Project Report: Parallel Query Processing in PostgreSQL

Kai Franz

https://kai-franz.github.io/15418-project/

Summary

I implemented a vectorized sequential scan in PostgreSQL that leverages instruction-level parallelism to perform filtering more efficiently. I then compared my implementation to the standard tuple-at-a-time sequential scan implementation in PostgreSQL.

Background

In a database management system (DBMS), declarative queries are executed via a query plan, which is generated by the system. These query plans are composed of building blocks called physical operators, which are implemented by programmers working on the DBMS. The sequential scan operator is one common physical operator, and it is used to scan a table of many (usually unsorted) tuples, filtering for tuples that match a predicate specified by the DBMS.

A common approach to implementing physical operators in a database management system is called the Volcano model. Under this style, each physical operator is implemented as an iterator object, and the query plan is implemented as a tree of these objects. Each result tuple of the given query is fetched by calling the getNext() operator on the root of the query plan; in turn, this node will call getNext() on its children one or more times, and so on.

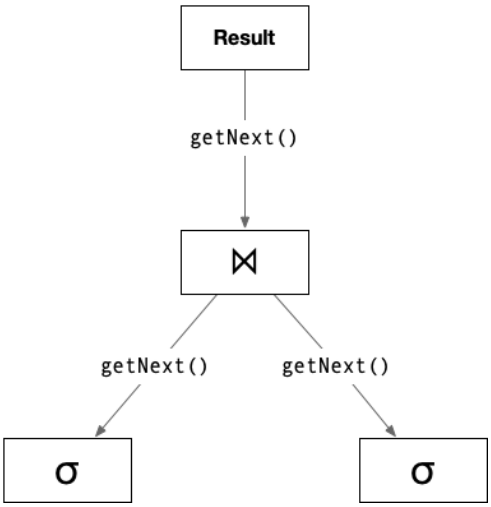


Figure 1: Example query plan using the Volcano model.

The Volcano model has several benefits, including simplicity, modularity, and ease of maintenance. However, the drawback of this approach is that it introduces a significant overhead into query execution. The interpretation overhead of jumping between different contexts for each physical operator waste valuable CPU cycles, while the tuple-at–a-time processing approach hides instruction locality from the processor's dynamic scheduler. [1]

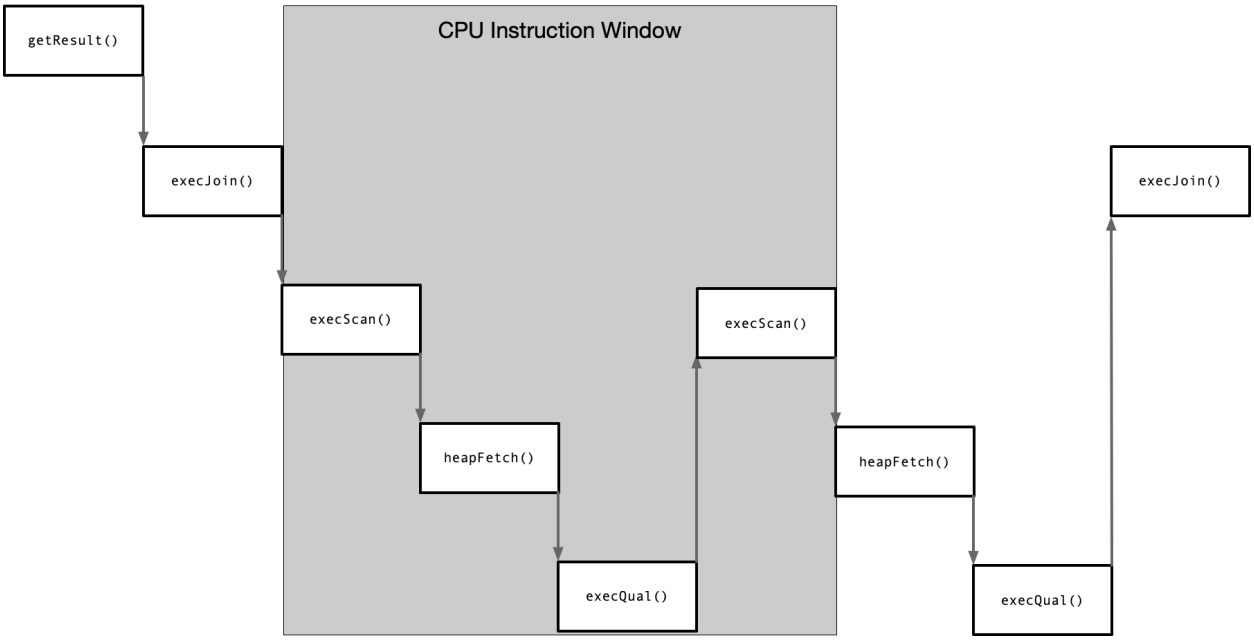


Figure 2: Sample tuple-at-a-time execution flow demonstrating unpredictable branching.

