Vrije Universiteit Amsterdam   Universiteit van Amsterdam

Master Thesis

On the design of a JVM-based vectorized Spark query engine

Author: Giorgi Kikolashvili (2613952)

1st supervisor:	Peter Boncz	
daily supervisor:	Alicja Łuszczak	(Databricks)
2nd reader:	Bogdan Ghit	

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 25, 2019

Abstract

In this project, we design and implement a JVM-based vectorized Spark query engine. Vectorized query processing operates on batches of data elements, which allows the compiler for the optimizations that are not possible on a single data element. Vectorized query model was introduced by the MonetDB/X100 engine, which is implemented in the native language. Implementing a similar idea for Spark, which is running on the JVM, is interesting for several reasons. First, the dynamic compilation in the JVM can optimize a vectorized engine in a way that is hard to achieve by static compilation. Second, we believe that vectorization can improve on the already existing data-centric engine in Spark, not only performance-wise, but also code-wise. However, the JVM may introduce performance issues in query execution, since the JVM abstracts away low-level hardware controls from the developers, which is otherwise accessible through native language API.

We compare our vectorized engine to the native vectorized engine and the already existing data-centric engine in Spark that uses code generation model. Based on our findings, the JVM can outperform the latter and get close in performance with the former.

Acknowledgements

I want to extend my deepest gratitude to my supervisors Peter Boncz and Bogdan Ghit, for their guidance, encouragement, and advice that they provided me during the development of my thesis. Peter's insistence on improvements pushed me to enhance my knowledge and led to a better quality paper. A special thanks to the Databricks team, namely Julek, Herman, Tom, Bogdan, Nico, and Kris, who spent their time and effort to help me understand the key principles of Spark, and who contributed to my professional step forward. My daily supervisor, Alicja Łuszczak, deserves a special recognition for her recommendations and endorsements that made it possible to complete this work.

Last but not least, I would like to thank all my friends at the university for their support and friendship and the life-changing experience I was able to gain over these two years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the past decade, analytical query processing architectures have diverted from the traditional architectures. The execution of analytical workloads is optimized either through code generation or vectorization. Recent work [1] showed that both approaches are efficient but have different strengths and weaknesses. In transaction processing, where a query may only touch a single tuple, vectorization has a little benefit over traditional tuple-at-a-time iteration. However, code generation can compile all queries of a stored procedure in an efficient machine code fragment. As for disadvantages, code compilation latency may dominate the query execution time in analytical workloads. Profiling, debugging, and code maintenance is also an issue for code generation engines. Vectorization, on the other hand, is less prone to the mentioned issues.

Spark chose the code generation approach, as this model better fits with the existing tuple-at-a-time iterator model, causing less invasive architectural changes. However, compilation latency is an issue for analytical workloads. Code generation also introduces problems with profiling, debuggability, and code maintainability. Furthermore, Spark‘s JIT-compiled path, while being a significant performance improvement over the original query executor, is still far from peak performance. This fact, in part, could be related to it being implemented in the Java Virtual Machine (JVM), and in part to suboptimal design decisions.

Given these issues, it is an interesting question whether a JVM-based vectorized query engine in Spark could come close to peak performance delivered by systems like Vectorwise [2] and HyPer [3], which are on the leading edge of vectorized and code generation query engines, respectively.

1.1 Research questions

These are the research questions we would like to answer to have a better understanding of the impact caused by vectorized execution in Spark:

RQ1: How should a vectorized engine be designed in Java?

A Java program is compiled into bytecode which is interpreted by the JVM. Interpretation comes at a high overhead and is much slower than the similar native program. However, the JVM collects statistics during program execution and identifies hotspots (e.g., loops) that are most time-consuming. It profiles and Just-In-Time (JIT) compiles small native routines for these hotspots. After the compilation is done, the JVM starts using these routines for native execution of the program.

As such, the JVM native compilation can exploit runtime statistics which under the circumstances can lead to faster native code than, for example, when a C++ program is compiled statically. Whether a program running in the JVM is slower or faster than an equivalent program depends on several factors:

i the JVM program should be aware of a memory management strategy to avoid heavy garbage collection (for example, huge memory footprint, multiple short-lived objects) that will slow down the application execution.

ii it should have few performance-critical hotspots that the JVM will JIT compile.

iii it should be relatively long-running because JIT code generation takes time to kick in.

We believe that a vectorized query engine could fulfill these criteria. We can architect it such that most data (columns, vectors) is represented as arrays of scalars (i.e., not as individual objects). The vectorized primitives concentrate most vector operations [4] and are suitable candidates for JIT compilation, and Spark tasks are relatively coarse-grained and long-running. Furthermore, JITed vectorized primitives might even be held in a JVM JIT cache and be re-used inside the same executor by different tasks.

RQ2: Is the JVM able to use and benefit from SIMD in vectorized execution? If so, which JVMs are more suitable for it?

Instruction-level parallelism in vectorized primitives allows better CPU utilization, such as loop unrolling, parallel cache miss resolution, and SIMD (Single instruction, multiple data). SIMD‘s role in query execution becomes more critical with vectorization‘s ability to deliver more work to the CPU. It has been explored by several studies [1, 5, 6]. We look at SIMD from the perspective of a JVM-based vectorized engine. Here, unlikely to native code, we rely only on a JIT compiler‘s implementation to detect the code shape that is a candidate and auto-SIMDize it. Auto-SIMD is brittle and cannot cover all cases. Popular compiler suites (e.g., GCC, LLVM) for native code give more flexibility to the developers by providing more robust auto-SIMD functionality and also allowing them to manually write SIMD commands if the former fails.

We believe that our vectorized primitives are simple enough to hint a JIT compiler on SIMD potential. We test several JVM implementations on a benchmarking framework to reveal those that can better handle the mentioned task.

RQ3: How does a vectorized engine on the JVM perform comparing to the already existing data-centric engine, WSCG?

The data-centric query model replaced the tuple-at-a-time execution engine in Spark, resulting in improved query execution performance. Spark‘s data-centric engine is referred to as whole-stage code generation (WSCG), which is also running in the JVM [7, 8]. For example, this model brings 10x speedup over the old interpreted engine in global aggregation and join operations. However, it created difficulties with compilation, query execution latency, profiling, and maintainability.

A JVM vectorized engine in Spark has the potential to bring improvements over the WSCG. These improvements are not only on the performance side but also include better profiling and debuggability, more natural development process. Vectorization is more amenable to compiler optimization. For example, the vectorized engine operates on a batch of data elements, which allows compilers to apply optimizations using intra-CPU parallelism, such as SIMD.

We discuss their features in Section 2.3.4, and evaluate these models in Chapter 5.

RQ4: How does a vectorized engine on the JVM perform comparing to a native vectorized engine?

Alternatively, one could ask whether a native vectorized engine like Vectorwise would not be a better replacement for Spark, than the existing WSCG engine or the Java vectorized engine proposed here. However, in the case of Spark, a native engine has several drawbacks such as maintainability problems, mixing on-heap and off-heap memory and performance penalties with Java/Scala UDFs (User Defined Function). Spark needs to use a native interface to communicate with processes outside of its environment. Interfaces that cross the JVM memory interface are on or over the edge of the standard API. Spark 2.x, for instance, is incompatible with the more recent JVM versions because it uses the deprecated, so-called “Unsafe” memory API to manage off-heap memory. When using a mix of on- and off-heap memory cannot happen dynamically but statically before the application is launched. Therefore, it is hard to predict how memory should be divided between Spark and a native engine. The incorrect configuration will cause Spark jobs to fail with out-of-memory errors. The users would become responsible for making this memory configuration decision, which would make it (even) harder to write reliable Spark pipelines. With a JVM-based vectorized query engine we can avoid these problems and also stay reasonably competitive comparing to it.

RQ5: How do vectorized engines in the JVM and native language compare when it comes to Java-native UDF performance?

Regarding UDF performance penalties, switching to a native engine for Spark will have to deal with the difficulty that Spark users do not write native code snippets. Instead, they write Python, Java, and Scala UDFs. In the current WSCG, Java and Scala UDFs are relatively efficient as they can be inlined in the generated Java code. The native engine would have to marshall memory parameters into and out of these UDF paying data conversion overhead, and call back into the JVM interpreter to execute the UDFs. Hence, queries that use UDFs intensively will likely get slower, not faster in a native engine. The JVM-based vectorized engine can more naturally integrate it into the execution model. We discuss more details on this topic in Section 3.2.1.

Chapter 2

Background

This chapter introduces necessary information about computer hardware and database query models to help the reader understand the cause of the problem and the proposed solutions. We compare data-centric and vectorized query engine models in more detail and explain how Spark implements the former and how a JVM-based vectorized engine in Spark can reduce query execution time.

2.1 Computer hardware

Being aware of modern computer hardware will help us understand the design choices in modern query engines and the reason why they perform better than the traditional query engine.

2.1.1 CPU pipelining

Before explaining how superscalar CPUs work, we first need to explain what CPU pipelining is. The instructions that are executed by CPU are divided into a series of sequential stages. On each CPU cycle, one stage of instruction is executed. On the next cycle, the next stage is executed. This way, an instruction progresses through the CPU pipeline. Each stage has a clear, separated role which is implemented by a hardware circuit. On every cycle, a new instruction is issued and the instructions that were "in-flight" progress to the next stage. This way, the CPU is executing different stages of different instructions in parallel and tries to keep every CPU part busy.

In the ideal case, on each cycle, a new instruction enters the pipeline. The case when this may not happen is described in the following section. Pipelines on modern processors have 12-20 stages. Modern CPUs have 4-6 pipelines and multiple instruction units that can issue as many instructions as there are pipelines. We can have 48-120 instructions in-flight as long as they are independent. A CPU with multiple execution units is called a superscalar processor. It can execute multiple instructions in parallel using multiple instruction units. Furthermore, the order instructions are executed may not follow the order defined by the program. This technique is called out-of-order execution.

The metric Instruction Per Cycle (IPC) measures how many instructions are executed per cycle. The theoretical top of 4-6 is hardly achieved. IPC of 2-3 is considered as good. It has been observed that traditional database workloads get IPC of 1 or even below.

2.1.2 Data and control dependency

Pipelined execution efficiently uses hardware units, but the data and control dependency between instructions may harm the throughput. The former happens when the result computed by one instruction is used in the next instruction. Here, instructions cannot progress through pipeline stages unless the instruction they depend on produces a result. The latter, control dependency, occurs when an outcome of one instruction determines the location of the next one. If this condition is not yet known, as the condition is computed by an instruction that still has to finalize all its pipeline stages, it is unclear which instruction should be taken into execution next.

Listing 2.1: Data dependency

```
while (i < a.length) {
    a(b(i)) += 1
    i += 1
}
```

Listing 2.2: Control dependency

```
while (i < a.length) {
    if (a(i) == 0)
        counter += 1
    i += 1
}
```

Listings 2.1 and 2.2 shows the examples of data and control dependency, respectively. Listing 2.1 presents a loop where each iteration contains an instruction that depends on the result from the previous iteration. Listing 2.2 shows an example of control dependency. The if-then branch creates a dependency between loop iterations. Using techniques called branch prediction CPU can guess the next value of program counter, and using a technique called speculative execution CPU starts executing instructions form the guessed location. If it is later determined that the branch was mispredicted, the CPU pipeline is flushed, and the correct instructions are executed.

2.1.3 SIMD

Even higher parallelism can be achieved using SIMD instructions. They operate on vectors of data using a single instruction. These vectors are held in special registers called `xmm`, `ymm` and `zmm` with 128, 256 and 512 bits in size, respectively. For example, `xmm` register can hold four 32-bit integers or sixteen 8-bit integers, and it can operate on this data at the same time. In Intel‘s CPUs this technology was introduced in 1996 with the name MMX that supports 64-bit registers. In 1999 and 2008, Intel introduced new SIMD technologies called *Streaming SIMD Extensions* (SSE), *Advanced Vector Extensions* (AVX), which support 128- and 256-bit registers, respectively. The 512-bit extension to AVX, called AVX-512, was introduced in 2015.

Using SIMD functions in the program requires special machine instructions to be issued. We can (1) manually write SIMD intrinsics in the source code and use a compiler to produce SIMDized machine code, or (2) we can use a compiler that is smart enough to detect SIMD potential from the source code and produce SIMDized machine code. For example, GCC supports manual SIMD intrinsics, and it can also automatically SIMDize the code. Java, on the other hand, does not support manual SIMD yet. Therefore we depend on the JVM‘s implementation to recognize the source code and produce SIMDized machine code. Section 3.3.3 explores SIMD support for Java and how it can benefit our project in particular.

2.1.4 Memory hierarchy

Physical constraints make it challenging to create fast storage that is also large. Therefore computer storage is arranged in a hierarchy. The top of the hierarchy is CPU registers with a hard drive in the bottom. Table 2.1 shows the approximate access latency and size of different memory levels.

When the CPU executes a load instruction from an address in the main memory, it sends this address to the L1 cache. If it is a match, the load instruction reads data from the cache. Otherwise, it checks lower levels of caches and addresses a memory or disk, in the worst case. Cache friendly programs can benefit from this hierarchical memory organization by exhibiting temporal or spatial locality. Temporal locality implies that the same object can be reused. Once a cache miss brings data in the cache, we can access the same neighboring address without incurring cache miss again. Spatial locality means that after a cache miss brings a memory block in the cache, subsequent accesses will be inside

Type	Latency (cycles)	Size
CPU registers	0	Few thousand bytes
L1 cache	4	32 KiB
L2 cache	10	256 KiB
L3 cache	50	8 MiB
Main memory	200	10s of GB
Flash memory	100.000	100s of GB
Disk storage	10.000.000	100s of GB

Table 2.1: Memory hierarchy and the latency to access different levels [9]

this block and they will benefit from the faster memory access. For example, this effect is observed during sequential array access.

Cache misses can damage pipeline performance by stalling all the work in the pipeline while the cache miss is resolved. The waiting time becomes even longer when a cache miss causes a page fault, which results in thousands of CPU cycles to fetch the data and update the cache. However, non-blocking caches allow having multiple outstanding cache misses all being resolved in parallel. The hardware keeps track of the physical memory addresses being resolved and can merge requests in the same block.

Having multiple outstanding cache misses is referred to as *Memory Level Parallelism* (MLP). The parallel misses can be generated by out-of-order execution or prefetching. The information about outstanding cache misses is stored in a hardware structure called *Miss Status Handling Register* (MSHR). This register, among other information, holds data about the physical address of the memory block, data about the words which are accessed in the block, a destination register number, and a store buffer entry address. On every cache miss, an entry is allocated in MSHR. This way CPU keeps track of pending memory requests. If there is no space left in MSHR, the memory request is stalled. CPU‘s memory management unit can use this information and issue parallel memory request. Modern CPUs can have 4-8 outstanding cache misses [10].

2.2 Java virtual machine

The Java virtual machine (JVM) interprets a code written in Java bytecode which is produced by compiling a Java source code. The JVM, among other features, offers automatic memory management and dynamic bytecode compilation. In this thesis, we will not focus on memory management. We do assume it to be common knowledge that garbage collected memory is not suited for the purpose of big data processing, especially when huge volumes of memory get continuously allocated and de-allocated by the creation and destruction of large amounts of small objects. This will trigger garbage collection, halting the progress of JVM applications for considerable amounts of time. As such, we take it as a design principle that a database query processor should not create many objects, e.g., one object per processed column value or even per processed row. Instead, we will try to understand how dynamic compilation works and how it can affect a vectorized query engine.

