 6.3



(b) Overflow detection in 32-bit integer SIMD multiplication.

Figure 6.7.: Different overflow detection implementations due to missing *signed right shift* instructions.

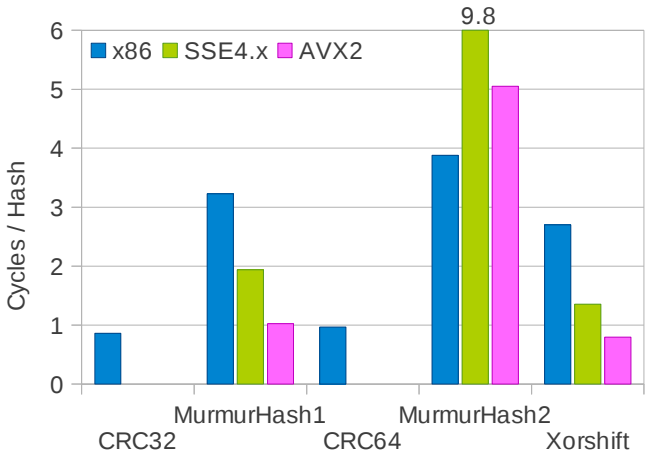


Figure 6.8.: Hash computation performance.

processed in parallel.

For our experimental evaluation, we choose the MurmurHash 1 & 2 [1] and the Xorshift [11] functions. The latter is actually a pseudo-random number generator but it can be alienated as a hash function. The Xorshift is very light-weight as it is computed in only seven bit operations, whereas the MurmurHash2 is considered more heavy-weight as it performs four 64-bit multiplication plus seven additional logical operations. Due to the fact that 64-bit multiplication is relatively slow in SIMD, we decided to additionally implement the MurmurHash2 *64B* version, which is optimized for 32-bit architectures as it only relies on 32-bit multiplications. With the 64B version we observed a 18% performance increase with AVX-2 compared to the 64A version of MurmurHash2. With SSE, the 64B version outperforms the 64A version by 20%. Therefore, we omit the results of MurmurHash2 *64A*.

We have to mention, that our implementations of MurmurHash2 slightly differs from the original versions. Originally, the function is designed to hash inputs of variable length, whereas our modified implementation is limited to hash a single 64-bit input value. The same applies for the MurmurHash1, except that it hashes 32-bit inputs.

Since Intel implemented the cyclic redundancy check (CRC) [16] in hardware, beginning with the Nehalem architecture, the CRC function also became very attractive to be used as a hash function. Therefore, we also consider the 32- and 64-bit CRC functions in our evaluation.

The results in figure 6.8 clearly show that MurmurHash2 function implemented in SIMD cannot compete with its sequential counterpart. In contrast, the 32-bit MurmurHash1, which computes eight hash values at ones, shows a speedup of more than 3 x. However, the CRC functions only take ~1 CPU cycle, and the only function that can slightly underbid this, is Xorshift implemented in AVX-2.

6.7. Summary

In this chapter we investigated the efficiency of SIMD arithmetics. Especially additions/-subtraction can highly benefit from SIMD instructions. With SSE we observed speedups that are close to the number of packed elements P: $S_{\text{SSE}} = 0.86P$. In contrast, the relative speedup of 256-bit SIMD is significantly lower: $S_{\text{AVX-2}} = 0.64P$, but in absolute numbers, AVX-2 still outperforms SSE by a factor of 1.5. Even though that overflow detection has

to be implemented manually in SIMD, we only observed 9% lower speedups compared to the scalar operations.

With multiplications the picture changes dramatically. In particular the SIMD 64-bit multiplication, that has to be implemented using 32-bit multiplications, cannot compete with its scalar counterparts. Detecting overflows causes significant additional runtime costs, that makes the SIMD-256 implementation about 40% slower than the scalar implementation. Due to the fact, that modern CPUs can perform up to four scalar operations in parallel, arithmetics barely benefit from SIMD-128 instructions. While on the other hand it is safe to rely on SIMD-256 instructions for integer types of up to 32-bit.

The missing 64-bit multiplication also has noticeable performance impacts on hashing. The evaluation of the MurmurHash2 function has shown, that the AVX-2 version is more than 20% slower than the scalar version. Only the 32-bit MurmurHash1 and the 64-bit Xorshift functions can compete with the CRC function which is directly implemented in hardware.

7. Type Conversion

In the context of database research, type conversions are a very uncommon topic, because most database systems are developed in a high-level programming language, and from the perspective of a high-level programming language, type conversions are usually performed implicitly. E.g. if a program accesses a scalar 8-bit integer value, it is most likely loaded into a 64-bit general purpose register. During program execution a `movsbl` is performed that extends the sign bits to the bit-width of the register while the data is loaded from memory. In practice, it makes (almost) no different w.r.t. execution time, if we load a 64-bit or an 8-bit integer. Differences in runtime are usually due to memory bandwidth limitations and not due to type conversions. In general, we can say that a single scalar value reserves one register independently of its data type and conversions between different integer types do not incur additional costs.

However, if we employ SIMD instructions the picture changes dramatically. In SIMD processing, where a single register contains multiple packed elements type conversions has the implication that data may no longer fit into a single register after a type conversion. Further, dependent on the source- and target-data type, a conversion may also employ multiple instructions. Thus, type conversions become an *explicit* task that need to be handled efficiently.

The necessity for type conversions arises by the facts that (i) almost all binary operations we perform in SIMD during query processing, require both operands to be of the same data type, and (ii) we favor that each attribute is stored (and processed) in the narrowest possible type, as this increase the degree of parallelism. Thus, whenever we want to perform a binary operation on two attributes that are of different types, we have to explicitly convert them into their common super-type. Furthermore, in chapter 5 we already mentioned, that type conversions also appear during the evaluation of selection predicates. In that case, the selection mask is type-converted to match the bit-width of the selection-attribute, which may happen multiple times within a single query.

In this chapter we first give an example for type conversions in the context of the map operator and we discuss the implications. Then we present our implementations for type conversion in SIMD in the subsequent section. Again we focus on the new AVX-2 instruction set, but we'll also show a performance comparison with the older SSE instructions. In the last section, we take a look at the performance impacts of the conversion from vectors to scalar values and vice versa.

7.1. Arithmetic Expressions

Type conversions often occur during the evaluation of arithmetic expressions. In general, the evaluation of arithmetic expressions in SIMD is very similar in terms of code generation with its scalar counterpart. Internally, expressions are represented as *binary expression tree*s. During code generation, the tree is traversed in post-order and the corresponding code for each operation is emitted.

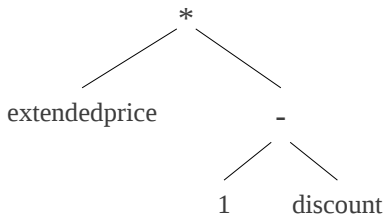


Figure 7.1.: An example binary expression tree.

For instance, figure 7.1 depicts the expression $extendedprice * (1 - discount)$ as it appears in TPC-H Q1. The *discount* attribute is internally represented as an 8-bit integer and the *extendedprice* attribute is a 32-bit integer. Due to the nature of x86-SIMD, both operands of an binary operation needs to be of the same data type and in general, this is achieved by converting the smaller operand into the type of the larger operand. In our example, a special case arises, because the final multiplication is *type-extending*. In that case, both operands need to be converted into 64-bit, as shown in figure 7.2.

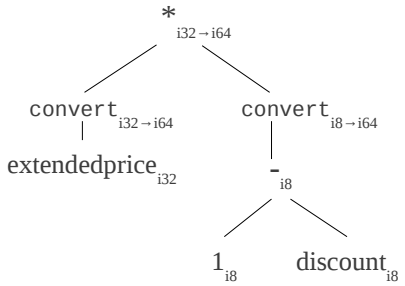
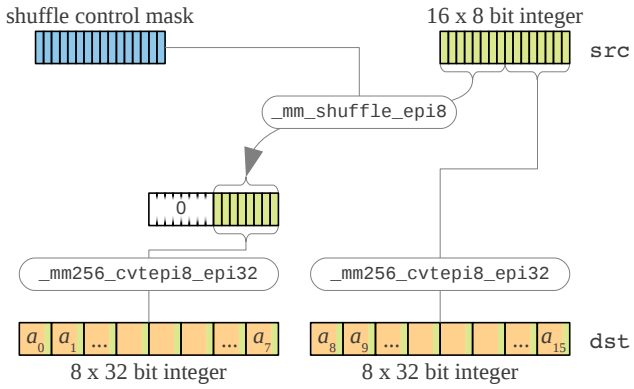


Figure 7.2.: A binary expression tree with type annotations and conversions.

In block-wise processing, each operation is performed on an entire block of tuples. This implies, that the degree of parallelism decreases with every type-conversion. Further, it results in a higher register pressure as the memory consumption of the attributes increases by factors. Assuming the block-size in the above example is set to $B = 32$, then the (sub-) expression $1 - discount$ is evaluated within a single instruction, whereas only two SIMD registers are allocated. In contrast, the subsequent multiplication is performed on 64-bit lanes and therefore, the thirty-two 8-bit elements need to be converted. After the type-conversion, the memory consumption has been increased from 32 bytes to 256 bytes which corresponds to 8 YMM register. The same applies for the second operand. Theoretically, all available SIMD register are then allocated only for the evaluation of a simple expression, whereas other attributes (and the selection mask) has been spilled to memory. However, the underlying LLVM compiler reorders the operations so that conversions and arithmetic operations are interleaved. Therefore, in practice the total number of allocated registers is significantly lower. Nevertheless, the example shows that type conversions in SIMD play a performance critical role.

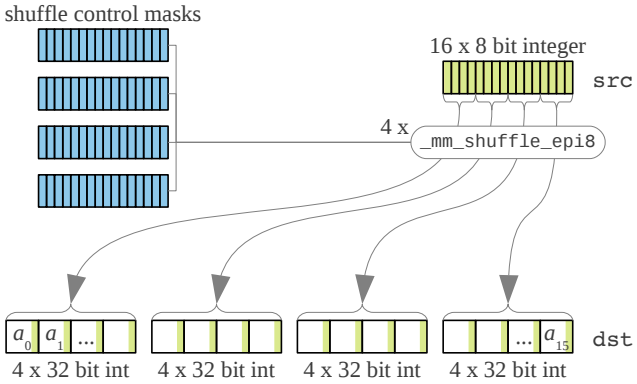
7.2. Implementation Details

In AVX-2, all converting instructions read the content of a XMM register and store their results in YMM registers. This implies, that if we cast into the next wider type, we again




```
# load 16 packed 8-bit integers
vmovdqa (%rsi,%rax,1),%xmm0
# convert and store the
# first 8 elements
vpmovsxbd %xmm0,%ymm1
vmovdqa %ymm1,(%rdx,%rax,4)
# load control mask and shuffle
# remaining elements
vpshufb 0x2110b8(%rip),%xmm0,%xmm0
# convert and store the
# remaining 8 elements
vpmovsxbd %xmm0,%ymm0
vmovdqa %ymm0,0x20(%rdx,%rax,4)
```

