Vectorization vs. Compilation in Query Execution

这篇论文是vectorwise的作者，主要讲vectorwise如何将向量化与编译执行结合起来达到最优的性能表现。

1. ABSTRACT

对于单线程执行框架，向量化和即时编译是常用的两个优化手段，即时编译和向量化都可以减少执行过程中的解释开销，优化code locality以及利用SIMD优化，本文主要关注如何将两者结合，通过对常见的Project，Select（filter）, Hash Join算子在Vectorwise数据库上的两种优化方法对比得出结论：编译执行应该和向量化结合起来达到更好的性能表现。

2. INTRODUCTION

vectorwise使用的是经典的iterator-model，通常每个operator都包含三个方法：open(), next(),close(), next()方式遍历operator tree向下迭代请求并返回tuple，结果沿tree 向上返回。这种a-tuple-at-a-time的执行方式比较明显的缺点是interpretation overhead，也就是说解释模型和函数调用消耗的时间超过了实际计算的时间，并且这种执行方式也无法利用现代处理器的CPU流水线技术以及SIMD（single instruction multiple data）技术。

所以通用的解决方案就是向量化执行技术。另一篇文章中介绍过，向量化最初用在Monet-X100上，后来被迁移到vectorwise上。此时next()一次处理多条tuple（100-10000）。向量化带来的好处显然就是减少了解释执行以及函数调用，所以可以带来tight loop，利用好编译器的一些优化以及CPU的乱序，SIMD，以及分支预测。在OLAP提升可以达到50倍。

loop-complication是另外常用的方法通过即时编译来减少interpretation overhead的影响。使用到编译技术的系统包括了ParAccel和Hyper，目前主要在c++中是通过LLVM来实现即时编译。

以下通过三个case来比较两者的影响。这里只给出一些结论，不做算法的解释。有兴趣的同学可以具体看一下论文中伪代码的实现。

3. PROJECT

测试query：SELECT l_extprice * (1-l_discount)* (1-l_tax) FROM lineitem;

其中以上三个变量都是精度为2的小数，vectorwise将其乘以100变为可以表示实际数值的最小整形，l_extprice是4字节整形，l_discount,l_tax都是单字节整型，所以最终相乘的结果保存在4字节数值里即可以保证不溢出。

通常gcc以及icc都可以进行SIMD编译优化。在该场景下，单个SSE在x86系统上可以处理16个单字节的加减， 8个单字节整数的乘积，或 4个4字节整数的乘积。对于16tuples需要大概60个instructions（22 load/stores， 8 calculation，30 casts padding looping）， 而不使用SIMD处理一个tuple需要16(4✖4)个instructions,不使用SIMD优化大约是使用SIMD的指令数的4倍。