Figure 2.1 shows how Java source code is transformed into an intermediate representation, called bytecode and later compiled to machine code. Java compiler (javac on the figure below) does not optimize the code at this stage. It accepts Java source code as input and intermediate representation called bytecode. The application is launched by loading the bytecode in the JVM. This process is shown on the image by the arrow connecting Bytecode and Interpreted code rectangles.

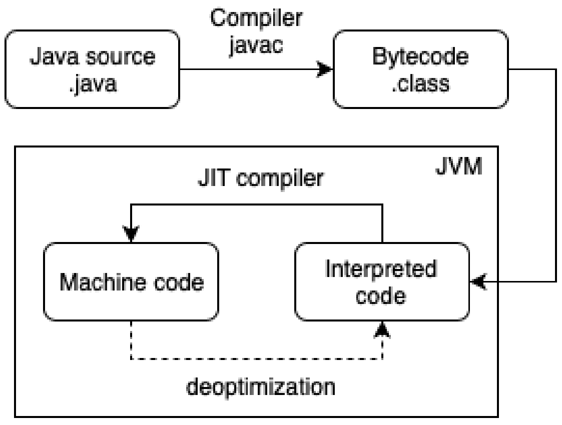


Figure 2.1: Compilation process in the JVM

The JVM starts application execution by interpreting and profiling the bytecode. If the application runtime notices that a function is executed sufficiently many times, it triggers the JIT compiler and produces machine code that replaces the interpreted function. Usually, there are five compilation tiers in the JVM, 0 being the interpreter. Higher tier levels are part of the JIT compiler (C1 and C2). Tiers 1-3 are part of the C1 compiler. These tiers are activated based on the function call frequency (hotness). The difference between these tiers is how aggressively they optimize the code and the profiling information they collect about the code execution. Tier 4 is called the C2 compiler which optimizes the produced machine code more aggressively than the C1 compiler. The JVM uses a compilation cost model (different from static compilers) which depends on the code execution patterns and system load. This model decides which tier should be used for machine code generation.

Bytecode interpretation starts from tier 0, and the JVM periodically switches to higher tiers based on the profiling information. The JVM reverts to interpreted code if conditions observed during profiling change. This event is called deoptimization [11]. It is shown with a dashed line on Figure 2.1. The deoptimized methods are profiled and recompiled. This event may happen multiple times for a single function, and the recompilation process may damage the application‘s max throughput. However, the generated machine code tends to be more optimized than its previous version.

2.2.1 Dynamic compilation

The JVM is running in tier 0 until runtime statistics are collected. Code interpretation is slower than running a machine code. The time before JIT compilation is activated is known as the JVM warmup period. When the JVM collects sufficient profiling information, it uses the C2 compiler and produces machine code. This machine code is optimized for the specific hardware that hosts the application, and the optimizations applied are based on the observed data patterns. Runtime profiling is expensive since the computer resources are limited and shared with the application. Therefore, some JIT compilers are more conservative with applying aggressive optimizations and settle with the suboptimal machine code. Other compilers, like Azul Zing‘s Falcon JIT compiler, go for slower compilation time in exchange for higher max throughput.

The opposite model of dynamic compilation is static, or ahead of time (AOT) compilation. It directly translates the source code to machine code during the compilation stage, and there is no runtime optimization taking place. AOT compilation thoroughly analyses and heavily optimizes the source code because computer resources are less constrained. The downside of AOT is that sometimes, static information is not enough to produce the optimal machine code.

JIT compilation produces the optimized machine code based on the runtime profiling that can provide valuable information for the compiler that is not available during the static compilation. Some of the optimization examples that are available only due to the runtime profiling are (a) devirtualization and (b) range check elimination.

2.2.1.1 Devirtualization

Before explaining devirtualization, first, we need to understand why it is needed. A virtual call (or a dynamic dispatch) is a process of choosing an implementation of a polymorphic function at runtime. When a class defines a virtual function, a hidden pointer is created inside the class that points to a virtual method table, which holds pointers to appropriate function implementations. The virtual table is used during the runtime to call the appropriate function implementation. This level of indirection introduces an overhead during a virtual function call. This model is commonly used in programming languages, such as Java or C++, and it is used for polymorphism.

The JVM has a technique to mitigate the performance penalty of virtual calls. Using the Class Hierarchy Analysis (CHA), the JVM collects information about the loaded classes. If it determines that only a handful number of subclasses of the parent class are loaded, a virtual call to this class can be devirtualized. It will be called like a regular function and can even be inlined if the target method is small enough. Devirtualization is expected for mono- and bimorphic calls. Devirtualization refers to a technique when the JVM assumes that there are only one or two receivers for the method call and during the future calls to this method, control is transferred directly to the receiver methods without consulting a virtual table. The CHA can statically identify the monomorphic cases and devirtualize them. The bimorphic calls are devirtualized based on the runtime profiling information. In the generated assembly code, the most frequent virtual function call will be placed on the straight path, and the less-frequent version will be branched. The branch prediction will follow the straight path and take the branch in exceptional cases only. This technique is known as basic blocks. In the case of megamorphic calls, when there are more than two receiver types, the C2 compiler usually fails to devirtualize them. However, some JVMs may still devirtualize the call if one of them dominates all other calls by 90% or more.

2.2.1.2 Range check elimination

Range check elimination is an optimization in the C2 compiler that removes explicit array index access checks from loops. These range checks are generated in compiled code, and they guarantee to execute uncommon traps if we access the array out of bounds [12]. Listing 2.3 contains the code obtained from the OpenJDK website. It shows the example of the loop accessing an array and the generated code with a range check.

Listing 2.3: Java loop and how the JVM checks array access validity

```
for (int index = Start; index < Limit; index++) {
  Array[index] = 0;
}

// JVM generated loop
for (int index = Start; index < Limit; index++) {
  if (index < Array.length) {
    Array[index] = 0;
  else
    uncommon_trap(range_check);
}
```

Branching in a loop is an expensive operation because it introduces control dependency between consecutive loop iterations. C2 compiler manages to go around this problem by splitting the iteration into three sections. The main loop‘s ranges are chosen to be as large as possible without violating the array access requirements. Listing 2.4 is an example of the transformed loop without an explicit range check.

Listing 2.4: Range check elimination by the C2 compiler

```
int MidStart = Math.max(Start, 0);
int MidLimit = Math.min(Limit, Array.length);
int index = Start;
for (; index < MidStart; index++) { // PRE-LOOP
  if (...) { // RANGE CHECK
    Array[index] = 0;
  else {...}
}
for (; index < MidLimit; index++) { // MAIN LOOP
  Array[index] = 0;   // NO RANGE CHECK
}
for (; index < Limit; index++) {   // POST-LOOP
  if (...) { // RANGE CHECK
    Array[index] = 0;
  else {...}
}
```

The JVM enforces bound checking by examining every array access. JIT compiler is allowed to remove bound checking when it observes that the index is always within bounds.

2.2.2 Loop unrolling

Loop unrolling is a loop modification technique that increases the amount of work done on a single iteration and helps superscalar CPUs to launch several operations in parallel. This technique is not unique to dynamically compiled languages. We can apply it in statically compiled languages as well. Listing 2.5 shows 2x unrolled loop.

Listing 2.5: Manually unrolled loop with two accumulators

```
for (int i = 0; i < a.length; i += 2) {
  sum1 += a[i];
  sum2 += a[i + 1];
}
return sum1 + sum2;
```

Notice that we have two accumulators in order to break the dependency between the pipeline instructions. This procedure can be applied to unroll with a higher degree of parallelism. However, we are constrained by the number of functional units in the CPU, the number of registers, and the instruction cache size.

Manual unrolling is not advised since compilers already use techniques to optimally unroll loops for us. Manual unrolling can introduce a degree of parallelism that is higher than the available number of registers. In this case, the compiler will fall back to register spilling by storing some of the variables on runtime stack instead of registers. Furthermore, manual unrolling produces larger machine code which may suffer from instruction cache missed.

2.2.3 Code cache

JVM keeps the JIT-compiled code in a memory area separate from JVM heap space, called *CodeCache*. By default, the cache size is 48 MB, but it can be resized using a JVM flag. Machine code stored here is called *nmethod*. Data from the code cache is evicted through the technique called code cache flushing which comes in two modes: default and speculative. By default, nmethods are evicted using a scanning and sweeping mechanism. During scanning, nmethods are marked if they are not entrant. A method can become not entrant if (1) the class which owns the method is unloaded or (2) it is deoptimized. Class unloading is caused by garbage collecting the class loader that loaded the class. This scenario is unusual for most of the Java applications. Deoptimization happens when an uncommon trap is taken in the method. An uncommon trap can be taken if the optimization proves to be invalid [13]. One of the triggers for deoptimizations is incorrect devirtualization due to new class load.

The sweeping mechanism is triggered before the compiler obtains new jobs from the compilation queue. The sweeper visits nmethods in the code cache and marks them as zombie methods if they were marked during scanning method and they are not on a call stack. Existing callers are still allowed to use not entrant methods. If no more stack frames hold the program counter to the deoptimized code, then it is marked as a zombie method during a sweeping step. If a sweeper sees zombie nmethods, they are marked for reclamation and flushed from the code cache.

The processes discussed above happen during flushing of CodeCache using the default flushing mechanism. With this mechanism code cache size usually keeps growing unless it gets full, which causes JVM to fall back to the interpreted mode. The speculative flushing mode is activated when a free space in CodeCache is below a certain threshold (1500 KB by default), the used cache size is increasing rapidly or a certain amount of time has passed since the last sweep. The nmethods are being evicted from the cache if they become too cold. A method is cold if a method‘s counter, which is decreased by every sweep operation, goes below a certain threshold. The evicted methods are first moved to the old list, and if they are not referenced during the next two sweeps, they are flushed.

In the case of our JVM-based vectorized engine, it is essential to keep the JIT-compiled vectorized primitives in the code cache and also prevent the cache from filling up. Methods in Java are class-based, not instance-based. All instances of the same class will use the same JIT-compiled code. Having vectorized primitives in the cache will reduce the warmup time for the new queries. Queries sharing the same expressions will benefit from reusing the cached code. However, the primitives that are no longer in use will be flushed from the cache only if the CodeCache becomes full. In long-running programs, like Spark, hopefully, the optimization cost is paid only once, and then the following queries can use already optimized code. This can be beneficial for short, interactive queries that require low latency, such as analytical workloads.

Figure 2.2 shows how query execution time is affected in our JVM-based vectorized engine when the code cache is full. We simulated the experiment by reducing the JVM code cache size to 3 MB. Based on our observation, Spark with our JVM-based vectorized engine uses approximately 12 MB of code cache to run TPCH Q1. Listing 2.6 is an example of vectorized primitives used in our project.

Listing 2.6: A vectorized primitive implemented in Scala

```scala
def addInts(a: Array[Int], b: Array[Int], out: Array[Int], length: Int): Unit = {
  var i = 0
  while (i < length) {
    out(i) = a(i) + b(i)
    i += 1
  }
}
```

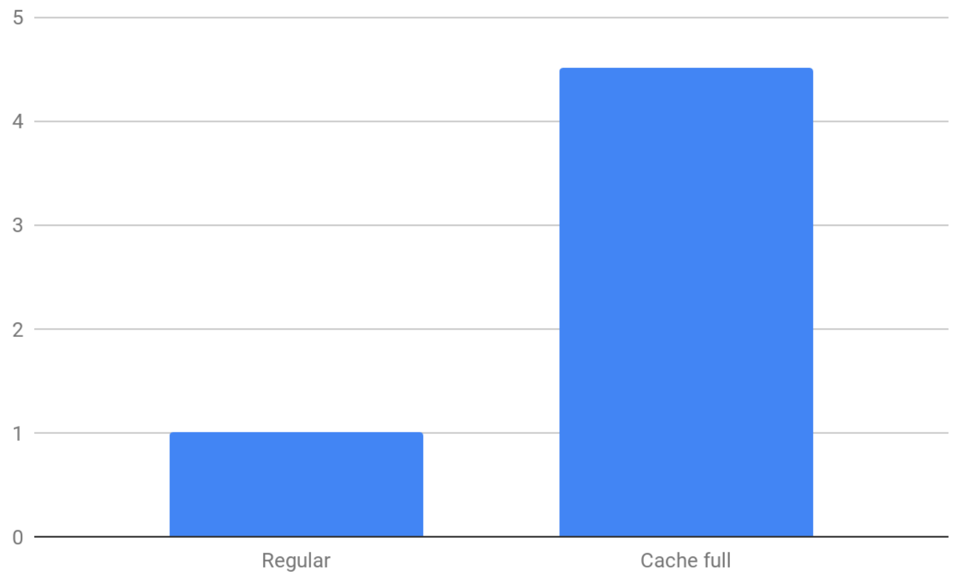


Figure 2.2: JVM-based vectorized engine performs 4.5x slower when JVM's code cache is full

The JIT-compiled code produced from this function takes 512 bytes in the code cache. This information is provided by the JVM compiler if `-XX:CompileCommand=print,className` flag is provided. Produced machine code size depends on the JIT implementation, data patterns, and hardware running the JVM.

2.3 Query execution models

Traditionally, query executors have followed Volcano [14], also known as the tuple-at-a-time model. This model is inefficient for CPU-bound workloads, for example, in in-memory databases. Systems based on Volcano model tend to spend more time interpreting a query plan than evaluating the query result [1, 15].

We explain the problem related to the Volcano model and how two new query engine architectures, data-centric code generation [3] and vectorization [2], try to solve it.

2.3.1 Tuple-at-a-time

To better understand the problem, we can look at figure 2.3. We see that evaluating a single tuple requires operator and expression tree traversal. The trees are traversed using `next()` and `eval()` virtual functions. Here, data operation instructions are interleaved with virtual function resolution instructions. Virtual function calls come with an overhead that prevents deep CPU pipelining [16]. As a result, the tuple-at-a-time model spends more time in query plan interpretation than on actual data operation.

Operator tree execution starts by calling the root node‘s `next()` function. Each node calls its child‘s `next()` function until a leaf node is reached. The data is pulled from the leaf node toward the root node. This data iteration model is called pull-based model.

Figure 2.3 shows the generated operator and expression trees from the SQL query shown in Listing 2.7. The pseudocode for operators and an expression node is embedded in the figure.

