Vectorization vs Compilation In Query Execution

汇报人：朱道冰

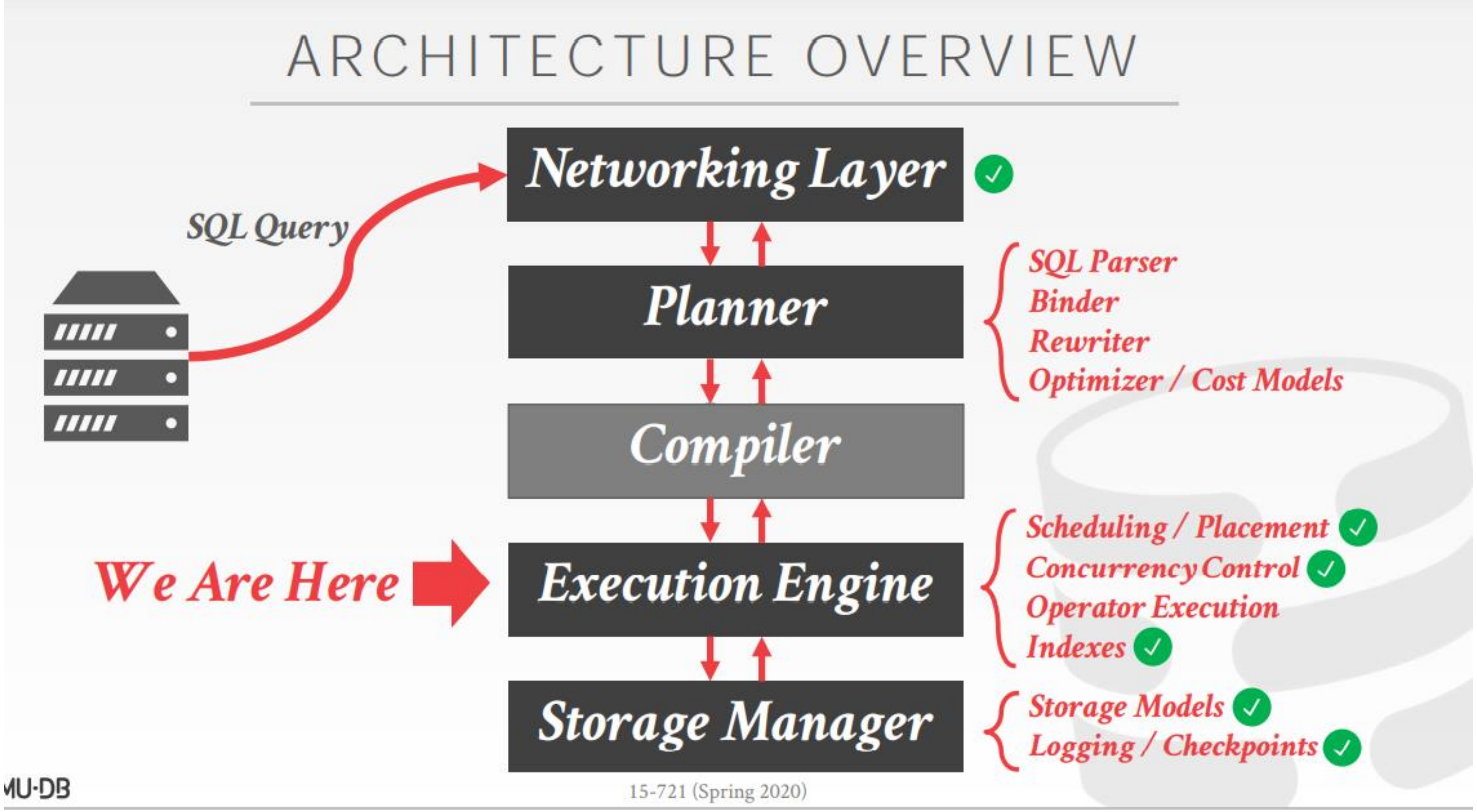
分布式存储与计算实验室

目录

01 / Execution背景

01 Execution背景

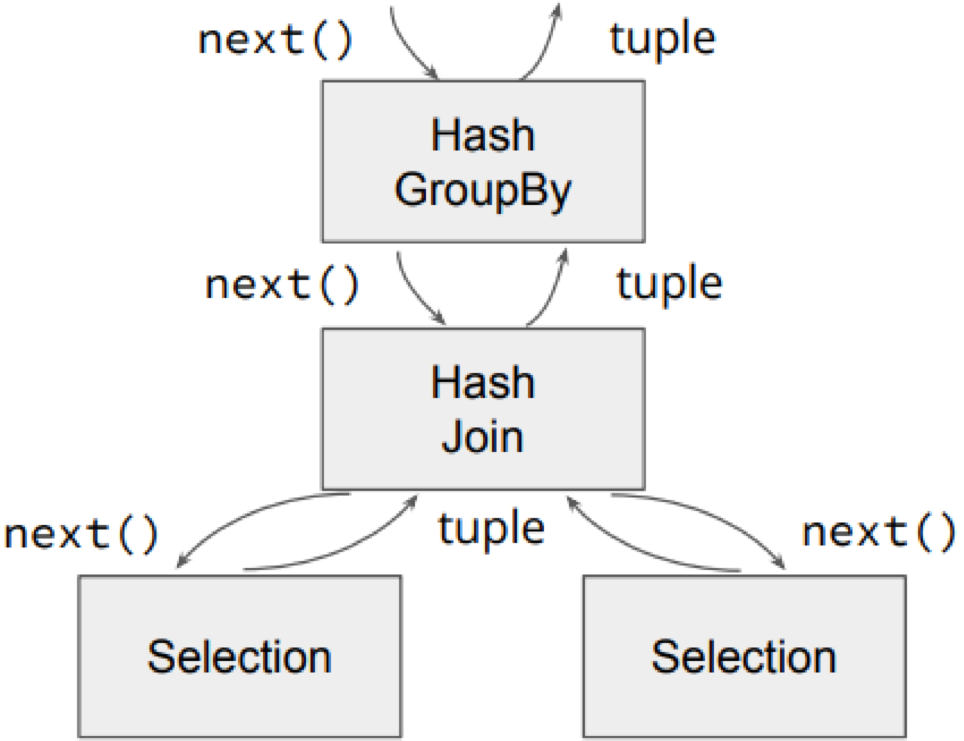
1.1 Execution所在的位置



Execution背景

1.2 传统的Execution的方案

1. Volcano
    1. Open-Next-Close
    2. Tuple-at-a-time
    3. Pull-based model



Execution背景

1.5 分享背景

1. 主要针对OLAP负载

2. In-Memory

3. 内存中以列存的形式进行计算

02 / Vectorization

Vectorization - MonetDB/X100

2.1 为何要Vectorization

1. CPU的发展符合摩尔定律
   1. 制程工艺的提升
   2. Pipeline技术

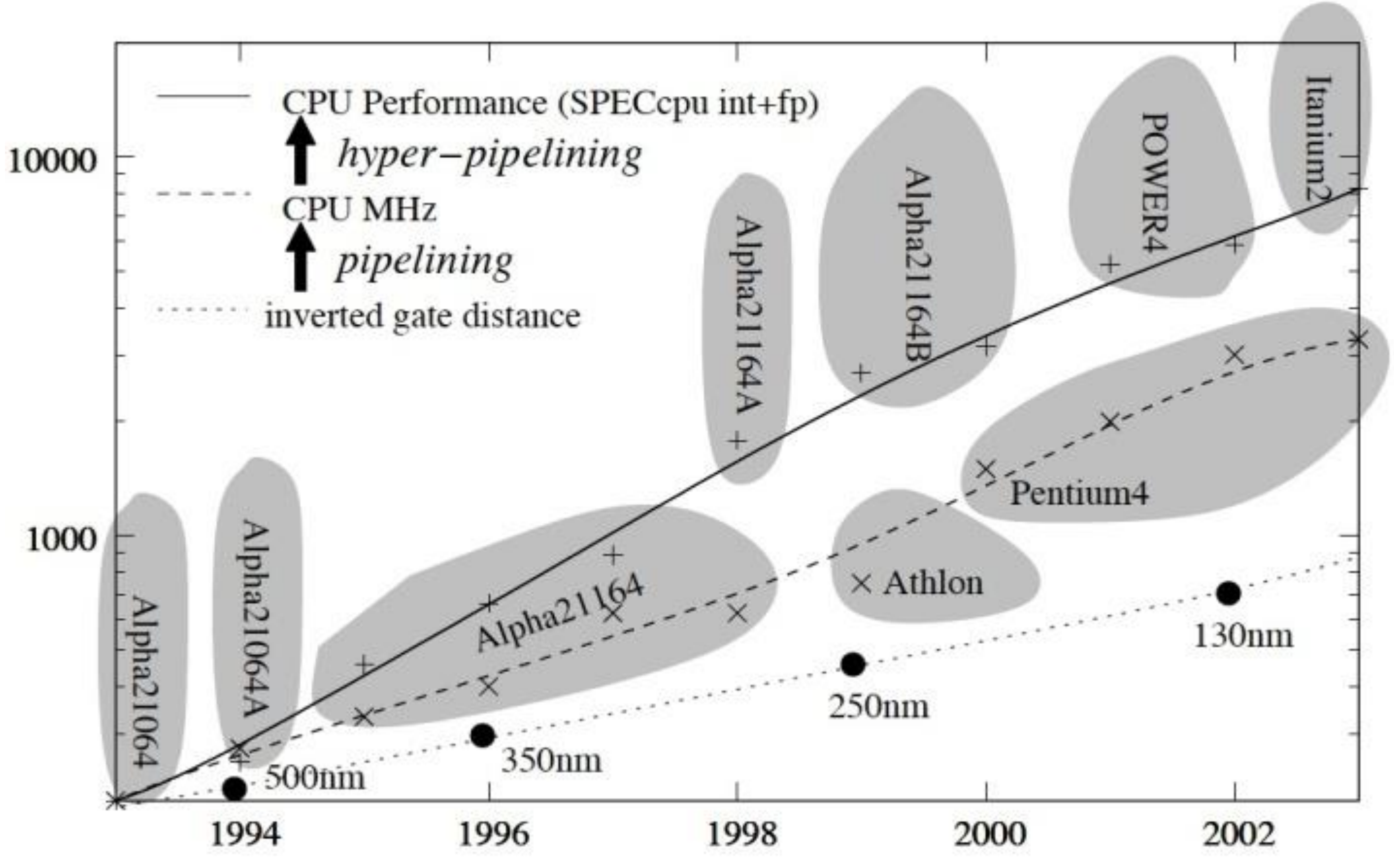


Figure 1: A Decade of CPU Performance

Vectorization - MonetDB/X100

2.1 为何要Vectorization

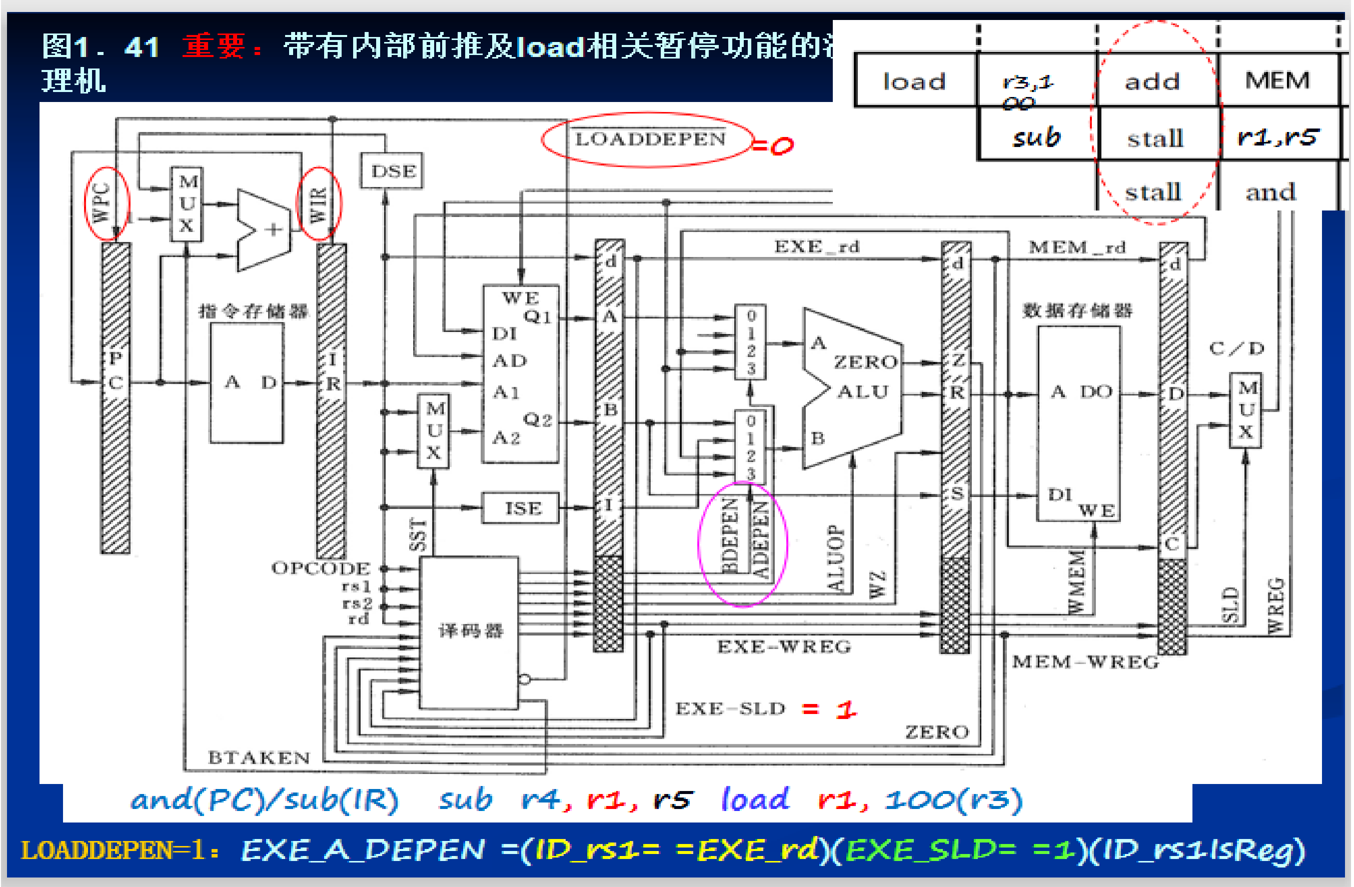
2. Pipeline的问题

   1. 不同Stage指令之间如果存在数据依赖不能交换顺序

   2. 分支预测错误会打断流水线

      流水线会对下一条指令进行预读和操作，如果发现下一条要执行的指令不是这一条需要回滚这条指令，这种情况打断了cpu pipeline。

3. 只有避免这些情况才能更充分发挥cpu的pipeline能力。



Vectorization - MonetDB/X100

2.2 Volcano架构的问题分析

1. 执行的查询语句 Tcp-H query 1 特征分析
   1. 功能：从lineitem表查询一定时间范围内按l_returnflag和l_linestatus分组的统计信息。
   2. 特征
      1. 谓词过滤性很差，5.9M/6M符合条件
      2. Group By的分组很少，只有4组，可以直接使用hash的方式进行aggregation。

```
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 '1998-09-02'
GROUP BY l_returnflag, l_linestatus
```