Figure 7.3.: Type conversion of sixteen packed 8-bit integers into 32-bit integers.




```
# load shuffle control masks
vmovdqa 0xb7f1(%rip),%xmm0
vmovdqa 0xb7f9(%rip),%xmm1
vmovdqa 0xb801(%rip),%xmm2
vmovdqa 0xb809(%rip),%xmm3
# load 16 packed 8-bit integers
vmovdqa (%rsi,%rax,1),%xmm4
# shuffle - store, 4 times
vpshufb %xmm0,%xmm4,%xmm5
vmovdqa %xmm5,(%rdx,%rax,4)
vpshufb %xmm1,%xmm4,%xmm5
vmovdqa %xmm5,0x10(%rdx,%rax,4)
vpshufb %xmm2,%xmm4,%xmm5
vmovdqa %xmm5,0x20(%rdx,%rax,4)
vpshufb %xmm3,%xmm4,%xmm4
vmovdqa %xmm4,0x30(%rdx,%rax,4)
```

Figure 7.4.: Unsigned type conversion of sixteen packed 8-bit integers into 32-bit integers.

can store the result within a single register. Therefore, the conversions i8 → i16 → i32 → i64 can all be performed within a single instruction. In contrast, if we increase the bit-width fourfold or eightfold, we have to issue multiple instructions to convert all packed elements in the source register, because the output of a single instruction is limited to a single register. E.g. the conversion into 32-bit integers is limited to eight elements per instruction, because at most eight 32-bit integers can be packed into a YMM register. If for example the source type is an 8-bit integer, the source register contains sixteen elements in total. In that case, the `_mm256_cvtepi8_epi32` (`vpmovsxbd`) instruction consumes eight source elements from the lower half of the source register, while the remaining elements, stored in higher halve, are ignored. In a second step, the remaining source elements are moved into the lower halve of the SIMD register and a second convert instruction is performed. This procedure is illustrated in figure 7.3. The listing right beside the figure shows the corresponding assembly code. Note, that due to missing right shift instructions, we have to use a shuffle instruction to move elements. The shuffle has the drawback, that a *control mask* needs to be loaded into a XMM register, which naturally requires an additional move instruction. Analogous, if we extend the bit-width eightfold (i8 → i64), three shuffles and four converts are required.

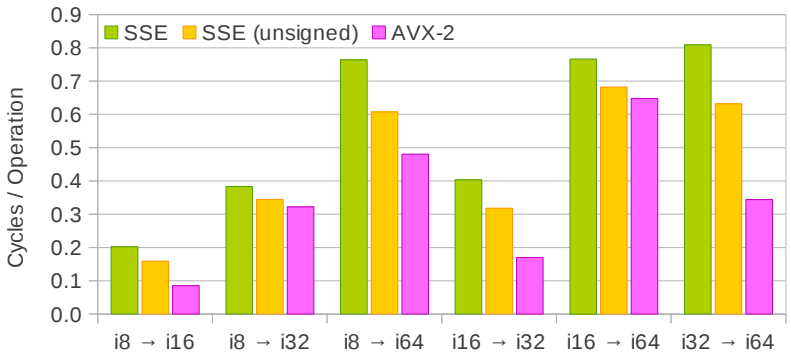


Figure 7.5.: Type conversion performance comparison.

7.3. Performance Comparison SSE vs. AVX-2

In SSE, things are quite different, as SSE instructions only operate on XMM registers. Here, basically all type conversions require additional shuffle instructions, this also includes the conversions i8 → i16 → i32 → i64. However, in SSE we can improve the overall performance by 22% in average, if we directly shuffle the source elements to their corresponding target positions without issuing a convert instruction. This optimization works as long as both operands are positive, because the sign bits are not extended this way. Unfortunately, this optimization does not work with AVX-2, because the shuffle instruction operates on XMM registers only and therefore does not fully utilize SIMD capabilities. As shown in figure 7.4, the unsigned conversion of sixteen packed integers would requires 13 instructions in total, whereas the signed conversion is performed in just 6 instructions.

To the best of our knowledge, the presented implementations are the most efficient ones. In average, the conversion takes 0.34 cycles per element with AVX-2. Compared to SSE, we observed speedups of 1.8 x and 1.5 x compared to "SSE-unsigned" (as shown in figure 7.5).

7.4. Mixing SIMD with Scalar Operations

As mentioned in previous sections, in some cases the query compiler has to fall back to scalar operations if the corresponding SIMD operation is not implemented in hardware. Accessing the individual components of a vector does not necessarily incur a type conversion, but from a high level perspective, it can be seen as some sort of conversion as well, as we convert vectors in multiple scalar values and vice versa. In this section we investigate the performance impacts of mixing scalar operations into a vectorized query pipeline.

As a subject of the investigation, we choose query number six (Q6) from the TPC-H benchmark. For our experiment, we synthetically sequentialized the computation of the arithmetic expression $discprice = extendedprice * discount$ and compared the runtime and the assembly generated by the LLVM compiler. The preceding type conversion of *extendedprice* and *discount* into 64-bit integers ensures, that both operands are first located in SIMD registers. The later aggregation $revenue = sum(discprice)$ is performed in parallel, thus the *discprice* must then be loaded back into SIMD registers again, and the circle is closed. In our experiment, we replaced the SIMD multiplication by a sequential loop:

```
for (uint64_t i = 0; i < 32; i+=1) {
     discprice[i] = extendedprice[i] * discount[i];
}
```

With this small change, the overall performance decreased by 66%. Due to the relatively large block-size of 32, no loop-unrolling is performed by the LLVM backend. As a result, ~15% of the overall runtime is spend in checking the loop condition and in branching. To remove the conditional branch, we unrolled the loop manually. This also paved the way for LLVM to produce more efficient code, but we still observed a performance decrease of 47%. An analysis of the assembly code has shown, that the conversion from vectors to scalars comes in many different "flavors". The LLVM compiler creates different sequences of instructions to perform the same task, which we do not want to discuss in detail. However, in all cases we observed that the data is spilled to memory at least once. E.g. in Q6, both operands are directly moved from SIMD to regular registers, whereas the results of the multiplications are first spilled to memory and then moved back into SIMD registers later on.

It is noteworthy, that data that is located in the higher 128 bits of an YMM register is never moved directly into regular registers. Instead, the higher 128 bits are first moved into a XMM register using the `vextracti128` instruction and then, the `vpextrq` instruction is used to extract the individual scalar values from the XMM register. The extraction of the higher 128 bits is usually done once for each vector-operand. Thus, the number of issued instruction is independent from the number of packed elements, but it additionally allocates one temporary SIMD register per input vector.

Figure 7.6 shows the runtime profile of the sequentialized code part of Q6 and table 7.1 gives a short description about the involved instructions.

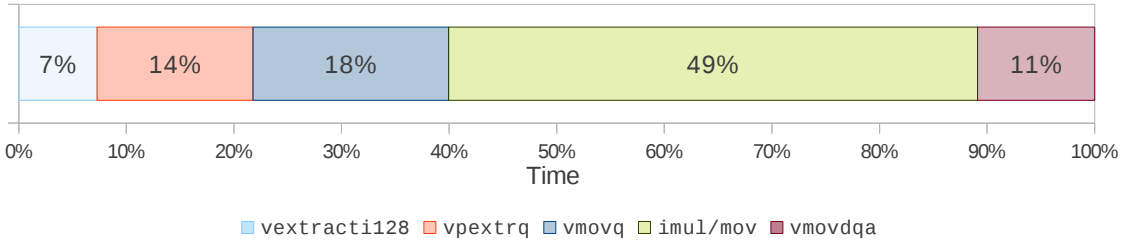


Figure 7.6.: Runtime profile of the sequential part of Q6.

`vextracti128`	move high 128 bits of an YMM- into a XMM-register
`vpextrq`	extract a scalar value from XMM- to regular register
`vmovq`	extract the first scalar value from a XMM register
`mov`	move data between regular register and memory
`vmovdqa`	move data between YMM registers and memory

Table 7.1.: Instructions that are involved in moving data between vector- and regular registers.

7.5. Summary

In this chapter we have shown that SIMD enabled query processing comes with additional costs for type conversions, what we would get almost for free if we would process data

sequentially. In average we have to count 0.34 additional cycles per element. Especially in databases, were attributes are preferably stored in narrow data types, type conversions may appear more often. Typically, queries that perform aggregation are potentially affected by type conversions, because the aggregates usually have a wider type than the individual attributes themselves. In general, conversion can be reduced if we adjust the all attributes to wider types. However, this would greatly reduce parallelism and the memory bandwidth might become the bottleneck. Beside type conversions we have shown that mixing sequential code into a vectorized query pipeline has severe impact on the overall performance. The costs that comes with sequential code parts are threefold: (i) Moving data from a SIMD register into a regular registers requires up to three instructions. (ii) In most cases, the results of scalar operations are spilled to cache memory, and must therefore be loaded into a SIMD register later on (filling); and (iii) the natural loss of parallelization has serious impact on the overall performance.

8. Aggregation and Grouping

In this chapter we show how aggregation is implemented in SIMD. We basically distinguish two different types of aggregation. 1.) The *scalar aggregation*, where aggregation functions are applied to one or more attributes. This kind of queries produce exactly one output tuple containing at least one aggregated value. 2.) The *group-by aggregation*, where aggregated values are computed individually for each group of tuples.

Even though, both implementations are quite different, they both rely on efficient arithmetics, which we presented in chapter 6. Note, that we focus on the most common aggregation function. More precisely, we only support aggregations that can be computed using a constant amount of space. E.g. median is not supported, due to its linear space complexity. On the other hand, average *is* supported, because it can be easily decomposed into a summation and a division: $\text{avg}(x) = \text{sum}(x)/\text{count}(x)$

8.1. Scalar Aggregation

Queries that perform scalar aggregations, are of the following form: `SELECT` $\texttt{f}_0\texttt{(a}_0\texttt{), ..., f}_n\texttt{(a}_n\texttt{)}$ `from <relations> where <conditions>` , whereas f_i are aggregation functions like `sum`, `avg`, `count`, etc. These functions reduce the attribute values a_i of each qualifying tuple to a single scalar value. The aggregation can be performed incrementally as the following pseudo-code snippet shows:

```
initialize variable aggr
for each qualifying tuple t do {
  aggr ← f(aggr, t.attr)
}
output aggr
```

We have to declare and initialize a variable for each aggregate, then we call the corresponding aggregation function for each qualifying tuple.

The basic idea for SIMD processing, which was originally presented in [24], is to compute multiple partial aggregates in parallel and store them in an array. Then, after all tuples have been consumed, a final reduction is performed on that array. In our block-wise approach, this array is initialized with a size of the block-size B. In each iteration, we process B tuples and update the corresponding partial aggregate, where the i^{th} element is the aggregation of the i^{th} tuple of each block.

In contrast to the sequential version, in our block-wise approach we have to take into account the non-qualifying tuples; as described in the earlier chapter 5. In the aggregation phase, we therefore make use of the *selection mask* to mask out all non-qualifying tuples. The key idea is to replace the affected attribute values in the current block by a neutral element w.r.t. the aggregation function. Then, the modified block is passed into the aggregation function. E.g. for summations, a bitwise `AND` is performed with the selection mask, which sets all elements that belong to non-qualifying tuples to zero. The following listing shows the (generic) algorithm in pseudo-code:

```
initialize array partial_aggr[B]
for each qualifying block b do {
  tmp[B] ← neutralize(b.attr, selection_mask)
  partial_aggr ← f(partial_aggr, tmp)
}
aggr ← reduce(partial_aggr)
output aggr
```

To keep things simple, the final reduction is done sequentially. Because the block-size is quite small (usually $B \leq 32$), the reduction phase is not a sweet spot for optimizations. Especially, because the reduction is independent from the input size.