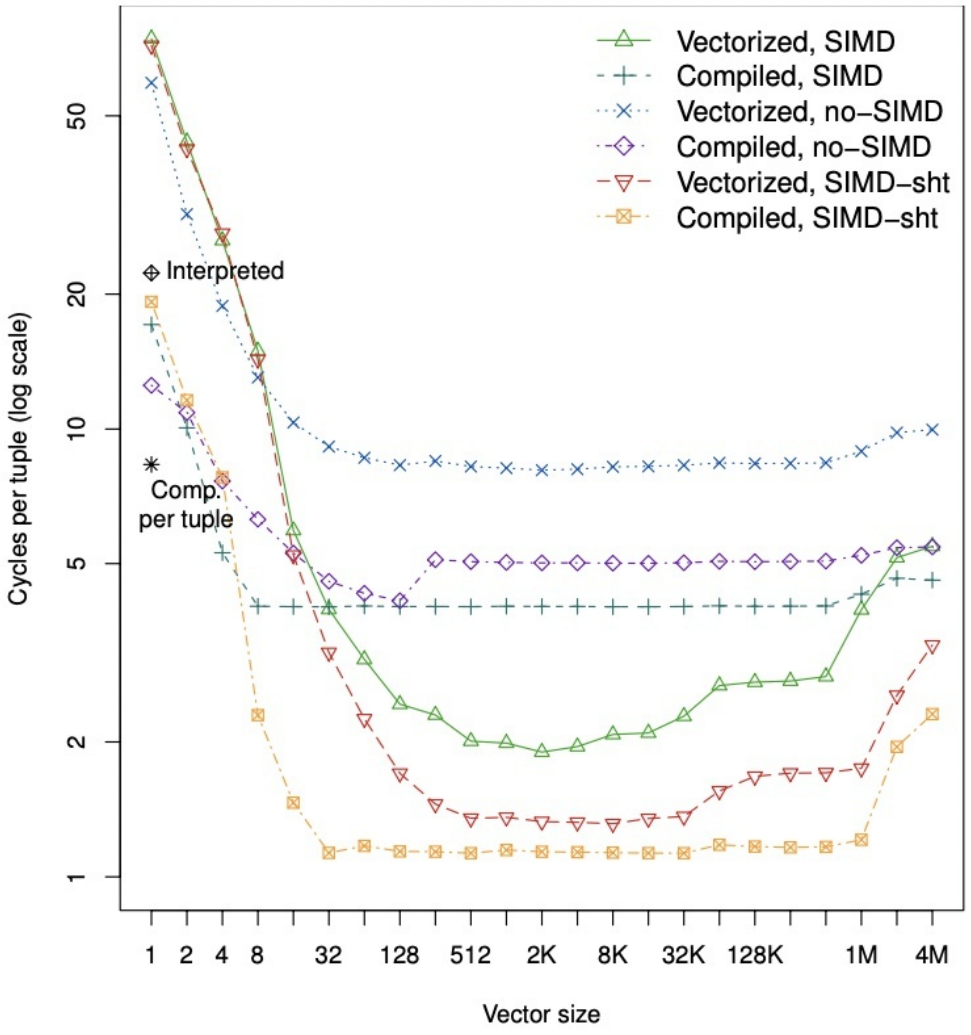

Figure 1: Project micro-benchmark: with and without {compilation, vectorization, SIMD}. The "SIMD-sht" lines work around the alignment sub-optimality in icc SIMD code generation.

知乎 @AwakeLjw

1. 所有曲线的走势都是随着vectorsize的增长，需要的cpu cycles减小后增加，这是因为当vector size大于L1和L2cache size后发生的cache miss将会增加。
2. complication的优势在于减少load/stores指令。但由于SIMD对齐的问题，并不一定可以比vector优化更好。本文中的例子是使用4字节对齐，导致1字节和2字节的SIMD operation没有得到优化，如果将l_extprice改为2字节的话也就是（SIMD-sht曲线代表）编译的提升大于向量化的提升。
3. tuple-at-a-time经过编译优化后，性能可以从21cycles提升到7cycles，而向量化执行的优化可以从（7-1.2cycle），绝对性能要提升更多。