Figure 3: TPC-H Query 1

Vectorization - MonetDB/X100

2.2 Volcano架构的问题分析

1. Volcano模型不能很好地发挥CPU的性能(**MySQL**)
    1. Tuple-at-a-time 函数调用导致函数调用太多，而且在现实中往往是虚函数的调用，开销比普通函数的调用更大。 右图中mysql在执行过程中只有10%的cpu时间在执行coputaion。

cum.	excl.	calls	ins.	IPC	function
11.9	11.9	846M	6	0.64	ut_fold_ulint_pair
20.4	8.5	0.15M	27K	0.71	ut_fold_binary
26.2	5.8	77M	37	0.85	memcpy
29.3	**3.1**	**23M**	**64**	**0.88**	**Item_sum_sum::update_field**
32.3	3.0	6M	247	0.83	row_search_for_mysql
35.2	**2.9**	**17M**	**79**	**0.70**	**Item_sum_avg::update_field**
37.8	2.6	108M	11	0.60	rec_get_bit_field_1
40.3	2.5	6M	213	0.61	row_sel_store_mysql_rec
42.7	2.4	48M	25	0.52	rec_get_nth_field
45.1	2.4	60	19M	0.69	ha_print_info
47.5	2.4	5.9M	195	1.08	end_update
49.6	2.1	11M	89	0.98	field_conv
51.6	2.0	5.9M	16	0.77	Field_float::val_real
53.4	1.8	5.9M	14	1.07	Item_field::val
54.9	1.5	42M	17	0.51	row_sel_field_store_in_mysql..
56.3	1.4	36M	18	0.76	buf_frame_align
57.6	**1.3**	**17M**	**38**	**0.80**	**Item_func_mul::val**
59.0	1.4	25M	25	0.62	pthread_mutex_unlock
60.2	1.2	206M	2	0.75	hash_get_nth_cell
61.4	1.2	25M	21	0.65	mutex_test_and_set
62.4	1.0	102M	4	0.62	rec_get_1byte_offs_flag
63.4	1.0	53M	9	0.58	rec_1_get_field_start_offs
64.3	0.9	42M	11	0.65	rec_get_nth_field_extern_bit
65.3	**1.0**	**11M**	**38**	**0.80**	**Item_func_minus::val**
65.8	**0.5**	**5.9M**	**38**	**0.80**	**Item_func_plus::val**

Table 2: MySQL gprof trace of TPC-H Q1: +,-,*,SUM,AVG takes <10%, low IPC of 0.7

02 Vectorization - MonetDB/X100

2.2 Volcano架构的问题分析

1. Volcano模型不能很好地发挥CPU的性能(**MySQL**)
    2. 不能利用loop pipeline优化
        1. Item_func_plus::val
            1. Ins 38
            2. IPC 0.8 (instructions per cycle)
            3. Total cycle: 38/0.8=49=20(操作) + 29(函数调用)

```
operation +(double src1, double src2)

LOAD src1,reg1
LOAD src2,reg2
ADD reg1,reg2,reg3
STOR dst,reg3
```

3. tuple不能常驻寄存器和cache,需要**物化**到内存中

    next函数递归调用，下层节点给上层节点返回数据时需要**物化**到内存再传给调用的父亲operator

4. tuple-at-a-time 不好利用SIMD指令

Vectorization - MonetDB/X100

2.2 Volcano架构的问题分析 总结

1. 缺点
   1. 函数调用开销大
   2. Pipeline不友好
      1. Tuple-at-a-time 不好进行loop pipeline优化
      2. 虚函数打断pipeline
   3. Tuple不能常驻寄存器和内存
   4. SIMD不友好，不能利用现代CPU的SIMD特性

02

Vectorization - MonetDB/X100

2.3 column-at-a-time MonetDB/MIL

1. Column-at-a-time向量化的极端
    1. 向量化后计算的操作时间栈了99%。
    2. SF=0.001 TPC-H的表很小，所有数据能放入cpu cache里面，最高的处理带宽高达1.5GB/s。
    3. SF=1时，数据不能全部放入cpu cache里面，执行速度受限于memory 带宽，最多500MB/s。
    4. 这种模型需要在各种计算之间对全列数据做物化，虽然避免了大部分的interpret的成本，但也在执行中引入了大量的memory IO，速度严重受限于memory IO bandwidth，从而影响了CPU的执行效率。

SF=1		SF=0.001		tot	res	(BW = MB/s)
ms	BW	us	BW	MB	size	**MIL statement**
127	352	150	305	45	5.9M	s0 := select(l_shipdate).mark
134	505	113	608	68	5.9M	s1 := join(s0,l_returnflag)
134	506	113	608	68	5.9M	s2 := join(s0,l_linestatus)
235	483	129	887	114	5.9M	s3 := join(s0,l_extprice)
233	488	130	881	114	5.9M	s4 := join(s0,l_discount)
232	489	127	901	114	5.9M	s5 := join(s0,l_tax)
134	507	104	660	68	5.9M	s6 := join(s0,l_quantity)
290	155	324	141	45	5.9M	s7 := group(s1)
329	136	368	124	45	5.9M	s8 := group(s7,s2)
0	0	0	0	0	4	s9 := unique(s8.mirror)
206	440	60	1527	91	5.9M	r0 := [+](1.0,s5)
210	432	51	1796	91	5.9M	r1 := [-](1.0,s4)
274	498	83	1655	137	5.9M	r2 := [*](s3,r1)
274	499	84	1653	137	5.9M	r3 := [*](s12,r0)
165	271	121	378	45	4	r4 := {sum}(r3,s8,s9)
165	271	125	366	45	4	r5 := {sum}(r2,s8,s9)
163	275	128	357	45	4	r6 := {sum}(s3,s8,s9)
163	275	128	357	45	4	r7 := {sum}(s4,s8,s9)
144	151	107	214	22	4	r8 := {sum}(s6,s8,s9)
112	196	145	157	22	4	r9 := {count}(s7,s8,s9)
3724		2327		**TOTAL**		

Table 3: MonetDB/MIL trace of TPC-H Query 1

Vectorization - MonetDB/X100

2.4 MonetDB/X100

对tuple-at-a-time和column-at-a-time的折衷

1. Cache
   1. 利用Volcano style的vectorized执行，vector是一个足够小的基本处理单元，可以fit in cache中，减少与memory的交互，提高效率
   2. 函数调用次数减少，load/store开销也被均摊
   3. 需要实现制定type的vector primitives
      1. 通过模板的方式实现

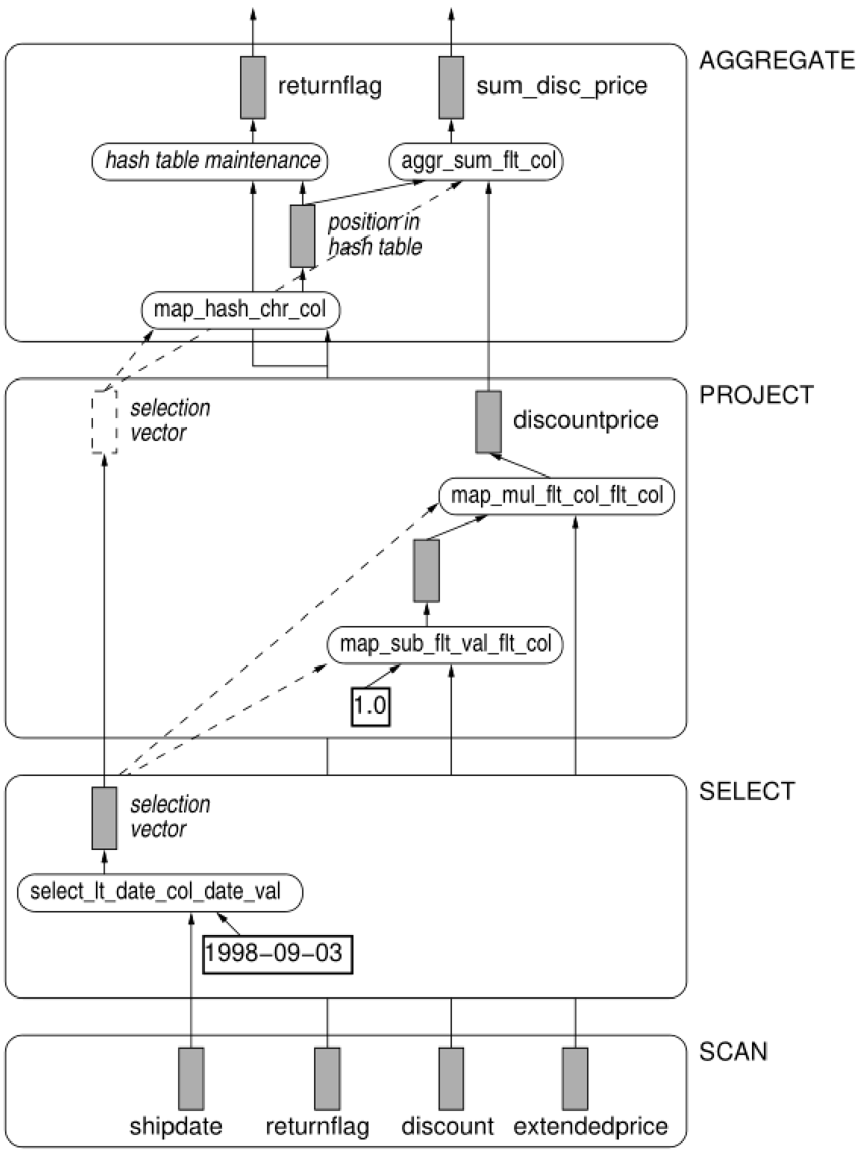


Figure 6: Execution scheme of a simplified TPC-H Query 1 in MonetDB/X100

Vectorization - MonetDB/X100

2.4 MonetDB/X100

1. CPU
   1. 由于每个vector是针对一列的切分chunk，vectorized primitives符合loop-pipeling的优化条件，可以重复利用CPU的并行流水线。

   2. 针对复杂表达式，通过对vectorized primitives做组合，进一步提高执行效率。

   load -> exec1 -> store

   load -> exec1 -> reg -> exec2 -> store

```
map_plus_double_col_double_col(int n,
  double*__restrict__ res,
  double*__restrict__ col1, double*__restrict__ col2,
  int*__restrict__ sel)
{
  if (sel) {
    for(int j=0;j<n; j++) {
      int i = sel[j];
      res[i] = col1[i] + col2[i];
    }
  } else {
    for(int i=0;i<n; i++)
      res[i] = col1[i] + col2[i];
} }
```

```
/(square(-(double*, double*)), double*)
```

Vectorization - MonetDB/X100

2.5 MonetDB/X100性能

1. Primitives需要的时间周期比较少->loop pipeline
2. Vector能放入cpu cache内，所以计算带宽非常高，最高达到7.5GB/s（同样的操作在MIL只有500MB/s）

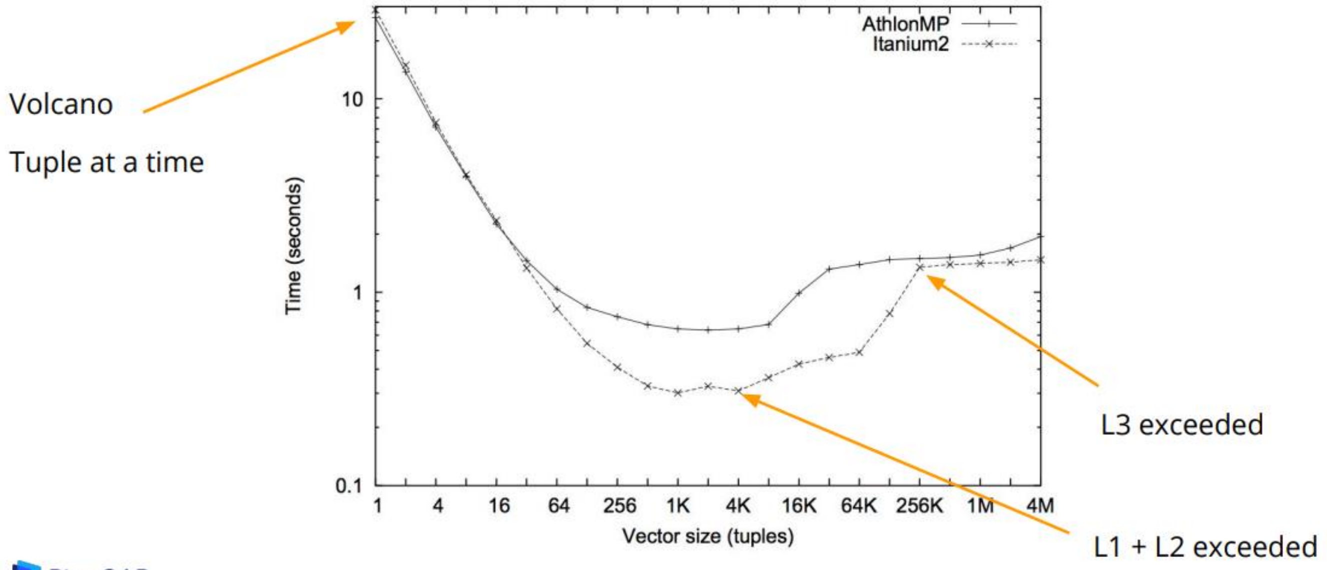
input count	total MB	time (us)	BW MB/s	avg. cycles	**X100 primitive**
6M	30	8518	3521	1.9	map_fetch_uchr_col_flt_col
6M	30	8360	3588	1.9	map_fetch_uchr_col_flt_col
6M	30	8145	3683	1.9	map_fetch_uchr_col_flt_col
6M	35.5	13307	2667	3.0	select_lt_usht_col_usht_val
5.9M	47	10039	4681	2.3	map_sub_flt_val_flt_col
5.9M	71	9385	7565	2.2	map_mul_flt_col_flt_col
5.9M	71	9248	7677	2.1	map_mul_flt_col_flt_col
5.9M	47	10254	4583	2.4	map_add_flt_val_flt_col
5.9M	35.5	13052	2719	3.0	map_uidx_uchr_col
5.9M	53	14712	3602	3.4	map_directgrp_uidx_col_uchr_col
5.9M	71	28058	2530	6.5	aggr_sum_flt_col_uidx_col
5.9M	71	28598	2482	6.6	aggr_sum_flt_col_uidx_col
5.9M	71	27243	2606	6.3	aggr_sum_flt_col_uidx_col
5.9M	71	26603	2668	6.1	aggr_sum_flt_col_uidx_col
5.9M	71	27404	2590	6.3	aggr_sum_flt_col_uidx_col
5.9M	47	18738	2508	4.3	aggr_count_uidx_col
					X100 operator
0		3978			Scan
6M		10970			Fetch1Join(ENUM)
6M		10712			Fetch1Join(ENUM)
6M		10656			Fetch1Join(ENUM)
6M		15302			Select
5.9M		236443			Aggr(DIRECT)

Table 5: MonetDB/X100 performance trace of TPC-H Query 1 (Itanium2, SF=1)