8.1.1. Implementation Details

8.1.1.1. Count(*)

Counting the qualifying tuples is basically a summation, where the constant 1 is passed into the aggregation function for each qualifying tuple. In SIMD, we can optimize the computation of a `count(*)` by simply summing up the selection mask. If a tuple qualifies, all bits at the corresponding position of the selection mask are set to ones, which represents the integer value -1 in two's complement. Thus, if we pass the selection mask into the aggregation function, then we count in negative direction. After the final reduction phase is finished, we only have to toggle the sign of the aggregate. Zhou and Ross already mentioned this "trick" [24], but did not make use of it. In total, this reduces the number of instruction from three down to one per vector, because we don't have to broadcast (or load) the constant 1 into a vector register, nor do we have to neutralize elements.

A `SELECT count(*) FROM lineitem` shows speedups of 1.6 x with SSE and 2.3 x with AVX-2 instructions, compared to sequential code.

8.1.1.2. Min/Max

For selecting the minimum or maximum attribute values we use the extreme values of the corresponding data type as neutral elements. Attribute values that belong to non-qualifying tuples are replaced by the largest possible integer, in case of minimum selection, or the smallest possible integer, in case of maximum selection. These extreme values are broadcasted into a vector register, then we *blend* (`vpblendvb`) the attribute values and extreme value into a third register using the selection mask as a control mask (as shown in section 6.5). Compared to sequential code, we observed an average speedup of ~6 x with AVX-2 (see table 8.1).

In contrast to `count` and `sum`, we achieve much higher speedups with min/max-selection. The key difference is, that min/max is performed on the original data type, whereas a sum is usually stored in a wider type, e.g. in a 64-bit integer. Thus, min/max benefits from a higher degree of parallelism and does not cause costly type-conversion.

8.1.2. NULL-Handling

In scalar aggregation a special case occurs if no tuple qualifies. In that case, the aggregates contain `NULL` values. If we apply the aggregation functions as shown, we might get one of the neutral elements as a result. In that case it is undecidable, whether the aggregation resulted in an extreme value or no tuple survived the selection. To handle these corner

	SSE	AVX-2
`int 8`	11.0	12.2
`int 16`	5.2	6.7
`int 32`	2.7	3.5
`int 64`	1.4	1.5

Table 8.1.: Speedup of SIMD minimum/maximum selection compared to sequential execution.

cases, the system has to additionally count the qualifying tuples. In case of min/max-selection, the introduction of an additional `count(*)` aggregation might have negative performance impacts, due to the mentioned loss of parallelism and conversion costs.

8.2. Group-By Aggregation

In this section we take a closer look at the *hash-based* group-by aggregation. Compared to the scalar aggregation, that perfectly enables SIMD processing, the way of implementing a group-by aggregation is not that self-evident. Actually, the column oriented aggregation does not work anymore when the aggregation contains a group-by clause. At least, it can't be implemented efficiently on the current Haswell architecture, due to missing hardware instructions. Nevertheless, this is going to change with the next processor generation, therefore we still sketch the idea of a column oriented group-by aggregation. After that, we present a fine-tuned row-oriented implementation.

8.2.1. Column-Oriented Group-By

In the block-wise processing scheme, the individual attribute values arrive in a columnar format. Compared to the scalar aggregation, the key difference is, that in general these attributes belong to different groups which are stored in different hash buckets. Therefore, the corresponding aggregates are no longer stored in contiguous memory locations. This implies that, if we want to apply the aggregation function to P tuples in parallel, we would have to *gather* the corresponding P aggregates from their memory locations into a single vector register. After the function is applied, the results must be *scatter*ed back to their original locations.

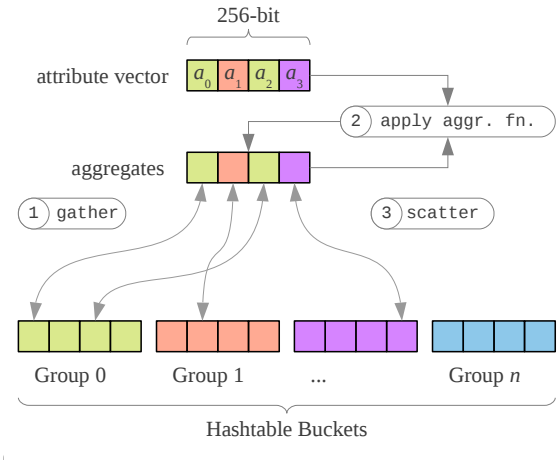


Figure 8.1.: Column-oriented group-by aggregation using gather- and scatter-instructions.

Further, in this approach conflicting write operations occur, if multiple attributes belong to the same group. E.g. if two aggregates are written to same memory location during the back-scattering, the results are partially overridden. This can be solved using the partial aggregation approach as shown in the scalar aggregation section. For this, we allocate multiple memory "slots" for each aggregate and depending on the position of attribute in the column vector, the

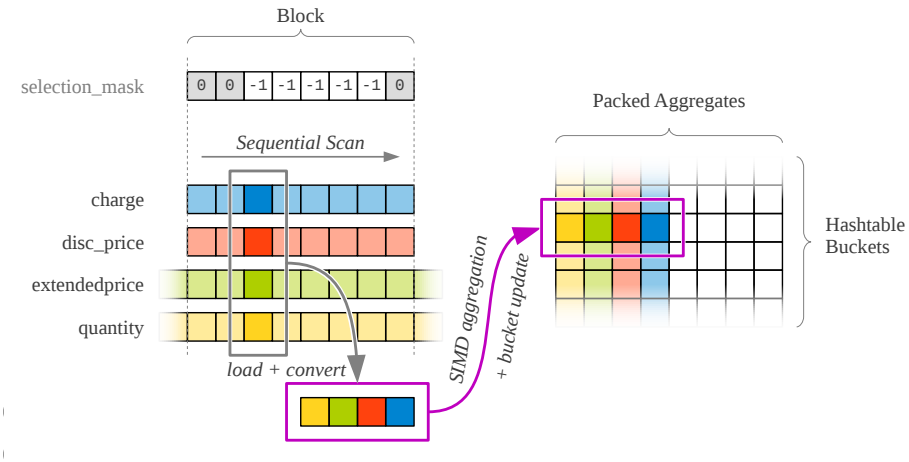


Figure 8.2.: Row-oriented group-by aggregation.

corresponding aggregate is updated as shown in figure 8.1. This way, all write-accesses become data-independent and can be performed in parallel. E.g. in the figure the attribute a_0 and a_2 hash to the same group 0. Depending on their position in the attribute vector, the corresponding aggregates within the group are updated. In principle, this approach trades memory consumption for parallelism, as it reserves 256-bits for each bucket independently of the attributes data type. Thus, if P elements fit into a single 256-bit register, the hash table consumes P times more space. However, the show-stopper is the missing scatter instruction in Intel's current Haswell architecture. Indeed, the scatter can be manually implemented using `maskstore`s but this would lead to a complete loss of parallelism.

8.2.2. Row-Oriented Group-By

As the name suggests, the following implementation of a group-by aggregation breaks with the column-oriented processing model. The row-based approach tries to overcome the aforementioned shortcomings by employing SIMD instructions in a tuple-at-a-time processing. The key idea is to update multiple aggregates of a single group in parallel. This is achieved by arranging aggregates in the hash bucket in such a way, that neighboring aggregates are compatible w.r.t. to their aggregation function. The corresponding attribute values can then be packed into vector registers and the hash bucket is updated using vector instructions. Typically, aggregates are 4- or 8-byte values, therefore we can update up to eight aggregates in parallel. This approach implies, that data has to be transformed from column-vectors into tuples. Further, the maximum degree of parallelism no longer depends on the block size, instead it depends on the number of compatible aggregations. Therefore, only queries that perform multiple aggregations may benefit from SIMD instructions. In general, the vectorized group-by, as it is sketch in figure 8.2, is orthogonal to block-wise processing.

In the following sections we first present how SIMD instructions can be employed to efficiently update the aggregations in hash buckets. Then, we show how the row-oriented approach integrates into the block-wise processing scheme.

8.2.2.1. Bucket Layout and Updates

To apply SIMD updates, the hash buckets have to be of a proper format. For our approach, we subdivide the structure of a hash bucket into *packed aggregates* (PA). All aggregates that belong to the same PA basically have the same aggregation function and the same data type. More precisely, the underlying arithmetic operation needs to be the same. Therefore, `sum`s and `count`s can be packed together, naturally. A single PA reserves 256 bits, thus it takes exactly one instruction to perform an update on all containing elements. Further, each PA needs to be 32-byte aligned otherwise a general-protection exception will be thrown. The following listing shows the structure of an exemplary hash bucket consisting of two PAs:

```
struct Entry {
    int64_t packedAggr0[4] __attribute__ ((aligned (32)));
    int32_t packedAggr1[8] __attribute__ ((aligned (32)));
};
```

Finding a proper bucket layout is part of the query compilation process. For our prototypal query compiler we developed an algorithm, that minimizes the number of instructions that are required to update all packed aggregates. Thereby, the algorithm tries to avoid overheads caused by 1.) under-utilization of SIMD registers and 2.) unnecessary space consumption by the hash buckets.

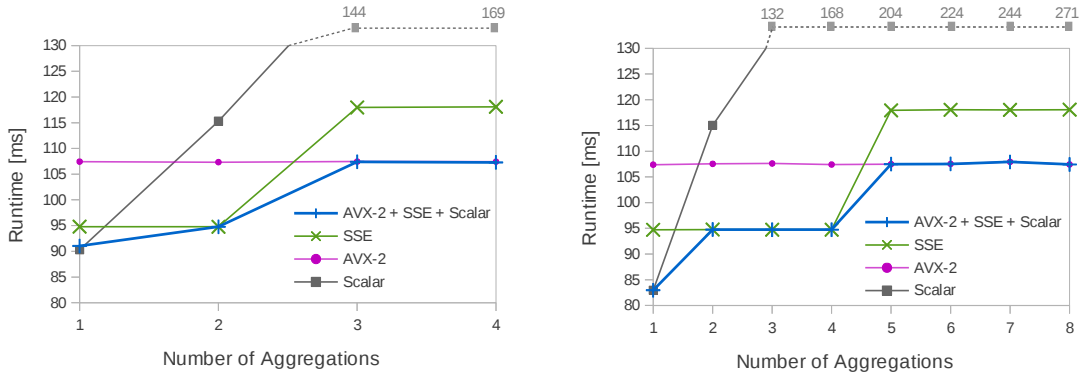
The algorithm determines the bucket layout in two phases. The first phase groups together all compatible aggregations, forming up to three disjoint sets of aggregations: the sum-group, the min- and the max-group. In the second phase, the algorithm determines a proper data layout for each group individually as follows:

First, the algorithm counts the number of aggregates for each data type. If the number of aggregates of a certain type is greater or equal to the number of aggregates that can be stored within a single PA, then new PAs are created and the corresponding aggregates are assigned to it. All PAs created within this step contain the maximum number of aggregates, thus they fully utilize SIMD instructions and each aggregate reserves the minimum amount of memory. Therefore, they are considered as optimal.

In the second step, the algorithm considers the remaining aggregates that have not been assigned in the previous step. In general, the remaining aggregates are of different data types (typically 32- and 64-bit integers) and the algorithm assigns them to PAs so that the number of instructions required for updates and the costs are minimal. During this step, the algorithm considers *up-casting* the aggregates from smaller into larger types to achieve the mentioned optimization objectives. We discuss the details of this optimization step later in this section.