Most modern processors make use of superscalar execution to achieve maximum efficiency. While machine code is generated as a linear sequence of instructions, rather than executing one instruction at a time, superscalar processors will look at a window of several hundred instructions and execute them out of order if their operands are ready, executing several instructions in parallel on each clock cycle. In order to best leverage the capabilities of a superscalar processor, programmers must allow the CPU to understand which instructions are likely to be executed in the future. A tuple-at–a-time execution model does this poorly, as jumps between physical operators are difficult to predict and can thus lead to suboptimal CPU utilization. [1]

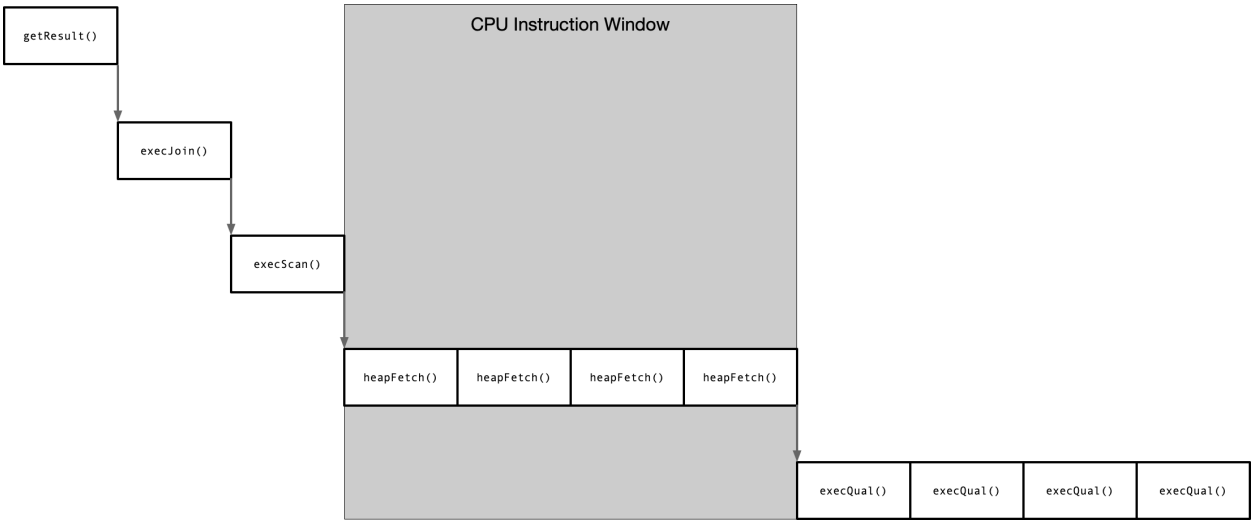


Figure 3: Vectorized query execution flow.

Sequential Scans

Revisiting the sequential scan operator: this operator typically functions as follows. First, its parent node calls getNext() on it. This causes it to call getNext() on its child, which returns the next tuple from the database heap, which stores the data. Upon receiving this tuple, the sequential scan operator checks if this tuple satisfies the assigned predicate. If it does not, it continues to call getNext() on its child until either (1) it returns a satisfactory tuple, or (2) it returns NULL. In both cases, it returns the supplied tuple to its parent operator (a NULL tuple represents the end of the operator's output).

This behavior presents a problem for superscalar processors because scanning different tuples can result in divergent control flow; if a tuple satisfies the predicate then execution jumps to the parent operator; if it fails, the processor must jump to the child operator. These predicates are often complex, and the CPU's branch predictor will not be able to exploit patterns in the data due to the unsorted nature of the database heap. However, this is not a fundamental problem with database query processing; each predicate evaluation is independent and can thus be safely executed in parallel. In this project, I aim to target this particular drawback of sequential scans and be able to better exploit instruction-level parallelism.