Listing 2.7: Global aggregation with predicates

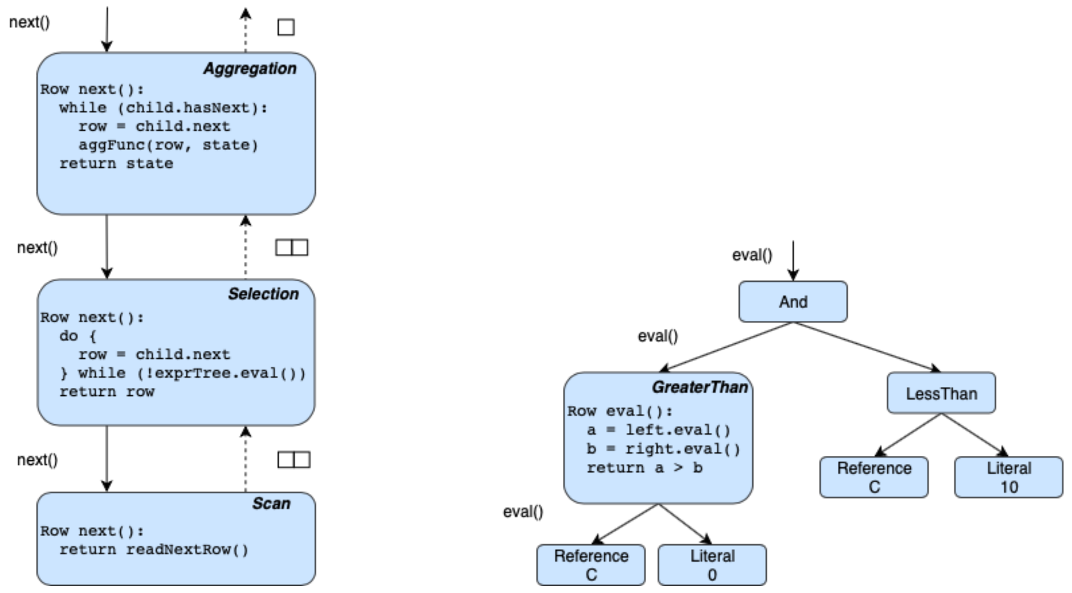
```sql
select sum(A) from table where C > 0 and C < 10
```

On Figure 2.3a, scan operator, which is a leaf node, reads a single tuple from the data store and returns it to the caller. Selection operator checks the predicate (`C > 0` and `C < 10`) against the received tuple. Those tuples that pass the predicate test are sent up in the operator tree. Next, the aggregation operator reads a tuple from the selection operator and adds up attribute `A` received from the child operator into an aggregation state. Aggregation only returns the result after it reads all tuples from its child operator.

Expressions are represented using trees, shows in Figure 2.3b. The expression tree is evaluated by calling `eval()` function on a root node. Each node calls `eval()` of their children, traversing the tree down to the leaf node. Calling `eval()` on a leaf node returns appropriate attribute from the tuple, which is passed to the caller.

The scan operator on every `next()` function call reads a tuple (`readNextRow()`) and returns it. The filter operator receives a tuple from the scan operator and evaluates it using the expression tree (`exprTree.eval()`). The result of `eval()` function indicates if the filter predicate is satisfied or not. If it is satisfied, the tuple is passed up to the operator tree. Otherwise, the filter operator calls its child to retrieve the next tuple. The aggregation operator applies the aggregation function (e.g., sum) to the tuples returned from its child. In our example, the aggregation function (`aggFunc(row, state)`) adds the value from the received tuple to the aggregation state. The result from the aggregation operator is returned once its child returns all tuples.

The functions `next()` and `eval()` are polymorphic functions. Calling these functions means that the runtime has to look up the virtual function table to determine the next location in the code to be executed. Because of this reason, the tuple-at-a-time model is an interpreted engine. From the Figure 2.3 we can see that the virtual functions `next()` and `eval()` is called multiple times while pulling the tuples through the operator and



```
Row next():
  while (child.hasNext):
    row = child.next
    aggFunc(row, state)
  return state
```

```
Row next():
  do {
    row = child.next
  } while (!exprTree.eval())
  return row
```

```
Row next():
  return readNextRow()
```

```
Row eval():
  a = left.eval()
  b = right.eval()
  return a > b
```

(a) Query operator tree **(b)** Query expression tree

Figure 2.3: Example of a query plan built from the query in Listing 2.7

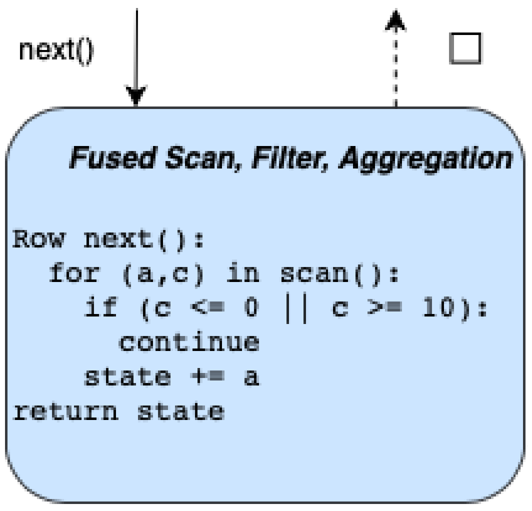
expression trees. Query interpretation imposes extra overhead in the engine and prevents it from benefiting from modern hardware features discussed in Section 2.1.1.

2.3.2 Code generation

Data-centric code generation was pioneered by HyPer [3] as an alternative to the tuple-at-a-time execution model. The idea is to avoid query interpretation and its overheads by compiling query-specific, efficient machine code. This process is also known as JIT-compiling. This model diverges from the traditional pull-based iteration and introduces a push-based interface. It generates specialized code for a given query that is later compiled to efficient machine code with a reduced number of function calls. This model, like the tuple-at-a-time model, operates on a single tuple on each iteration.

On Figure 2.4, we see a single operator and the generated code for this operator. We can observe that there is only one virtual function call, and there is no expression tree to evaluate the predicate. The model fuses a pipeline of relational operators that do not need to materialize intermediate results. An expression tree is "inlined" inside the operator's `next()` function, and there are no more virtual function calls needed to evaluate the predicate result. Using this strategy, we reduce query interpretation overhead.

We have briefly mentioned the drawbacks of code generation in Section 1.1. Compilation latency, code maintainability, and debuggability problems were among the issues. More



```
Row next():
  for (a,c) in scan():
    if (c <= 0 || c >= 10):
      continue
    state += a
return state
```

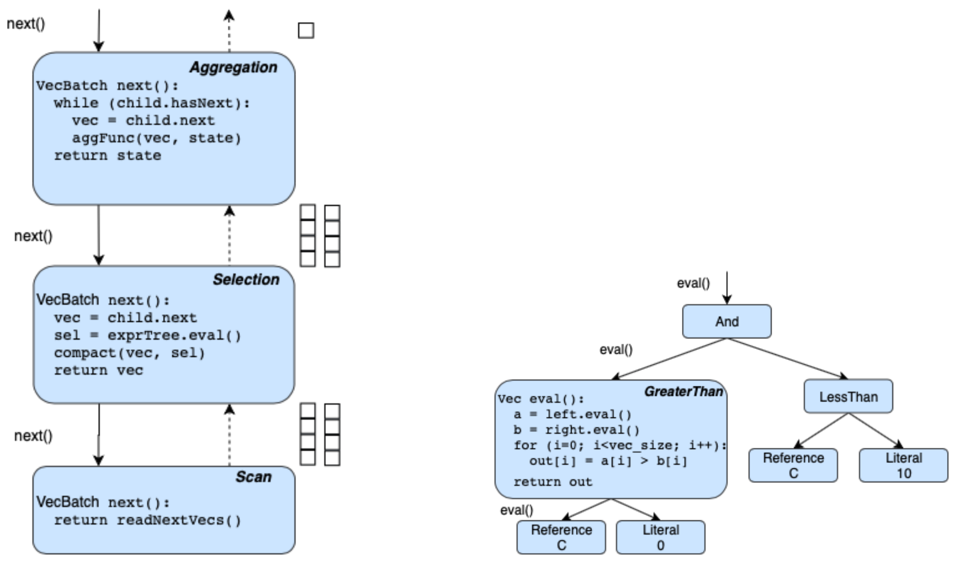
Figure 2.4: Example of a fused operator tree built from the query in Listing 2.7

information on this topic can be found in Section 2.3.4. Additionally, leveraging SIMD instructions in the data-centric model seems to be harder than in vectorized engines, since different types of operations are applied to every single tuple, while SIMD instructions requires multiple data elements to operate on [1, 5].

2.3.3 Vectorization

MonetDB [17] introduced an alternative to the tuple-at-a-time processing model. Interpretation overhead is reduced by operators fully processing their input before invoking the next execution stage. Hence, tuple-at-a-time iteration is replaced by bulk processing in which only one single operation is performed to produce one output column. This column-at-a-time execution allows to hardcode implementations, thus eliminating interpretation overhead. However, this method requires high memory bandwidth due to the materialization of intermediate results. MonetDB/X100 [2], later renamed to Vectorwise, further improved the idea and pioneered vectorized execution, providing an implementation of the concept. The main idea of vectorized execution is to process data in batches large enough to amortize the interpretation overhead, while at the same time small enough to keep it hot in the CPU cache at all times, avoiding crossing the cache-memory boundary multiple times and the associated materialization cost. Another benefit of the vector-wise models is compiler optimization amenability. Data is processed in tight loops, which makes it easier for compilers to loop-unroll and use SIMD instructions. Several works [5, 18] explored the benefits of SIMD in database applications, which allows high instruction-level parallelism and eliminates conditional branches and branch mispredictions.

Figure 2.5 shows the operator and expression trees in a vectorized query engine. Instead of rows, operators and expressions operate on and return vectors. When evaluating an expression tree, we still have the same number of virtual function calls as in the tuple-at-a-time model, but the interpretation overhead is amortized over vector size. Therefore, the total number of virtual function calls is reduced by a factor of vector size. There are several details to notice in the figure. Query operators may return multiple vectors from different columns that are wrapped in `VecBatch` type. The details about a selection operator implementation are presented in Section 4.3.



```
VecBatch next():
  while (child.hasNext):
    vec = child.next
    aggFunc(vec, state)
  return state
```

```
VecBatch next():
  vec = child.next
  sel = exprTree.eval()
  compact(vec, sel)
  return vec
```

```
VecBatch next():
  return readNextVecs()
```

```
Vec eval():
  a = left.eval()
  b = right.eval()
  for (i=0; i<vec_size; i++):
    out[i] = a[i] > b[i]
  return out
```

(a) Vectorized query operator tree **(b)** Vectorized query expression tree

Figure 2.5: Example of a vectorized query plan built from the query in Listing 2.7

2.3.4 Comparison

These models are fundamentally different in the way they process the data. Nevertheless, it is challenging to say which query execution model is superior [1]. Data-centric code generation is a compiled query engine, vectorization, on the other hand, is an interpreted query engine. Here, compilation means that when the engine receives a query, it generates a specialized code that has reduced number of virtual function calls. In the case of vectorization, program runtime has to interpret operator and expression tree, which is usually done through a class interface. This process requires virtual function resolution during query execution. Therefore, it is an interpreted query engine. Table 2.2 summarizes the main design features of the three mentioned architectures.

	Iterator	**Code generation**	**Vectorization**
Granularity dimension	Row	Row	Vector
Compilation	Fully interpreted	Fully compiled	Fully interpreted

Table 2.2: Key features of different query engines

Vectorization is more amenable to compiler optimization such as SIMD and loop unrolling. Vectorized expressions consist of simple, tight loops that can be translated to data-parallel code. Therefore, operating on a batch of data points allows a query engine to reduce interpretation overhead and use intra-CPU parallelism. For example, vectorized hash lookup eliminates dependencies found in tuple-at-a-time hash lookup. This means, that the CPU can generate multiple concurrent cache misses, which significantly improves the memory throughput achieved.

Queries bound by data access, and not by computation, do not benefit much from SIMD. Regardless of this fact, SIMD may have a more significant impact on query execution performance as hardware evolves. Increasing CPU cache size and SIMD register size will allow vectorized query engines to process data with higher throughput. For example, SATA hard drive disks achieve ∼250 MB/sec sequential read speed, while NVMe SSDs can get ∼1.5 GB/sec throughput. Intel‘s Cascade Lake CPUs have higher core count and newer technologies like AVX-512. These hardware improvements are becoming more accessible through cloud providers.

As we discussed above the vectorized engine uses an interpreted execution. This means that a vectorized engine can benefit from runtime predicate reordering based on the predicate cost and selectivity. The predicate order cannot be adjusted in case of generated code. Additionally, vectorization is not vulnerable to query execution latency, which is a drawback for code generating query engines that need to compile the generated code [19].

The distinction is also visible when we focus on factors like profiling and debuggability, where vectorization has advantages. Strict boundaries between operators or vectorized primitives make it easier to identify performance issues caused by distinct query plan components. Furthermore, sampling the clock cycle count for each vectorized primitive that processes a batch of tuples (usually 1024 to 4096 at a time) adds only marginal overhead to query execution.

The problems with profiling and debuggability are visible in the WSCG, which is based on the code generation model. WSCG brought performance improvements, but it also introduced challenges for the developers. Erasing boundaries between query operators and fusing their functionality creates maintainability issues. It may be impossible to split runtime between the fused operators. This results in reduced performance understanding on the part of developers and end-users. Another problem with profiling is that because the query code is generated, small changes in a query may produce a code that suffers from performance issues. A generated Java class sometimes crosses the 64 KB size limit that prevents the compiler from processing it. Similarly, a method bytecode larger than 8 KB is not optimized and JIT-compiled. Spark developers had to put extra engineering effort to overcome the problems related to Java compiler.

As for debuggability, the WSCG engine is written in Java, but it actually generates temporary Java code that executes a task. Hence, the buggy code does not have a permanent source code. When something goes wrong, it is difficult to find the temporary source code as this gets deleted. Also, to fix a WSCG bug, one should fix, not change the code that is wrong, but the code that generated that code, which is another abstraction level. These maintainability and development challenges created a foundation for the next generation query engine in Spark.

These factors serve as an example that vectorization is more favorable to code maintenance and debuggability than code generation.

2.4 Related work

Two schools of thought redesign a query execution model to deal with the problems revealed by the iterator model [14]: code generation and vectorization. The database community has done substantial work to explore these two models [2, 3, 14, 19, 20, 21, 22, 23]. Code generation and vectorization are two fundamentally different, orthogonal query execution models. Some academic or industry projects explored the design space between them [15, 24, 25, 26].

The idea of code generation is to specialize the query code, thereby avoid the interpretation overhead. The idea of code generation was introduced in System R [20] which directly produced assembly code. The project was later abandoned due to the high maintenance cost of the code generating engine [27]. HIQUE [19] uses code templates to generate a code for each query operator. Predicate evaluation and low-level access code is customized based on data types. Operator output is materialized in memory and consumed by the following operator. This model can be characterized as operator-centric. HyPer [3] popularized data-centric (push-based) model in code generation engines. Here, the data flow is reversed. Tuples are pushed from child to parent operator. Furthermore, the data-centric code generation manages to fuse several query operator logic into one code that uses CPU registers to operate on a tuple. The code generation model introduced by HyPer was also adopted by other query engines [7, 21, 26].

Vectorization amortizes the interpretation overhead by batch processing. MonetDB [17] experimented with column-at-a-time processing. Here, operators fully consumed their input before invoking the next execution stage. However, this method requires high memory bandwidth due to the materialization of intermediate results. The idea was further improved by MonetDB/X100 [2], later renamed to Vectorwise, which pioneered vectorized execution. The batch size is big enough to amortize the interpretation overhead without materializing the data in memory. Other engines that are built on this model are IBM DB2 BLU [22] and Quickstep [23]. The fact that the query code is not compiled allows for changes in the query plan during the runtime. For example, predicates can be reordered based on their selectivity to reduce the overall predicate evaluation cost [4]. Furthermore, based on the observed data statistics, it is possible to change the attribute data types dynamically to reduce the memory footprint of the batched data [5].

There exists several hybrid systems that take the best from both worlds. Peloton [24] is built on a code generation engine that uses a form of batching to make query expressions SIMDizable. This type of batching serves as a prefetching mechanism for operators that issue random memory access. Tupleware [26] examines UDFs are examined for SIMD opportunity and splits them into vectorizable and non-vectorizable groups. Vectorized UDFs process data in batches and store the output result in cache-resident data blocks. However, UDFs need to use the Tupleware‘s provided API in order to be examined by the system. Sompolski et al. [15] explored compilation strategies in Vectorwise. The scope of the project was vectorized primitive fusion into a single loop, called “loop-compilation.” This technique can avoid materialization of intermediate query results when calling consecutive query expressions.