Instruction Selection Optimization After the bucket layout is determined, the query compiler performs an additional optimization step during the instruction selection phase. Based on the *utilization* of the PAs, it selects the smallest instruction for updating the aggregates. Small, in terms of number of packed elements or vector size respectively. In worst case, a PA contains only a single element. In that case the code generator selects a scalar instruction.

To make the effects of the instruction selection optimization visible, we conducted an experiment where we performed multiple `count(*)` aggregations and compared the query run-times. We started with a single count and subsequently increased the counts to the number of elements in a PA. As data set we used the `lineitem` table of the TPC-H benchmark with scale factor 10 and the grouping attributes are `l_returnflag` and



(a) 64-bit integer aggregates

(b) 32-bit integer aggregates

Figure 8.3.: SIMD aggregation performance with varying number of aggregates and different instruction selection strategies.

`l_linestatus`:

```
SELECT count(*) c1, ... , count(*) cn
FROM lineitem
GROUP BY l_returnflag, l_linestatus
```

The experiment is repeated with different instruction selection strategies and different data types. First, we restricted the query compiler to use either scalar-, SSE- or AVX-2 instructions exclusively. Then, we let him select the smallest viable instruction from all instruction sets. The plots in Figure 8.3 show the results with either (a) 4 x 64-bit packed aggregates and (b) 8 x 32-bit packed aggregates.

Beside the fact, that SIMD updates can significantly increase performance, the results also show that

1. under-utilization of SIMD instructions causes noticeable overheads,
2. and a packed aggregate should be updated within (at most) one single instruction.

Both observations seem to be contradictory, as in some cases we cannot avoid under-utilization without employing an additional instruction. In these cases the system should select a vector instruction even if the packed aggregate contains empty slots (blue lines in Figure 8.3).

Based on this results we can formalize how the costs of the individual instructions are related to each other. Assuming the underlying hardware implements the SSE and the AVX-2 instruction sets, we distinguish between scalar x86 instructions, denoted as I_{64}, and 128-/256-bit SIMD instructions denoted as I_{128} and I_{256}, respectively. If we apply some (arbitrary) cost function C, then the following system of inequalities must be satisfied:

$$
\begin{aligned}
&1) \quad C(I_{64}) < C(I_{128}) < C(I_{256}) \\
&2) \quad C(I_{256}) < C(k \cdot I_{128});\ k \in \mathbb{N}\backslash\{1\} \\
&3) \quad C(I_{256}) < C(k \cdot I_{64}) \\
&4) \quad C(I_{128}) < C(k \cdot I_{64}) \\
&5) \quad C(I_{128} + I_{128}) < C(I_{256} + I_{64})
\end{aligned}
\tag{8.1}
$$

Thus, employing more than one instruction is in any case more costly. E.g. updating two 32-bit aggregates using a SIMD-128 instruction is less costly than using two scalar instructions, even though the SIMD register is only utilized by 50%. The overhead of under-utilization becomes visible if we compare the SIMD-128 with the SIMD-256 instruction. Whereas the SIMD-256 instruction still outperforms the two scalar instructions, we only observe a performance gain of ~7% compared to ~17% with SIMD-128.

Minimizing Costs Through Up-casting As mentioned before, we can apply additional optimizations if the aggregates are of different data types. Due to the nature of SIMD, all aggregates that are packed together need to be of the same data type. Therefore, if we naively assign aggregates depending on their types to the corresponding PAs, we might employ more instructions during hash bucket updates than necessary. Further, as a PA always reserves 32-bytes, the space consumption of the hash buckets may also be suboptimal. For example, the following two packed aggregates are under-utilized: $\texttt{PA}_0^{\text{i64}}$ = [$\texttt{a}_0$, $\texttt{a}_1$, Ø, Ø], $\texttt{PA}_1^{\text{i32}}$ = [$\texttt{b}_0$, Ø, Ø, Ø, Ø, Ø, Ø, Ø], where Ø denotes an empty slot. By up-casting $\texttt{b}_0$ from i32 to i64 we can assign it to $\texttt{PA}_0^{\text{i64}}$ = [$\texttt{a}_0$, $\texttt{a}_1$, $\texttt{b}_0$, Ø] and therefore reduce the costs for the hash bucket updates, because $C(I_{256}) < C(I_{128}) + C(I_{64})$.

To quantify the results, we model the costs using the latencies of the `add` instruction, which we observed in the micro-benchmarks of chapter 6:

$$Lat(I_B) = \begin{cases} 1.2 & \text{if } B = 64 \\ 1.6 & \text{if } B = 128 \\ 2.4 & \text{if } B = 256 \end{cases} \tag{8.2}$$

Further, we define the costs as a function that maps a given PA to the latency values based on the number of assigned aggregates n and the data type T:

$$C_T(n) = \begin{cases} 0 & \text{if } n = 0 \\ Lat(I_{64}) & \text{if } n = 1 \\ Lat(I_{128}) & \text{if } n > 1 \wedge \frac{n \cdot \text{bitwidth}(T)}{256} \leq \frac{1}{2} \\ Lat(I_{256}) & \text{if } n > 1 \wedge \frac{1}{2} < \frac{n \cdot \text{bitwidth}(T)}{256} \leq 1 \\ +\infty & \text{otherwise} \end{cases} \tag{8.3}$$

If we apply the cost function to the above example, then we get a reduced latency of 0.2 (-14%). Further, as a side effect, the overall space consumption has been halved through this optimization step.

In the following we present an algorithm that minimizes the costs for a given list of under-utilized packed aggregates: The algorithm expects the list to be sorted by the bit-width of the PA's data type in a descending order. The *up-casting optimization* starts at the PA with the widest data type. Thereby, the algorithm computes how the costs would change if additional aggregates will be up-casted into the current PA. These costs deltas are computed for every empty slot and the results are stored in a vector $\overrightarrow{delta} = (d_0, \ldots, d_{maxIn})^{\text{T}}$. Thus, the components d_i denotes the cost delta if i elements are up-casted from one of the subsequent PA's, where d_0 is always 0 at the top-level PA and the size of the vector ($= maxIn + 1$) denotes the number of available slots. After the deltas have been computed, the algorithm recurses to the subsequent PA of a narrower type with the delta-vector passed in. In each recursion step, a new cost delta-vector is computed similarly, but this time the algorithm does not only considers *incoming* aggregates, but also *outgoing* aggregates. For all possible combinations the changes in costs are computed

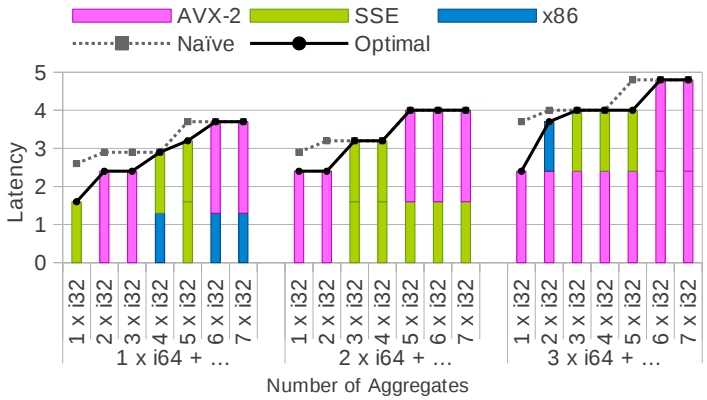


Figure 8.4.: Costs comparison

and the results are again stored in a delta-vector. When the recursion reaches the last PA, the algorithm can make a global decision, because the delta-vector reflects the cost changes of all preceding PAs. As a result, the algorithm returns for each PA the number of elements that need to be up-casted. In the (trivial) introductory example, this would be $(0, 1)$.

The full algorithm is shown on the facing page. Within the pseudo-code we denote the maximum number of aggregates a given PA can contain with $||\texttt{PA}||$, and $|\texttt{PA}|$ denotes the number of currently assigned aggregates.

Figure 8.4 shows a comparison of the naive assignment strategy and the up-casting strategy with varying numbers of (mixed) aggregates. Each of the colored bars denotes a single instruction, and the heights of the bars refer to the latencies of the corresponding instructions as defined in equation 8.2. If we mix only 32- and 64-bit aggregates, as shown in the Figure, we gain performance improvements of up to 35% and in general, the optimization potentials are higher for a smaller number of (mixed) aggregates, which is more likely in practice.

8.2.2.2. Switching from Block- to Tuple-Wise Processing

Another challenge with the group-by aggregation is to efficiently switch between the block-wise to tuple-at-a-time processing. The simplest and most general way is to introduce a inner loop that sequentially iterates over the current block, as shown in listing 8.1.

```
for (uint64_t i = 0; i < BLOCK_SIZE; i += 1) {
    if (selection_mask[i]) {
        // ... convert attributes from DSM to NSM
        // ... update hash bucket
    }
}
```

Listing 8.1: Sequential aggregation loop

As we do not expect non-qualifying tuples within a sequential code part, we additionally introduce a conditional branch. This naturally provokes the well known performance problems caused by branch mispredictions and the resulting execution pipeline stallings. Only blocks that contain either qualifying or non-qualifying tuples are considered harmless, because the branch is highly predictable for the CPU. In general, the predictability of the branch depends on how often the selection mask changes between -1 and 0, and therefore on the distribution of the qualifying tuples. For instance, a block that contains 50%

Algorithm 8.1 UpcastAggregates

Input: an ordered sequence of under-utilized packed aggregates $P = (p_i)$,
current position *pos* in sequence P,
a vector $\overrightarrow{upCosts}$ of size k that reflects the costs changes when up-casting k elements.

Output: a sequence (u_i) of length $|P|$, where u_i denotes the number of aggregates in P_i that need to be up-casted to minimize the costs.

$pa = p_{pos}$
$T = \texttt{type-of}(pa)$
$currentCosts = C_T(|pa|)$
$maxOut = |\overrightarrow{upCosts}| - 1$
$maxIn = maxOut + ||pa|| - |pa|$
$\overrightarrow{delta} = (d_0, \ldots, d_{maxIn})^{\mathrm{T}} = +\infty$
$\overrightarrow{up} = (u_0, \ldots, u_{maxIn})^{\mathrm{T}} = 0$
for $i = 0$ **to** $maxIn$ **do**
    for $j = 0$ **to** $maxOut$ **do**
        $\Delta c = 0$
        $util = |pa| + i - j$
        if $util < 0$ **then**
            $\Delta c = +\infty$
        else
            $\Delta c = C_T(util) + upCosts_j - currentCosts$
        end if
        if $\Delta c < d_i$ **then**
            $d_i = \Delta c$
            $u_i = j$
        end if
    end for
end for
if $pos < |P|$ **then**
    $R = (r_i) =$ UpcastAggregate(P, $pos + 1$, $\overrightarrow{delta}$)
    return $(u_{r_0}) \circ R$
else
    $m = \arg\min_i d_i$
    return (u_m)
end if

qualifying tuples where each qualifying tuple follows a non-qualifying tuple represents the worst case for the CPU's branch predictor.