Approach

I addressed the aforementioned problem using vectorization. This approach is similar to software pipelining, where a loop with several operations that are independent across loop iterations but dependent on each other within an iteration, is transformed such that the independent operations execute in series. Similarly, with vectorization, rather than checking the predicate on each tuple, one at a time, the DBMS checks the predicate on a vector of, say, 1024 tuples. This allows an out-of-order processor to schedule predicate evaluation in parallel, increasing processor utilization.

A simplified version of this transformation is shown below in pseudocode:

```
Scan->getNext():
  tuple = heap->getNext()
  while tuple is not NULL:
    if satisfiesPredicate(tuple):
      return tuple
  tuple = heap->getNext()
  return NULL
```

Before vectorization

```
Scan->getNext():
  if vector.empty():
     for i = 0 .. 1024:
        vector[i] = heap->getNext()
     for i = 0 .. 1024:
        bitmap[i] = satisfiesPredicate(vector[i])
     nextTuple = 0
   while !bitmap[nextTuple]:
     nextTuple++
```

After vectorization

I implemented this optimization starting from PostgreSQL 14.0, which is open source.

Iteration

Originally, I based my implementation off of the materialize physical operator, which fetches all the tuples of its child operator and makes a copy of them, which can then be rescanned by its parent operator.

Experimental Setup

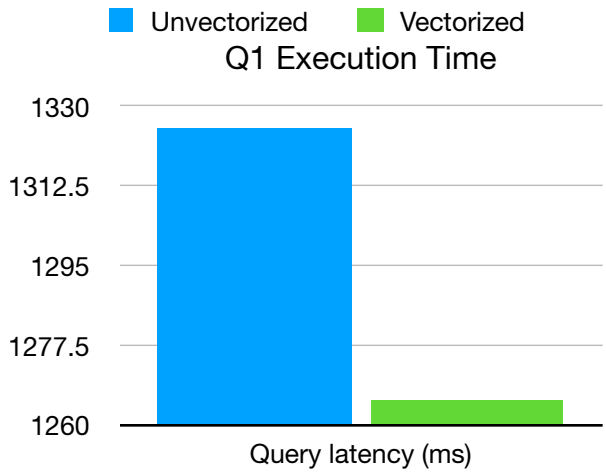
I evaluated my setup using microbenchmark SQL queries that I constructed. Compiling PostgreSQL from source, I used the following settings:

- Table of 2^20 random double precision floating-point numbers
- Prewarmed buffer pool
- Single-threaded
- JIT expression compilation enabled for every query
- Intel(R) Xeon(R) W-1290 CPU @ 3.20GHz, 64 GB RAM
- Ran each experiment 5x

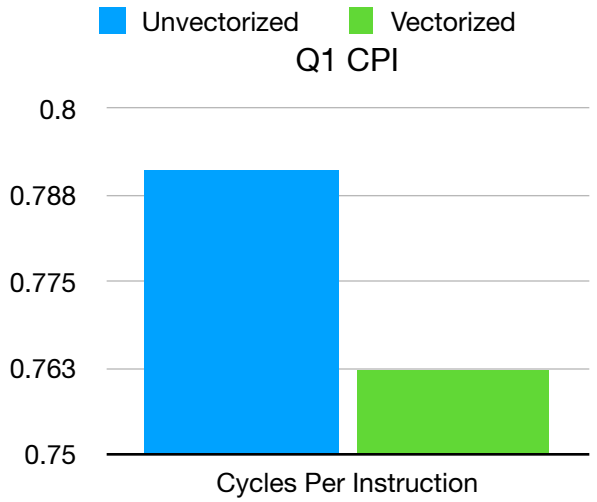
Since I included the ability to turn off vectorization using PostgreSQL's SQL interface, my baseline comparison was on the same database with vectorization turned off.