Vectorization - MonetDB/X100

2.5 MonetDB/X100性能

1. 与vector size的关系
   1. 1 (volcano) -> L1+L2 cache size大小，执行时间减少。
   2. L1+L2->L3 执行时间增大，L3速度比L12慢。
   3. 超过L3cache大小后，将所有中间结果物化到内存，性能和MIL方案相似。

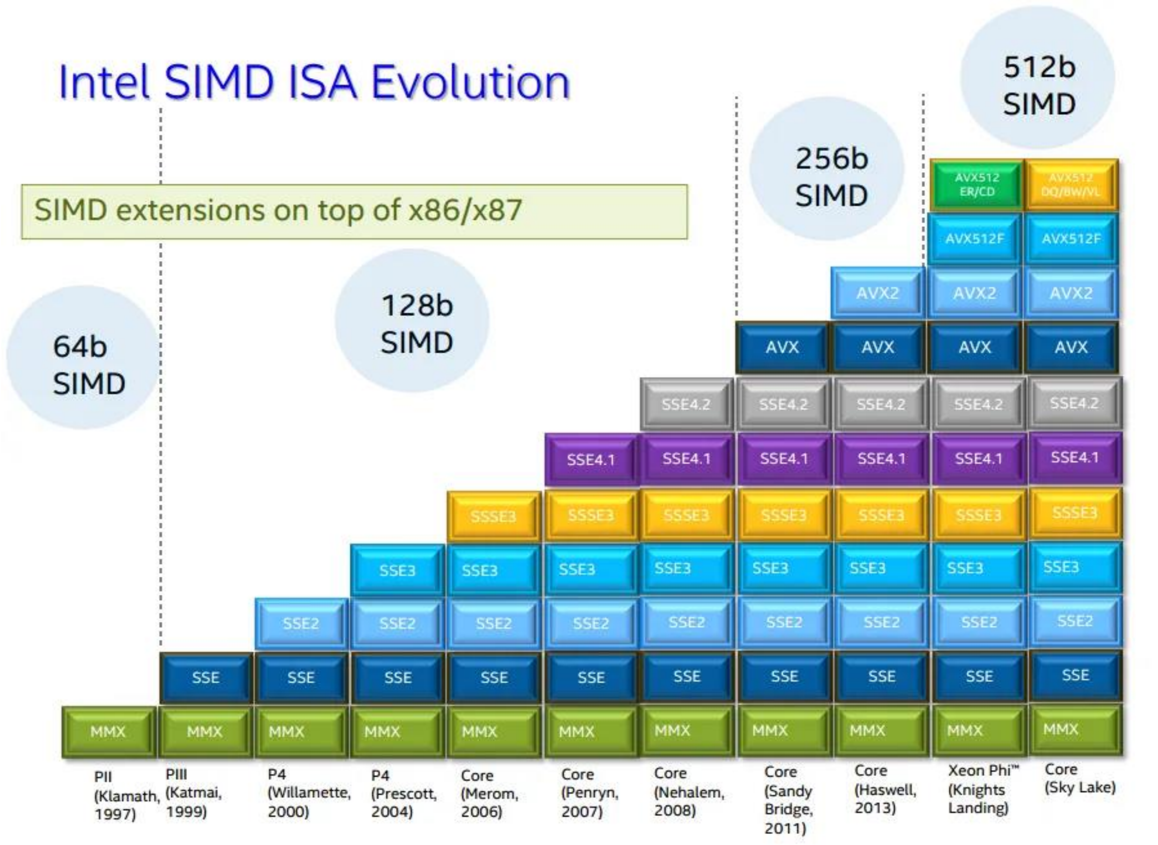


PingCAP

Vectorization-SIMD

2.1 SIMD

1. 指单指令多数据流技术，可用一组指令对多组数据通进行并行操作。SIMD指令可以在一个控制器上控制同时多个平行的处理微元，一次指令运算执行多个数据流，这样在很多时候可以提高程序的运算速度。
2. 需要特殊的cpu硬件支持，右图中的bit即为需要的寄存器。
3. AVX512 就可以同时处理8个int64的数据。



Vectorization-SIMD

2.2 基本操作指令

1. Selective load
    1. 根据mask从Memory->Vector(SIMD寄存器)
2. Selective store
    1. 根据mask从Vector(SIMD寄存器)->Memory

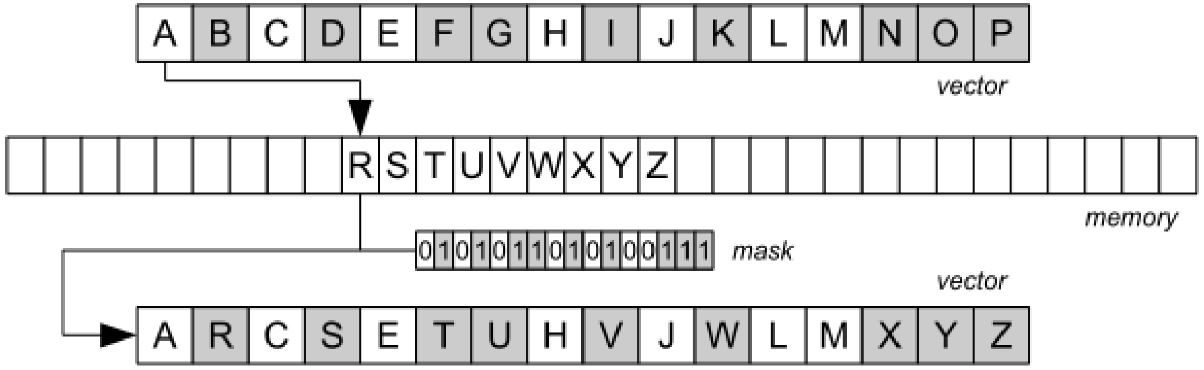


Figure 2: Selective load operation

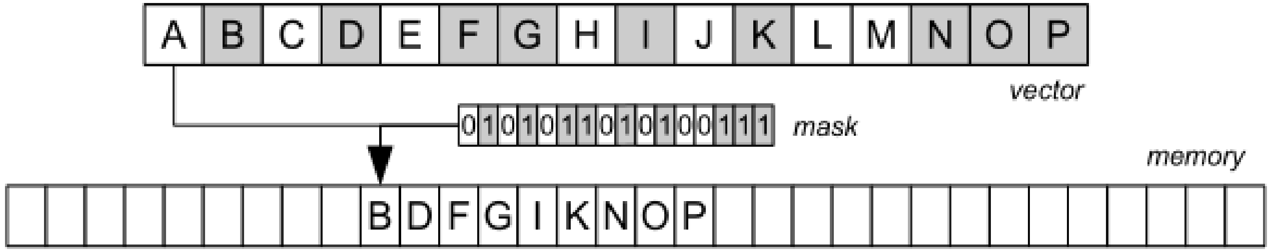


Figure 1: Selective store operation

02 Vectorization-SIMD

2.2 基本操作指令

1. Gather operation
   1. 根据index vector从Memory->Vector(SIMD寄存器)

2. Scatter operation
   1. 根据index vector从vector(SIMD)->Memory

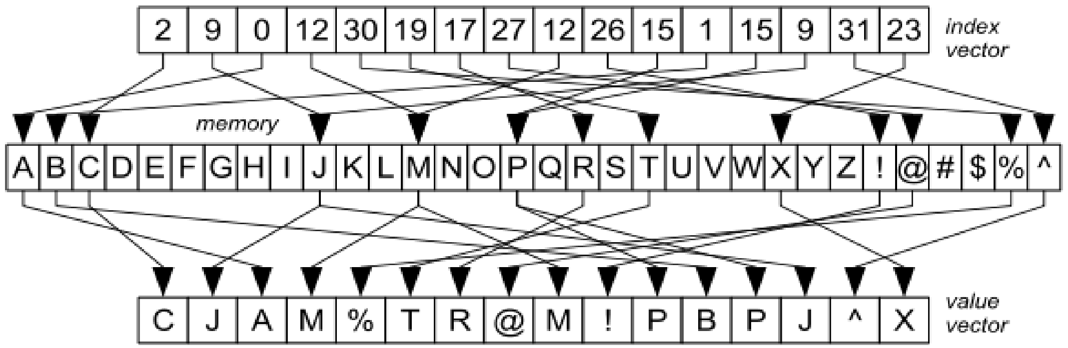


Figure 3: Gather operation

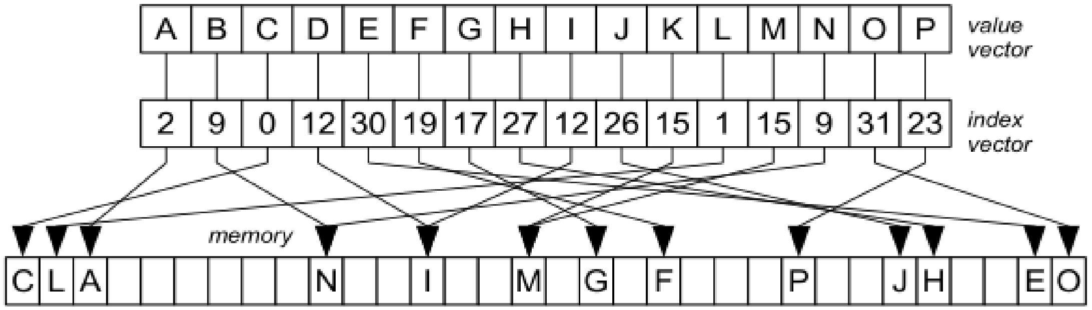


Figure 4: Scatter operation

Vectorization-SIMD

2.3 数据库操作

1. SELECTION SCANS Scalar(标量)

Algorithm 1 Selection Scan (Scalar - Branching)

$j \leftarrow 0$ ▷ *output index*
for $i \leftarrow 0$ **to** $|T_{keys\_in}| - 1$ **do**
  $k \leftarrow T_{keys\_in}[i]$ ▷ *access key columns*
  if $(k \geq k_{lower})$ && $(k \leq k_{upper})$ **then** ▷ *short circuit and*
    $T_{payloads\_out}[j] \leftarrow T_{payloads\_in}[i]$ ▷ *copy all columns*
    $T_{keys\_out}[j] \leftarrow k$
    $j \leftarrow j + 1$
  end if
end for

Algorithm 2 Selection Scan (Scalar - Branchless)

$j \leftarrow 0$ ▷ *output index*
for $i \leftarrow 0$ **to** $|T_{keys\_in}| - 1$ **do**
  $k \leftarrow T_{keys\_in}[i]$ ▷ *copy all columns*
  $T_{payloads\_out}[j] \leftarrow T_{payloads\_in}[i]$
  $T_{keys\_out}[j] \leftarrow k$
  $m \leftarrow (k \geq k_{lower}\ ?\ 1 : 0)\ \&\ (k \leq k_{upper}\ ?\ 1 : 0)$
  $j \leftarrow j + m$ ▷ *if-then-else expressions use conditional ...*
end for ▷ *... flags to update the index without branching*

02

Vectorization-SIMD

2.3 数据库操作

1. SELECTION SCANS

Vector(向量)

Algorithm 3 Selection Scan (Vector)

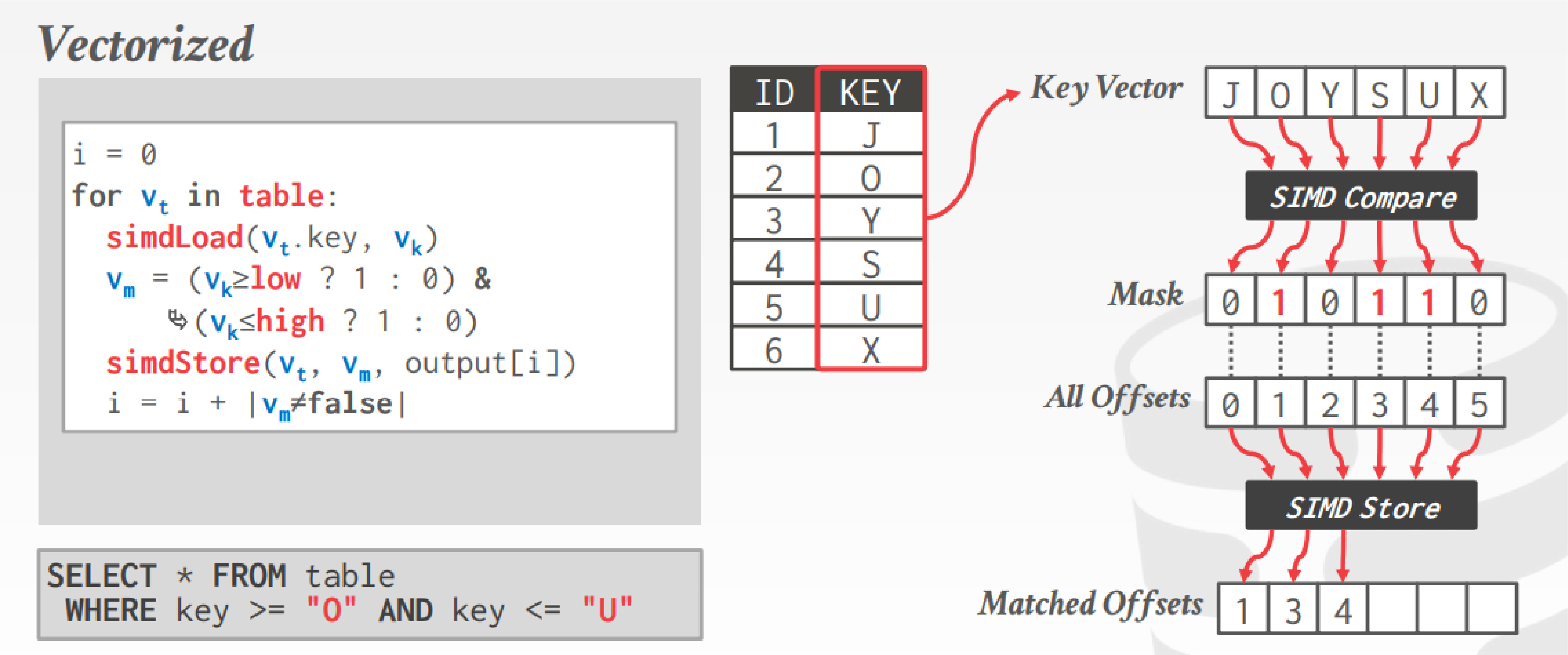
$i, j, l \leftarrow 0$ ▷ *input, output, and buffer indexes*
$\vec{r} \leftarrow \{0, 1, 2, 3, ..., W-1\}$ ▷ *input indexes in vector*
for $i \leftarrow 0$ **to** $|T_{keys\_in}| - 1$ **step** W **do** ▷ *# of vector lanes*
  $\vec{k} \leftarrow T_{keys\_in}[i]$ ▷ *load vectors of key columns*
  $m \leftarrow (\vec{k} \geq k_{lower})\ \&\ (\vec{k} \leq k_{upper})$ ▷ *predicates to mask*
  if $m \neq$ false **then** ▷ *optional branch*
    $B[l] \leftarrow_m \vec{r}$ ▷ *selectively store indexes*
    $l \leftarrow l + |m|$ ▷ *update buffer index*
    if $l > |B| - W$ **then** ▷ *flush buffer*
      for $b \leftarrow 0$ **to** $|B| - W$ **step** W **do**
        $\vec{p} \leftarrow B[b]$ ▷ *load input indexes*
        $\vec{k} \leftarrow T_{keys\_in}[\vec{p}]$ ▷ *dereference values*
        $\vec{v} \leftarrow T_{payloads\_in}[\vec{p}]$
        $T_{keys\_out}[b + j] \leftarrow \vec{k}$ ▷ *flush to output with ...*
        $T_{payloads\_out}[b + j] \leftarrow \vec{v}$ ▷ *... streaming stores*
      end for
      $\vec{p} \leftarrow B[|B| - W]$ ▷ *move overflow ...*
      $B[0] \leftarrow \vec{p}$ ▷ *... indexes to start*
      $j \leftarrow j + |B| - W$ ▷ *update output index*
      $l \leftarrow l - |B| + W$ ▷ *update buffer index*
    end if
  end if
  $\vec{r} \leftarrow \vec{r} + W$ ▷ *update index vector*
end for ▷ *flush last items after the loop*