The code generating engines discussed so far [3, 19, 22, 23, 24] are implemented in the native language (e.g., C/C++), and they produce native code or an IR (e.g., LLVM IR) that is compiled to machine code. Several papers [7, 14, 21, 28] introduce code generating engines implemented in JVM-based languages. LegoBase [21] is implemented in Scala. It uses LMS framework [29] to store the query execution logic in highly-customizable IR and to generate the final C code. JAMDB [6] is a system implemented in Java that produces query-specific Java code that is compiled and loaded into the running JVM. The authors claim that generated Java code can benefit from the dynamic optimizations by the JVM‘s JIT compiler. Spark‘s [7] is based on the model proposed by HyPer. It produces Java source code that is compiled and loaded into the JVM. Presto is another JVM based code generating engine based on the HyPer system. Carefully generating the bytecode can take advantage of optimizations provided by the JIT compiler. Presto developers are experimenting with GraalVM [30] in scenarios where the JVM is not able to generate optimal machine code (e.g., SIMD optimization). To the best of our knowledge, no academic paper explores a JVM-based vectorized engine.

Chapter 3

The design of a JVM-based vectorized query engine

It is interesting to see whether the advantages of vectorization can also be achieved inside the JVM and to what extent, since this would ease backward compatibility for Java and Scala UDF code as well as RDD code. Furthermore, it would lead to a simpler JVM-only architecture, compared to the native engine‘s approach.

3.1 Vectorized primitives

The tight loops that operate on vectors are called vectorized primitives. They execute all of the work specified in the query. These short functions perform arithmetic or boolean operations. Vectorized primitives access memory to read or write vectors. Listing 3.1 shows the vectorized primitive implemented in Scala.

We make an assumption that our JVM-based vectorized engine will support 6 numerical types (`byte`, `short`, `int`, `long`, `float`, `double`). A vectorized primitive of a single arithmetic operation that accepts all combinations of parameter types would result in $6 * 6 = 36$ different implementations for arithmetic operations (also assuming that no explicit casting is applied to the output). Primitives can be reused for associative arithmetic operations (e.g., addition, multiplication) and associative data types (`byte`, `short`, `integer`, `long`). Supporting four types of arithmetic operations (add, subtract, multiply, divide) would yield $4 * 36 = 144$ vectorized primitives. The variations of vectorized primitives increase even more as some primitives need to consult NULL selection vector or indirection array before operating on a data point. NULL selection vector indicates whether the data element on index `i` was originally null. Indirection array is used in case of grouping aggregation. It holds the positions of the hash table values that need to be updated.

Listing 3.1: A vectorized primitive that is aware of a NULL selection vector

```scala
def addInts(a: Array[Int], b: Array[Int], out: Array[Int], length
    : Int, nulls: Array[Int]) = {
  var i = 0
  if (nulls != null) {
    while (i < length) {
      if (!nulls(i)) {
        out(i) = a(i) + b(i)
      }
      i += 1
    }
  } else {
    while (i < length) {
      out(i) = a(i) + b(i)
      i += 1
    }
  }
}
```

Notice that the function shown in Listing 3.1 accepts only integer parameters. Supporting all other combinations of the parameters would lead to the problem known as the combinatorial explosion. For high-level languages, it is natural to address the issue with generic functions. With generics, we can write one function that is used for multiple different parameters. For example, Listing 3.2 shows a vectorized primitive implemented with Scala generics.

Listing 3.2: Vectorized primitive implemented with Scala generic function

```scala
def addInts[T](a: Array[T], b: Array[T], out: Array[T], length:
    Int) (implicit num: Numeric[T]): Unit = {
  import num._
  var i = 0
  while (i < length) {
   out(i) = a(i) + b(i)
   i += 1
  }
}
```

Due to Java's language restrictions, primitive types cannot be used to instantiate generics. Instead, we have to use object classes which wrap primitive types. The conversion between primitive and the compatible object types are handled automatically by the Java compiler. This process is called autoboxing. It introduces a performance penalty for our vectorized primitives. Operations on elements require unboxing, actual operation, and then boxing again. Section 3.3.1 describes the test we did in order to measure the impact of autoboxing on vectorized primitives.

Another solution we applied to address the problem in vectorized primitives was to cast one of the parameters so that both parameters‘ data types match. This technique reduces the number of implementations of vectorized primitives. However, casting an array of primitive values comes with a cost of materialization. Data from input should be read, cast, and written to the output array. We observed that running two separate primitives (one casting, one doing the actual arithmetic) is slower than running one primitive that combines both operations and accepts mixed types. Because of this reason, we avoid explicit casting of vectors. Instead, we implement vectorized primitives that accept mixed types of parameters, and the Java runtime does the implicit casting (e.g., `1.0 + 1`) inside vectorized primitives.

Another solution is code generation. This is a similar technique to the one applied in data-centric code generation. However, instead of generating code for fused physical plans, we can generate vectorized primitives, compile, and dynamically load them in the JVM. For example, the “generator” class accepts three parameters for generating binary primitives: binary operation (e.g., plus, greater than, logical and) and two data types for the parameters. Once the generated vectorized primitive is compiled and loaded into the JVM, it can be reused for subsequent calls. The primitive still needs to go through the compilation tiers before it gets compiled and stored in the CodeCache.

In our project, we have not implemented the mentioned technique for a vectorized primitive generation due to the time limitations. We implemented four arithmetic and four predicate expressions that support three data types which already created code explosion problem on a small scale like this.

3.2 Contrast with a native vectorized engine

Writing performance-oriented code in a high-level language like Java comes at its cost. In Chapter 2.2, we discussed some of the reasons why Java may be a less compelling choice when it comes to applications with high throughput. One of these reasons is JVM warmup. This chapter explains some of the aspects of native vectorized engine and how it compares to the engine implemented in the JVM. Table 3.1 compares the JVM and native vectorized query engines from Spark‘s perspective.

Benefit	**JVM**	**Native**
Better control over execution code	–	✓
No GC pauses, no warmup	–	✓
Homogenous memory model	✓	–
Efficient Java/Scala UDF support	✓	–
Faster dev cycles	✓	–

Table 3.1: Feature comparison between JVM-based and native vectorized engines

Vectorized engines are operating on multiple values in a vector and can benefit from intra-CPU parallelisms, such as SIMD instructions. Complex code shape may prevent compilers from SIMDizing the code, and the developers have to fall back to manually writing SIMD intrinsics [5, 18]. Comparing native code and the JVM, the former supports both manual and automatic SIMD. The latter has limited support for the native API. JVM languages rely on automatic SIMD support, which is still brittle [1, 31, 32]. Project Panama[1] aims at providing SIMD intrinsics in Java to allow for manual SIMDization, but currently, it is in early development only.

A native engine in Spark has to operate on data that is stored in off-heap memory. Off-heap memory is a memory area that resides outside of the JVM heap space. On the one hand, it can operate on large amounts of data in memory without a performance penalty of garbage collection. On the other hand, this provides extra concern for Spark since the JVM cannot control off-heap memory. Combining extensive use of on-heap and off-heap memory in the same application creates challenges with memory management. The on-heap memory is managed by the JVM using a garbage collector, while the off-heap memory has to be meticulously freed or controlled by a separate memory management system. There is no easy way for these two systems to negotiate what amount of physical memory each of them is allowed to use at any given time, while statically allocating a fixed amount of memory to each of the systems will result in poor memory utilization.

3.2.1 UDF support

Calling user-defined functions (UDFs) is another performance consideration when it comes to a choice between using native code or the JVM. UDFs have been a critical extension point for Spark. Calling a Spark UDF, implemented in Java or Scala, bears less invocation overhead compared to calling a Python or R function. Invoking such functions requires data

[1]`https://openjdk.java.net/projects/panama/`

serialization from the JVM to the non-JVM environment, which has a high performance cost. Besides, running another environment side by side with the JVM means that these environments are competing for memory [7]. For the native engine, all existing UDFs are non-native, and even Java and Scala functions will have to pay this performance penalty.

3.2.2 Vectorized primitive implementation

Section 3.1 discussed the problem related to vectorized primitive implementation in the JVM. In the case of C++, this problem has a more straightforward solution. Here, we can use function templates to write a generic function that accepts different types of parameters, and also have the performance similar to a specialized function. There is a difference between C++ and Java templates. During the C++ compilation phase, if a template is instantiated with a specific argument, the template function is compiled to a type-specific function. While this language feature solves the code explosion in terms of code the programmer needs to write and maintain, it does not solve the problem of compiled code size and compilation time. Listing 3.3 is an example of a template class which allows parameter and operation substitution, as well.

Listing 3.3: An example of implementing and calling a C++ vectorized template

```
 1 template <class Op, class T, class S>
 2 class VecPrimitives {
 3   Op f{};
 4   public:
 5     void execute (T* a, S* b, T* out, int length) {
 6       for (int i=0;i<length;i++) {
 7           out[i] = f(a[i], b[i]);
 8       }
 9     }
10 };
11
12 long* a = new long[length];
13 int* b = new int[length];
14 long* out = new long[length];
15 VecPrimitives<std::plus<long>, long, int> add_long_int;
16 add_long_int.execute(a, b, out, length);
```

Line 7 in Listing 2.1.1 is a function call to execute an operation. The compiler is able to inline the call and produce machine code that directly executes the operation.

3.3 Microbenchmarks

We created microbenchmarks for operations used frequently in a vectorized system. We used the microbenchmark results to guide our design decisions while working on the vectorized engine in the JVM. The benchmarks simulate vectorized primitives: tight loops that iterate over arrays of primitives and applying an arithmetic operation on the array elements. The vector size is fixed to 1024 elements (even though our JVM-based vectorized engine in Spark uses 4096 elements). We use OpenJDK‘s benchmarking tool JMH, which provides a convenient interface to create test cases for benchmarking. Each test is run five times. Each run consists of five JVM warmup runs and five measured runs. Methods under test are run with one thread, and benchmark runs do not share the JVM environment. For these benchmarks, we used OpenJDK 8, unless otherwise stated. The benchmarks results indicate how many CPU clock cycles were spent to process a single tuple (cycle/tuple). Lower cycles per tuple indicate higher throughput. Table 3.2 shows the experimental setup for benchmarking environment.

Name	**CPU**	**Clock speed**	**Cores**	**L1,2,3 per core**	**Memory**
c5.2xlarge	Xeon Platinum 8124M, Skylake	3.4 GHz	18	32 KiB 1 MiB 1375 MiB	16 GB

Table 3.2: Hardware specifications of the machine running microbenchmarks

3.3.1 Vectorized primitives in Scala

Scala programming language seems to be a logical choice when it comes to implementing vectorized primitives in Spark. Existing query expressions are implemented in Scala, and introducing vectorized counterparts will not cause invasive architectural changes. We conducted experiments to prove that tight loops written in Scala have similar throughput as Java primitives. Scala code was written simplistically: a `while` loop iterating over arrays of primitives and applying an arithmetic function. Microbenchmark results verified our expectation that both Scala and Java vectorized primitives perform similarly. The benchmark simulated projection and aggregation expressions. We consider the difference between Scala and Java to be insignificant. Table 3.3 shows the results.

We investigated the assembly code generated by HotSpot‘s C2 compiler. In both versions, the code for projections‘ tight loop was similar. Except, Scala was compiled to larger machine code. Listing 3.4 shows the compiled loop. SIMD is used in both cases.

Benchmark	**Scala**	**Java**
Min aggregation with Math.min	2.0	2.0
Min aggregation with if-then condition	1.2	0.8
Sum aggregation	1.0	1.0
Add projection	0.6	0.5

Table 3.3: Java and Scala cycles-per-tuple performance on an integer array.

Listing 3.4: The fragment of JIT-compiled *Add* vectorized primitive

```
0x00007fc8ec635ff0:  vmovdqu    0x10(%rdx,%rdi,4),%xmm0
                     vpaddd     0x10(%rcx,%rdi,4),%xmm0,%xmm0
                     vmovdqu    %xmm0,0x10(%r8,%rdi,4)    ;*iastore
                     add        $0x4,%edi                 ;*iadd
                     cmp        %r10d,%edi
                     jl         0x00007fc8ec635ff0   ;*if_icmpge
```

3.3.2 Vectorized primitives as generic methods

Hardcoding vectorized primitive implementation with all possible combinations of parameter types leads to the code explosion problem discussed in Section 3.1. This JMH benchmark tests how generic vectorized primitives perform compared to the relevant type-specific implementation. Based on the results, generic primitives are more than 15x slower than the type-specific primitives. Furthermore, loops containing autoboxing operations do not get SIMDized. Based on these observations, we decided to implement type-specific primitives. Table 3.4 shows the experiment results.

Benchmark	**Type-specific**	**Generic**
Min aggregation	1.2	19.5
Sum aggregation	1.0	17.9
Add Projection	0.5	7.9

Table 3.4: Type-specific and generic function‘s cycles-per-tuple performance on an integer array

3.3.3 SIMD support in JVMs

Automatic SIMD support depends on JVM implementation and how well it can recognize SIMDizable code patterns. We decided to check which JVM implementation is suitable for a vectorized query engine. We tested OpenJDK 8 and 11, GraalVM Enterprise Edition, and Azul Zing. OpenJDK 8 is the default runtime for Spark. Zing uses LLVM-based JIT compiler called Falcon.

We checked JVMs‘ ability to SIMDize tight loops written in Java. Based on these results, Zing achieves the highest performance, followed by OpenJDK 11 and GraalVM[30], and OpenJDK 8. The Figure 3.1 shows the throughput of different JVMs on different operations. On the figure below, A+B denotes adding the array elements position-wise. SUM(A) and MIN(A) adds up array elements in a single variable and finds the smallest value element in the array, respectively. These operations were to simulate projection and aggregation expressions during query execution. These operations were simple enough to give us the idea of JVMs‘ ability to recognize and SIMDize tight loops.

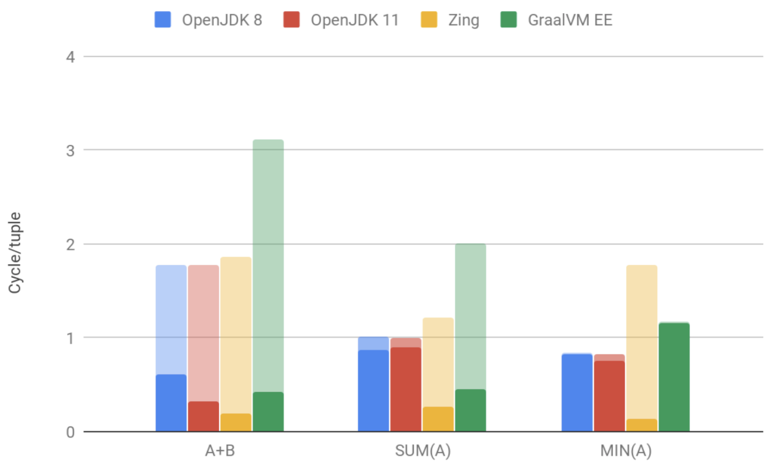


Figure 3.1: Vectorized primitives on different JVMs. Transparent bars depict the performance without SIMD

Looking into the JIT-compiled machine code reveals that Zing manages to use the widest, 512-bit SIMD registers for all three vectorized primitives. While OpenJDK 8 and 11 only manages to use 128- and 512-bit registers in A+B benchmark, respectively. GraalVM uses 512- and 256-bit SIMD registers for A+B and SUM(A), respectively.