The first microbenchmark, Q1, I constructed scans a table and searches for values that hash to a value beginning with a 0 byte, making the predicate unpredictable even if the tuples are sorted.

```
select count(*) from t where digest(c1::text::bytea, 'sha256') like '%00%';
```

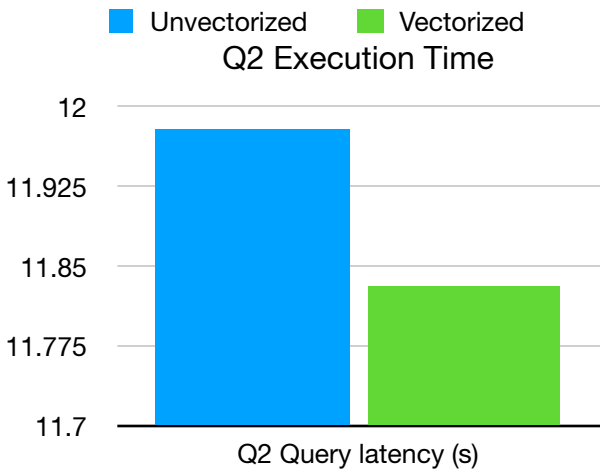


Vectorization resulted in a 5% speedup. Diving deeper, I profiled the performance of Q1 using VTune and measured the cycles per instruction (CPI) of the predicate functions for both implementations.



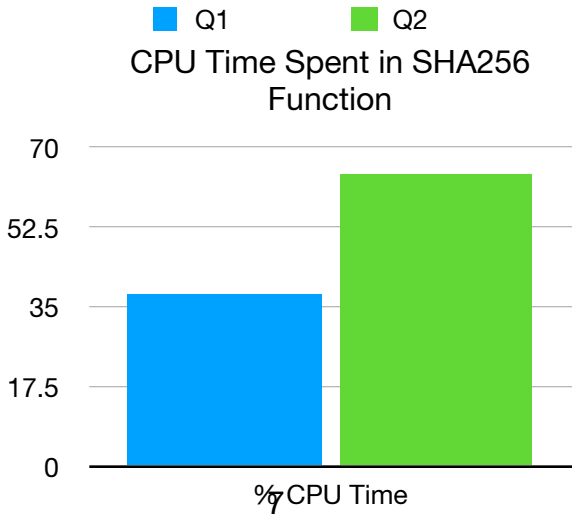
The next query, Q2, uses a query that is much more expensive to evaluate than the one in Q1. This query computes the SHA256 hash of a floating point before checking if the final hash starts with a 0 byte.

```
SELECT COUNT(*) FROM t WHERE
SUBSTRING(digest(digest(digest(digest(digest(digest(digest(digest(digest(digest(
digest(digest(digest(digest(C1::text::bytea, 'sha256'), 'sha256'), 'sha256'),
'sha256'), 'sha256'), 'sha256'), 'sha256'), 'sha256'), 'sha256'), 'sha256'),
'sha256'), 'sha256'), 'sha256'), 'sha256') FROM 1 FOR 2) = '00';
```



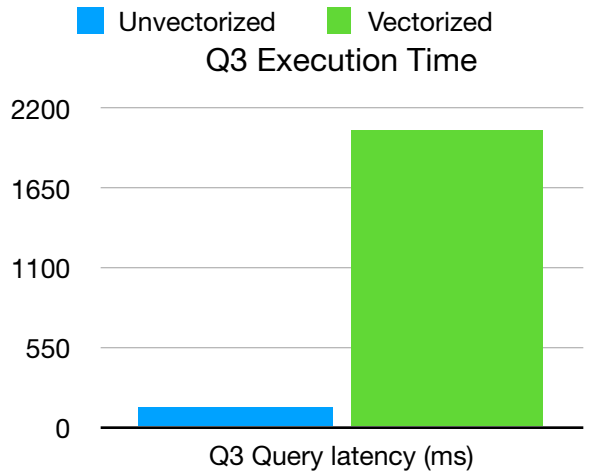
Vectorization results in a 1% speedup.

Profiling with VTune, I compared the percentage of time spent deep inside the predicate calculating SHA256 hashes. Q2 spends much more time than Q1 in the hash function, resulting in a lower vectorization speedup for Q2.