<- 读取对应column W个值
<- 适用SIMD compare并生成match mask
<-适用selectively store把符合条件的index存储到结果Buffer里面
<-Flush Buffer 这边Buffer不是重点
<-下W个数据

02 Vectorization-SIMD

2.3 数据库操作

1. SELECTION SCANS Vector(向量) example



02

Vectorization-SIMD

2.3 数据库操作

1. Linear Probing Hash – Probe (探测) Scalar

Algorithm 4 Linear Probing - Probe (Scalar)

$j \leftarrow 0$ ▷ *output index*
for $i \leftarrow 0$ **to** $|S_{keys}| - 1$ **do** ▷ *outer (probing) relation* <-对每个要probe的seek key
  $k \leftarrow S_{keys}[i]$ <-取用于hash的key和负载列
  $v \leftarrow S_{payloads}[i]$
  $h \leftarrow (k \cdot f) \times\!\uparrow |T|$ ▷ "$\times\uparrow$": *multiply & keep upper half* <-计算hash值
  while $T_{keys}[h] \neq k_{empty}$ **do** ▷ *until empty bucket* <-开放寻址hash
    if $k = T_{keys}[h]$ **then**
      $RS_{R\_payloads}[j] \leftarrow T_{payloads}[h]$ ▷ *inner payloads*
      $RS_{S\_payloads}[j] \leftarrow v$ ▷ *outer payloads*
      $RS_{keys}[j] \leftarrow k$ ▷ *join keys*
      $j \leftarrow j + 1$
    end if
    $h \leftarrow h + 1$ ▷ *next bucket* <-冲突的话下一个位置+1
    if $h = |T|$ **then** ▷ *reset if last bucket*
      $h \leftarrow 0$
    end if
  end while
end for

02 Vectorization-SIMD

2.3 数据库操作 1. Linear Probing Hash – Probe（探测） Vector

Algorithm 5 Linear Probing - Probe (Vector)

$i, j \leftarrow 0$ ▷ *input & output indexes (scalar register)*
$\vec{o} \leftarrow 0$ ▷ *linear probing offsets (vector register)*
$m \leftarrow$ true ▷ *boolean vector register*
while $i + W \leq |S_{keys\_in}|$ **do** ▷ *W: # of vector lanes*
  $\vec{k} \leftarrow_m S_{keys}[i]$ ▷ *selectively load input tuples*
  $\vec{v} \leftarrow_m S_{payloads}[i]$
  $i \leftarrow i + |m|$
  $\vec{h} \leftarrow (\vec{k} \cdot f) \times \uparrow |T|$ ▷ *multiplicative hashing*
  $\vec{h} \leftarrow \vec{h} + \vec{o}$ ▷ *add offsets & fix overflows*
  $\vec{h} \leftarrow (\vec{h} < |T|) \ ? \ \vec{h} : (\vec{h} - |T|)$ ▷ *"m ? $\vec{x}$: $\vec{y}$": vector blend*
  $\vec{k}_T \leftarrow T_{keys}[\vec{h}]$ ▷ *gather buckets*
  $\vec{v}_T \leftarrow T_{payloads}[\vec{h}]$
  $m \leftarrow \vec{k}_T = \vec{k}$
  $RS_{keys}[j] \leftarrow_m \vec{k}$ ▷ *selectively store matching tuples*
  $RS_{S\_payloads}[j] \leftarrow_m \vec{v}$
  $RS_{R\_payloads}[j] \leftarrow_m \vec{v}_T$
  $j \leftarrow j + |m|$
  $m \leftarrow \vec{k}_T = k_{empty}$ ▷ *discard finished tuples*
  $\vec{o} \leftarrow m \ ? \ 0 : (\vec{o} + 1)$ ▷ *increment or reset offsets*
end while

<-根据mask(一开始全为true) selectively load一批要probe 的keys

<- SIMD 计算hash函数

<- 根据hash地址，从hash表中gather buckets

<- 判断读取的key是否相等，如果相等对应位置1

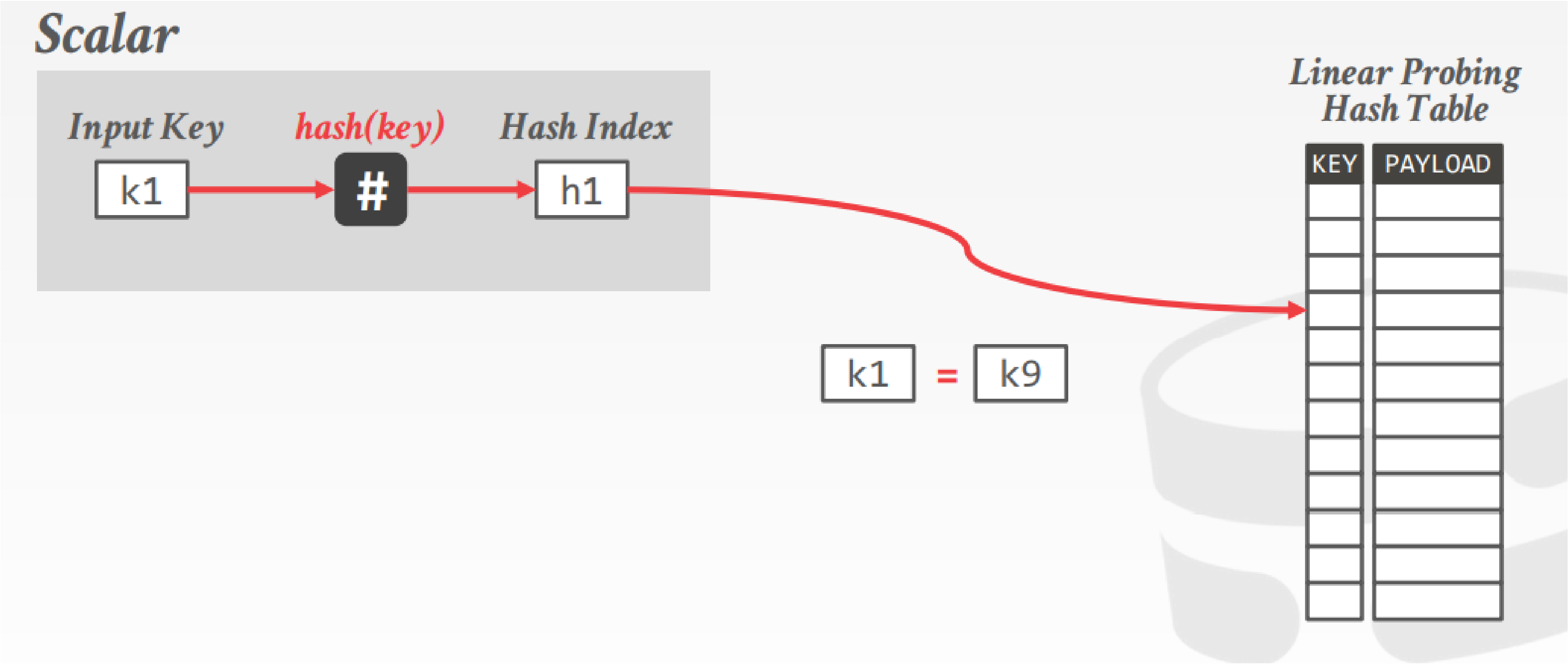
<- 使用selectively store把值存储到RS结果集里面

<-如过读取的K是empty，则结束搜索，可以读取新的key进来（selectively load），如果非空且不相同，保持key不动，通过offset开放检测下一个位置

Vectorization-SIMD

2.3 数据库操作

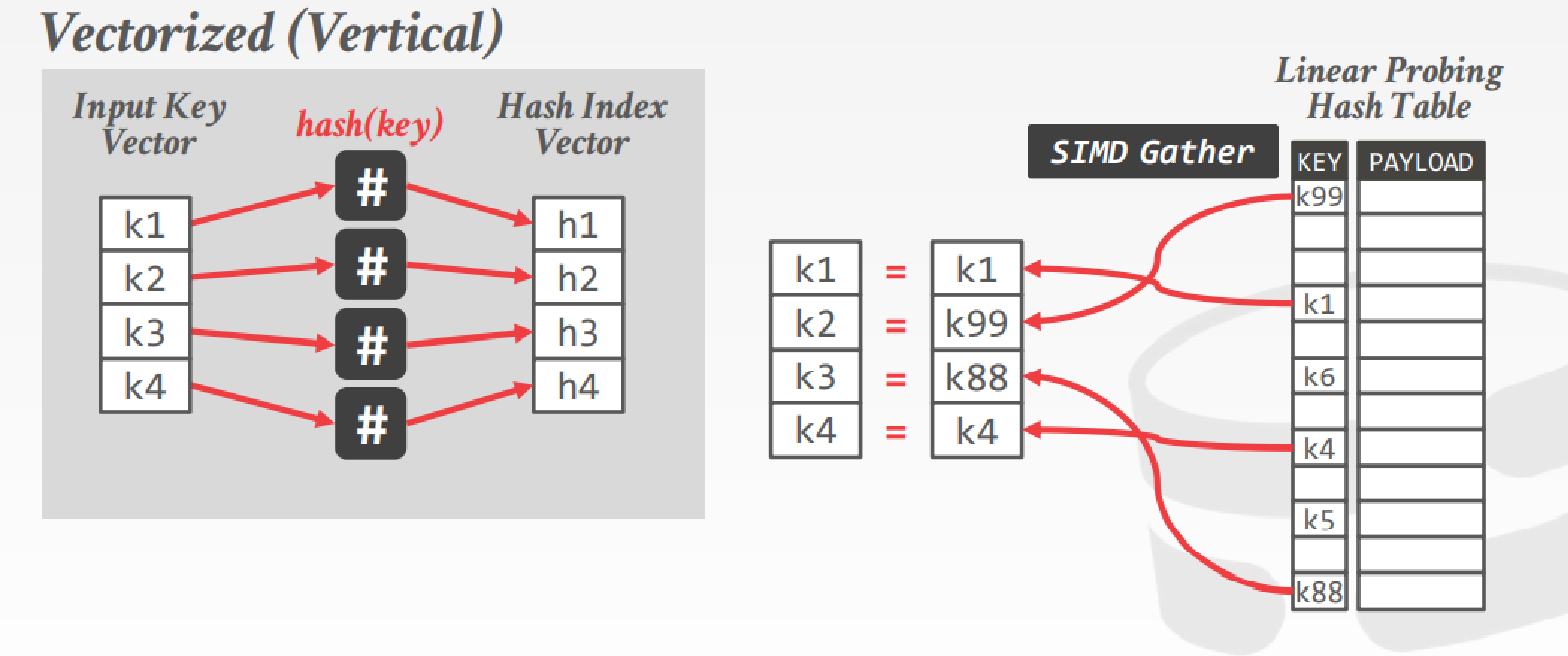
1. Linear Probing Hash – Probe (探测) Scalar Example



Vectorization-SIMD

2.3 数据库操作

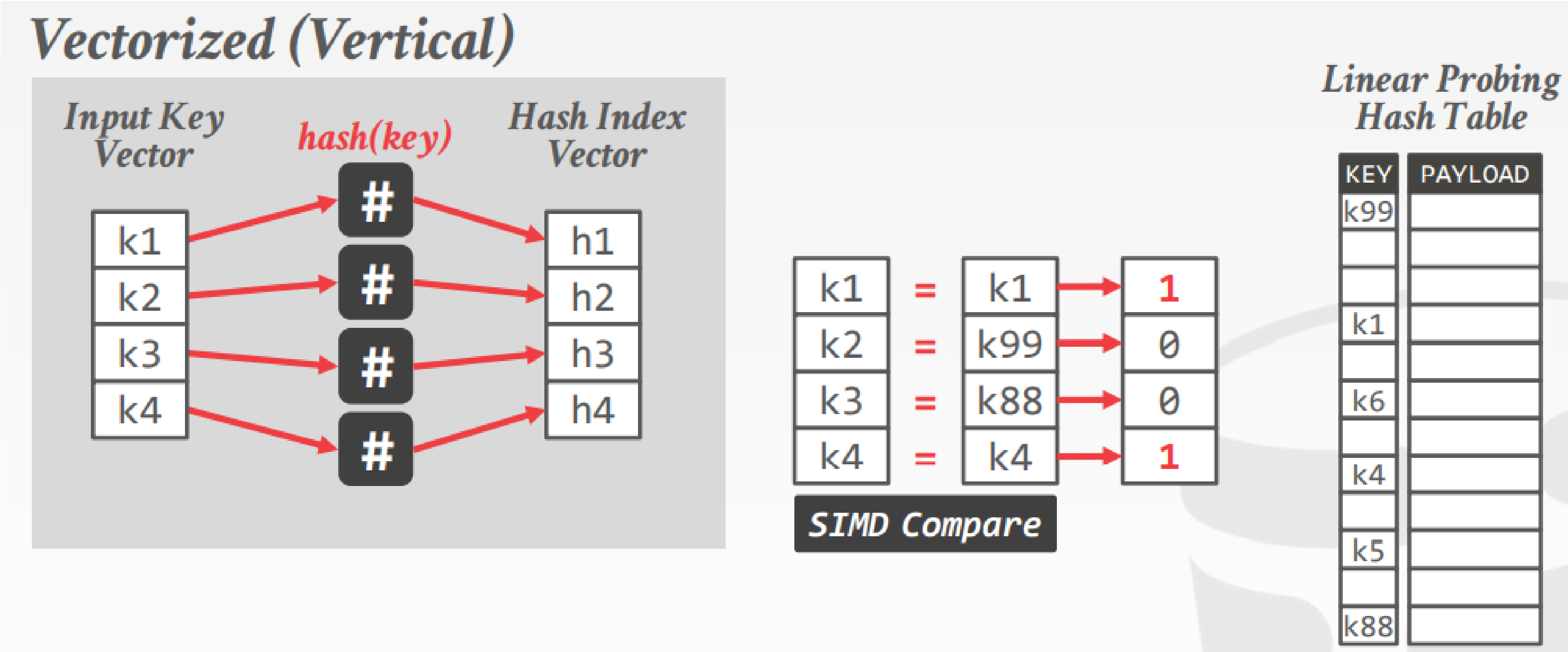
1. Linear Probing Hash – Probe (探测) Vector Example