Figure 8.5a shows the costs induced by the sequential loop, assuming a uniform distribution of the qualifying tuples. In worst case, the loop adds 300 CPU cycles to process a single block consisting of 32 tuples. Therefore, we additionally evaluated and compared two branch-free approaches:

1) Transforming the selection mask into a selection vector: In this approach we adapt the concept of the *selection-vector* from the MonetDB database system [3]. A selection-vector basically consists of positions of tuples that survived the selection. For instance, `discount[selection_vector[i]]` refers to the attribute value of `discount` of the i^{th} qualifying tuple. This approach makes the conditional branch superfluous, as shown in listing 8.2. The transformation itself is implemented branch-free as well. The code is based on the *No-Branching Selection primitive* presented in [18].

2) Masking aggregates that belong to non-qualifying tuples: The second branch-free approach adapts the idea of masking out non-qualifying tuples using bitwise `AND` operations. Similar as in the scalar aggregation, the aggregation functions are always applied, but in case of non-qualifying tuples, the attribute values are replaced by neutral elements beforehand. Listing 8.3 shows the masking code for `SUM` aggregations.

Figure 8.5b shows a performance comparison of the two branch-free approaches and the sequential loop using a conditional branch. As mentioned before, the conditional branching is very sensitive to the number of qualifying tuples in a block. In contrast, the selection-vector approach greatly eliminates misprediction costs, but introduces a constant amount of costs for transforming the selection mask. However, only qualifying tuples reach the sequential code part, which improves performance of queries with lower selectivities. The costs for the masking approach are entirely independent from the number of qualifying tuples, but masking doubles the number of instructions needed for aggregation. These additional cycles become noticeable in high selectivity queries like Q1, where 80% of all blocks contain only qualifying tuples.

	Relative runtime
No branch (selection-vector)	+30.2%
No branch (masking)	+13.6%

Table 8.2.: Relative runtime of Q1 using different transition strategies from parallel to sequential code compared to sequential loop.

In conclusion, all three approaches have their strengths. The selection-vector performs best in queries with a selectivity of < 0.65 and masking shows good results with high selectivity queries. In case of Q1, the conditional branch is so well predictable, that the naive approach still outperforms the branch-free approaches.

8.3. Summary

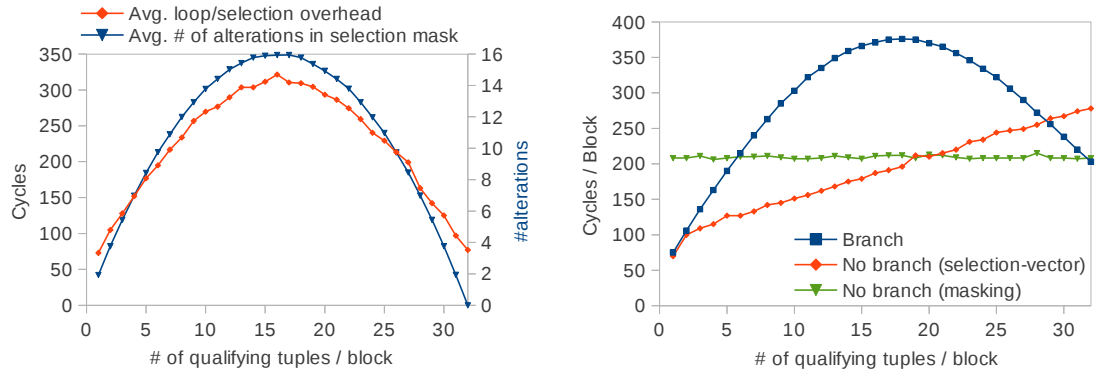
In this chapter we covered the scalar- and the group-by aggregation. We have shown that the former one perfectly enables SIMD processing, which resulted in speedups reaching from 1.5 x up to more than 12 x, depending on the data types. In contrast, the hash-based group-by aggregation naturally involves random memory accesses, which are hard to handle using SIMD instructions. Eventually, the missing scatter instruction in current

```
uint64_t selection_vector[BLOCK_SIZE];
uint64_t num_qualifying_tuples = 0;
for (size_t i = 0; i < BLOCK_SIZE; i++) {
    selection_vector[num_qualifying_tuples] = i;
    num_qualifying_tuples += (selection_mask[i] == -1);
}
for (uint64_t i = 0; i < num_qualifying_tuples; i += 1) {
    // ... convert attributes from DSM to NSM
    // ... update hash bucket
}
```

Listing 8.2: Aggregation loop with branch-free selection (selection-vector)

```
const vector4_int64 UPDATE_4xI64 {-1,-1,-1,-1};
const vector4_int64 NO_UPDATE_4xI64 {0,0,0,0};
const vector4_int64 MASKS_4xI64[2] {UPDATE_4xI64, NO_UPDATE_4xI64};
...
for (uint64_t i = 0; i < BLOCK_SIZE; i += 1) {
    mask = MASKS_4xI64[selection_mask[i] + 1];
    // ... convert attributes from DSM to NSM
    // ... update hash bucket
}
```

Listing 8.3: Aggregation loop with branch-free selection (masking)



(a) Average costs for scanning a block of size 32 with varying number of qualifying tuples, including branch mispredictions costs. (uniformly distributed qualifying tuples)

(b) Comparison of selection cost, caused by non-qualifying tuples when falling back to sequential aggregation code. (using aggregations from Q1)

Figure 8.5.: Branch prediction costs and branch-free alternatives.

Haswell architectures forced us to break with the column-oriented processing model. We therefore optimized aggregation in a NSM. The key idea, of updating multiple aggregates of a single group in parallel, was originally presented by Zukowski et al. in [26]. We generalized this approach to algorithmically determine an optimal data layout for hash buckets for arbitrary queries. The underlying cost function considers the latencies of instructions from different instruction sets. We demonstrated the instruction selection optimization with x86, SSE and the AVX-2 instruction sets. Further, we introduced the up-casting strategy to optimally utilize SIMD capabilities.

The implication of row-based aggregation is that the database system has to switch the processing model on-the-fly within the query processing pipeline, which might cause considerable costs due to branch mispredictions. We therefore evaluated two different branch-free implementations to reduce this "friction loss" that arise at the breaking point, which resulted in a saving of up to 150 CPU cycles per block.

We did not cover the topic of (re-)designing the hash table data structure in general, as we don't see great improvement potential through SIMD. In principle, applying the hash function to the attributes can be done in parallel. But as already shown in section 6.6, the benefits of SIMD is very limited. However, Q1 is a rare exception, because the grouping attributes are two 8-bit values we can apply the hash function `(l_returnflag << 8)| l_linestatus` which can be efficiently computed using SIMD instructions. Nevertheless, we argue that existing hash table implementations can take advantage of a SIMD-optimized bucket layout.

9. Prototype

The implemented prototype follows the classical database system architecture and basically consists of a *compile time system* (CTS) and a *runtime system* (RTS). The CTS is further subdivided into a *parser* that transforms the query into a relational operator tree; an *optimizer* that performs logical optimizations and transforms the operator tree into a physical algebra; and a *code generator* that compiles the queries into executable code. In this case, the code generator emits C++ code that is compiled by the LLVM/Clang++ compiler into a shared object file. The RTS dynamically loads and executes the generated shared object file and prints the results to the standard output stream. A small web-server glues everything together and provides a REST-interface and a web frontend for automatic and manual testing/benchmarking.

9.1. Low-Level Query Language

For experimental evaluations we developed a small *low-level query language* (LLQL) which gives control over the query compilation process. The LLQL especially allows the user to specify the data types of intermediate results and aggregates. Further it provides a *branch annotation* for comparison operators that allows the user to introduce conditional jumps ad libitum. In principle, LLQL is considered as a tool that simplifies experiments and benchmarks. Nevertheless, it gives insights into the query compiler and its range of functions. Therefore, we briefly introduce LLQL and show the full source code of the TPC-H queries 1 and 6 as they where used in benchmarks.

9.1.1. Language Features

LLQL is an easy to compile, statically typed and lexically scoped language that allows to formulate read-only database queries. It supports arbitrary complex arithmetic expressions and conjunctive boolean expressions. The language basically supports queries over multiple relations, but the underlying compiler is restricted to a single relation. The following paragraphs give an overview about the LLQL syntax and semantics. The listings 9.1 and 9.2 on page 64 show how Q1 and Q6 are implemented in LLQL.

Type Definitions Attributes and variables in LLQL are statically typed by specifying a *triple* consisting of the high-level SQL type, the low-level (internally used) type and an optional range that specifies the min. and max. values. For example, an attribute `discount` can be declared as

```
num(3,2):i8:[0,100] discount
```

Where `num(3,2)` is the short form of the SQL type `Numeric(3,2)` or `Decimal(3,2)` respectively. The attribute values of `discount` are internally represented as 8-bit integers (`i8`). The third component specifies, that the attribute values are within the range from 0 to 100, where the range refers to the internal representation. In this example, a discount of 0.03 is internally represented as 3.

The high-level types are restricted to: `num(precision,scale)`, `date` and `char(1)`. For the internal representations, 8-, 16-, 32- and 64-bit signed integers are supported which are denoted as `i8`, `i16`, `i32` and `i64`. Note that `char(1)` can only be mapped to `i8`.

Because boolean values are only supported in selection predicates, they must not be typed explicitly. Instead LLQL provides the `predicate` keyword that binds boolean expressions to variables:

```
predicate cond = attribute1 == attribute2
```

Like in many common programming languages, the single equality sign is the assignment operator whereas the == is an equality check.

Type Inference For simplicity, the query compiler does not infer any data types. Thus, it does not support literals in arithmetic and boolean expressions. - Literals can only be used to declare statically typed constants. For instance the expression $e = 1 - discount$ is translated into:

```
num(3,2):i8 one = 100
num(12,2):i8 e = one - discount
```

Tuple Variables LLQL provides *tuple variables* to access the contents of relations. For instance, the statement `t <- relation` is equivalent to the SQL statement `SELECT ... FROM relation AS t`. Like in SQL, the individual attributes can then be accessed using the *dot-notation*, e.g. `t.attribute`.

Branch Control By default, all selection predicates are compiled without conditional jumps, thus the entire relation is read during query processing. LLQL allows to override the default behavior, by appending an "*" (asterisk) symbol to the comparison operators. The query compiler then introduces a conditional jump as shown in chapter 5.

In the following example, a branch is introduced right after the evaluation of the given range predicate. Note, that the comparisons are performed in *reverse order* as they are specified in the source.

```
date:i16 shipdate_lo = ...
date:i16 shipdate_hi = ...

select l.l_shipdate >* shipdate_lo
   and l.l_shipdate < shipdate_hi
```

9.1.2. Manual Overflow Prevention in LLQL

As mentioned in chapter 6, the prototypal query compiler does not automatically prevent integer overflows, but it is able to either generate code with or without overflow checks. In case of disabled overflow checks, the responsibility for choosing the right data types is exposed to the user through LLQL.

For instance, the following LLQL code snippet evaluates the arithmetic expression $1 - discount$ as it appears in TPC-H query 1, where the type of *discount* is `num(3,2):i8 :[0,100]`.

```
num(12,2):i8:[100,100] one = 100 // constant declaration
num(12,2):i8:[0,100] t = one - discount
```

There are two things to consider:

- First, for the constant declaration the user has to consider the *scale* of the numeric type. The scale of both operands need to be compatible w.r.t. the arithmetic operation. Here, both operands have two decimal places. Therefore, the constant's value 1.00 is represented as 100.
- Second, the result value is guaranteed to fit into an 8-bit integer. More precisely, $t \in [0, 100]$ because if we perform the subtraction on the given intervals, we get $[100, 100] - [0, 100] = [100, 0] = [0, 100]$.