4. SELECT

测试query：WHERE col1 < v1 AND col2 < v2 AND col3 < v3

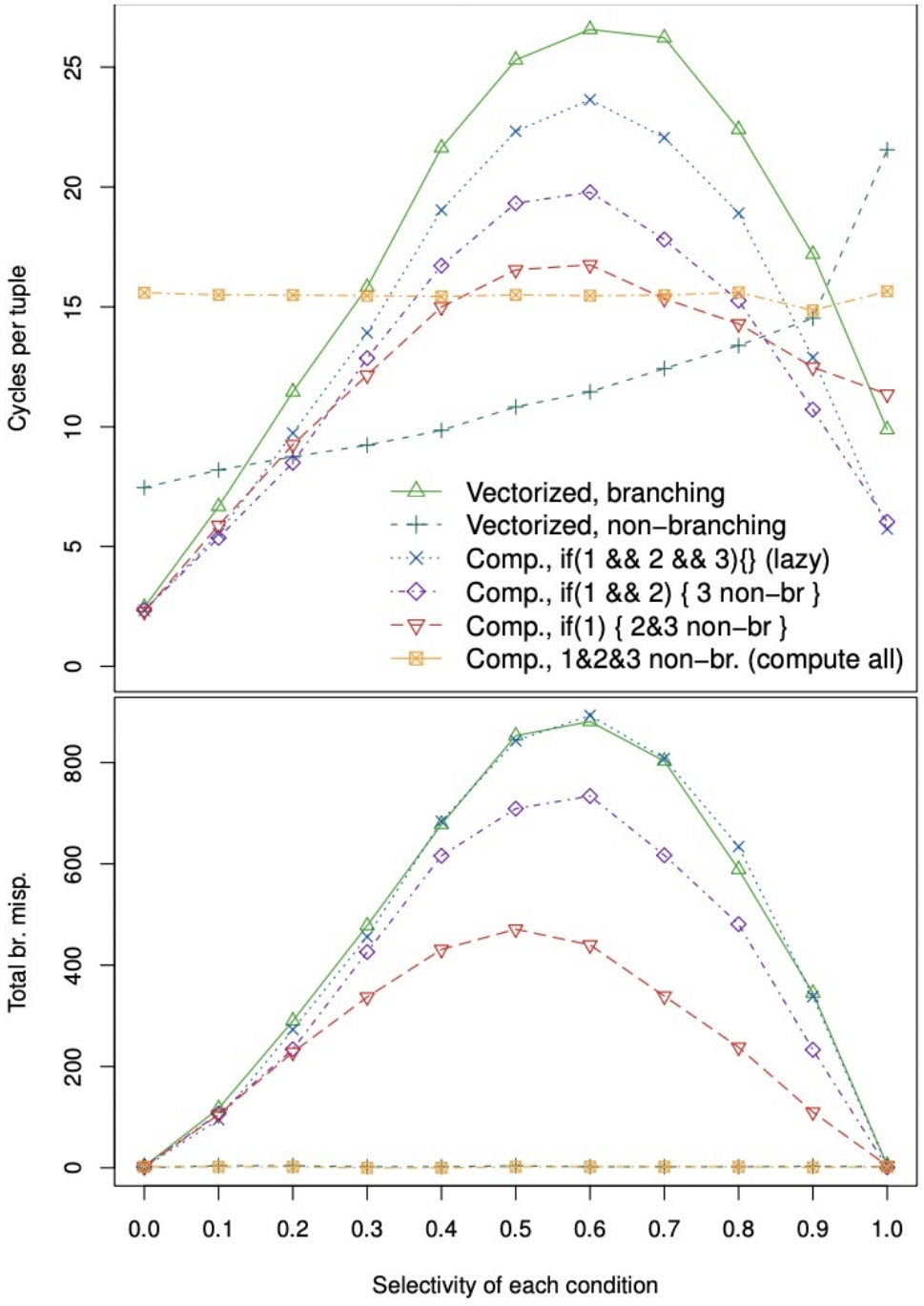

Figure 2: Conjunction of selection conditions: total cycles and branch mispredictions vs. selectivity

知乎 @AwakeLjw

1. branching和no-branching的差别就是使用if条件判断。
2. conjunctive select场景下总体编译技术no-branching不如向量化no-branching表现好。选择率会影响branch mispredictions。在中等选择率上，branch misprediction最大。

3. 对于多个选择条件，当选择率在50%左右时，向量化+no branching的表现最好，但当选择率靠近边界时，brancing表现会更好。因为当靠近边界时，部分条件不满足就可以退出而no-branching则需要所有的条件计算。所以可以根据选择率来选择select的执行方法。

5. HASH JOIN

重点来看HASH JOIN。测试query使用的是等于条件。

测试query:SELECT build.col1, build.col2, build.col3 WHERE probe.key1 = build.key1 AND probe.key2 = build.key2 FROM probe, build

这里vectorwise使用的hash table不是传统的链表法，而是bucket-chaining。这个bucket-chaining的实现原理是当遇到hash冲突时，如果bucket中的value不为0，将bucket中的value放入next对应的offset中，比如hash values都是3的两个row number，后插入的也就是4放在bucket里面，而原来的1放在next的第4个offset。DSM类似于列存，而NSM对应的是行存。