Vectorization-SIMD

2.3 数据库操作

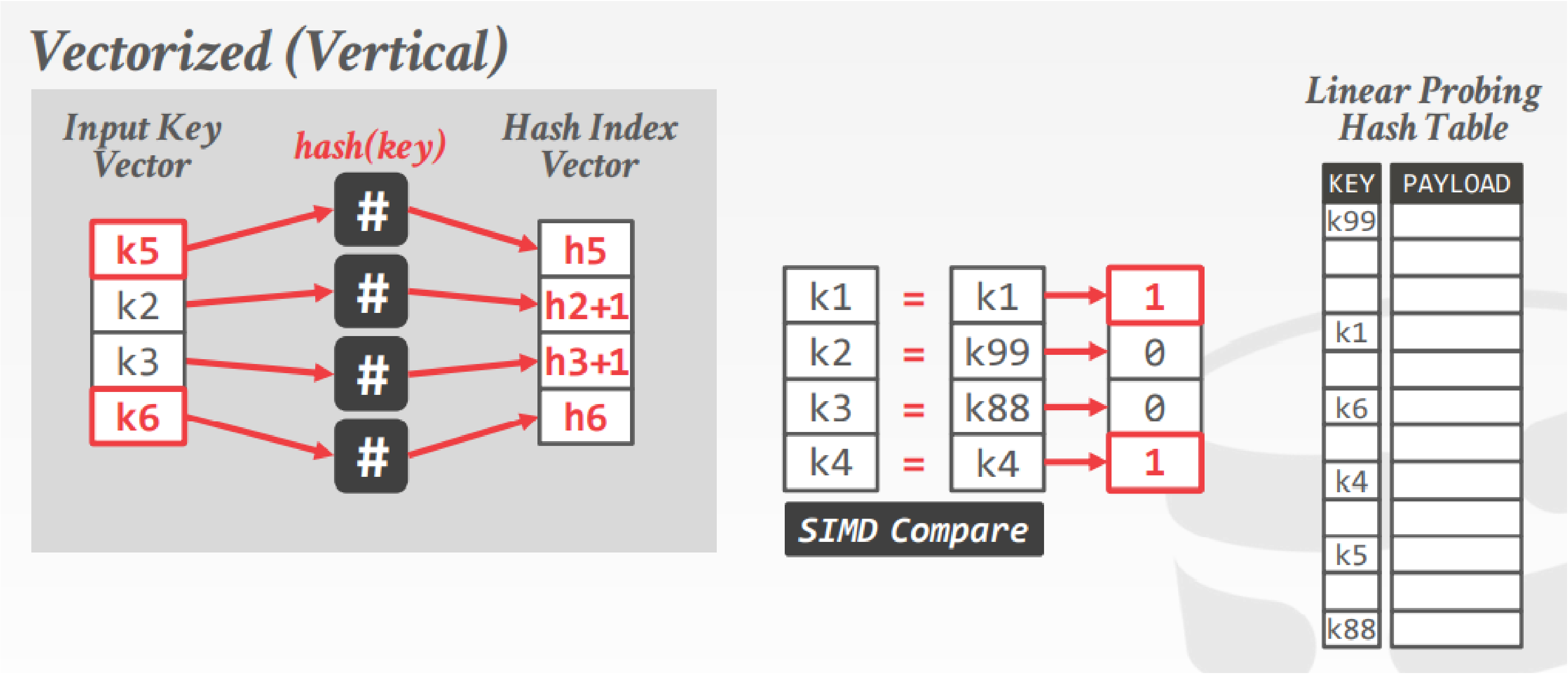
1. Linear Probing Hash – Probe (探测) Vector Example
    1. 注意：下次读取新批次key时会把SIMD的比较结果作为读取mask,如果为0会+1，探测下一个位置；如果为1的说明已经找到，为了提高处理速率可以读取其他需要探测(probe)的keys到SIMD寄存器中用于下一轮探测。



Vectorization-SIMD

2.3 数据库操作

1. Linear Probing Hash – Probe (探测) Vector Example
   1. 注意：下次读取新批次key时会把SIMD的比较结果作为读取mask,如果为0会+1，探测下一个位置；如果为1的说明已经找到，为了提高处理速率可以读取其他需要探测(probe)的keys到SIMD寄存器中用于下一轮探测。



02 Vectorization-SIMD

2.3 数据库操作

1. Linear Probing Hash –Build Scalar Example

Algorithm 6 Linear Probing - Build (Scalar)

for $i \leftarrow 0$ **to** $|R_{keys}| - 1$ **do** ▷ *inner (building) relation*
  $k \leftarrow R_{keys}[i]$
  $h \leftarrow (k \cdot f) \times \uparrow |T|$ ▷ *multiplicative hashing*
  while $T_{keys}[h] \neq k_{empty}$ **do** ▷ *until empty bucket*
    $h \leftarrow h + 1$ ▷ *next bucket*
    if $h = |T|$ **then**
      $h \leftarrow 0$ ▷ *reset if last*
    end if
  end while
  $T_{keys}[h] \leftarrow k$ ▷ *set empty bucket*
  $T_{payloads}[h] \leftarrow R_{payloads}[i]$
end for

02

Vectorization-SIMD

2.3 数据库操作 1. Linear Probing Hash – Build Vector

Algorithm 7 Linear Probing - Build (Vector)

$\vec{l} \leftarrow \{1, 2, 3, ..., W\}$ ▷ *any vector with unique values per lane*

$i, j \leftarrow 0$, $m \leftarrow$ true ▷ *input & output index & bitmask*

$\vec{o} \leftarrow 0$ ▷ *linear probing offset*

while $i + W \leq |R_{keys}|$ **do**

  $\vec{k} \leftarrow_m R_{keys}[i]$ ▷ *selectively load input tuples*

  $\vec{v} \leftarrow_m R_{payloads}[i]$

  $i \leftarrow i + |m|$

  $\vec{h} \leftarrow \vec{o} + (\vec{k} \cdot f) \times\!\uparrow |T|$ ▷ *multiplicative hashing*

  $\vec{h} \leftarrow (\vec{h} < |T|)\ ?\ \vec{h} : (\vec{h} - |T|)$ ▷ *fix overflows*

  $\vec{k}_T \leftarrow T_{keys}[\vec{h}]$ ▷ *gather buckets*

  $m \leftarrow \vec{k}_T = k_{empty}$ ▷ *find empty buckets*

  $T[\vec{h}] \leftarrow_m \vec{l}$ ▷ *detect conflicts*

  $\vec{l}_{back} \leftarrow_m T_{keys}[\vec{h}]$

  $m \leftarrow m\ \&\ (\vec{l} = \vec{l}_{back})$

  $T_{keys}[\vec{h}] \leftarrow_m \vec{k}$ ▷ *scatter to buckets ...*

  $T_{payloads}[\vec{h}] \leftarrow_m \vec{v}$ ▷ *... if not conflicting*

  $\vec{o} \leftarrow m\ ?\ 0 : (\vec{o} + 1)$ ▷ *increment or reset offsets*

end while

<- 使用selectively store把值存储到RS结果集里面

<-根据m selectively load一批要probe 的keys

<- SIMD 计算hash函数

<- 根据hash地址，从hash表中gather buckets

<-如过读取的K是empty，则不冲突，可以直接放置，下次selectively load会读取新的值；
如果非空说明冲突了则key保持不动，需要通过offset开放检测下一个位置。

02 Vectorization-SIMD

2.3 数据库操作

1. Radix Partitioning - Histogram

Algorithm 11 Radix Partitioning - Histogram

$\vec{o} \leftarrow \{0, 1, 2, 3, ..., W-1\}$

$H_{partial}[P \times W] \leftarrow 0$ ▷ *initialize replicated histograms* <- 使用selectively store把值存储到RS结果集里面

for $i \leftarrow 0$ **to** $|T_{keys\_in}| - 1$ **step** W **do**

  $\vec{k} \leftarrow T_{keys\_in}[i]$ <- 读取W个key

  $\vec{h} \leftarrow (\vec{k} << b_L) >> b_R$ ▷ *radix function* <- SIMD 计算radix函数值

  $\vec{h} \leftarrow \vec{o} + (\vec{h} \cdot W)$ ▷ *index for multiple histograms*

  $\vec{c} \leftarrow H_{partial}[\vec{h}]$ ▷ *increment W counts* <- 增加对应位置的值

  $H_{partial}[\vec{h}] \leftarrow \vec{c} + 1$

end for

<- 累加

for $i \leftarrow 0$ **to** $P - 1$ **do**

  $\vec{c} \leftarrow H_{partial}[i \cdot W]$ ▷ *load W counts of partition*

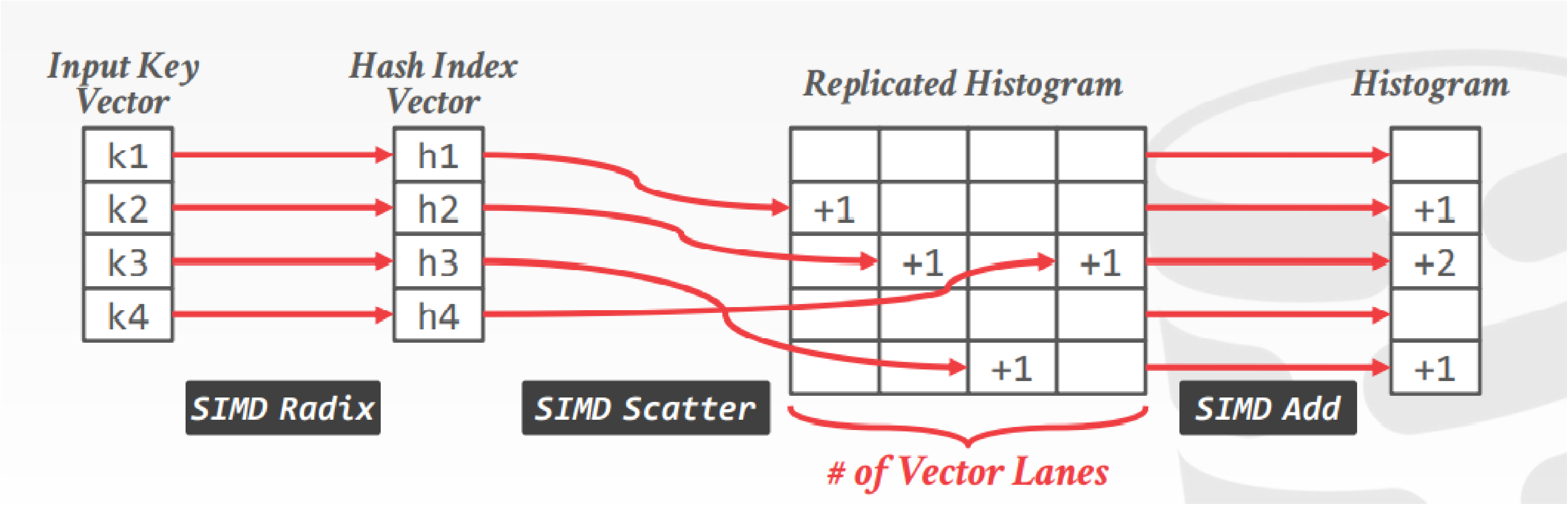
  $H[i] \leftarrow \text{sum\_across}(\vec{c})$ ▷ *reduce into single result*

end for

Vectorization-SIMD

2.3 数据库操作

1. Radix Partitioning - Histogram



Vectorization-SIMD

2.3 数据库操作

1. 论文还介绍了sort和hash join的向量化实现思路。

2. 论文附录提供了这些数据库操作更详细的代码。

3. 可以结合intel的指令手册进行学习。

   software.intel.com/sites/landingpage/IntrinsicsGuide/

4. **_mm<bit_width>_<name>_<data_type>**

C. SELECTIVE LOAD & STORE

```
void _mm512_mask_packstore_epi32(int32_t *p, // pointer
                                 __mmask16 m, // mask
                                  __m512i v) // vector
{ _mm512_mask_packstorelo_epi32(&p[0],  m, v);
  _mm512_mask_packstorehi_epi32(&p[16], m, v); }
```

```
__m512i _mm512_mask_loadunpack_epi32(__m512i v,
                                     __mmask16 m,
                                   const int32_t *p)
{ v = _mm512_mask_loadunpacklo_epi32(v, m, &p[0]);
  v = _mm512_mask_loadunpackhi_epi32(v, m, &p[16]);
  return v; }
```

D. SELECTION SCANS

```
for (i = j = k = 0;  i < tuples;  i += 16) {
  /* load key column and evaluate predicates */
  key = _mm512_load_epi32(&keys[i]);
  m = _mm512_cmpge_epi32_mask(key, mask_lower);
  m = _mm512_mask_cmple_epi32_mask(k, key, mask_upper);
  if (!_mm512_kortestz(m, m)) {     // jkzd
    /* selectively store qualifying tuple indexes */
    _mm512_mask_packstore_epi32(&rids_buf[k], m, rid);
    k += _mm_countbits_64(_mm512_mask2int(m)); }
    if (k > buf_size - 16) {
      /* flush the buffer */
      for (b = 0;  b != buf_size - 16;  b += 16) {
        ptr = _mm512_load_epi32(&rids_buf[b]);
        /* dereference column values and stream */
        key_f = _mm512_i32gather_ps(ptr, keys, 4);
        pay_f = _mm512_i32gather_ps(ptr, pays, 4);
        _mm512_storenrngo_ps(&keys_out[b + j], key_f);
        _mm512_storenrngo_ps(&pays_out[b + j], pay_f);}
      /* move extra items to the start of the buffer */
      ptr = _mm512_load_epi32(&rids_buf[b]);
      _mm512_store_epi32(&rids_buf[0], ptr);
      j += buf_size - 16;
      k -= buf_size - 16; } }
  rid = _mm512_add_epi32(rid, mask_16); }
```

03 / Compilation

Compilation Hyper Basic Compilation

3.1 手写代码

1. MonetDB性能基准(Baseline)
   1. 手写的UDF(user define function)
   2. Restrict关键字能够利用上loop pipeline

2. Hand-written 是 task specific
   1. 代码更加紧凑，往往是几个函数就把所有操作完成了，减少了函数调用。
   2. 代码具有更好的data locality，将把一个tuple读取到cpu cache后，会尽量把所有能做的computaion计算完成，尽量减少与memory的交互，以及cache misses。

```
static void tpch_query1(int n, int hi_date,
  unsigned char*__restrict__ p_returnflag,
  unsigned char*__restrict__ p_linestatus,
  double*__restrict__         p_quantity,
  double*__restrict__         p_extendedprice,
  double*__restrict__         p_discount,
  double*__restrict__         p_tax,
  int*__restrict__            p_shipdate,
  aggr_t1*__restrict__        hashtab)
{
 for(int i=0; i<n; i++) {
  if (p_shipdate[i] <= hi_date) {
    aggr_t1 *entry = hashtab +
      (p_returnflag[i]<<8) + p_linestatus[i];
    double discount = p_discount[i];
    double extprice = p_extendedprice[i];
    entry->count++;
    entry->sum_qty += p_quantity[i];
    entry->sum_disc += discount;
    entry->sum_base_price += extprice;
    entry->sum_disc_price += (extprice *= (1-discount));
    entry->sum_charge += extprice*(1-p_tax[i]);
}}}
```