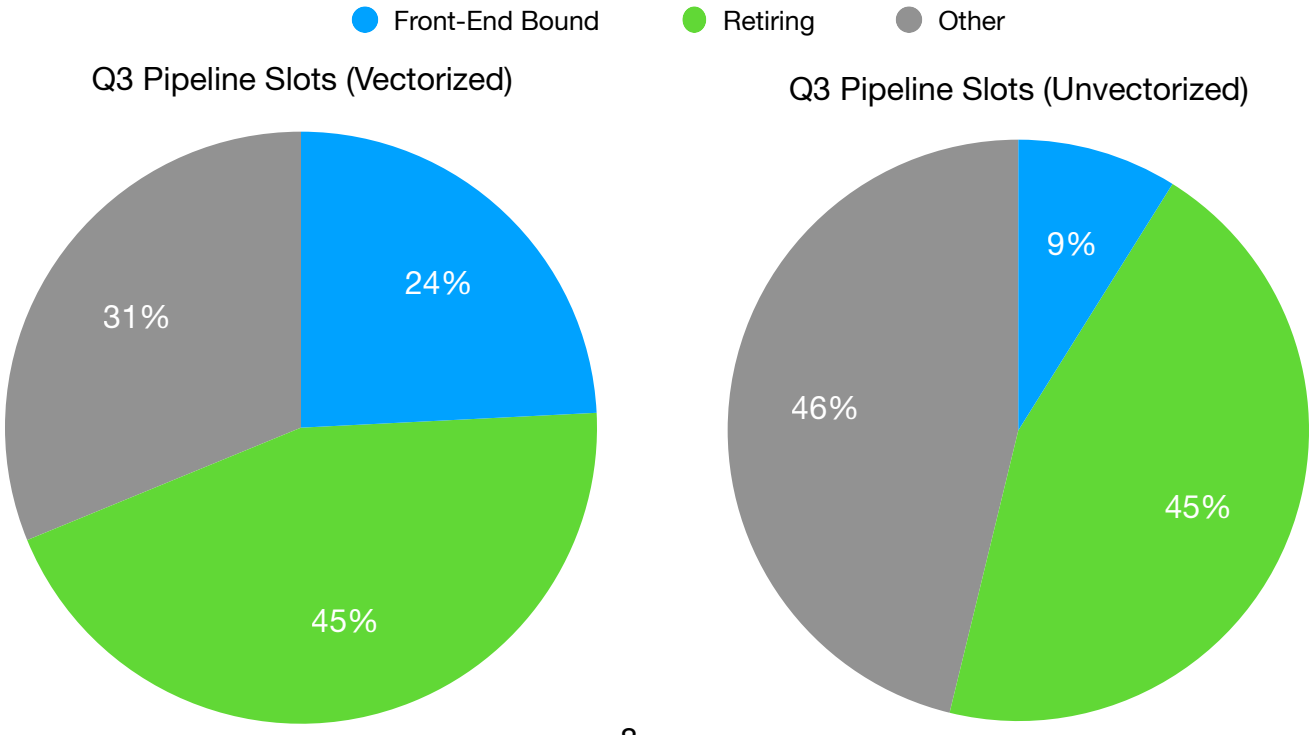


The final query, Q3, has a predicate that is cheap to evaluate: it simply checks if a floating-point value is less than a constant.

```
select count(*) from t where c1 < 150000000;
```



Vectorization results in a 15x slowdown. Using VTune, I profiled Q3 running both vectorized and unvectorized. I found that the vectorized version of Q3 spent much more of its pipeline slots front-end bound than the tuple-at-a-time version.



Discussion

Q1

As shown above, the performance gains of vectorization are extremely workload-dependent. The best efficiency gains were observed on a query that had a predicate that required a modest number of instructions to evaluate: not too many or too few. We see that Q1 achieved the best speedup out of all three workloads.

Q2

Q2's much more expensive predicate caused diminishing performance gains due to instruction-level parallelism. As shown by the profiler, much more time was spent evaluating hash functions inside the predicate, resulting in a low vectorization speedup for Q2.

Q3

Q3 is interesting. As it contains the simplest predicate, we might expect it to have the highest percentage gains from instruction-level parallelism, yet it performed almost 15x slower with vectorization enabled. One of the drawbacks of vectorization is that it requires copying tuples into a vector while scanning them, while in tuple-at-a-time processing, tuples can be efficiently pipelined between operators. This extra copying introduced by vectorization demands more memory bandwidth, a resource that is already scarce in DBMSs. Additionally, the vectorized operator needs to store the predicate results so that it knows which tuples it can return in the future. Reading and writing this information also increases memory bandwidth.