9.2. Optimizations

During the query compilation process the system performs the following logical optimizations:

1. *Push Mappings*: Pushes all map operators down the operator tree.
2. *Push Selections*: Pushes all selections down the operator tree and conjunctive selection predicates are broken-up beforehand.
3. *Decompose `AVG` Aggregations*: Decomposes `AVG` aggregations into a `SUM` and a `COUNT(*)` aggregation. Further a map operator is introduced that performs a division. Thereby, the system makes use of already existing aggregations.

Further, the group-by operator (physically) optimizes the bucket layouts as described in chapter 8.

9.3. Query Compiler Flags

Parameters like the block size, available instruction sets and overflow handling are specified as compiler flags. The compiler supports the instruction sets `default`, `sse2`, `sse41`, `sse42` and `avx2` where `default` refers to x86 instruction set.

Throughout this work we only distinguish between x86, SSE and AVX-2, whereas SSE includes `default` $\cup$ `sse2` $\cup$ `sse41` $\cup$ `sse42` and AVX-2 := SSE $\cup$ `avx2`. Thus, a query that is compiled with AVX-2 may also make use of instructions from earlier instruction sets.

```
q1 = {
    l <- tpch.lineitem

    date:i16 d = 10473  // = 1998-12-01 - 90 days + 1
    select l.l_shipdate < d

    num(3,2):i8 one = 100 // = 1.00
    num(12,4):i64 disc_price = l.l_extendedprice * (one - l.l_discount)
    num(12,4):i64 charge = disc_price * (one + l.l_tax)

    group by l.l_returnflag, l.l_linestatus
        num(12,4):i64 sum_disc_price = sum(disc_price)
        num(12,2):i64 sum_qty = sum(l.l_quantity)
        num(12,2):i64 sum_base_price = sum(l.l_extendedprice)
        num(12,6):i64 sum_charge = sum(charge)
        num(12,2):i64 count_order = count(*)
        num(12,2):i64 avg_qty = avg(l.l_quantity)
        num(12,2):i64 avg_price = avg(l.l_extendedprice)
        num(12,2):i64 avg_disc = avg(l.l_discount)
}

q1()
```

Listing 9.1: TPC-H Query 1 in LLQL

```
q2 = {
    l <- tpch.lineitem

    date:i16 shipdate_lo = 8766 // 1994-01-01 - 1
    int:i16 one_year = 366  // one year + 1
    date:i16 shipdate_hi = shipdate_lo + one_year // 1995-01-01
    num(12,2):i8 discount_lo = 4 // = 0.05 - 0.01
    num(12,2):i8 discount_hi = 8 // = 0.07 + 0.01
    num(12,2):i8 quantity_hi = 24

    num(12,4):i64 r = l.l_extendedprice * l.l_discount

    select l.l_shipdate > shipdate_lo
       and l.l_shipdate < shipdate_hi
       and l.l_quantity < quantity_hi
       and l.l_discount > discount_lo
       and l.l_discount < discount_hi

    aggregate
        num(12,4):i64 revenue = sum(r)

    project revenue
}

q2()
```

Listing 9.2: TPC-H Query 6 in LLQL

10. Experimental Evaluation

In this chapter we present the results of our experimental evaluations with TPC-H Q1 and Q6. We examine different aspects of the query compilation process and their impacts on query runtimes. Further we show by comparison how wider data types affect query runtimes.

In this chapter we often refer to the two different internal representations of the TPC-H data set, namely "narrow" and "wide" which we introduced in section 2.4. If not stated otherwise, we set the block size B in all experiments so that B elements of the narrowest attribute a_{min}, referenced by the query Q, fill an entire SIMD register:

$$a_{min} = \underset{a \in \text{schema}(Q)}{\arg\min} \text{ bit-width}(a)$$

$$B = \frac{\text{bit-width(SIMD-REG)}}{\text{bit-width}(a_{min})} \tag{10.1}$$

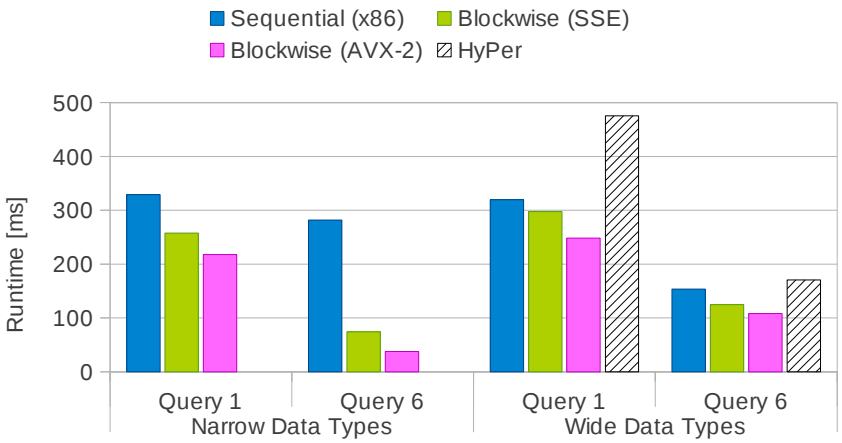
10.1. TPC-H Queries

Figure 10.1 on the following page depicts the runtimes of Q1 and Q6 with the narrow and the wide data set. The hatched bar denotes the query runtimes on the HyPer database system [6].

Q1: With the block-wise approach, Q1 shows a speedup of 1.5 x compared to the sequential execution. Q1 primarily benefits from efficient arithmetics in the map and the group-by operator. 70% of the performance improvements are due to the row-oriented aggregation in the group-by operator. Whereas the evaluation of arithmetic expressions in the map operator are dominated by type conversions. In contrast, the type conversions in the group-by operator do not incur additional cost, because they are performed in regular CPU registers. With the wider data types, we see a 14% decreased performance, although no type conversions occur. The performance degradation is due to the lower degree of parallelism in the map operator. Especially the 64-bit integer multiplications have negative effects on the overall performance. A single 64-bit multiplication on four elements issues 9 vector instructions, which results in a DoP of 0.44. Therefore the performance of Q1/wide can be improved by *pulling* the 64-bit multiplications into the sequential code part.

The relatively large differences in performance compared to HyPer are mainly caused by the different overflow handling strategies. In this experiment we applied the overflow *prevention* strategy, thus no overflow checks are performed at runtime. The impacts of overflow detection are shown in the later section 10.2.

Q6: In contrast to Q1 Q6 benefits significantly from narrow data types. Compiled with the AVX-2 instruction set, the overall query runtime can be improved by factor 2.8 compared to the "wide" version. The poor sequential performance with the narrow data types



Speedups	Narrow		Wide	
	Q1	Q6	Q1	Q6
SSE	1.8	2.3	1.6	1.4
AVX-2	2.2	4.5	1.9	1.6

Figure 10.1.: Performance comparison with HyPer (overflow prevention)

is a result of compiler "optimizations" performed by LLVM/Clang++. The underlying compiler automatically introduces conditional jumps in native code, although the generated C++ code is entirely branch-free. As we focus on SIMD processing, we do not investigate further on the sequential performance. Instead we consider this as an outlier.

The entire query pipeline from the table scan (source) to the aggregation operator (sink) can be compiled into SIMD code. The pipeline consists of twelve operations like comparisons, logical ANDs and arithmetics, as show in table 10.3a on page 68. Whereas the involved type conversions are considered to be as SIMD-overhead. With narrow data types the query issues 73 vector instructions per block of 32 tuples, whereas 36 instructions are due to type conversions. With the wide data types the query is compiled into 39 instructions, where only 2 are considered to be as overhead. However, the block size is four times smaller and therefore the total number of instructions issued during query processing is more than two times higher compared to the narrow version. In terms of DoP, with the narrow data types 5.3 elements are processed per instruction in average. With wider types the DoP degrades to 2.5 $\left[\frac{\text{elements}}{\text{instruction}}\right]$.

As mentioned in chapter 5, introducing branches does not improve performance with narrow data types for two reasons: (i) The block selectivity is much higher than the tuple selectivity and (ii) re-ordering the predicates by their selectivity causes additional type conversion costs. With the wider types the picture changes (at least) for small block sizes used with SSE. Introducing a conditional jump can improve query performance by ~8%. However, executing the query branch-free with AVX-2 is still ~14% faster. The tables 10.1 and 10.2 show the selection predicates of Q6 and how they affect query runtime when the query is compiled with branches.

#	Predicate	Selectivity
p_1	l_shipdate > 1993-12-31 ∧ l_shipdate < 1995-01-01	0.15
p_2	l_discount > 0.04 ∧ l_discount < 0.08	0.27
p_3	l_quantity < 24	0.46

Table 10.1.: Selection predicates and selectivities. (Q6)

	p_1	$p_1 \wedge p_2$	$p_1 \wedge p_2 \wedge p_3$
Sequential (x86)	-14.9%	-25.3%	-20.1%
Blockwise (SSE)	-8.1%	-1.5%	+0.7
Blockwise (AVX-2)	+3.7%	+24.1%	+26.9%

Table 10.2.: Relative changes in runtime with Q6/wide when introducing branches after predicate evaluation.

10.2. Overflow Detection

In chapter 6 we presented fine-tuned implementations to detect integer overflows in SIMD arithmetics. Micro-benchmarks have shown that manual overflow checks in AVX-2 can outperform hardware supported checks in scalar arithmetics with one exception, the 64-bit multiplication, which is 30% slower than the scalar multiplication. Thus it should be avoided whenever possible. Especially Q1 suffers from the performance impacts of multiplications. In the narrow case, the multiplication in expression $charge = disc\_price * (1 + l\_tax)$ is performed in 64-bit, and in the wide case the expression $disc\_price = l\_extendedprice * (1 - l\_discount)$ additionally results into a checked 64-bit multiplication. Due to the fact that Q1 falls back to sequential execution, the overall performance can be improved by moving 64-bit multiplications from the parallel code section to the sequential section. Through this optimization the runtime decreased by 14% (to 338 ms) with narrow data types and by 20% (to 375 ms) with wide data types.

Q6 also suffers from the 64-bit multiplication when the wide data set is used. In case of Q6 the entire query consists only of a parallel code section, thus the move-optimization cannot be applied. Instead the query compiler has to *mix-in* scalar instructions into the parallel code part, which happens by default if the inefficient parallel 64-bit multiplications are removed from the query compilers instruction set. The disadvantages of mixing scalar and vectorized code have been discussed in section 7.4. However, due to the high inefficiencies of a check 64-bit multiplication in SIMD, mixing in scalar code results in 16% faster query execution with AVX-2. With SSE instructions, the benefits are even higher: 25% through scalar multiplication plus another 25% through introducing a branch, which results in a runtime of 135 ms. Q6 in general benefits from introducing a branch after selection, because the costs of the remaining operators have been increased due to overflow detection. Figure 10.2 on page 69 shows a comparison of the best performing implementations.