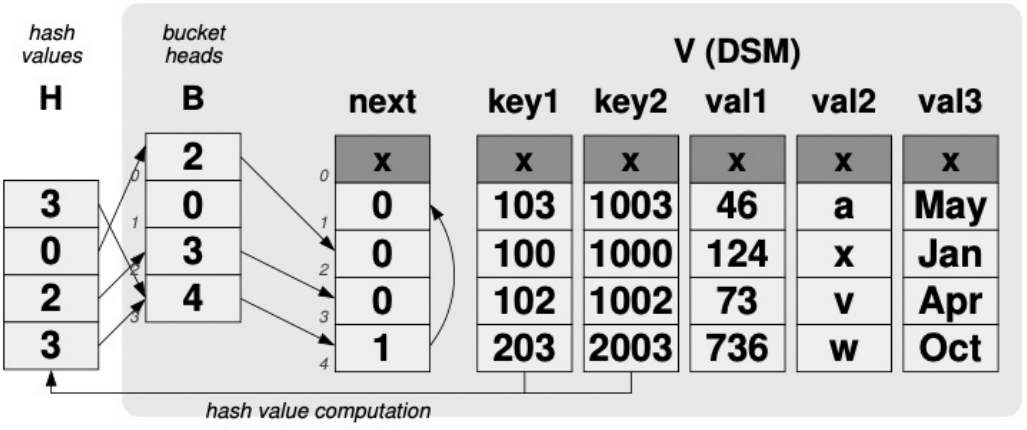

Figure 3: Bucket-chain hash table as used in VectorWise. The value space V presented in the figure is in DSM format, with separate array for each attribute. It can also be implemented in NSM, with data stored tuple-wise.

知乎 @AwakeLjw

```
procedure HTPROBE(V, B[0..N − 1], K⃗_{1..k}(in), R⃗_{1..v}(out))
    // Iterative hash-number computation
    H⃗ ← map_hash(K⃗_1)
    for each remaining key vectors K_i do
        H⃗ ← map_rehash(H⃗, K⃗_i)
    H⃗ ← map_bitwiseand(H⃗, N − 1)
    // Initial lookup of candidate matches
    P⃗os, M⃗atch ← ht_lookup_initial(H, B)
    while M⃗atch not empty do
        // Candidate value verification
        C⃗heck ← map_check(K⃗_1, V_{key_1}, P⃗os, M⃗atch)
        for each remaining key vector K⃗_i do
            C⃗heck ← map_recheck(C⃗heck, K⃗_i, V_{key_i}, P⃗os, M⃗atch)
        M⃗atch ← sel_nonzero(C⃗heck, M⃗atch)
        // Chain following
        P⃗os, M⃗atch ← ht_lookup_next(P⃗os, M⃗atch, V_next)
    H⃗its ← sel_nonzero(P⃗os)
    // Fetching the non-key attributes
    for each result vector R⃗_i do
        R⃗_i ← map_fetch(P⃗os, V_{value_i}, H⃗its)
```

知乎 @AwakeLjw

文章中重点介绍了hash table probe的过程，map_hash原语表示将给定的Key向量计算生成一个hash值向量 和map_rehash原语表示复合键（也就是多个等于条件）是需要对第一个hash值向量与第二列key向量进行hash计算，最后通过map_bitwiseand（H &（N - 1））， N一般是2的倍数，来计算产生bucket number。

接下来是ht_lookup_initial原语，对Bucket向量中的值保存在Pos向量中，其中bucket不为零的offset放置到Match向量中。

然后是map_check原语和map_recheck原语，对于hash值相等的tuple需要对每个key进行比较，map_check表示对单一key的检查，结果保存在check向量中，map_recheck在check向量基础上继续对剩下的key进行比较，更新check向量，check中保存的是失败的position。通过sel_nonzero原语更新Match向量，通过ht_lookup_next原语来对next数组中进行遍历，直到next对应的offset为0，或者已经找到对应的key则停止，最后将结果通过map_fetch原语写到结果集中返回。

以上大致就是整个vector hash join的执行过程。

partial compilation

在以上的执行过程中有三处可以进行编译优化，

1. 将hash/rehash/bitwise合成为一个原语。
2. 将check/recheck/select>0合成一个原语。如果hash table调优较好的话选择率可以达到75%，所以使用no-branching的优化。
3. 将fetch multiple columns优化成一个composite fetch原语。这一优化主要对行存有优势。

向量化执行一般是一次一列，所以对于列存需要column次数的fetch。随着fetched column增加，因为vector-size的大小一定大于TLB cache size并且可能超过cache line的大小，所以随着fetched columns增加TLB miss线线性的增长，而compiled NSM则不会随着columns的增长而增长（一次多列实现的data locality）。上图也可以看出向量化的NSM和DSM的性能相似，tuple-wise也就是行存由于更好的data locality更有优势。