Compilation Hyper Basic Compilation

3.2 为何需要Compilation

1. 向量化达不到hand-written代码的性能

2. Compilation
    1. 将执行计划转换成hand-written一样紧凑的代码

3. Aka Code Generation

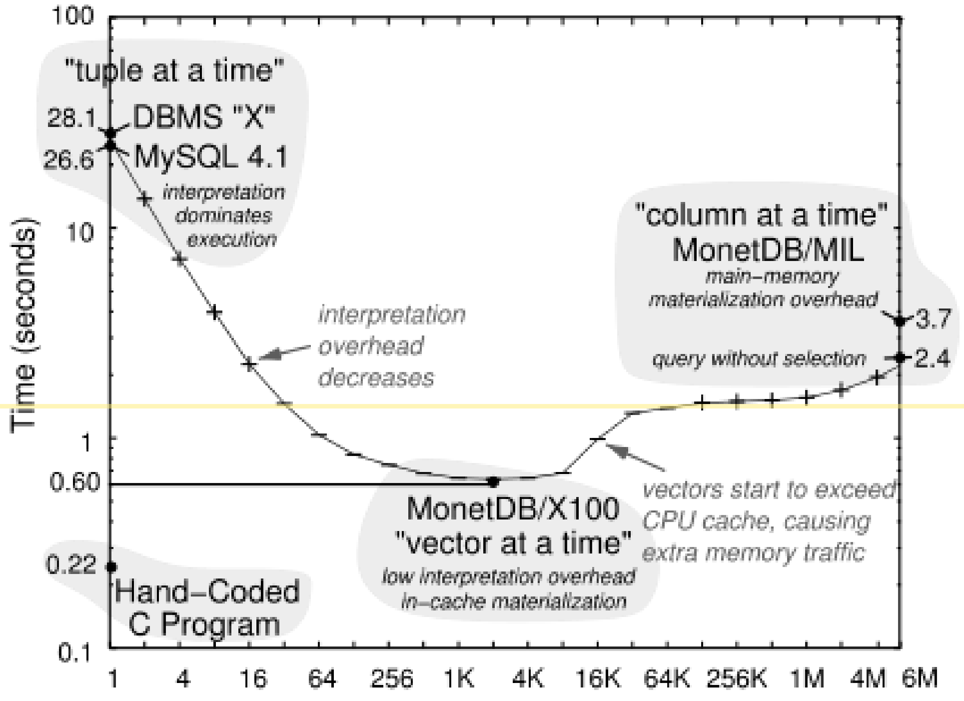


Figure 1: Hand-written code vs. execution engines for TPC-H Query 1 (Figure 3 of [16])

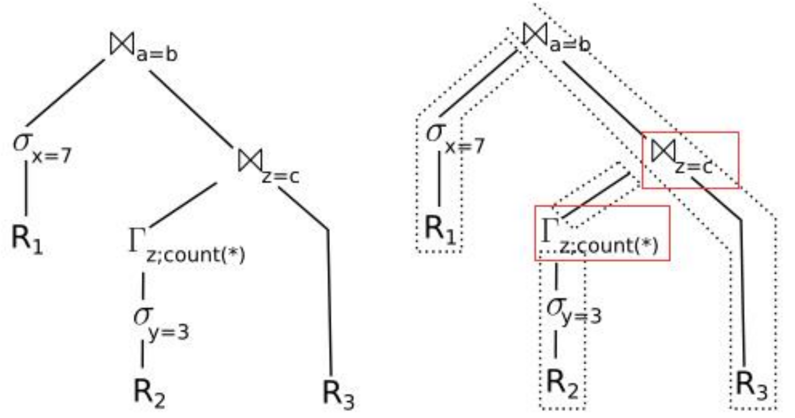
Compilation Hyper Basic Compilation

3.2 Hyper Basic Compilation 重要概念

1. Pipeline
    1. Pipeline内tuple能一直在寄存器内
2. Pipeline breaker
    1. 需要将tuple从寄存器和cache物化到内存
    2. 如hash join，hash aggregation
3. Data-centric
    1. 以数据为驱动，尽量提高data locality，减少cache misses
4. Pull -> Push
    1. 底层节点生成数据并物化到内存后，再让上层节点进行处理，数据往上层节点推。

```
select   *
from     R1,R3,
         (select   R2.z,count(*)
          from     R2
          where    R2.y=3
          group by R2.z) R2
where    R1.x=7 and R1.a=R3.b and R2.z=R3.c
```

Figure 2: Example Query



original　　with pipeline boundaries

Figure 3: Example Execution Plan for Figure 2

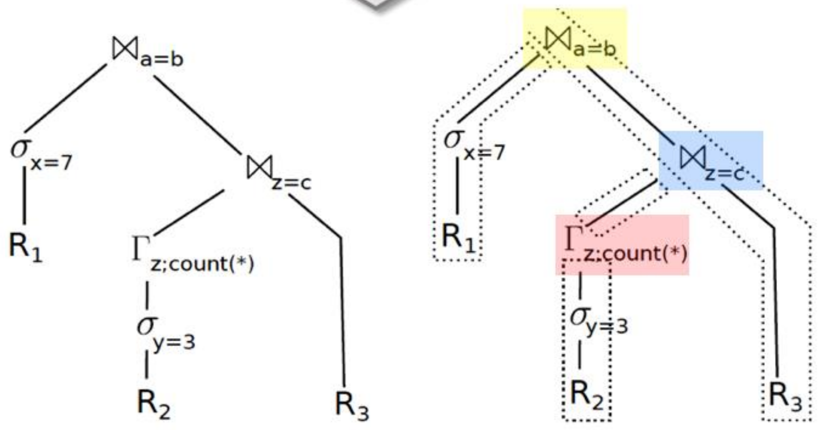
03

Compilation Hyper Basic Compilation

3.3 Code Generation Example

1. example

```
select      *
from        R1,R3,
            (select    R2.z,count(*)
             from      R2
             where     R2.y=3
             group by  R2.z) R2
where       R1.x=7 and R1.a=R3.b and R2.z=R3.c
```



original

with pipeline boundaries

initialize memory of $\bowtie_{a=b}$, $\bowtie_{c=z}$, and Γ_z

for each tuple t in R_1
    if $t.x = 7$
        materialize t in hash table of $\bowtie_{a=b}$

for each tuple t in R_2
    if $t.y = 3$
        aggregate t in hash table of Γ_z

for each tuple t in Γ_z
    materialize t in hash table of $\bowtie_{z=c}$

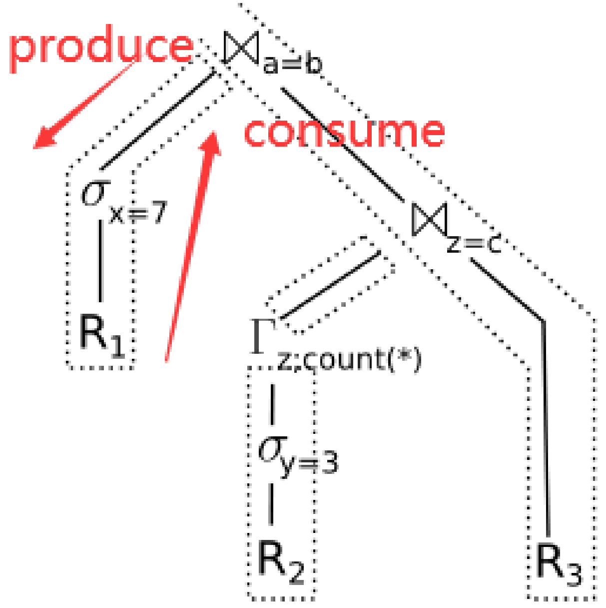
for each tuple t_3 in R_3
    for each match t_2 in $\bowtie_{z=c}[t_3.c]$
        for each match t_1 in $\bowtie_{a=b}[t_3.b]$
            output $t_1 \circ t_2 \circ t_3$

Compilation Hyper Basic Compilation

3.4 如何Compilation

1. Abstraction
   - produce()
   - consume(attributes,source)

2. 概念上的接口，主要用于code gen



with pipeline boundaries

join.left.produce->
filter.produce->
scan.produce->
filter.consumer->
join.consumer

$\bowtie$.produce	$\bowtie$.left.produce; $\bowtie$.right.produce;

$\bowtie$.consume(a,s)	if (s==$\bowtie$.left)
	print "materialize tuple in hash table";
	else
	print "for each match in hashtable["
	+a.joinattr+"]";
	$\bowtie$.parent.consume(a+new attributes)

σ.produce	σ.input.produce
σ.consume(a,s)	print "if "+σ.condition; σ.parent.consume(attr,σ)
scan.produce	print "for each tuple in relation" scan.parent.consume(attributes,scan)

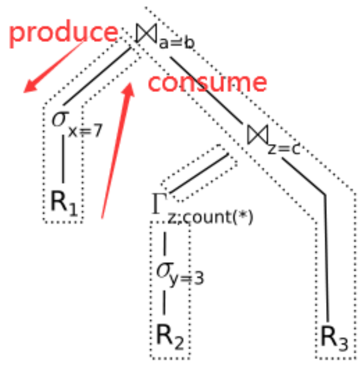
Figure 5: A simple translation scheme to illustrate the *produce*/*consume* interaction

03 Compilation Hyper Basic Compilation

3.5 如何Compilation

1．深度优先遍历

join.left.produce->
filter.produce->
scan.produce->
filter.consumer->
join.consumer



with pipeline boundaries

$\bowtie$.produce	$\bowtie$.left.produce; $\bowtie$.right.produce;

$\bowtie$.consume(a,s)	if (s==$\bowtie$.left)
	print "materialize tuple in hash table";
	else
	print "for each match in hashtable["
	+a.joinattr+"]";
	$\bowtie$.parent.consume(a+new attributes)

σ.produce	σ.input.produce
σ.consume(a,s)	print "if "+σ.condition; σ.parent.consume(attr,σ)
scan.produce	print "for each tuple in relation" scan.parent.consume(attributes,scan)

Figure 5: A simple translation scheme to illustrate the *produce*/*consume* interaction

initialize memory of $\bowtie_{a=b}$, $\bowtie_{c=z}$, and Γ_z
for each tuple t in R_1
  if $t.x = 7$
    materialize t in hash table of $\bowtie_{a=b}$
for each tuple t in R_2
  if $t.y = 3$
    aggregate t in hash table of Γ_z
for each tuple t in Γ_z
  materialize t in hash table of $\bowtie_{z=c}$
for each tuple t_3 in R_3
  for each match t_2 in $\bowtie_{z=c}[t_3.c]$
    for each match t_1 in $\bowtie_{a=b}[t_3.b]$
      output $t_1 \circ t_2 \circ t_3$

Figure 4: Compiled query for Figure 3

Compilation Hyper Basic Compilation

3.6 Hyper LLVM Compilation

1. 为何不用C++
    1. C++编译器转换为machine code比较慢
    2. C++不能对生成的machine code完全控制
2. 使用了更低层次的llvm
    1. JIT 运行时编译执行技术
    2. 只需要在**关键路径上**的多次调用的函数使用llvm来实现，通过llvm将原来的c++代码串联起来。

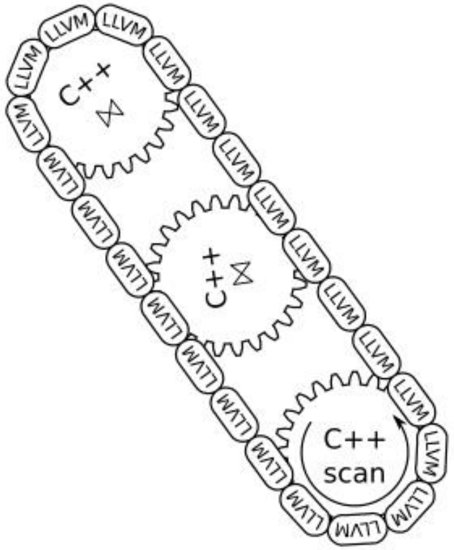


Figure 6: Interaction of LLVM and C++

Compilation Hyper Basic Compilation

3.7 与MonetDB对比的效果

1. data-centric,模糊了operator之间的界限，生成的代码更加紧凑，分支更少，指令数更少（pipeline内的操作一起进行了，selection -> build hash table）。
2. Cache 命中率更高，一个tuple可以把多个计算完成，直到下一个pipeline breaker点。

	Q1		Q2		Q3		Q4		Q5	
	LLVM	MonetDB	LLVM	MonetDB	LLVM	MonetDB	LLVM	MonetDB	LLVM	MonetDB
branches	19,765,048	144,557,672	37,409,113	114,584,910	14,362,660	127,944,656	32,243,391	408,891,838	11,427,746	333,536,532
mispredicts	188,260	456,078	6,581,223	3,891,827	696,839	1,884,185	1,182,202	6,577,871	639	6,726,700
I1 misses	2,793	187,471	1,778	146,305	791	386,561	508	290,894	490	2,061,837
D1 misses	1,764,937	7,545,432	10,068,857	6,610,366	2,341,531	7,557,629	3,480,437	20,981,731	776,417	8,573,962
L2d misses	1,689,163	7,341,140	7,539,400	4,012,969	1,420,628	5,947,845	3,424,857	17,072,319	776,229	7,552,794
I refs	132 mil	1,184 mil	313 mil	760 mil	208 mil	944 mil	282 mil	3,140 mil	159 mil	2,089 mil

Table 3: Branching and Cache Locality

Compilation Hyper Adaptive Compilation

3.1 Compilation Time 很重要

1. 对OLTP这种本身比较小的query来说 compilation time 可能比本身的执行时间还大，compilation不太适合。

2. 执行只需1ms，llvm compilation 54ms

```sql
SELECT c.oid, c.relname, n.nspname
FROM pg_inherits i
JOIN pg_class c ON c.oid = i.inhparent
JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE i.inhrelid = 16490 ORDER BY inhseqno
```

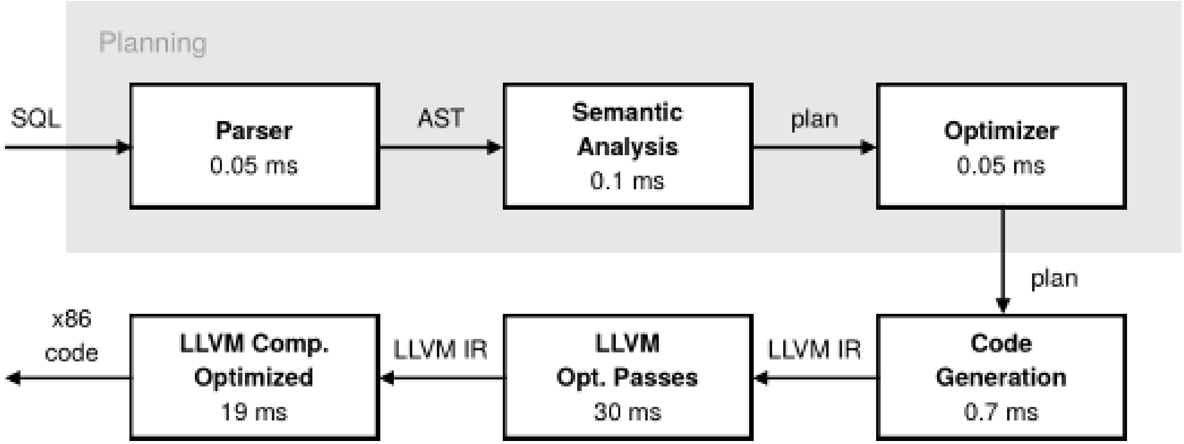


Fig. 1. Architecture of compilation-based query engines.