10.3. Block Size

Processing multiple tuples at a time was the decisive modification on the query execution model to enable SIMD processing. Throughout this work we used the smallest possible

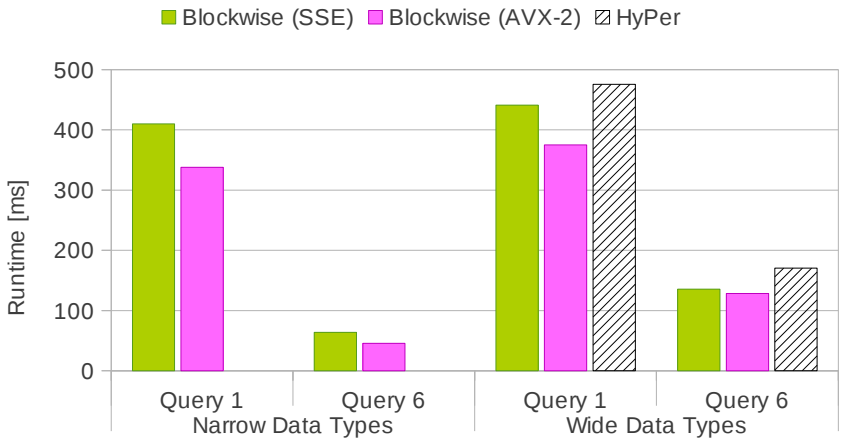
	#	Operation	P	I	I^B
Select	1	sel = l_discount $<_{i8}$ 0.08	32	1	1
	2	t0 = l_discount $>_{i8}$ 0.04	32	1	1
	3	sel = sel $\&_{i8}$ t0	32	1	1
	4	t1 = l_quantity $<_{i8}$ 24	32	1	1
	5	sel = sel $\&_{i8}$ t1	32	1	1
	6	sel0 = **convert**$_{i8\rightarrow i16}$(sel)	16	1	2
	7	t2 = l_shipdate $<_{i16}$ 1995-01-01	16	1	2
	8	sel0 = sel0 $\&_{i16}$ t2	16	1	2
	9	t3 = l_shipdate $>_{i16}$ 1993-12-31	16	1	2
	10	sel0 = sel0 $\&_{i16}$ t3	16	1	2
Map	**11**	l_extendedprice0 = **convert**$_{i32\rightarrow i64}$(l_extendedprice)	4	1	8
	12	l_discount0 = **convert**$_{i8\rightarrow i64}$(l_discount)	32	14	14
	13	r = l_extendedprice0 $*_{i32\rightarrow i64}$ l_discount	4	1	8
Aggr.	**14**	sel1 = **convert**$_{i16\rightarrow i64}$(sel0)	8	3	12
	15	r_masked = sel1 $\&_{i64}$ r	4	1	8
	16	revenue = revenue $+_{i64}$ r_masked	4	1	8
				$\sum$	73

(a) Q6/narrow; B=32

	#	Operation	P	I	I^B
Select	1	sel = l_shipdate $<_{i32}$ 1995-01-01	8	1	1
	2	t0 = l_shipdate $>_{i32}$ 1993-12-31	8	1	1
	3	sel = sel $\&_{i32}$ t0	8	1	1
	4	sel0 = **convert**$_{i32\rightarrow i64}$(sel)	4	1	2
	5	t1 = l_discount $<_{i64}$ 0.08	4	1	2
	6	sel0 = sel0 $\&_{i64}$ t1	4	1	2
	7	t2 = l_discount $>_{i64}$ 0.04	4	1	2
	8	sel0 = sel0 $\&_{i64}$ t2	4	1	2
	9	t3 = l_quantity $<_{i64}$ 24	4	1	2
	10	sel0 = sel0 $\&_{i64}$ t3	4	1	2
Map	11	r = l_extendedprice0 $*_{i64}$ l_discount	4	9	18
Aggr.	12	r_masked = sel0 $\&_{i64}$ r	4	1	2
	13	revenue = revenue $+_{i64}$ r_masked	4	1	2
				$\sum$	39

(b) Q6/wide; B=8

Table 10.3.: Operations performed by Q6 including type conversion overheads.



Speedups	Narrow		Wide	
	Q1	Q6	Q1	Q6
SSE	1.2	2.7	1.1	1.3
AVX-2	1.4	3.7	1.3	1.3

Figure 10.2.: Performance comparison with HyPer (enabled overflow detection).

block size (according to the equation 10.1 on page 65) as we preferred to keep attribute values in registers as long as possible. In all previous experiments the minimum block size led to the best results in terms of lower query runtimes. In general, the block size has significant impacts on the overall performance. If the block size is too small, the compiler can not make use of SIMD instructions and therefore must rely on scalar instructions. On the other hand, larger block sizes results in higher register pressure and eventually to register spilling. Figure 10.3 on the following page illustrates this on the example of Query 1. The smallest attribute accessed by Q1 is 8-bit wide. Thus, a block size of sixteen is required to fully utilize SSE instructions, and a block size of thirty-two for AVX-2. Reducing the block size leads to many small arrays in the generated code which are sequentially processed in regular registers using scalar instructions. As a result, attribute values are spilled to (cache) memory at operator boundaries, due to the limited amount of registers. On the other hand, larger blocks also result in performance decreases, but it degrades much slower due to higher capacities of SIMD registers. The speed of degradation naturally depends on the (minimum) number of registers that are required to process a single block of tuples, where not only the block size plays a crucial role. Other important factors are:

- *type conversions*: Up-casting attributes into larger types increases the register usage by factors.
- *number of intermediate results*: For the evaluation of arithmetic expressions, registers are allocated to hold the intermediate results. Especially when the expression tree is bushy, the minimum number of registers increases.
- *the total number of referenced attributes*: Especially attributes that are *live* across multiple operators increase register pressure. In our model, the selection mask can also be considered as a attribute, and the selection mask is live from the table scan to the next pipeline breaker (or sequentialization point).

- *operation complexity*: Typically, checked arithmetics issue multiple instruction for a single (primitive) operation, which in all cases leads to higher register usage.

Further, smaller block sizes are beneficial for low selectivity queries because smaller blocks can more likely be eliminated during selection what effectively increases the degree of parallelism.

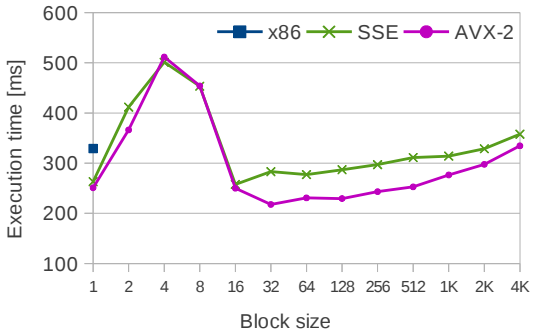


Figure 10.3.: Runtime of Q1 with varying block sizes.

10.4. Query 1: Generated vs. Handwritten

Finally, we present a performance comparison of the generated code and the handwritten code which we introduced at the beginning of this thesis in chapter 3. Figure 10.4 shows, that the performance of the generated code is very close to the performance of the handwritten code. The generated Q1, is only 3% slower than the handwritten version. Compared to the *hand-tuned* version, our query compiler produces 6.4% slower code. However, as mentioned at the beginning, the hand-tuned version makes use of some implementation "tricks" that are not applicable in database systems like HyPer.

The 3% performance difference between the generated and the handwritten version is basically due to lower materialization costs in the handwritten query. The materialization costs in the manual implementation have been reduced by cleverly interleaving the arithmetic and type conversion operations. Thus, the operator boundaries are even more blurred as in the generated code. However, this degree of interleaving cannot be achieved with the our extended produce/consume model.

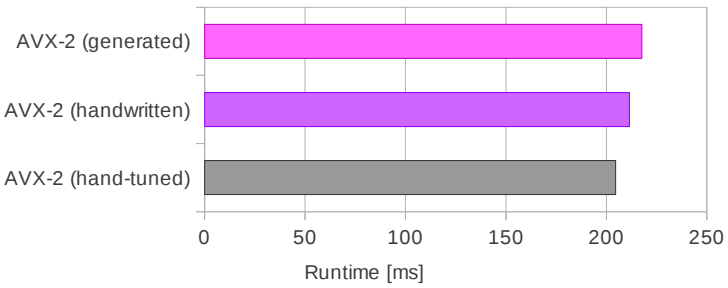


Figure 10.4.: Performance comparison of generated and handwritten code (Q1).

11. Conclusion and Future Work

11.1. Conclusions

In this work we have presented a method to accelerate analytical database queries using the SIMD capabilities of Intel's latest Haswell architecture. In contrast to previous works, we did not focus on a single database operator, instead we aimed to parallelize large parts of the query plans. Thereby, we investigated many aspects and revealed hidden costs that arise when multiple successive operators are parallelized using SIMD instructions.

The presented code generation model adapts features from existing high-performance database systems like HyPer and MonetDB (which evolved into the commercial product VectorWise). Even though, both systems follow very different philosophies in their query execution model, we fused together parts from both worlds and implemented a prototypal query compiler that produces very efficient code which is almost as fast as handwritten code. The presented block-wise approach is based on the data-centric code generation model of HyPer which originally processes tuple-at-a-time. The philosophy behind the block-wise approach is to process multiple tuple at a time without adding additional instructions that do not directly contribute to the query result. E.g. data movements are avoided along the query pipeline, only type conversions produce copies of the source attribute values. Especially at operator boundaries, the system does not prune non-qualifying tuples from the current block as this would introduce costly mem-copys and a level of indirection when accessing attributes. Instead, non-qualifying tuples remain in place and are passed to the subsequent operators. The performance evaluation with TPC-H query 6 has shown, that even low-selectivity queries with a large amount of non-qualifying tuples can highly benefit from this approach.

The overall query performance highly depends on the bit-width of the used data types and on efficient arithmetics. The fact that SIMD in x86 architecture is designed and implemented using fixed-width registers, has the consequence that the number of elements that can be processed in parallel depends on the bit-width of their data types. We have shown that narrowing down the attributes can increase the query performance by significantly, even though that attributes must be up-casted into wider types during query processing. Further, due to the fact that SIMD arithmetic lacks of hardware supported overflow detection, checked arithmetics had to be implemented manually and are therefore considerably more expensive than unchecked arithmetic operations. The presented implementations for checked arithmetics outperform scalar arithmetics by factors, with one exception, the 64-bit integer multiplication. However, applying an overflow prevention strategy can increase the performance significantly. E.g. the overall runtime of Q1 is reduced by almost 40% when no overflow checks are performed at runtime. But in general, relying only on overflow prevention can induce negative performance impacts when attributes are unnecessarily up-casted into wider data types. Due to the fact that overflow prevention and overflow detection are not mutual exclusive, we conclude that the overflow prevention strategy can be used to optimize arithmetics primarily with narrow data types.

We further evaluated the performance gains through SIMD in the aggregation operator which perfectly fits into the block-wise processing model. The more or less obvious implementation idea behind the scalar aggregation has been originally presented in [24] and the authors reported speedups in micro-benchmarks that are equivalent to the number of packed elements. However, in a "multi-operator scope" we also have to take type conversions into account. In our experiments, this resulted into speedups of less than $1/2 \cdot P$.

The presented group-by aggregation operator breaks with the column-oriented query processing scheme, due to missing hardware instructions. We therefore generalized the row-oriented approach presented in [26] where multiple aggregates are updated within a single SIMD instruction. Our contribution is an algorithm that determines an optimal layout for hash buckets that can be efficiently updated using SIMD instructions. Thereby the algorithms considers not only all available SIMD instruction sets, it also considers scalar instructions to minimize update latencies. In case of TPC-H Query 1, optimizing the hash bucket updates in the group-by operator led to a overall performance improvement of 26%.