We disabled SIMD support in the JVMs to see how much do SIMD commands contribute to tight loop performance. The transparent bars on Figure 3.1 show the performance of tight loops when SIMD support is disabled in the JVM. We observe that all of the JVMs perform poorly on the A+B benchmark without SIMD. On the MIN(A) benchmark, only Zing gets slower. The explanation is that on the A+B test, all of the JVMs were able to use SIMD, so by disabling it, every JVMs‘ performance got slower. On the MIN(A) only Zing was able to SIMDize the vectorized primitive. Therefore only Zing‘s performance was affected by disabling SIMD support.

Figure 3.2 shows the effect of SIMD on tight loop performance. The experiment was done on Zing, which allows for more granular control of the active SIMD instruction set. We ran the A+B benchmark with five different SIMD configurations: SIMD disabled, only `xmm`, `ymm` or `zmm` SIMD registers enabled. One may argue that this experiment does not prove that SIMD instructions can also benefit query execution performance in a JVM-based vectorized engine where we have virtual function calls. We come back to this topic in Section 5.4.

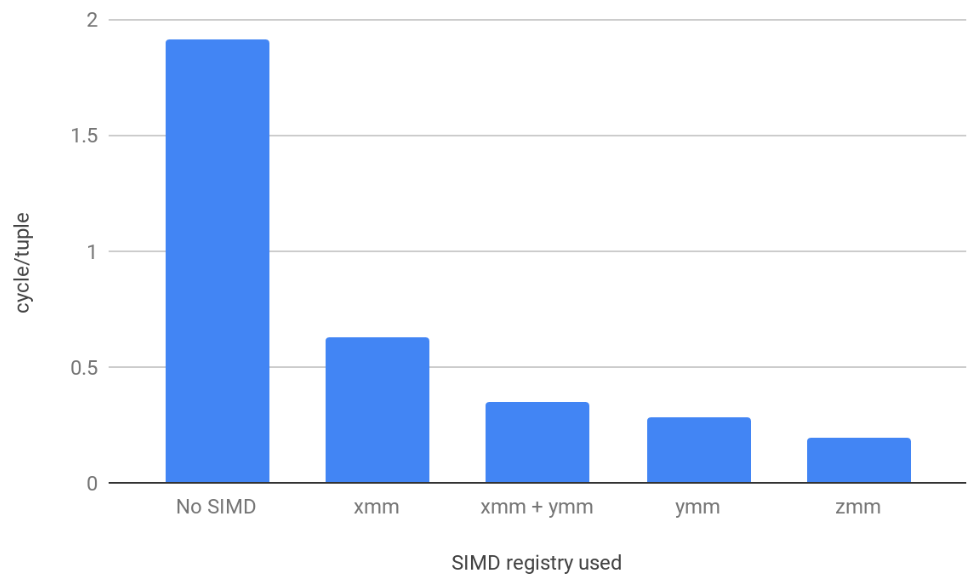


Figure 3.2: How JVM-based vectorized engine running on Zing performs A+B operation on integer array of 1024 elements based on SIMD registry width

3.3.4 Control and data dependency

Pipelines in modern CPUs can run multiple instructions in parallel. However, there may be data or control dependencies between the instructions, that harm pipeline‘s instruction throughput. The mentioned issues are caused by how pipelines function, and they are not

JVM specific. This topic is discussed in Section 2.1.2. Listing 3.5 displays a loop from our project, and it has a control dependency.

Listing 3.5: Loop with control dependency

```
while (i < a.length) {
  if (a(i) < 0) {
   miss(counter) = i
   counter += 1
  }
  i += 1
}
```

The tight loop contains an expensive branch inside, which hurts its throughput. The code in Listing 3.6 yields a similar result to the code above. However, here, we have data dependency instead of control dependency. The value of the counter variable determines the memory location of the next update in array miss.

Listing 3.6: Loop with data dependency

```
var i = 0
while (i < a.length) {
  miss(counter) = i
  counter += a(i) >> 31 & 1
  i += 1
}
```

In Java, as opposed to C++, boolean values cannot be automatically interpreted as integers (with 0 or 1 value) by the compiler. Therefore we need to apply a workaround using bitwise operations to have a numerical representation of a predicate as shown in Listing 3.6. In this case, we may be issuing more write commands, but we are avoiding the expensive branch in the loop [16].

Figure 3.3 shows the cycle-per-tuple performance of the tight loops shown in Listings 3.5 and 3.6. The line for control dependency displayed on the figure does not conform to the same pattern, as shown by other researches [4, 16]. In the case of OpenJDK 8, we see that both versions of tight loops produce a flat line. For the code snippet in Listing 3.5, there are fewer branch mispredictions for very low or very high selectivity, and this fact should result in faster performance in the mentioned cases.

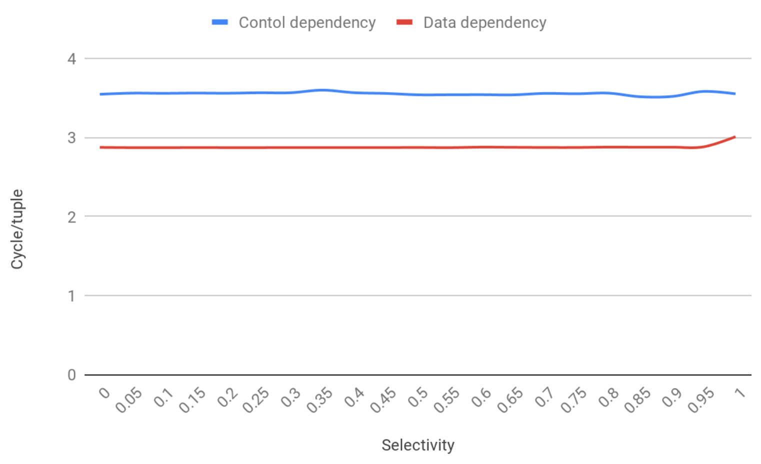


Figure 3.3: How selectivity influences tight loop performance with control and data dependency

Chapter 4

The implementation of a JVM-based vectorized query engine in Spark

One of the main objectives of this research is to implement vectorized execution in Spark. In this chapter, we will discuss how vectorized physical operators, projection, selection, and aggregation were implemented. Section 4.2 discusses a vectorized projection. Section 4.3 explores a vectorized selection and its implementation techniques. Finally, Section 4.4 describes a vectorized operator for grouping and global aggregations. Furthermore, we present a vectorized algorithm for building a hash table used by a grouping aggregation operator.

4.1 Overview

Spark‘s present query engine, whole-stage code generation (WSCG) is based on a data-centric code generation model. This model, by nature, operates on a single row at a time. The vectorized engine requires batch execution, which operates on several thousand data points at a time. A scan operator in Spark is already producing data in columnar format. Therefore, we only adapted Projection, Selection and Aggregation to make them compatible with batch processing.

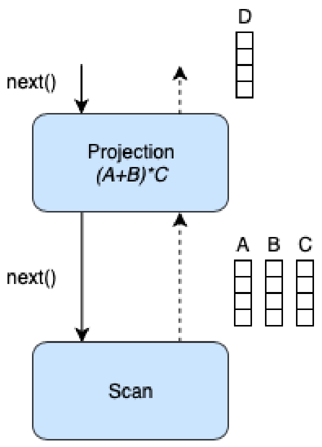
4.2 Vectorized projection

A projection operator takes a set of input columns and produces new columns by applying an operation on them. Listing 4.1 shows a SQL query that is translated to a query plan with a projection operator, which produces a new column `D` by adding columns `A` and `B` and multiplying the result by column `C`.

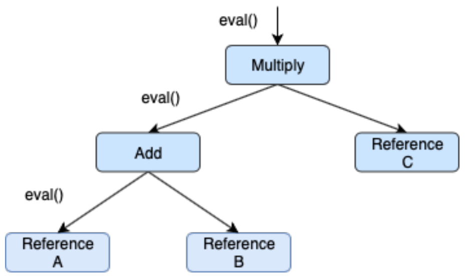
Listing 4.1: An example of SQL query that produces a query plan shown on Figure 4.1

```
select (A+B)*C as D from table
```

Figure 4.1 shows a vectorized projection operator and the expression tree. The projection operator pulls batches from the scan operator and evaluates the expression tree. Leaf nodes in the tree return appropriate vectors from multiple vectors provided by the scan operator. *Add* or *Multiply* node receives inputs from their children and applies the operation on them, producing the output. Here, the result of the topmost node is also the result of the operator.



(a) Vectorized query operators passing vectors



(b) Vectorized query expression tree for (A+B)*C

Figure 4.1: Vectorized projection and the expression tree called from the projection operator

4.2.1 Vectorized query expression

Listing 4.2 shows how the *Add* expression's `eval()` function is represented in our JVM-based vectorized engine. The `eval()` function is polymorphic, and its return type is of type `Any`, which is a superclass of all data types in Scala. Therefore, we have to check the result types returned by the left and right child nodes and cast them accordingly before calling a type-specific vectorized primitive (Listing 4.3).

The class `ColumnarBatch` is a container class for vectors. It also stores a vector size (`batchSize`) and the output vector (`out`) that holds a vectorized primitive result. The `batchSize` variable controls the loop bounds in vectorized primitives. Even though vectors are represented as fixed-size arrays, the actual data the array holds may be less than the array size. This may happen when an operator returns an incomplete batch (e.g., Scan, Select, HashAggregation).

Listing 4.2: The `eval()` function from the expression tree‘s *Add* node, which calls two vectorized primitives

```
def eval(batch: ColumnarBatch): Any = {
 val resL = left.eval(batch)
 val resR = right.eval(batch)
 if (resL.isInstanceOf[Array[Int]] && resR.isInstanceOf[Array[Int
     ]]) {
   VecPrimitives.add(resL.asInstanceOf[Array[Int]],
     resR.asInstanceOf[Array[Int]],
     batch.out.asInstanceOf[Array[Int]],  /*stores result*/
     batch.batchSize)                     /*loop boundary*/
 } else if (resL.isInstanceOf[Array[Int]] && resR.isInstanceOf[
     Array[Long]]) {
   VecPrimitives.add(resL.asInstanceOf[Array[Int]],
     resR.asInstanceOf[Array[Long]],
     batch.out.asInstanceOf[Array[Long]],
     batch.batchSize)
 }
}
```

Listing 4.3: A vectorized primitive adding two integer arrays

```
class VecPrimitives {
  def add(a: Array[Int], b: Array[Int], out: Array[Int], len: Int
      ): Unit = {
    var i = 0
    while (i < len) {
      out(i) = a(i) + b(i)
      i += 1
    }
  }
  ...
}
```

4.3 Vectorized selection

A selection operator, also called a filter operator, is responsible for filtering out the query results that do no meet the predicate requirements. Listing 4.4 shows an example of a simple SQL query with selection and projection.

Listing 4.4: An example of SQL query that produces a query plan shown on Figure 4.2

```
select A/B from table where B > 0
```

Implementing a selection operator in iterator model is more straightforward than in the vectorized model. A tuple-at-a-time selection operator checks a predicate against every tuple pulled from a child operator (e.g., Scan). If the tuple passes the filter, it is passed to the next operator (e.g., Projection) in an operator tree. Otherwise, the tuple is discarded, and the next tuple is pulled from the child operator.

4.3.1 Representation of filtered data

A vectorized selection operator evaluates a predicate against a number of input vectors, producing information about vector elements that qualify the filter. This information can be represented as a boolean vector or a selection vector. Another way to handle filtered data is to compact every vector inside a batch by removing the filtered elements from the original input vectors.

Figure 4.2 shows the query plan and how the vectors are passed to the upstream operators. The operator produces a boolean vector as a result. Below we show how these methods compare to each other.

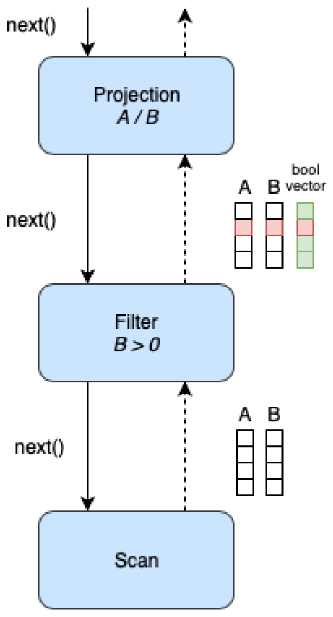


Figure 4.2: Vectorized selection operator producing boolean vector

4.3.1.1 Compaction

During vector compaction, only the elements that qualify are adjacently placed in a new array. Compaction seems to be a compelling solution because the subsequent expressions use the same vectorized primitives for the compacted or uncompacted data. This method has several drawbacks. First, low filter selectivity causes high materialization cost. More elements need to be copied in a new resulting vector. This process happens for every input column, and the cost is even higher if we are copying wide data types, such as Longs and Doubles. Second, if the filter selectivity is too high, the compacted vector tends to be too short to benefit from SIMD or to amortize function call overhead. The latter can be compensated by buffering qualifying elements unless their number reaches a certain threshold, and then returning them all together to the upstream operator.

4.3.1.2 Boolean vector or bitmap

A boolean vector marks the vector elements that passed the filter. A boolean vector can be represented by a bitmap, which reduces memory footprint. A boolean vector with 4096 elements takes 4 KB memory, while the bitmap needs 512 bytes to store the same information. This space efficiency is available if 64 boolean values (that take 64 bytes) are encoded in 64 bits or eight bytes. The bitmap can be used for the masked AVX-512 instructions for compaction which tend to be faster than the version with the explicit if-then branch. Furthermore, a sparse bitmap allows for optimization by checking 64-bit chunks at once and skipping the chunks that are 0 [16].

4.3.1.3 Selection vector

A selection vector uses an array to store the indices of active rows. Its memory footprint is proportional to the number of active rows. Each element in the vector may be represented with a 16-bit data type. For 4095 active rows, when only one row is inactive in 4096 size vector, the selection vector occupies 8190 bytes (about 8 KB) memory. If only one element is active, it takes only 2 bytes (vs. 512 bytes in a bitmap). Here, we have to make a compromise between data width to store active row indices and vector length. With 16-bit data types, we can index at most 65 536 vector elements, so we are constrained with vector length. A selection vector causes more memory pressure in the worst case. However, the number of active rows is always known because it is the size of the selection vector. Whereas, the boolean vector needs explicit iteration to count active rows.

In the case of the JVM, all these considerations hold, but there are several differences with a native engine. JDK 8 does not support manual SIMD, which is needed to use AVX-512‘s `vpcompressd` instruction. This instruction is used to copy values from one SIMD register to another based on a boolean mask and pack the values densely. The mentioned SIMD command can be used to optimize the compaction process described in Section 4.3.1.1. Another difference is that a boolean type cannot be used in an arithmetic operation. This constraint forces us to use branching in cases when it is avoided in native code. For example, counting the number of active rows in the boolean array requires an explicit value check for the elements in the array.

We implemented a compaction method with explicit if-then branching. This decision was made due to the constraint in time.

4.4 Vectorized aggregation

Aggregation operator is responsible for grouping the data based on one or many columns and applying a function to the elements in a group. If the number of groups is one, then it is a global aggregation. Listing 4.5 shows an example of grouping aggregation. The tuples are grouped by column `C`. Each group sums column `A` and finds a minimum value of column `B`.