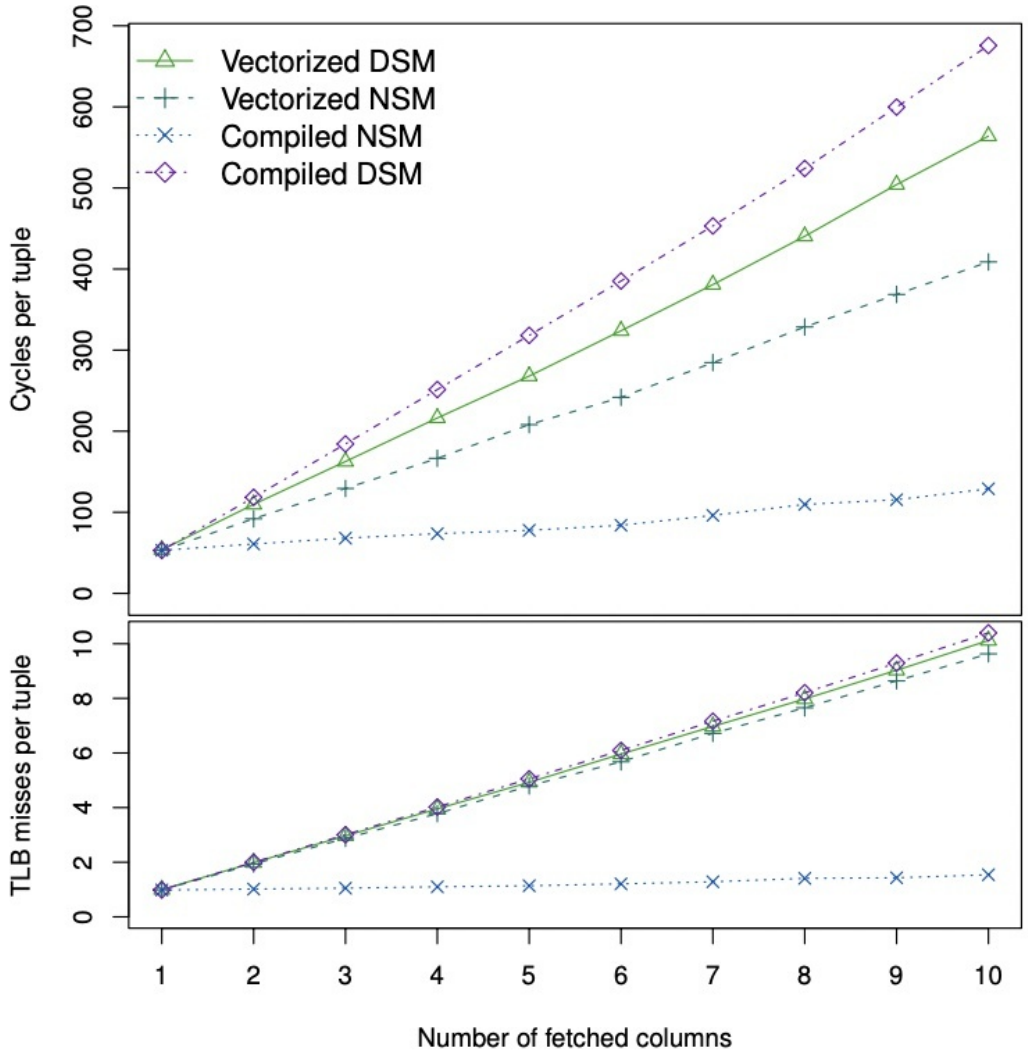

Figure 4: Fetching columns of data from a hash table: cycles per tuple and total TLB misses

知乎 @AwakeLjw

相对于以上的partial complication，又可以使用full complication。full complication将以上的probe过程集成到一个tight loop-complied algorithm中，但实际fully complied algorithm的表现不如partial compilation。fully complied algorithm算法如下。

Algorithm 5 Fully loop-compiled hash probing: for each NSM tuple, read hash bucket from B, loop through a chain in V, fetching results when the check produces a match

```
for(idx i=0, j=0; i<n; i++) {
  idx pos, hash = HASH(key1[i])^HASH(key2[i]);
  if (pos = B[hash&(N-1)]) do {
    if(key1[i]==V[pos].key1 &&
       key2[i]==V[pos].key2) {
        res1[i] = V[pos].col1;
        res2[i] = V[pos].col2;
        res3[i] = V[pos].col3;
        break; // found match
    }
  } while(pos = V.next[pos]); // next
}
```