As part of the group-by operator we have shown how block-wise processing integrates with non-parallelized query parts. We evaluated three different approaches on how the transition from parallel to sequential code sections can be efficiently performed, depending on the query's selectivity. The transition between SIMD- and non-SIMD code is considered important, as we in general do not expect that entire query plans can be compiled with the block-wise model. It should rather be seen as a supplement to existing models. Combining the three different approaches, the transition from block-wise to sequential can be performed in 16 cycles/tuple in average and in 6.3 cycles/tuple in best case. Where a block that only consists of qualifying tuples represents the best case.

In general, the block-wise approach fits best for high selectivity queries. Queries that are more selective typically reduce the effective degree of parallelism, because the non-qualifying tuples are passed through the query pipeline even though they do not contribute to the query result. Only blocks that contain no qualifying tuple at all can be skipped during query processing. Depending on the block-size, the block selectivity is much higher than the tuple selectivity, thus with a increasing block size it becomes more likely that a block contains at least one qualifying tuple. Due to the fact, that in our model, costs are incurred on a per-block basis, a query optimizer can apply Yao's formula to estimate the number of qualifying blocks and decide whether to introduce branches or to compile the query plan into sequential code. However, we did not establish a cost or decision model, thus we leave it for future work. Nevertheless, we identified the performance critical aspects that must be considered.

In conclusion, this work has shown that the benefits of using the widespread SSE instruction set in query processing is very limited. On the other hand, Intel's AVX-2 instruction set shows great potentials to accelerate analytical database queries. Therefore, with the spread of the new processor generation, we expect that query engines will adapt the data-parallel approach in near future. Beside Intel's Haswell micro-architecture, AMD released a very promising CPU generation (namely "Kaveri") with an integrated graphics processing unit (GPU). The special feature of this architecture is that the GPU has full access to the main memory, which allows a closer cooperation with the CPU. The integrated GPU consists of eight *compute units* where each compute unit can process 64 elements in parallel. We expect even higher speedups with the Kaveri architecture, because the number of elements that can be processed in parallel is independent from the used data types. Unfortunately, current operating systems do not support the new capabilities of

AMD's latest processor generation.

11.2. Future Work

As this work primarily wanted to answer the question, how much can database queries be accelerated using the latest x86 SIMD instruction set, we focused our work on aspects that are closely related to the underlying hardware. To experiment with the different aspects that arise with SIMD processing, we decided to develop a small query compiler from scratch instead of directly integrate SIMD processing into a existing database system, because this allowed us to deal with the different aspects of SIMD more quickly and flexible. Nevertheless, the design decisions have been made with a subsequent integration in mind. In this section we want to propose how the findings that have been made in this work can be integrated into the main-memory database system HyPer.

11.2.1. Integration Into a Existing Database System

In this section we will discuss the implications for the individual system components. Thereby the focus is on integrating the block-wise approach with the tuple-at-a-time model. The goal pursued through this proposal, is to parallelize possible large parts of query execution plans using SIMD. In general we expect each pipeline to be split into a parallel and a sequential section. Whereas the goal is to *push* as many operators into the prior parallel code section. In the ideal case, as we have seen in Q6, the sequential section does not contain any operators at all and the entire pipeline only consists of a parallel code section. The more common case is that the pipeline is *"sequentialized"* at some point like in Q1.

Integrating the block-wise approach affects many parts of the database system architecture. We briefly cover the affected system components and discuss the necessary changes.

Storage Only minor changes are required to prepare the storage for SIMD processing. The storage system has to ensure that each column (in each partition) is 32-byte aligned. Further, it must be guaranteed that the amount of allocated memory of each column is a multiple of 32 bytes. Otherwise, load instructions would access unallocated memory.

To fully utilize SIMD instructions, the system should represent attribute values using the smallest possible data type. Further, meta data like min/max values have to be collected to enable overflow prevention.

Code Generation As the code generation model presented in this work is basically an extension to the existing code generation in HyPer, it is possible to extend the affected operators to produce/consume multiple tuples at a time. However, this would increase the implementation complexity and might affect the maintainability of the code base. Therefore, we suggest to implement the parallel operators separately from their sequential counterparts. Thereby, special care is needed with pipeline breakers that materialize tuples to memory. Both implementations of the same operator have to be compatible w.r.t. the data structures, because materializing might take place in parallel and whereas reading the materialized data might be performed sequentially.

The more common case, where a pipeline consists of a parallel and a sequential section, can be handled by a special operator, that performs the transition to the sequential part. Conceptually, the SIMD and non-SIMD operators do not share the same interface. Thus,

an adapter needs to be implemented that consumes blocks and produces tuples. The following listing shows how the generated code might look like.

```
for each block b in R
  // parallel code section
  [ ... ]
  for each qualifying tuple t in block b
    // sequential code section
    [ ... ]
  end for
end for
```

This "sequentialization" operator basically emits a loop that iterates over the current block. Thereafter, the produce function of the subsequent sequential operator is called. It is noteworthy, that the presented query compilation method can also be integrated with database systems that make use of other execution model. E.g. the integration with the iterator model can be achieved by wrapping the generated parallel code into a iterator interface.

Query Optimizations Throughout this work we discovered and discussed many performance related aspects of query processing in SIMD. Depending on the query, different individual optimizations have been performed, such as introducing branches, pulling inefficient SIMD operations to the sequential code section, reordering of selections, etc. Integrating the block-oriented approach into a full-featured database system requires that these optimization step are performed by the query optimizer. Most database systems, including HyPer, use a cost-based optimizer to find efficient query plans. Therefore, a cost model needs to be established that covers the aforementioned aspects. Due to the fact that we have investigated only unary operators the optimizations can be performed on each pipeline individually in an additional optimization phase, after a (sequential) query plan has been determined.

A. Appendix

Implementation of 8-Bit Integer Multiplication Using AVX-2 Instructions Only

In the following, we show an alternative implementation of a type-preserving 8-bit multiplication. In contrast to the implementation presented in section 6.3.2, this version does not *mix* AVX-2 with SSE instructions and therefore avoids the data movements from the upper 128 bits of a YMM register into a XMM register. As mentioned before, this version is approximately 14% slower, due to the high latencies of the `_mm256_permute2x128_si256` instruction. However, if we add overflow detection we can hide these latencies and we observe a 13% better performance compared to the "mixed" version, which makes this implementation the first choice with the overflow detection strategy.

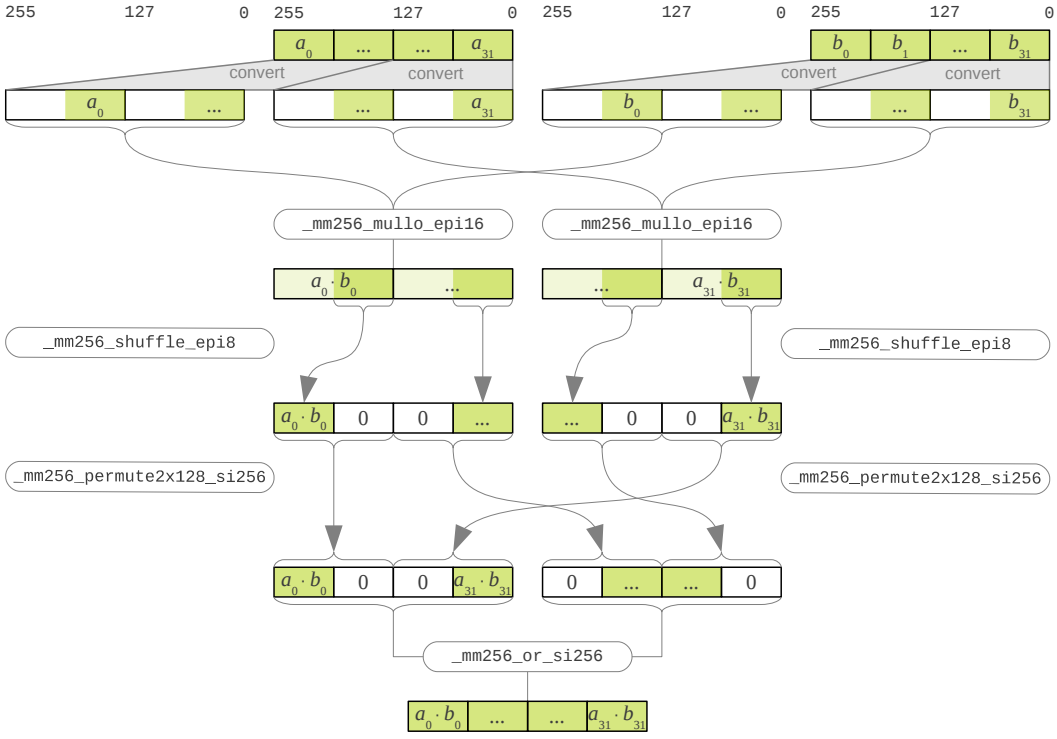


Figure A.1.: SIMD multiplication of 8-bit integers using AVX-2 instructions only.

Bibliography

[1] Austin Appleby. Murmurhash. `https://sites.google.com/site/murmurhash/`, 2014. [Online; accessed June 25, 2014].

[2] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and Tamer Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*, 2013.

[3] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.

[4] M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, Sept 1972.

[5] Intel Corporation. Intel intrinsics guide. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`, 2014. [Online; accessed August 14, 2014].

[6] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[7] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. In *ICDE'10*, pages 613–624, 2010.

[8] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[9] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 0:38–49, 2013.

[10] Raymond A Lorie. *XRM: An extended (N-ary) relational memory*. IBM, 1974.

[11] George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 7 2003.

[12] Guido Moerkotte. Building query compilers (pre-print). `http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf`, September 2009. [Online; accessed August 10, 2014].

[13] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Instant Loading for Main Memory Databases. *Proc. VLDB Endow.*, 6(14), August 2013.

[14] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[15] S. Padmanabhan, T. Malkemus, A Jhingran, and R. Agarwal. Block oriented processing of relational database operations in modern computer architectures. In *Proceedings of the 17th International Conference on Data Engineering*, ICDE '01, pages 567–, Washington, DC, USA, 2001. IEEE Computer Society.

[16] W.W. Peterson and D.T. Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235, Jan 1961.

[17] Orestis Polychroniou and Kenneth A. Ross. High Throughput Heavy Hitter Aggregation for Modern SIMD Processors. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, DaMoN '13, pages 6:1–6:6, New York, NY, USA, 2013. ACM.

[18] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. Micro adaptivity in vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1231–1242, New York, NY, USA, 2013. ACM.

[19] Transaction Processing Performance Council (TPC). TPC Benchmark H (Decision Support) - Standard Specification, 2013.

[20] H.S. Warren. *Hacker's Delight*. Pearson Education, 2012.

[21] SJ Waters. Hit ratios. *The Computer Journal*, 19(1):21–24, 1976.

[22] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *PVLDB*, 2(1):385–394, 2009.

[23] S. B. Yao. Approximating block accesses in database organizations. *Commun. ACM*, 20(4):260–261, April 1977.

[24] Jingren Zhou and Kenneth A. Ross. Implementing database operations using simd instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 145–156, New York, NY, USA, 2002. ACM.

[25] Marcin Zukowski, Sándor Héman, and Peter A. Boncz. Architecture-conscious hashing. In *DaMoN*, page 6. ACM, 2006.

[26] Marcin Zukowski, Niels Nes, and Peter Boncz. DSM vs. NSM: CPU Performance Tradeoffs in Block-oriented Query Processing. In *Proceedings of the 4th International Workshop on Data Management on New Hardware*, DaMoN '08, pages 47–54, New York, NY, USA, 2008. ACM.