Listing 4.5: An example of SQL query for the grouping aggregation

```
select sum(A), min(B) from table group by C
```

We will now discuss the implementation of the grouping aggregation operator using a bucket-chained hash table since it is relatively fast and easy to grow. Other alternatives, not discussed, could use a linear probing hash table, or use a B-tree index, or even be based on the data sorting. The hash aggregation operator receives data from downstream operators and inserts it in a hash table using grouping attributes as hash table keys, and storing partial aggregates in values. In the example above, column `C` acts as a key in the hash table, and the results of `sum(A)` and `min(B)` are values. An implementation of the aggregation operator that follows the iterator query model pulls a tuple from a child operator and inserts it in the hash table one by one. Listing 4.6 is an example of constructing a hash table for a grouping aggregation.

Listing 4.6: Tuple-at-a-time grouping aggregation

```
keys = getKeys(tuple);
bucketIndex = hash(keys);
index = bucket[bucketIndex];
while (index != -1) {
  if (keyCols[index].equals(keys))
        break;
  index = next[index];
}
```

```
if (index == -1) {
  index = insert(keys);
  bucket[bucketIndex] = index;
}
foreach aggregate
  updateState(aggregate, index, tuple);
```

Several steps are required to place the incoming key-values in the correct place in the hash table. First, keys are hashed, and the hash value points to the `bucket` array, which holds a reference to the key-value space (the memory area holding keys and aggregation states). Bucket value -1 indicates that the hash table does not contain a key hashing to the same hash value, and it is safe to append the incoming tuple at the end of the hash table. The bucket value greater than -1 indicates the position in the key-value space where the already-existing record should be updated. If the keys at the destination match the keys from the incoming tuple, the aggregation states (the hash table values) are updated. Otherwise, we have to follow the chain by consulting the array `next` that points to the next location in the hash table. The key collision is checked after every step in the chain. Aggregation state is updated if there is no collision anymore. Otherwise, we reach the end of the chain, and we insert the element at the end of the table. This process of inserting the data in a table is called *bucket chaining.*

In Listing 4.6, we can see that control logic and data processing is mixed. This approach takes one tuple through different processing stages. It is hard for the software prefetcher to guess the memory access pattern and make the data access more efficient (e.g., prefetching). We can help the CPU‘s instruction prefetcher and make the loops simpler with a vectorized approach by separating the processing of elements that are in different stages of hash table insertion. Loops that access memory becomes simpler, which allow outstanding cache misses. Section 2.1.4 discusses this parallel memory access in more details.

Figure 4.3 shows the layout of the vectorized hash table implemented in this project. The idea was introduced in [16] and later explored in more details by [15]. Our JVM-based vectorized hash table uses a vector-wise layout for key-value storage. This model is known as *Decomposed Storage Model* (DSM). Here, keys and values are stored in separate arrays. That is, keys from different groups are stored next to each other. An alternative model to DSM is *N-ary Storage Model* (NSM). In NSM, all keys and aggregation values from the same group are stored adjacent to each other in memory. Experiments [16] show that performance-wise NSM is a better model when the data is accessed randomly, and the hash table records reside outside of the L1 cache, since it provides better cache locality.

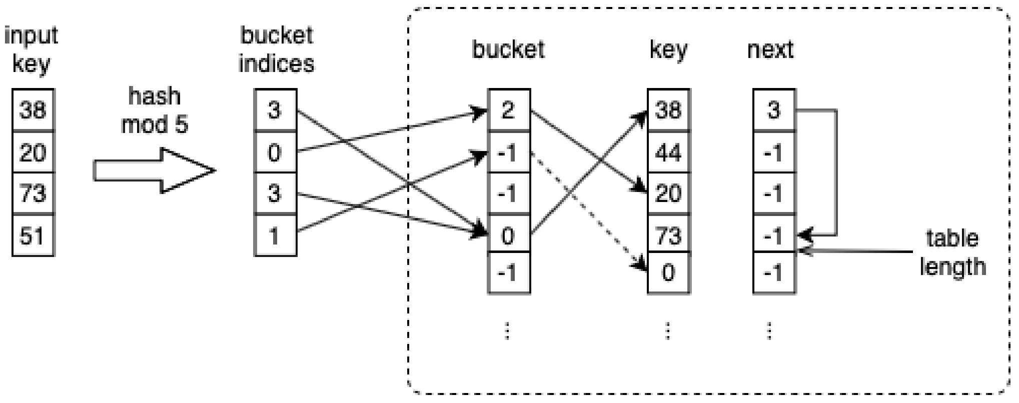


Figure 4.3: Bucket-chained hash table in DSM format. Value vectors are not displayed

Otherwise, the DSM model is more optimal performance-wise. DSM‘s faster memory access code can outperform the NSM model as long as the hash table data fits in the L1 cache. As future work, switching key-value storage model from DSM to NSM can make our JVM-based vectorized aggregation even faster compared to the WSCG‘s tuple-at-a-time hash map.

The vectorized hash table‘s construction stages are similar to the ones described above. However, instead of a single tuple, we have batches of tuples transitioning through the stages. The stages (states) are shown in a state diagram on Figure 4.4. Every stage accepts an array and produces two arrays (arrays are not created but reused to avoid expensive memory allocation). Step 1 is to hash incoming keys and obtain bucket indices. Step 2 is to lookup these indices in the bucket array. If the bucket value is -1, the tuple is stored for an append operation. Otherwise, the tuple is stored for an update operation. On step 3, tuples are split into two groups. Those who transition to a state new are all inserted at the end of the table. Tuple 51 on Figure 4.3 is inserted at the end of the `key` array. It is shown with a dashed arrow. Indices that are in the state *check* are examined for conflicts. If input tuples match the keys in the table, they transition to a state *match*, and they “wait” for the other, conflicting tuples to get resolved. Conflicting tuples go to the state *miss* and to the state *next* where they follow the chain by consulting the array `next`. On Figure 4.3, key 38 has a chain that continues on index 3 in the table.

A hash table with a tuple-at-a-time approach takes a single element through every state. In a vectorized hash table, on the other hand, every vector elements progress through the states on each step. This model allows a vectorized insertion algorithm to operate in tight loops and benefit from parallel cache resolution.

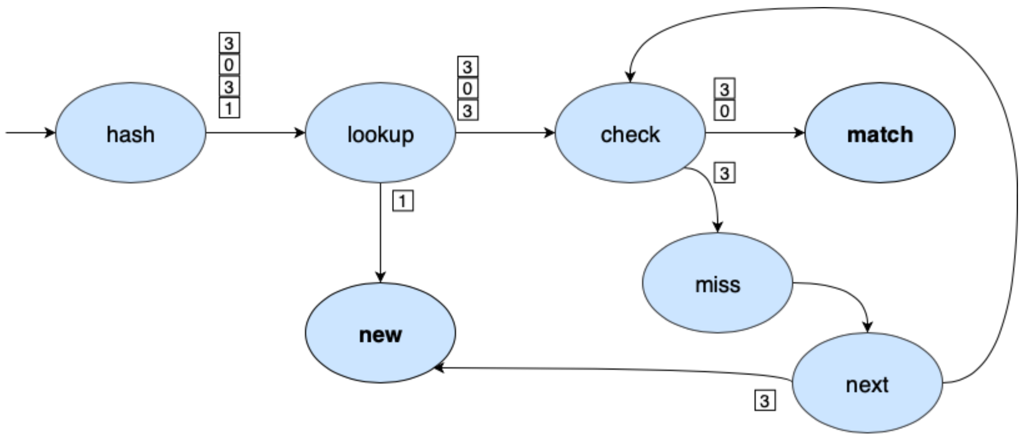


Figure 4.4: State machine representing the stages vector elements travel through. The vector displayed is the same as `bucketIndices` from Figure 4.3

The iteration over this state machine will stop when all input tuples end up in the state *new* or the state *match* (terminal states). Next, we update the aggregation states, which are not shown in the figures. An update to the aggregation state requires random memory accesses which becomes more expensive as the hash table size increases (Listing 4.8). Random memory access is also required in the states *next* and *check*, which are executed more frequently as the collision chain size in the hash table increases. Therefore, longer chains in a hash table have a detrimental effect on the performance. Still, a vectorized hash table can benefit from independent memory access and parallel cache resolution, which is backed by the experimental results presented in Section 5.2. Listing 4.7 shows pseudocode for building a vectorized hash table.

Listing 4.7: Vectorized grouping aggregation

```
keys = getKeys(tupleVec);
bucketIndices = hash(keys);
next = int[n];
matches, misses, conflict = int[n];
splitMatchMiss(misses, matches, bucketIndices); // state "lookup"
while (misses.length > 0) {
  insert(keys, misses); // "new"
  splitMatchConflict(keys, matches, conflicts); // "check"
  updateIndirection(misses, matches, indirection); // "check"
  followChain(misses, matches, conflicts, next); // "next"
}
foreach aggregate
  updateState(aggregate, indirection, tupleVec);
```

The method `splitMatchMiss` checks the bucket array values on indices provided by the `bucketIndices` array and splits the indices in the `misses` or `matches` arrays. The former holds the indices of the incoming keys that are inserted at the end of the hash table. The latter holds the indices of the incoming keys that already exist in the table (or they conflict with the existing keys). New keys are inserted with the function `insert`. The matches are further examined for conflicts using the function `splitMatchConflict`. The incoming keys that matched other keys in the table are stored in the `indirection` array, which is a mapping between the incoming values and the aggregation states. The conflicting keys have to follow the chain. On the next iteration, the array `misses` contains the indices of the original incoming keys that reached the end of the chain and can be inserted in the table while the array `matches` contain the indices of the keys that need to be checked for conflict.

The pseudocode in Listing 4.8 shows how the function `splitMatchMiss` is implemented. The `bucket` array is accessed in random order, which is guided by the values from `bucketIndices` array. Accesses to the `bucket` array are independent of each other. Independent memory accesses can benefit from the parallel cache resolution, which is hard to achieve in the tuple-at-a-time hash table model.

Listing 4.8: The `splitMatchMiss` function with independent random access in the `bucket` array

```
def splitMatchMiss(misses, matches, bucketIndices) {
  while (i < bucketIndices.length) {
    index = bucketIndices[i]
    if (bucket[index] == -1)
      misses[m1++] = i
    else
      matches[m2++] = i
    i++
  }
}
```

Chapter 5

Results and discussion

This chapter explains the experiments we conducted to assess the performance of our vectorized JVM-based query engine prototype. We ran the modified TPC-H query 1 and 6 and observed the overall query execution time, as well as, operator and vectorized primitive execution times.

5.1 Experimental setup

Our implementation of a vectorized query engine leverages Spark‘s existing query evaluation infrastructure. Spark builds similar logical plans and expression trees for the JVM-based vectorized engine, WSCG, and the experimental native vectorized engine. The same optimizations are applied to the plans in all three cases. The scope of the comparison is only limited to the performance of the physical operators: Selection, Projection, and HashAggregation. The vector size in our JVM-based vectorized engine was fixed to 4096.

Our JVM-based vectorized engine supports a limited number of data types and expressions. Therefore, we had to modify the original TPC-H query 1 and 6 to make them run on our engine. In the following sections, we will refer to the modified queries as TPC-H Q1* and Q6*. For the same reason, we transformed the synthetic data that was used in experiments. These factors are described below.

5.1.1 Synthetic data

Tests were done on TPC-H data with scale factor (SF) 100. The queries were executed against the lineitem table. The table was represented as a Parquet file, which was fully memory-resident. These conditions hold for all benchmarks unless otherwise stated. Table 5.1 shows the original and the modified data types in the `lineitem` table.

Column	Original type	Modified type
l_orderkey	long	long
l_partkey	long	long
l_suppkey	long	long
l_linenumber	integer	integer
l_quantity	decimal(12, 2)	integer
l_extendedprice	decimal(12, 2)	integer
l_discount	decimal(12, 2)	integer
l_tax	decimal(12, 2)	integer
l_returnflag	string	integer
l_linestatus	string	integer

Table 5.1: The `lineitem` table before and after attribute types were modified

Data type `decimal (12, 2)` in SQL standard is a fixed point data type which, in this case, is converted to integer type by multiplying the original value by 100 and casting the result into an integer. The fields `l_returnflag` and `l_linestatus` originally held numerical values represented as strings. We converted the field types to integers.

5.1.2 Test query

Listing 5.1 shows Q1* and Q6* which were modified for reasons of limited functionality in the various systems. The lines highlighted in red and green were removed and added, respectively. Q1* is a projection followed by a grouping aggregation with two keys and four SUM aggregation functions. Q6* is a global aggregation preceded by a projection and a selection operator.

Listing 5.1: The modified TPC-H query 1 and 6. The red lines were removed from and the green lines added to the original queries

```
# TPC-H Q1
select
  l_returnflag, l_linestatus,
  sum(l_quantity) as sum_qty,
  sum(l_extendedprice) as sum_base_price,
  sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
  sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as
     sum_charge,
  avg(l_quantity)} as avg_qty,
  avg(l_extendedprice) as avg_price,
  avg(l_discount) as avg_disc,
  count(*) as count_order
from
  lineitem
where  l_shipdate <= date '1998-12-01' - interval '90' day
group by
  l_returnflag,
  l_linestatus
order by  l_returnflag,  l_linestatus

# TPC-H Q6
select
  sum(l_extendedprice * l_discount) as revenue
from
  lineitem
where
  l_shipdate >= date '1994-01-01'
  and l_shipdate < date '1994-01-01' + interval '1' year
  and l_discount between .06 - .01 and .06 + .01
  and l_discount between 5 and 7
  and l_quantity < 24
  and l_quantity < 2400
```

5.1.3 Hardware

EC2 virtual machine was used for the experiments. The microbenchmarks were running on `c5.2xlarge` instance. For TPC-H tests, we had to switch to `c5.metal` instance, which allowed us to profile kernel events and read hardware counters. For this purpose, a program called *perf*[1] was used. Table 5.2 summarizes the machine specifications.

Name	CPU	Clock speed	Cores	L1,2,3 per core	Memory
c5.metal	Xeon Platinum 8275CL, Cascade lake	1.2 - 3.6 GHz	48	32 KiB 1 MiB 1375 MiB	192 GB

Table 5.2: Hardware specifications of the machine running benchmarks

[1]`https://perf.wiki.kernel.org/index.php/Main_Page`

5.2 TPC-H Q1*

Q1* is a grouping aggregation with four groups that is preceded by a projection operator. The scan operator reads five integer columns, 600 million (SF100) rows, in total 12 GB of data. Figure 5.1 shows the performance comparison between WSCG, our JVM-based vectorized engine, and the native vectorized engine. Our engine is running on OpenJDK 8 and Zing. As explained before, the scan operator reads data from memory. This fact allows all three engines to read the entire data in less than half a second. Therefore, the time spent on reading the data is negligible compared to the overall running time.