知乎 @AwakeLjw

Parallel memory access现在硬件内存的延迟大概在（100ns），cache line是64bytes，内存的带宽不是两者简单的除法。单个Nehalem核由于顺序访问的预读优化对于多个memory requests可以实现10倍的带宽提升。而对于随机访问，虽然cpu乱序执行可以防止指令阻塞，但乱序时带宽提升只有4倍并且成功率还不保证。虽然Speculation-friendly推测友好的代码比人工预读的代码更高效。

作者对于为什么fully loop-complied的实现不如partial complication的解释如下：对于partial compilation的，vectorized fetch 原语可以实现最大的CPU支持的未完成负载，因为负载之间是独立的。而fully compiled的probe，因为在探测bucket时可能被搁浅导致无法提前进行推测。因为如果while（pos=V.next[pos]）为false则表示没有collisions，而true的时候表示存在碰撞，则CPU则被阻塞在while循环中.由于while的结果不确定，所以不能提前分支预测。这就导致完全编译的hashing计算比vectorized hashing要慢四倍。类似的情况可以参考论文：Improving Hash Join Performance through Prefetching

figure5的左图表示，随着hash table size增加，性能裂化。这是因为hash table越大，cache miss和TLB miss增大。列存比行存 achieve less locality。中间的图表示随着命中率增加，probe的tuple数量也会增加，耗时自然也会增加。右图表示随着桶链长度增加，fully complied NSM由于缺少parallel memory access，受影响最大。在所有的情况下，partially complied NSM表现最好，因为它的multi-column

fetching （多个条件的checking/hashing）和在lookup，fetching和chain-following的并行内存访问。

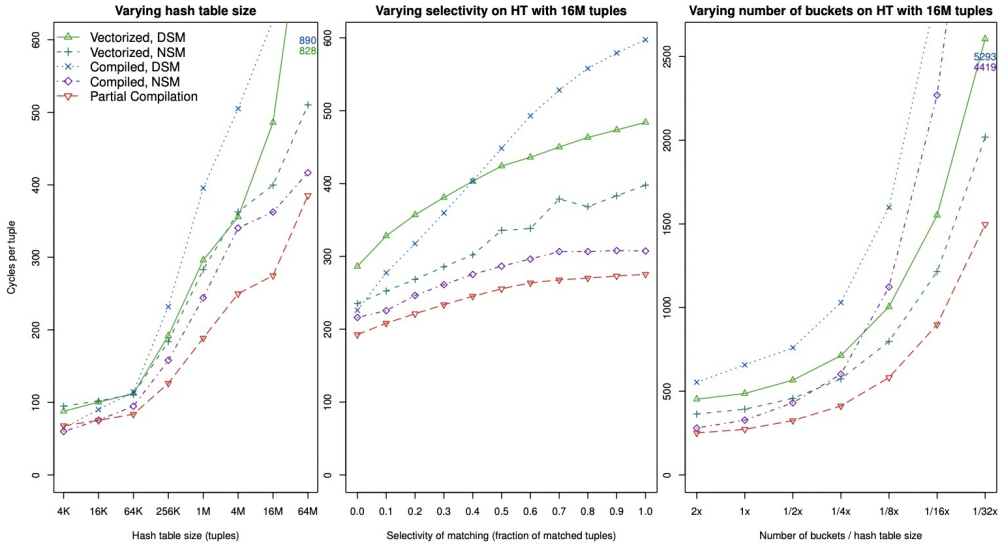

Figure 5: Hash table probing. Left: different sizes of the hash table value space V. Middle: different match rates of the probe. Right: different sizes of the bucket array B w.r.t. value space V (different chain lengths).

知乎 @AwakeLiw

6. CONCLUSION

目前大多数使用iterator model的数据库大多实现了向量化以及编译执行，这篇文章也是CLICKHOUSE的中文手册中推荐的一篇论文。作者建议

1. 不要在已有的a-tuple-a-time的API上修改，并且在这种修改中编译技术带来的提升不大
2. 应该将两者结合起来以获得更好的性能，向量化比loop-compilation更优因为（i）SIMD对齐，(ii)避免分支预测 （iii) 并行内存访问。
3. Also, it shows that vectorized execution, which is an evolution of the iterator model, thanks to enhancing it with compilation further evolves into an even more efficient and more flexible solution without making dramatic changes to the DBMS architecture： 向量化是基础，即时编译则是锦上添花。

发布于 2021-07-01 19:10
「真诚赞赏，手留余香」
赞赏