Compilation Hyper Adaptive Compilation

03

3.1 Compilation Time 很重要

1. Compilation的时间越长，生成的代码质量越高。
   1. LLVM IR
   2. LLVM bytecode 基于VM（类似于JVM）的bytecode 解释虚拟机
   3. LLVM unoptimized
   4. LLVM optimized
   5. handwritten

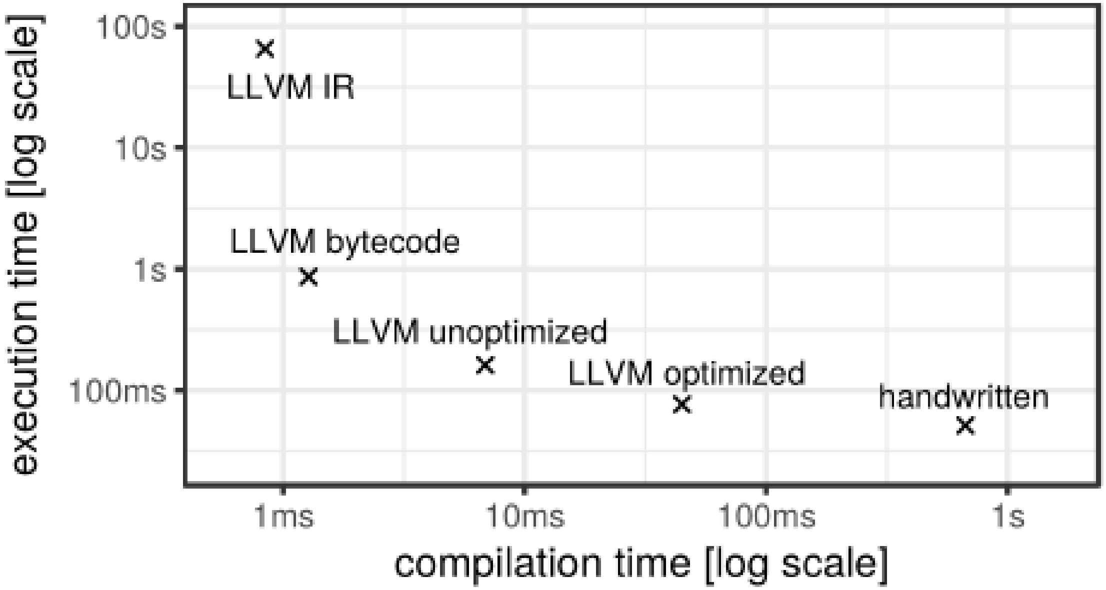


Fig. 2. Single-threaded query compilation and execution time for different execution modes on TPC-H query 1 on scale factor 1.

Compilation Hyper Adaptive Compilation

3.2 Adaptive Overview

1. 基于LLVM bytecode解析执行（高效），默认模式。
2. 基于LLVM IR做无优化的编译，生成machine code
3. 基于LLVM IR做有优化的编译，生成更高效的 machine code

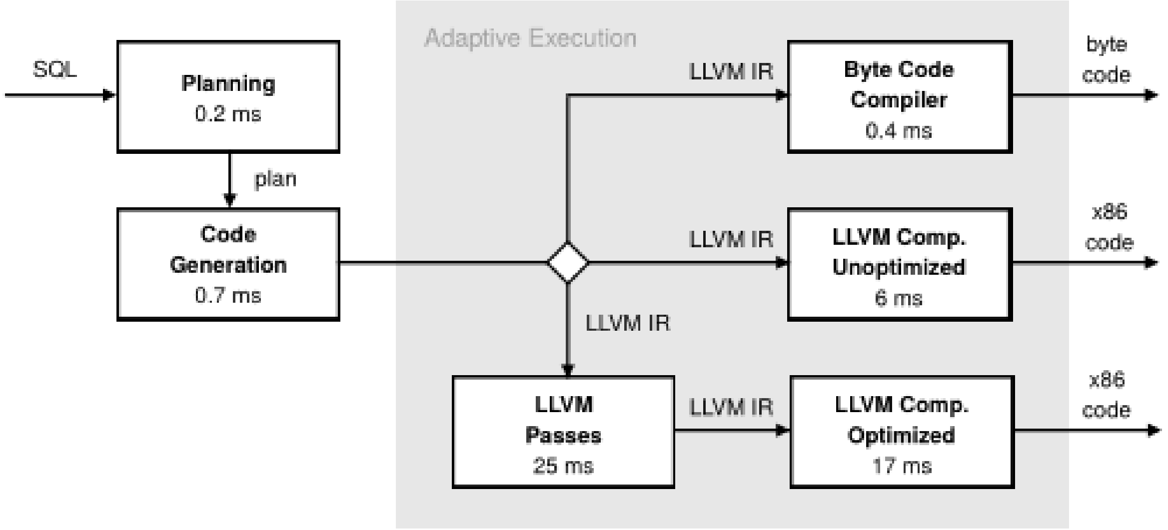


Fig. 3. Execution modes and their compilation times.

03 Compilation Hyper Adaptive Compilation

3.3 如何方便切换

1. 对pipeline进行Morsel-driven细分调度

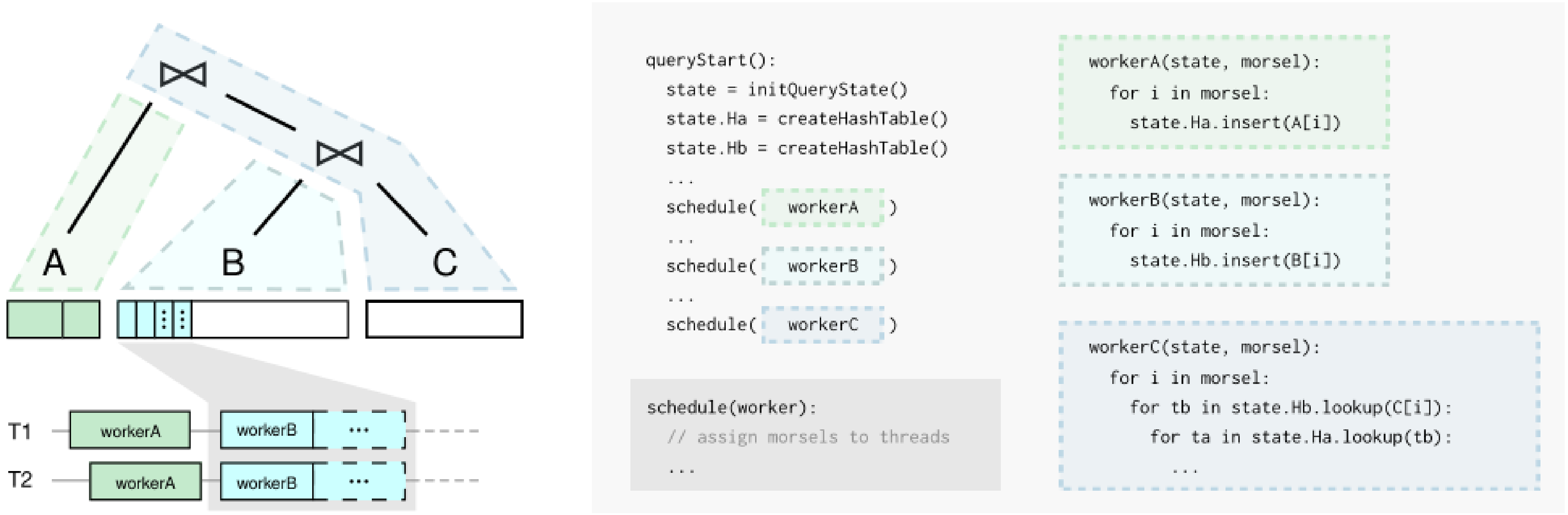


Fig. 4. Illustration of query plan translation to pseudo code. *queryStart* is the main function. Each of the three query pipelines is translated into a *worker* function. The lower left corner shows that the work of each pipeline is split into small morsels that are dynamically scheduled onto threads.

Compilation Hyper Adaptive Compilation

3.3 如何方便切换

2．函数封装方便模式切换

1．使用handle.fn对具体的实现方式进行封装，当需要切换模式时，只要切换handleB指向的函数指针即可。

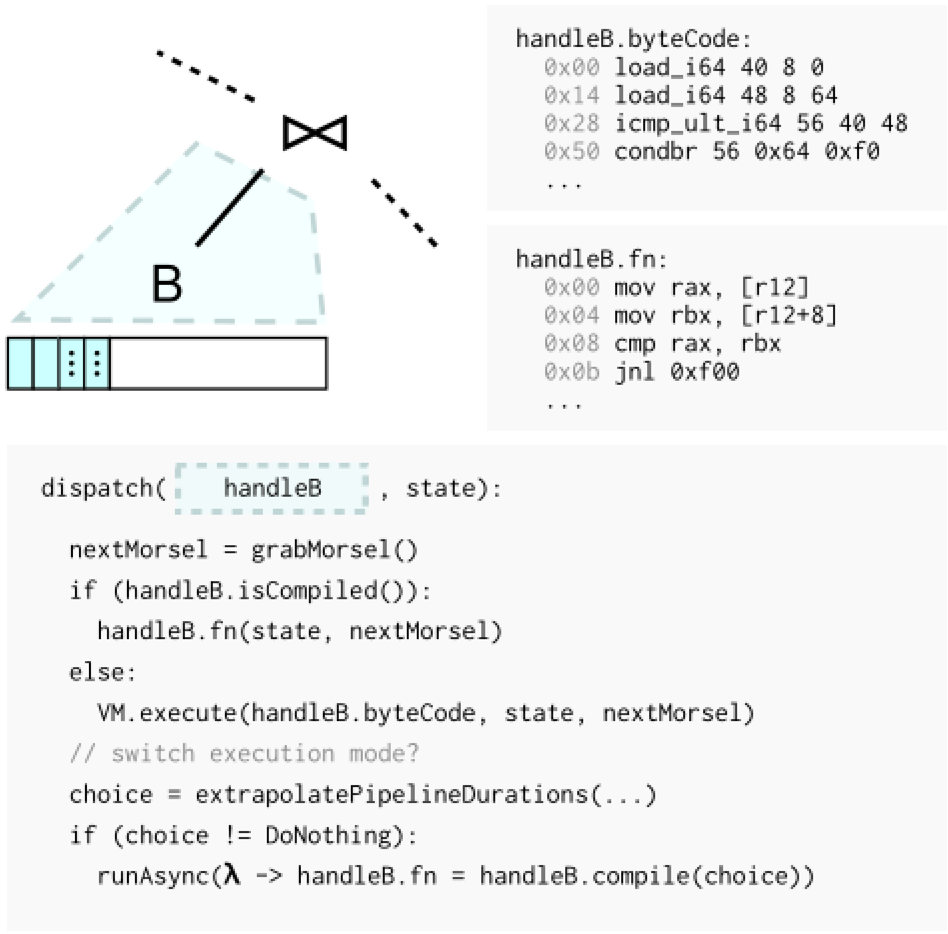


Fig. 5. Switching on-the-fly from interpretation to execution. The dispatch code is run for every morsel.

03 Compilation Hyper Adaptive Compilation

3.3 如何方便切换

3. 切换规则
   1. 解释mode:

      r0：所有进程的平均处理速度

      t0：w个进程处理剩余n个tuple需要的开销
   2. llvm unopt:

      r1：llvm unopt执行速度

      c1：llvm unopt的编译时间

      t1：编译时间 + 扣除编译期间解释执行后的剩余tuple数量，被w个进程处理所需要的时间
   3. 切换到预期开销最小的模式

```
// f: worker function
// n: remaining tuples
// w: active worker threads
extrapolatePipelineDurations(f, n, w):
  r0 = avg(rate in threadRates)
  r1 = r0 * speedup1(f); c1 = ctime1(f)
  r2 = r0 * speedup2(f); c2 = ctime2(f)
  t0 = n / r0 / w
  t1 = c1 + max(n - (w-1)*r0*c1, 0) / r1 / w
  t2 = c2 + max(n - (w-1)*r0*c2, 0) / r2 / w
  switch min(t0, t1, t2):
    case t0: return DoNothing
    case t1: return Unoptimized
    case t2: return Optimized
```

Fig. 7. Extrapolation of the pipeline durations.

Compilation Hyper Adaptive Compilation

3.4 效果

1. 4个进程
2. Bytecode模式：
    1. 全部使用bytecode，就是volcano模式，耗时比较长
3. Llvm unopt
    1. 编译期间其他thread需要空等
4. Adaptive
    1. 编译期间其他thread可以使用bytecode解释执行
    2. 编译完成后全部切换成编译执行
    3. 效果最好

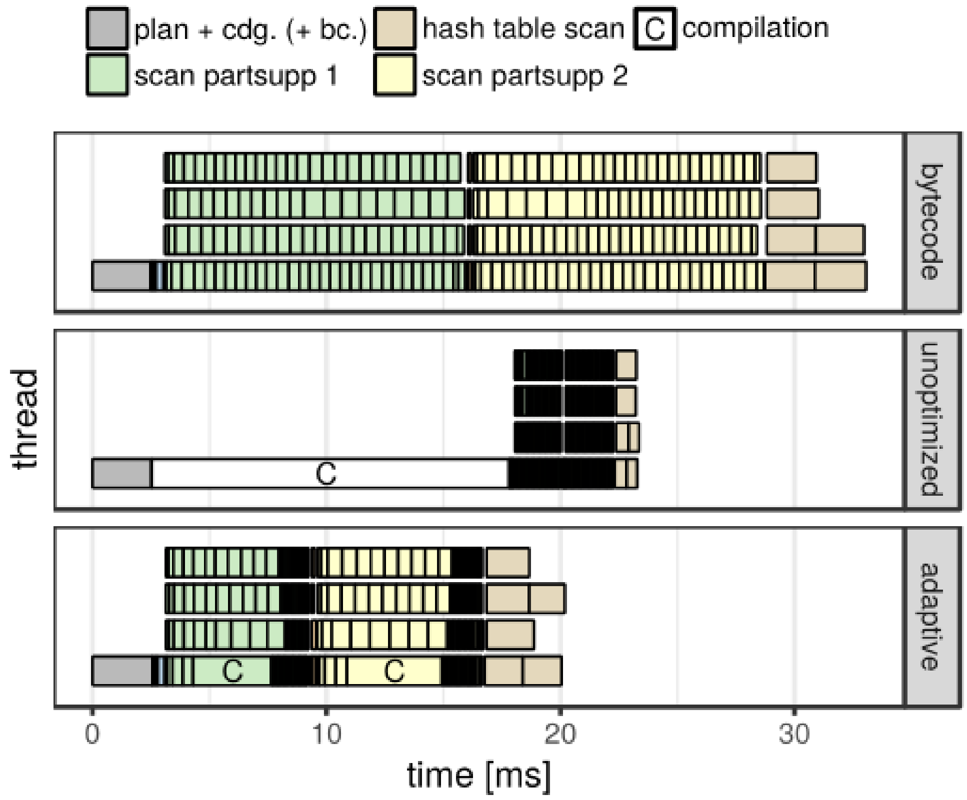


Fig. 14. Execution trace of TPC-H query 11 on scale factor 1 using 4 threads. The optimized mode is not shown, as its compilation takes very long (103ms).