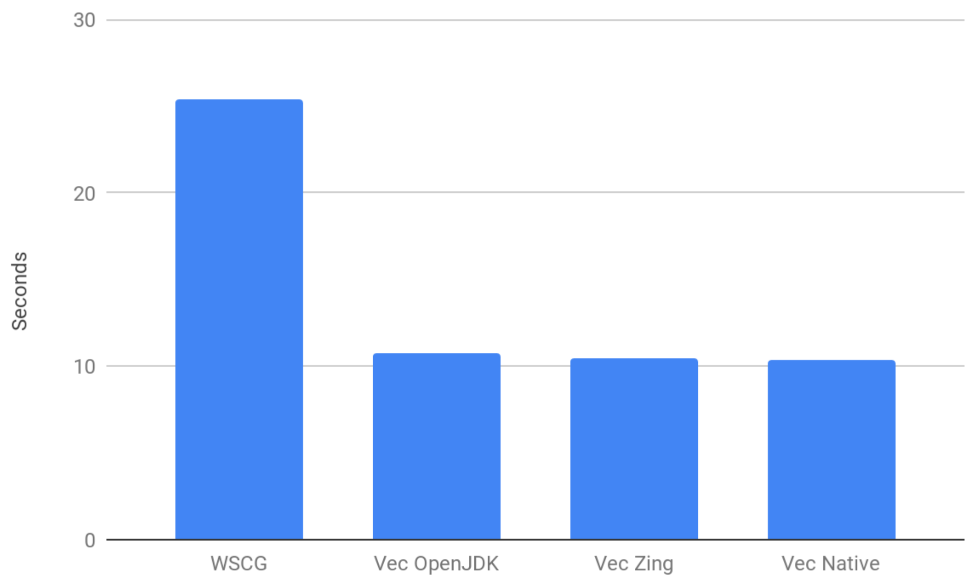


Figure 5.1: TPC-H Q1* performance of different engines on SF100

Our JVM-based vectorized query engine achieves more than 2.5x speedup. We assume that we are benefiting from the vectorized hash table. Even though it has more cache misses, it can resolve them in parallel. Parallel cache resolution is possible because vectorized processing has tight loops with independent data access. The parallel resolution, also known as Memory Level Parallelism, is discussed in Section 2.1.4.

A varying number of groups can better emphasize the effect of parallel cache resolution in grouping aggregation. Figure 5.2 shows the dependency between the number of groups and query execution time. For the experiment, we executed grouping aggregation with three integer keys and one sum aggregation operation. Our execution plan included projection with grouping aggregation. The number of groups was adjusted using bit-shifting one of the aggregation keys. Bit shift operation is cheap enough to consider its effect negligible for this experiment.

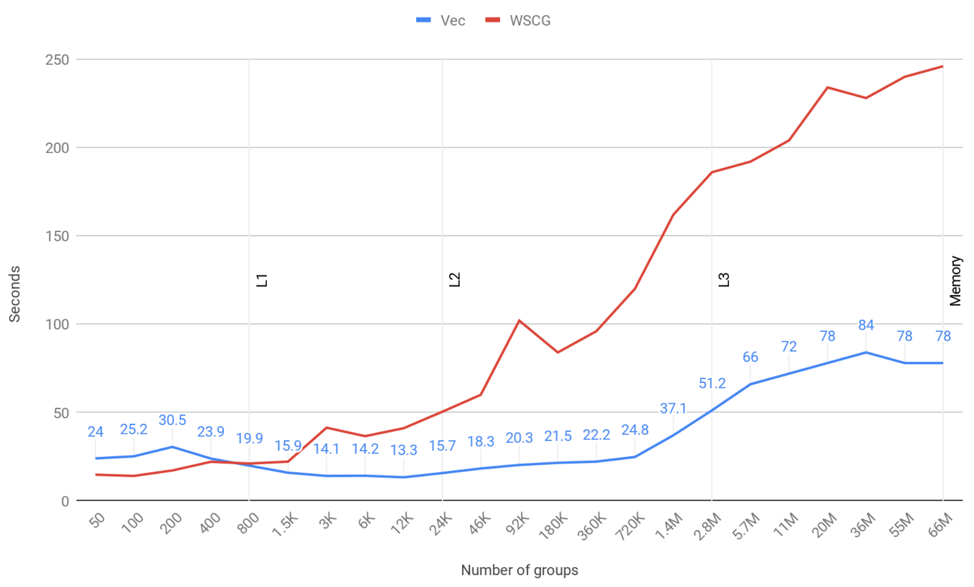


Figure 5.2: Comparison of our JVM-based vectorized and WSCG‘s tuple-at-a-time grouping aggregation with three keys and one sum aggregate

Figure 5.2 reveals that for 66 million groups, vectorized aggregation achieves almost 3x speedup compared to the tuple-at-a-time aggregation implemented in WSCG. Before every test run, we manually set bucket and key-value space sizes for the vectorized hash table. The number of buckets was chosen to be twice as large as the number of groups to achieve 40-50% fill factor. For WSCG hash table we could only control the number of rows. The number of rows was chosen based on the number of groups so that the table would fit all groups without resizing. Same for the vectorized table, the key-value space had the same size as there was a number of groups. From the figure, we see that the vectorized hash table is slower than WSCG‘s table for groups below 800. This is because our hash function produced multiple collisions for small group numbers that resulted in longer chains (average chain length 1.6 or higher). Chain resolution requires random memory accesses in tight loops. For aggregations with groups above 800 average chain length was 1.1-1.3.

The number of buckets in the hash table was controlled manually to make a fair comparison between WSCG‘s table and the vectorized hash table. There are two hash tables in WSCG collaborating for grouping aggregation: one that has a fixed row size (2^{17} by default) but does not resize (“fast” map) and the other that is used if there is no space left in the first one, but it can grow if needed (BytesToBytes map). By controlling the maximum number of rows in the fast map, we managed to fit the aggregation result within one hash table, and there was no need to fall back to BytesToBytes map.

The fast map was designed as an optimization layer for grouping aggregation. It is an append-only map and does not accept the data with null elements. The fast map limits the size of the table and focuses on reducing CPU overhead by keeping the records in the L2 cache. From Figure 5.2, it is visible that for group sizes below 46K WSCG is factor 2 slower than our JVM-based vectorized map.

It is an open question whether the fast map is the best WSCG can do. There are indications that further optimizations to it could yield a factor 1.5-2 improvement [33]. This would still place it somewhat behind our vectorized implementation. Our vectorized implementation could still be optimized somewhat by switching to a linear hash table rather than a bucket-chained one. This will reduce the amount of cache misses with a factor 1.6-1.3 [16].

Figure 5.2 marks CPU cache boundaries for the vectorized hash table. For example, for 2.8 million or more groups the map does not entirely fit in the L3 cache (∼63 MB), and more frequent memory accesses are issued to read or write the data in the hash table. Hardware counters from the CPU may explain the performance numbers.

Using the Linux tool called perf, we analyzed hardware counters to understand the reason behind the query execution performance difference. Table 5.3 shows the number of events captured by the perf tool for hash tables with 5.7 million groups.

Metric	**# of events in billions**	
	JVM vectorized engine	**WSCG**
# of cycles	80.50	254.69
# of instructions (IPC)	128.68 (1.60)	206.40 (0.81)
Bus-cycles (bus-cycle/cache miss)	1.67 (2.0)	5.30 (3.7)
L1-dcache-loads (misses)	34.92 (13.6%)	77.42 (5.9%)
LLC-loads (misses)	2.12 (40%)	2.43 (58%)

Table 5.3: Hardware counters collected after the query was executed

The first row in the table shows the total number of cycles spent to execute the query. The vectorized code executes the query three times faster, which explains why the vectorized engine spends three times fewer instructions. Using the number of cycles and the number of issued instructions, we can calculate instructions per cycle (IPC). High IPC can indicate better hardware utilization, less wasted CPU work. However, IPC alone cannot be used to explain the difference in query execution time between the hash tables. As this paper [1] points out, a query can have high IPC but still be slower in overall running time than the query that has lower IPC. We have to look at other metrics to better understand what causes the performance difference. `L1-dcache-loads` and `LLC-loads` (LLC stands for Last Level Cache, which is the L3 cache in our case) metric indicate the count of issued L1 and L3 cache read instructions. The value of `L1-dcache-loads` indicates that WSCG‘s hash table uses the L1 cache more efficiently, accessing it twice frequently and having the same number of cache misses as the vectorized hash table. The `LLC-loads` value suggests that both tables access the L3 cache with the same frequency, but the vectorized hash needs to fetch data from memory less often.

Dividing bus-cycles by the `LLC-loads` metric yields number that shows how many CPU cycles were spent on average every time when the instruction missed the L3 cache and had to access the memory. Vectorized hash table manages to spend twice fewer cycles waiting for memory fetch than WSCG‘s hash table. The data access pattern can explain this difference in the tight loops: sequential and independent array accesses. CPU can resolve memory access requests in parallel without queuing and executing them one after another.

Based on Figure 5.2 and Table 5.3 we can consider that WSCG‘s hash table has to fetch data directly from the main memory rather than cache to main memory more often while the vectorized hash table‘s data access patterns can better use L2 and L3 caches. The access patterns also allow the vectorized hash table to resolve cache misses in parallel and reduce the number of cycles spent on waiting for the cache resolution.

5.3 TPC-H Q6*

The query plan for Q6* consists of a selection (Filter) operator that is followed by a projection operator and global aggregation with a SUM aggregation function. Filter selectivity is 12%; in other words, only the minority of the rows made it past the filter. Our implementation details for the selection operator is presented in Section 4.3.

From Figure 5.3 it is visible that our vectorized engine achieves 1.42x speedup compared to WSCG running on OpenJDK 8. Exploring WSCG‘s generated code reveals that it contains multiple conditional branches that serve as null checks and short-circuit evaluation for the predicate condition `l_discount > 5 and l_discount < 7 and l_quantity < 2400`. WSCG may impose the penalty of a branch misprediction. Our vectorized filter

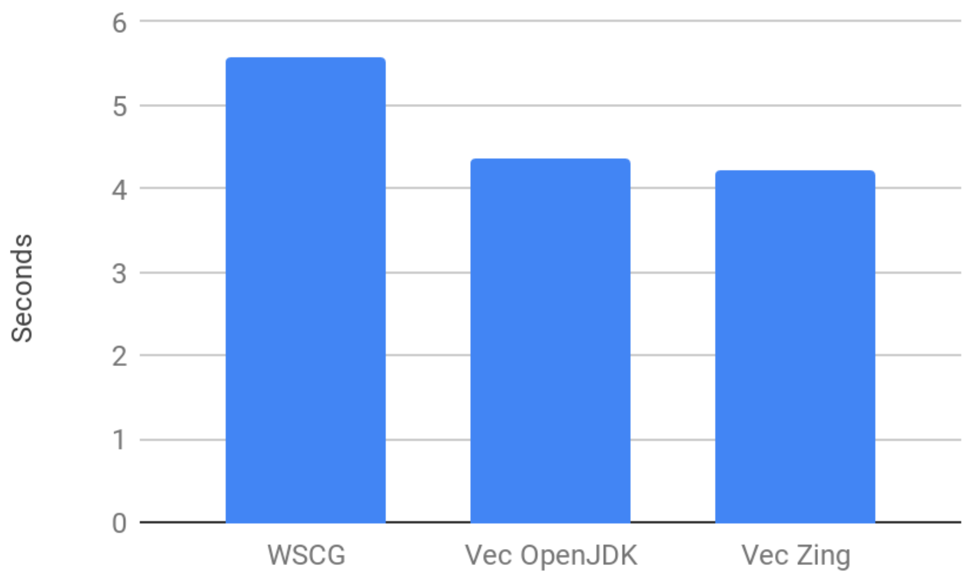


Figure 5.3: TPC-H Q6* performance on different engines on SF100

operator evaluates the predicates independently from each other, and, in the case of Q6*, we use logical AND operation to produce the final result of the predicate expressions. Using this technique, we transformed a control dependency to a data dependency in our tight loops. More information on selection operator strategies is presented in Section 4.3.

5.4 The SIMD effect

The following experiment demonstrates the performance of our engine using only a projection operation. The query performs seven multiplication operations on a single column (`select A*A*...*A from table`). The purpose of the query is to provide an operation that is relatively expensive and is easy to SIMDize. This query helps us see the contrast between the executions with or without SIMD instructions.

Figure 5.4 shows the performance of the JVM and native vectorized engines for the query mentioned above. The dataset of SF100 was read from memory. The first two columns show OpenJDK‘s performance with the JVM flag UseSuperword enabled and disabled respectively. OpenJDK does not allow granular SIMD instruction set control as Zing does. OpenJDK with SIMD enabled only managed to use `xmm` registers for a multiplication operation. From the figure, we see that Zing with only `xmm` registers performs faster than OpenJDK and native vectorized engine. We verified that the native engine also uses `zmm`

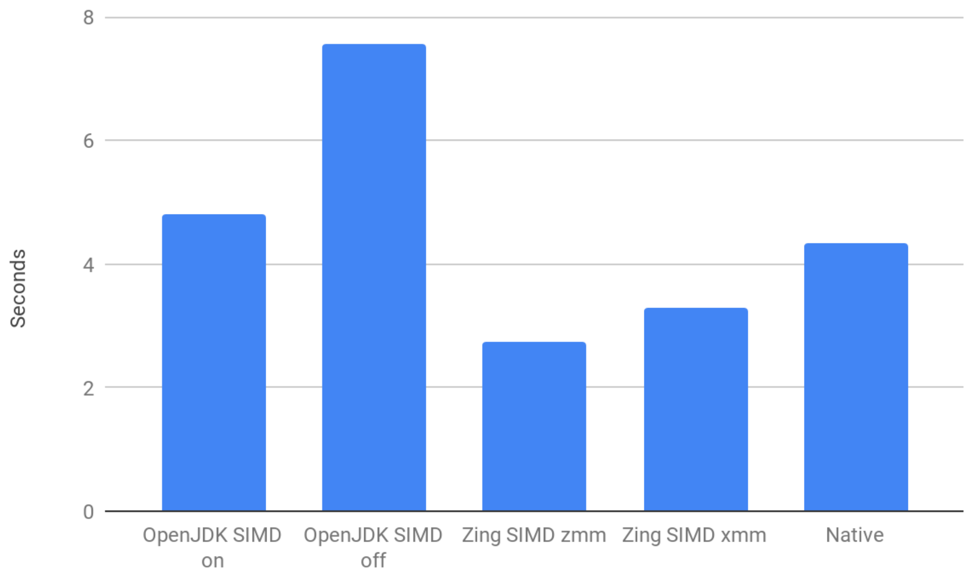


Figure 5.4: SIMD effect on Projection. The native engine uses `zmm` registers.

registers.

This experiment does not simulate real-world query, but it shows the difference between an execution with and without SIMD commands. Vectorized execution can benefit from SIMD even with the virtual function calls involved in the operator and expression tree iteration. This experiment also shows that JIT-compiled machine code can outperform statically compiled native code.

5.5 The JVM warmup effect

Here we explore the potential problem of query execution performance caused by the JVM warmup. The issue is discussed in Section 2.2. Here, we launched Spark‘s interactive shell and ran Q1*, reading the data from memory. Figure 5.5 shows the time for hash aggregation with SF100. For the JVM and native vectorized engines, we see how the second and consecutive query executions take less time compared to their initial run. For the native engine, this difference in execution time may be attributed to cold CPU caches, but for the JVM, this can be explained with JIT compilation time, as well as, with cold caches.

We designed another test to isolate the effect of JIT compilation and avoid cold cache influence on query execution time. The result of this test is shown on Figure 5.6. We executed Q1* on WSCG, and after several runs, we switch to vectorized execution. This technique brings the data from memory in CPU caches and JIT-compiles Spark‘s code paths. However, it does not compile our code in our JVM-based vectorized engine. The vectorized primitives are compiled after we switch the engines and activate the vectorized one. Notice, that switching the query engine happens within the same Spark process, by updating a configuration setting. This way, we can observe how JIT compilation affects query execution time.