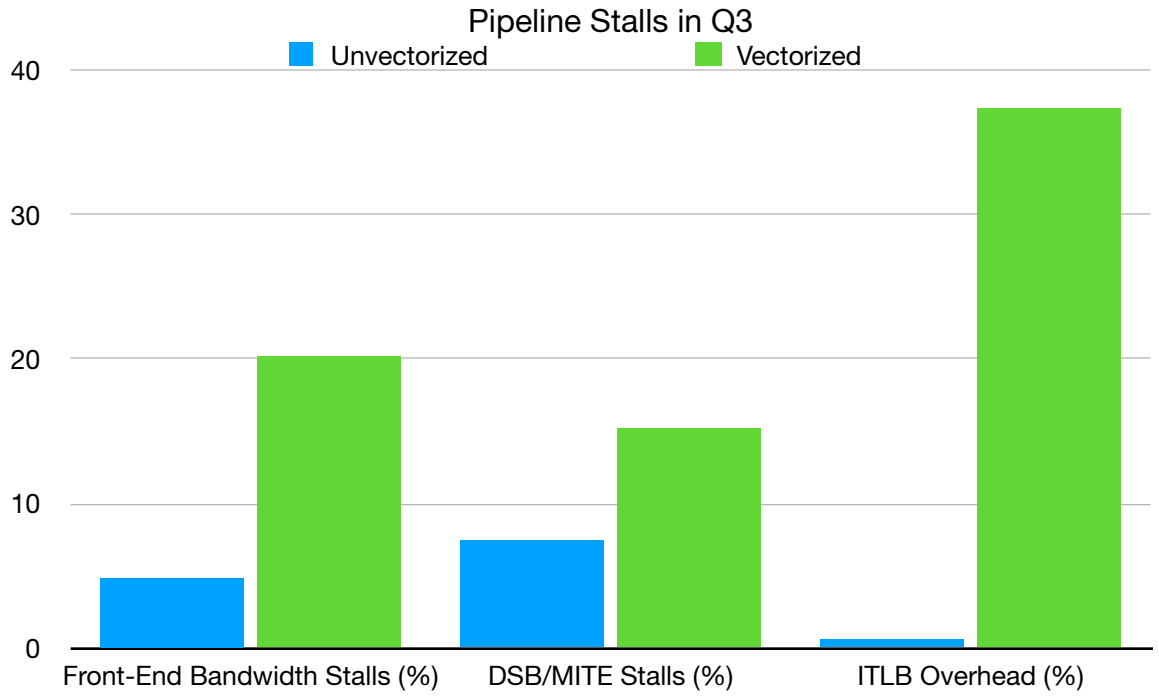
I suspected that this workload, with such a simple predicate, was heavily memory-bound, and introducing vectorization heightened memory pressure without providing much speedup (which is gained by efficiently scheduling computation). After profiling both the vectorized and unvectorized version of Q3 with VTune, I found that the vectorized version spent a large number (24%) of its pipeline spots front-end bound, while the tuple-at-a-time version spent

only 9% of its pipeline slots front-end bound. This confirms the idea that the increased memory pressure causes vectorized queries to perform poorly.

Looking into the profiler-annotated source code, 3 out of the 5 lines that took the longest to execute were reading from or writing to either the predicate result bitmap or the tuple vector. Interestingly, reading the saved predicate value was over 20x as expensive as computing it in the first place. Computing the predicate was about 15x faster on the vectorized version than the unvectorized version, demonstrating the significance of instruction-level parallelism.

Conclusion

According to VTune, the factor that limited my speedup the most was data movement, and, in particular, memory bandwidth.



The profiler output shows that, in addition to the aforementioned memory stalls, vectorization introduces stalls due to switching between DSB and MITE. The DSB (Decoded Stream Buffer) caches decoded micro-operations, but when there is a DSB miss, the instructions must be decoded again using the MITE (Micro-Instruction Translation Engine). According to VTune, a

high DSB/MITE penalty is often due to hot regions that are too large to fit into the DSB. In addition, the ITLB overhead accounts for over 60 times as many pipeline slots in the vectorized implementation than the tuple-at-a-time version. This also appears to be a symptom of having a hot region that does not fit into the I-cache: vectorization introduces significantly more instructions, a problem that may be exacerbated by loop unrolling the vectorized code. In conclusion, both the increased memory pressure from storing tuples an extra time and the increased code size caused by vectorization limit its performance gains.

References

1. Boncz, Peter A., Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution." Cidr. Vol. 5. 2005.