04 / Vectorization vs Compilation

Vectorization vs Compilation

4.1 Apples-to-apples 的比较

1. Pull-vectorizaton:

Vectorwize(前身是MonetDB) -> Tectorwize

2. Push-Compilation:

Hyper-> Typer

3. 不同的data type等各自优化点，会影响比较结果。

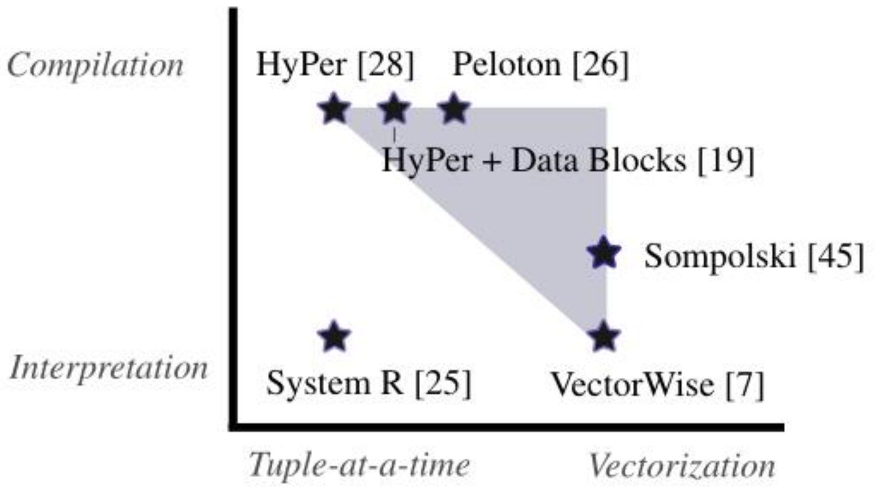


Figure 13: **Design space between vectorization and compilation** – *hybrid models integrate the advantages of the other approach.*

Vectorization vs Compilation

4.2 Single-Threaded Performance

1. 选择对TPC-H的5条SQL进行对比分析

2. Q1,Q18 计算为主 typer 性能高更好

3. Q3, Q9 有hash join, hash build 是 memory bound操作，需要从memory读取数据 Tectowice表现更好

Q1: fixed-point arithmetic, (4 groups) aggregation
Q6: selective filters
Q3: join (build: 147K entries, probe: 3.2M entries)
Q9: join (build: 320K entries, probe: 1.5M entries)
Q18: high-cardinality aggregation (1.5M groups)

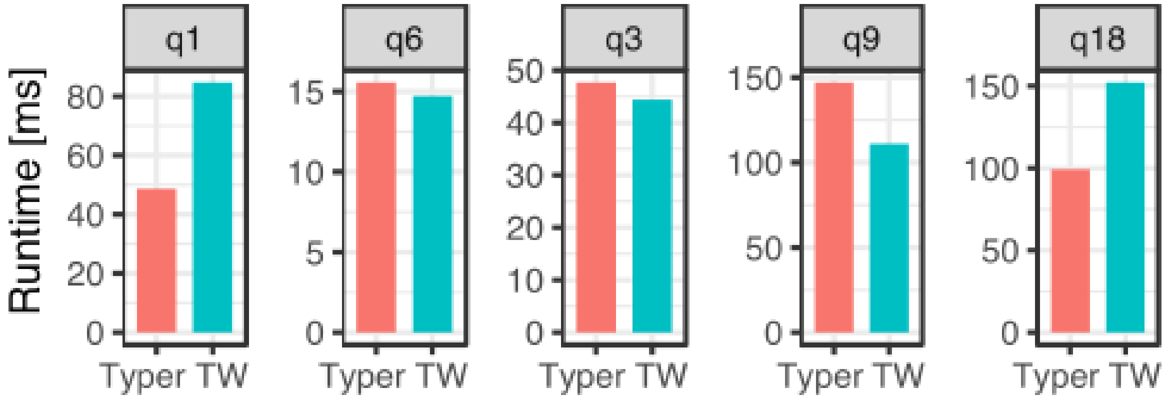


Figure 3: **Performance** – *TPC-H SF=1, 1 thread*

04

Vectorization vs Compilation

4.2 Single-Threaded Performance (**Q1,Q18**)

1. 以数学操作和chip in cache aggregation为主
2. Tectorwise < Typer
3. Instructions
    1. Typer模糊了operator之间的界限，指令数更少。
4. L1 cach misses
    1. Tectorwize有很多Load和Store的开销，一批量（这个批量刚好放在cache）计算好一个expresion之后，需要通过内存传给父亲节点，导致cache misses多。

如a+1+b：a+1物化到内存，再把a+1 load到寄存器，再计算a+1+b。

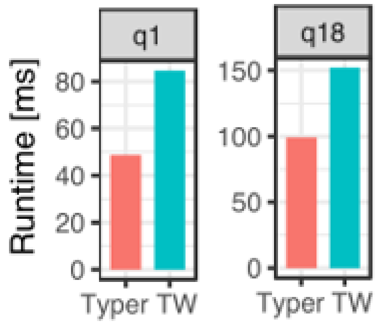


Table 1: **CPU Counters** – *TPC-H SF=1, 1 thread, normalized by number of tuples processed in that query*

		cycles	IPC	instr.	L1 miss	LLC miss	branch miss
Q1	Typer	34	2.0	68	0.6	0.57	0.01
Q1	TW	59	2.8	162	2.0	0.57	0.03
Q6	Typer	11	1.8	20	0.3	0.35	0.06
Q6	TW	11	1.4	15	0.2	0.29	0.01
Q3	Typer	25	0.8	21	0.5	0.16	0.27
Q3	TW	24	1.8	42	0.9	0.16	0.08
Q9	Typer	74	0.6	42	1.7	0.46	0.34
Q9	TW	56	1.3	76	2.1	0.47	0.39
Q18	Typer	30	1.6	46	0.8	0.19	0.16
Q18	TW	48	2.1	102	1.9	0.18	0.37

data size by a factor of 10, causes 0.5 additional cache misses per tuple“).

Q1: fixed-point arithmetic, (4 groups) aggregation
Q18: high-cardinality aggregation (1.5M groups)

04

Vectorization vs Compilation

4.2 Single-Threaded Performance **(Q3,Q9)**

1. 以memory-bound的hash join为主
2. Tectorwise > Typer
3. Typer的Memory stall数量比Tectorwise多

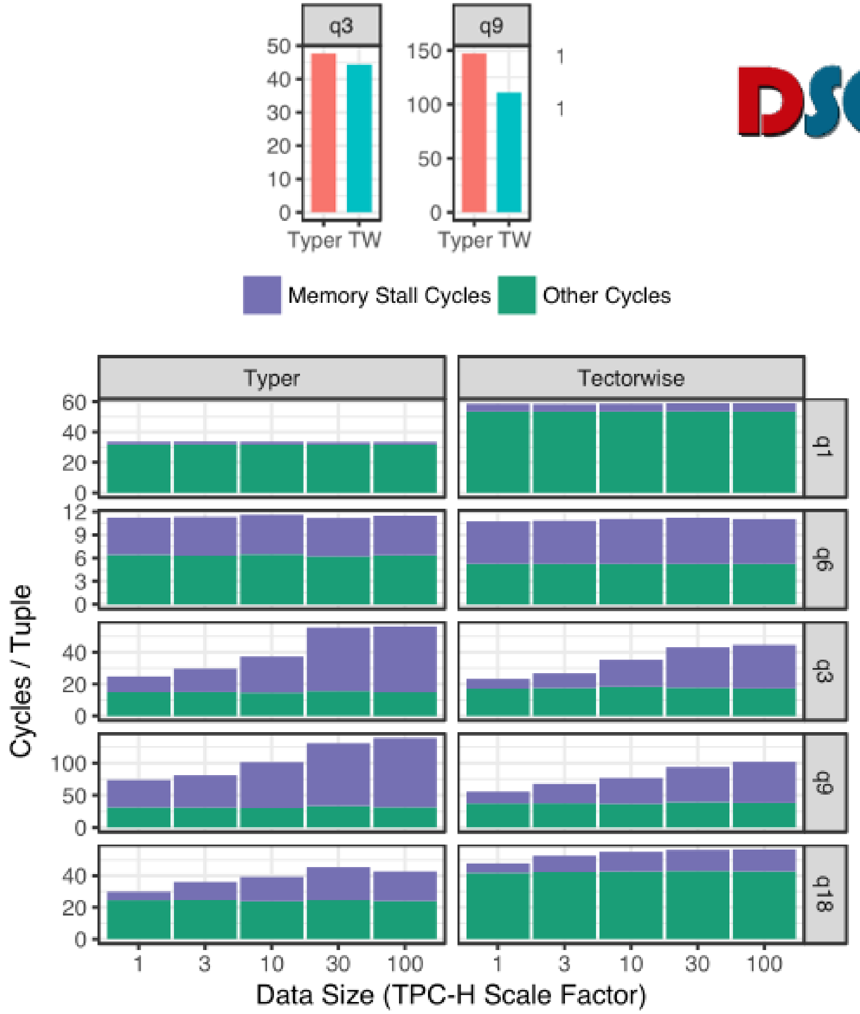


Figure 4: **Memory Stalls** – *TPC-H, 1 thread*

Q3: join (build: 147K entries, probe: 3.2M entries)
Q9: join (build: 320K entries, probe: 1.5M entries)

04 Vectorization vs Compilation

4.2 Single-Threaded Performance **(Q3,Q9)**

1. Typer 编译执行的 hash join

2. cpu对于上一个if的判断是匹配的，然后cpu可能会预测下一个if也是true会把对应的指令和数据都预加载到内存里面，如果分支预测失败的话，会有rollback的开销，并打断pipeline。

```
query(...)
  // build hash table
  for(i = 0; i < S.size(); i++)
    ht.insert(<S.att1[i], S.att2[i]>, S.att3[i])
  // probe hash table
  for(i = 0; i < R.size(); i++)
    int k1 = R.att1[i]
    string* k2 = R.att2[i]
    int hash = hash(k1, k2)
    for(Entry* e = ht.find(hash); e; e = e->next)
      if(e->key1 == k1 && e->key2 == *k2)
        ... // code of parent operator
```

(a) Code generated for hash join

Vectorization vs Compilation

4.2 Single-Threaded Performance **(Q3,Q9)**

1. Tectorwise 向量化执行的 hash join

2. Vector大小内tuple的 probeHash_, findCandidates_, compareKeys_因为没有数据依赖可以乱序执行形成pipeline。

3. 编译执行的If操作被转换成了SIMD的向量化执行，避免了分支预测。

```
class HashJoin
  Primitives probeHash_, compareKeys_, buildGather_;
  ...
int HashJoin::next()
  ... // consume build side and create hash table
  int n = probe->next()// get tuples from probe side
  // *Interpretation*: compute hashes
  vec<int> hashes = probeHash_.eval(n)
  // find hash candidate matches for hashes
  vec<Entry*> candidates = ht.findCandidates(hashes)
  // matches: int references a position in hashes
  vec<Entry*, int> matches = {}
  // check candidates to find matches
  while(candidates.size() > 0)
    // *Interpretation*
    vec<bool> isEqual = compareKeys_.eval(n, candidates)
    hits, candidates = extractHits(isEqual, candidates)
    matches += hits
  // *Interpretation*: gather from hash table into
  // buffers for next operator
  buildGather_.eval(matches)
  return matches.size()
```

(b) Vectorized code that performs a hash join

Vectorization vs Compilation

4.2 Single-Threaded Performance

1. 总结
    1. Typer 适合 computation 为主的 queries。
    2. Tectorwise 的cache misses会更少，对memory-bound的queries 比如要access大的hash tablejoin效果会比更好。

Vectorization vs Compilation

4.3 其他方面

1. SIMD
    1. 理论上vectorization能更好利用SIMD，在密集的数据上确实如此，但是实际上的分析型SQL语句的selectivity会比较稀疏，导致不会太大。
2. INTRA-QUERY PARALLELIZATION
    1. 两种方式都能很好地利用operator内的并行。
3. HARDWARE
    1. 在不同架构的cpu上两者各有胜负
4. Compilation Time
    1. 是一个问题，可以通过存储过程和udf来进行预编译
    2. Llvm 编译与SQL text的大小强相关，hyper对这部分做了自己的优化，优化编译算法。
    3. Adaptive的方式。

Vectorization vs Compilation

4.4 总结

Vectorized vs. Compiled	
Computation (<)	编译执行对computation为主的queries更适合，因其代码更紧凑，能在寄存器中一次性把相连的计算完成，减少了和内存的交互。
Parallel data access (>)	向量化执行更适合需要并行访问大量data的queries（如hash join为主），因其每个vector能构建有效的cpu pipeline。
SIMD (=)	向量化执行的SIMD收益在实际中有限，因现实中大多数operator都是memory-bound的。
Parallelization (=)	都能利用好多核cpu。
Hardware platforms (=)	在不同架构的cpu上两者各有胜负
Compile time (>)	向量化执行的primitives在编译时就完成，不需要在运行时编译，如何减少编译执行的编译时间是一个重要研究点。

05 总结

执行方式	特征	优缺点
Volcano	Open-Next-Close Pull-based Tuple-at-a-time	Pipeline不友好
		函数调用开销大
		cache命中率低
Vectorization	Open-Next-Close Pull-based Vector-at-a-time	一次处理一批，针对memory-bound的操作能有效形成pipeline
		均摊了函数调用和内存访问开销
		代码不够紧凑，向父亲节点返回结果时需要经过内存，不适合computaiton为主的quey
Compilation	Push-based Data-centric Tuple-at-a-time	代码紧凑，pipeline内操作一起完成，指令数少
		Cache利用率高
		Tuple-at-a-time不好形成pipeline，不适合memory-bound的query

04 文献

[01] - Boncz P A, Zukowski M, Nes N. MonetDB/X100: Hyper-Pipelining Query Execution[C]//Cidr. 2005, 5: 225-237.

[02] - Polychroniou O, Raghavan A, Ross K A. Rethinking SIMD vectorization for in-memory databases[C]//Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. 2015: 1493-1508.

[03] - Neumann T. Efficiently compiling efficient query plans for modern hardware[J]. Proceedings of the VLDB Endowment, 2011, 4(9): 539-550.

[04] - Kohn A, Leis V, Neumann T. Adaptive execution of compiled queries[C]//2018 IEEE 34th International Conference on Data Engineering (ICDE). IEEE, 2018: 197-208.

[05] - Kersten T, Leis V, Kemper A, et al. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask[J]. Proceedings of the VLDB Endowment, 2018, 11(13): 2209-2222.

[05] – 参考博文: https://www.one-tab.com/page/LmMkDhRhRWKUxU2bRRJgQA

[06] – cmu 15-721 2020 https://15721.courses.cs.cmu.edu/spring2020/

[07] - https://www.bilibili.com/video/BV1zb411G7ay/