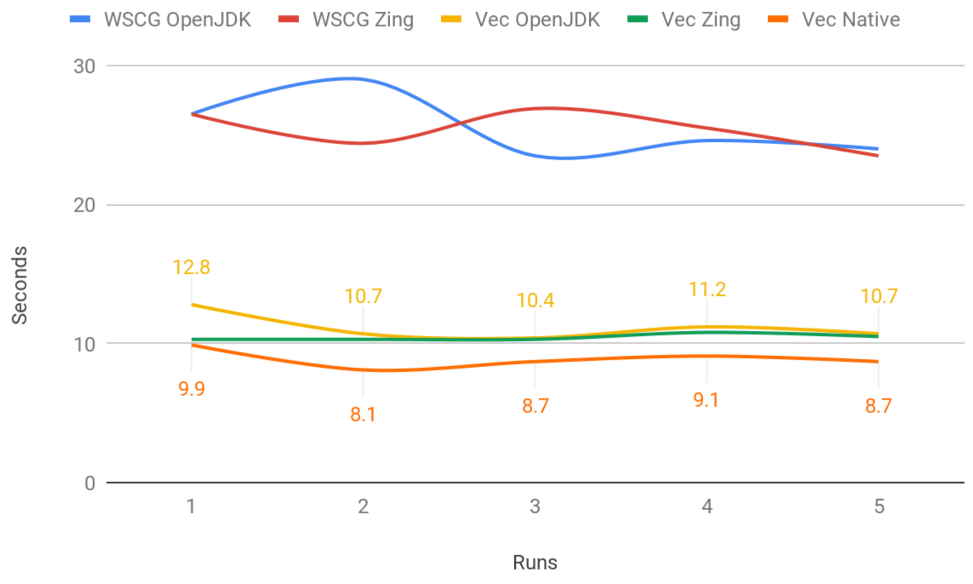


Figure 5.5: Q1* on SF100. Five runs in total, staring with a cold run

This time the data size was three times bigger (SF300) than in the previous test. The larger data size can make the performance differences between the engines clearer. On Figure 5.6, we see that the vectorized engine‘s execution time on OpenJDK fell by 3.3% in the second run compared to the first run. If we look at the subsequent runs, we can argue that this difference is caused partially by noise and partially by the JVM warmup.

Our observation is that the JVM manages to quickly JIT-compile hot paths. Consecutive calls to these paths run with a speed of machine code. The JVM compiles a function or a loop when it is executed ten thousand times by default. The SF100 test data contains six million rows. The query is unoptimized for the first three vector fetches, but consecutive vectors operations will be executed by machine code produced by the JIT compiler.

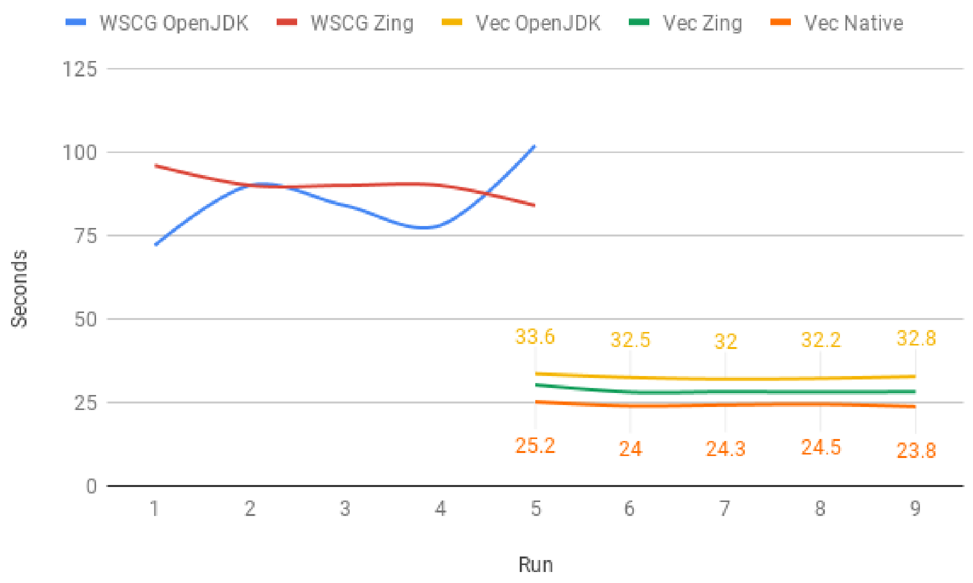


Figure 5.6: Q1* on SF300. Five runs with WSCG and then switching to the vectorized engines with five more runs. WSCG OpenJDK is followed by Vec OpenJDK. Same for Zing. Vec Native runs after WSCG OpenJDK

Chapter 6

Conclusion

In this project, we implemented a JVM-based vectorized query engine in Databricks Spark. We prove that a JVM-based vectorized engine improves query execution performance compared to the already existing JIT-compiling engine, and it comes close to the native vectorized engine.

How should a vectorized engine be designed in Java?

The JVM‘s JIT compilation introduces new challenges and a performance potential compared to a native engine. Based on our findings, the JVM warmup time has a negligible effect on query latency. Compiled methods are cached, and they can be reused for subsequent queries. JVM‘s devirtualization technique can inline virtual function calls and reduce a query interpretation time. However, it is unlikely to happen for megamorphic virtual calls. Therefore, their use should be avoided when calling vectorized primitives.

Vectorized primitives. Generics should be avoided in a JVM-based vectorized engine, due to the high overhead caused by auto-boxing. Hardcoding is another option for implementing the vectorized primitive. However, it leads to code explosion problem which harms code maintainability. For better code maintainability, a vectorized engine should preferably use JIT-generated primitives.

Is the JVM able to use and benefit from SIMD in vectorized execution? If so, which JVMs are more suitable for it?

We showed that in our JVM-based vectorized engine query execution performance can benefit from SIMD. However, SIMD support is still brittle in the JVM. The absence of manual SIMD puts the JVM-based approach at a disadvantage compared to a native engine. Switching to Zing JVM can bring better SIMD support compared to OpenJDK. However, moving to Zing is not a straightforward decision. OpenJDK may still be a practical solution for Spark‘s JVM-based vectorized engine. First, OpenJDK is a free and open-source tool, unlike Zing. Second, a vectorized engine implementation may get performance improvements on newer JVM versions.

How does a vectorized engine on the JVM perform comparing to the already existing data-centric engine, WSCG, and a native vectorized engine?

The comparison was made using a simplified TPC-H query 1 and 6, and the data was read from the main memory. Our JVM-based vectorized engine outperforms the JVM-based data-centric code generation engine (WSCG). The JVM‘s JIT compiler was able to quickly produce efficient machine code that comes close to the native engine‘s performance. Finally, the vectorized hash aggregation leverages out-of-order execution and resolves CPU cache misses in parallel which gives a vectorized engine a higher throughput compared to the tuple-at-a-time aggregation used in WSCG.

How do vectorized engines in the JVM and native language compare when it comes to Java-native UDF performance?

In the case of Spark, calling a UDF from a native query engine will require data marshaling, which is an overhead for query execution. The native vectorized engine will have this overhead for every batch. A JVM-based engine can avoid the mentioned overhead, since calling a Java/Scala native UDF does not require crossing environment boundary. Unfortunately, we could not implement UDF support in our JVM-based engine due to the constraint in time. Therefore, we cannot qualify how much benefit does a JVM-based engine bring in queries with UDFs.

6.1 Future work

Results obtained from this project are promising and open new research directions. As future work, UDF support on a JVM-based vectorized engine is still an active question. Theoretically, the JVM will achieve higher throughput with Java-native UDF. Therefore verifying this theory will make a stronger case in favor of JVM-based vectorized engines.

We hardcoded vectorized primitives and faced a code explosion problem. We proposed a solution which relies on JIT-compilation of vectorized primitives. We consider that generating vectorized primitives will not introduce the problems related to WSCG: the generated code is small and functionally similar. However, vectorized primitive compilation may introduce query execution latency. Implementing the mentioned technique for the vectorized primitive generation will demonstrate if the proposed solution is viable for a JVM-based vectorized engine.

Another interesting topic is the implementation of the selection operator. Active rows can be represented with selection vector or boolean vector. We can also use compaction by removing the filtered elements from the original vectors. We choose the compaction as it is relatively simple to implement compared to the other two options. However, it is interesting to see which selection model can yield higher throughput in the case of the JVM and if the JVM can SIMDize the mentioned operations.

Exploring new features introduced in newer Java versions (9 and above) will shed light on the future of a JVM-based vectorized engine. In the future, with project Panama, manual injection of SIMD instructions will become possible in the JVM. Another JVM feature, called the JVM Compiler Interface[1] (JVMCI) allows easy extension of JIT compiler‘s functionality. The customized JIT compiler can detect vectorized primitives‘ bytecode and produce SIMDized machine version, giving a programmer more control over the produced machine code. Another interesting feature in the JVM is Ahead-of-Time compilation[2] (JAOTC). Ahead of time compilation can reduce the application startup time, and make BI queries more apt by reducing the JVM warmup effect. The mentioned features increase Java‘s potential to be an even more viable option for a vectorized engine.

[1]https://openjdk.java.net/jeps/243
[2]https://openjdk.java.net/jeps/295

References

[1] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. **Everything you always wanted to know about compiled and vectorized queries but were afraid to ask**. *Proceedings of the VLDB Endowment*, **11**(13):2209–2222, 2018. 1, 2, 15, 18, 19, 27, 51

[2] Peter A Boncz, Marcin Zukowski, and Niels Nes. **MonetDB/X100: Hyper-Pipelining Query Execution.** In *Cidr*, **5**, pages 225–237, 2005. 1, 15, 18, 21, 22

[3] Thomas Neumann. **Efficiently compiling efficient query plans for modern hardware**. *Proceedings of the VLDB Endowment*, **4**(9):539–550, 2011. 1, 15, 17, 21, 22

[4] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. **Micro adaptivity in vectorwise**. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1231–1242. ACM, 2013. 2, 22, 33

[5] Tim Gubner and Peter Boncz. **Exploring query execution strategies for jit, vectorization and simd**. In *Eighth International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, 2017. 2, 18, 22, 27

[6] Jun Rao, Hamid Pirahesh, C. Mohan, and Guy M. Lohman. **Compiled Query Execution Engine using JVM**. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 23. IEEE Computer Society, 2006. 2, 22

[7] Bill Chambers and Matei Zaharia. *Spark: the definitive guide: big data processing made simple.* " O'Reilly Media, Inc.", 2018. 3, 22, 23, 28

[8] Sameer Agarwal, Davies Liu, and Reynold Xin. **Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop Deep dive into the new Tungsten execution engine**. `https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html`, 2016. Accessed: 2019-03-01. 3

[9] Randal E Bryant, O'Hallaron David Richard, and O'Hallaron David Richard. *Computer systems: a programmer's perspective*, **281**. Prentice Hall Upper Saddle River, 2003. vi, 8

[10] James Tuck, Luis Ceze, and Josep Torrellas. **Scalable cache miss handling for high memory-level parallelism**. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 409–422. IEEE Computer Society, 2006. 8

[11] Urs Hölzle, Craig Chambers, and David Ungar. **Debugging optimized code with dynamic deoptimization**. In *ACM Sigplan Notices*, **27**, pages 32–43. ACM, 1992. 10

[12] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. **Array bounds check elimination for the Java HotSpotâĎć client compiler**. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 125–133. ACM, 2007. 11

[13] Michael Paleczny, Christopher Vick, and Cliff Click. **The java hotspot TM server compiler**. In *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium*, **1**, 2001. 13

[14] Goetz Graefe. **Volcano - An Extensible and Parallel Query Evaluation System**. *IEEE Trans. Knowl. Data Eng.*, **6**(1):120–135, 1994. 15, 21, 22

[15] Juliusz Sompolski, Marcin Zukowski, and Peter A. Boncz. **Vectorization vs. compilation in query execution**. In Stavros Harizopoulos and Qiong Luo, editors, *Proceedings of the Seventh International Workshop on Data Management on New Hardware, DaMoN 2011, Athens, Greece, June 13, 2011*, pages 33–40. ACM, 2011. 15, 21, 22, 41

[16] M Zukowski. **Balancing Vectorized Query Execution with Bandwidth-Optimized Storage**. *Journal of Computational Physics - J COMPUT PHYS*, 01 2009. 16, 33, 39, 41, 50

[17] Peter Alexander Boncz et al. *Monet: A next-generation DBMS kernel for query-intensive applications.* Universiteit van Amsterdam [Host], 2002. 18, 22

[18] Jingren Zhou and Kenneth A. Ross. **Implementing database operations using SIMD instructions**. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, pages 145–156. ACM, 2002. 18, 27

[19] Konstantinos Krikellas, Stratis D Viglas, and Marcelo Cintra. **Generating code for holistic query evaluation**. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 613–624. IEEE, 2010. 20, 21, 22

[20] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. **System R: relational approach to database management**. *ACM Transactions on Database Systems (TODS)*, **1**(2):97–137, 1976. 21

[21] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. **Building efficient query engines in a high-level language**. *Proceedings of the VLDB Endowment*, **7**(10):853–864, 2014. 21, 22

[22] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. **DB2 with BLU acceleration: So much more than just a column store**. *Proceedings of the VLDB Endowment*, **6**(11):1080–1091, 2013. 21, 22

[23] Jignesh M Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. **Quickstep: A data platform based on the scaling-up approach**. *Proceedings of the VLDB Endowment*, **11**(6):663–676, 2018. 21, 22

[24] Prashanth Menon, Andrew Pavlo, and Todd C. Mowry. **Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last**. *PVLDB*, **11**(1):1–13, 2017. 21, 22

[25] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. **Hekaton: SQL server's memory-optimized OLTP engine**. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254. ACM, 2013. 21

[26] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stan Zdonik. **Tupleware: Redefining modern analytics**. *arXiv preprint arXiv:1406.6667*, 2014. 21, 22

[27] Donald D Chamberlin, Morton M Astrahan, Michael W Blasgen, James N Gray, W Frank King, Bruce G Lindsay, Raymond Lorie, James W Mehl, Thomas G Price, Franco Putzolu, et al. **A history and evaluation of System R**. *Communications of the ACM*, **24**(10):632–646, 1981. 21

[28] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. **Presto: SQL on Everything**. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1802–1813. IEEE, 2019. 22

[29] Tiark Rompf and Martin Odersky. **Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs**. In *Acm Sigplan Notices*, **46**, pages 127–136. ACM, 2010. 22

[30] Thomas Würthinger, Christian Wimmer, Andreas Wöss, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. **One VM to rule them all**. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 187–204. ACM, 2013. 23, 31

[31] Vladimir Ivanov. **Vectorization in HotSpot JVM**. `https://cr.openjdk.java.net/~vlivanov/talks/2017_Vectorization_in_HotSpot_JVM.pdf`, 2017. Accessed: 2019-03-04. 27

[32] Alen Stojanov, Ivaylo Toskov, Tiark Rompf, and Markus Püschel. **SIMD intrinsics on managed language runtimes**. In Jens Knoop, Markus Schordan, Teresa Johnson, and Michael F. P. O'Boyle, editors, *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018*, pages 2–15. ACM, 2018. 27

[33] Ionut Boicu. **Adaptive on-the-fly compressed execution in Spark**. 2019. 50