inspur 浪潮 | 云溪数据库

西安电子科技大学 XIDIAN UNIVERSITY

关系型数据库执行器演化发展& 编译执行自适应优化

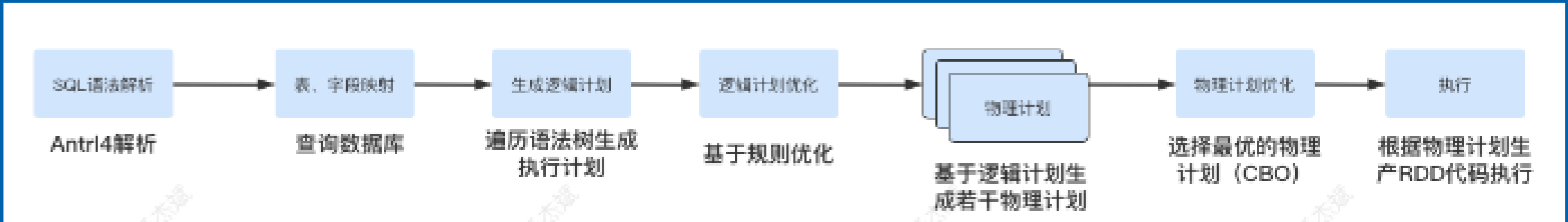
西安电子科技大学-浪潮云溪数据库A2组-王智彬

www.znbase.com
www.xidian.edu.cn

目录

Background

- 在数据库系统发展历史中，SQL语句的**查询优化器**优化一直是数据库SQL引擎行业关注的重点，但是在关系型数据库执行计算的过程中，还有同等重要的**查询调度器与计划执行器**对SQL优化器生成的执行计划进行解释优化执行



Background

- 在关系数据库发展的早期，受制于计算机IO能力的约束，计算在查询整体的耗时占比并不明显，这个时候**调度器和执行器**的作用被**弱化**，一个查询的好坏更主要取决于优化器对执行计划的选择好坏
- 随着计算机硬件的发展，**调度器和执行器**也逐渐彰显了它们的重要地位。

目 录

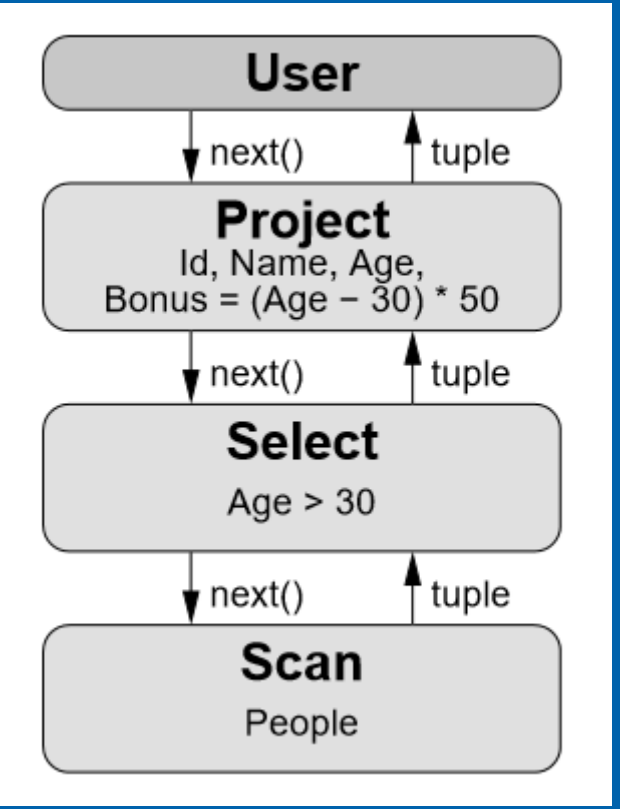
火山模型 V1.0

- Volcano Model是一种经典的基于行的流式迭代模型(Row-BasedStreaming Iterator Model)，在我们熟知的主流关系数据库中都采用了这种模型，例如Oracle，SQL Server，MySQL等

Graefe G，Mckenna W J . The Volcano optimizer generator: extensibility and efficient search[C]// International Conference on Data Engineering. IEEE, 1993.

火山模型 V1.0 拉取模型

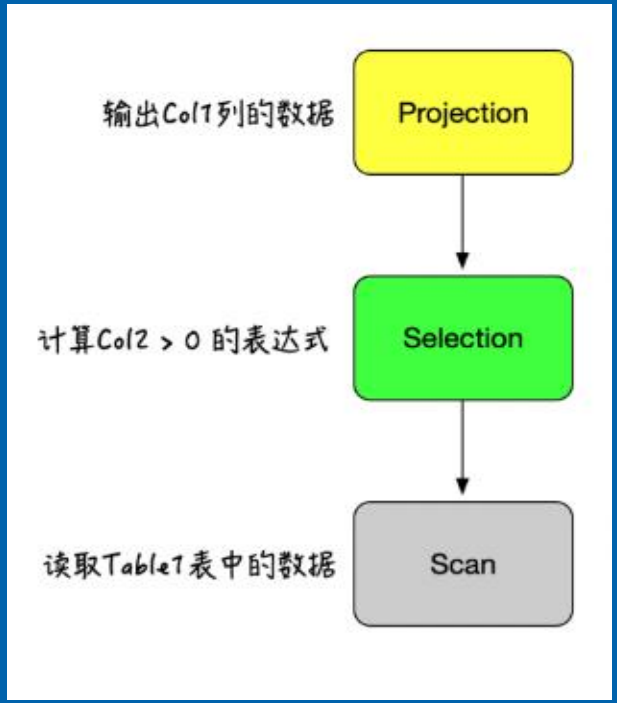
- 将关系代数当中的每一个算子抽象成一个迭代器。每个迭代器都带有一个 next 方法。每次调用这个方法将会返回这个算子的产生的一行数据（或者说一个 Tuple）。程序通过在 SQL 的计算树的根节点不断地调用 next 方法来获得整个查询的全部结果
- 这一过程就像是将数据经过层层调用从底层拉取到上层，因此这一模型又被称为**拉取模型**



G. Graefe. 1994. Volcano— An Extensible and Parallel Query Evaluation System. IEEE Trans. on Knowl. and Data Eng. 6, 1 (February 1994), 120–135.

火山模型 V1.0 拉取模型

- SELECT col1
- FROM table1
- WHERE col2 > 0




```
//Projection
    def next() = {
            return Row.of(Filter.next().col1);
}

//Filter
    def next() = {
            while(Scan.hasNext()){
              val row = Scan.next();
              if(row.col2 > 0){
                return row;
              }
            }
    return nil;
}

//Scan
    def next() = {
            return table1.readRow();
}
```

G. Graefe. 1994. Volcano— An Extensible and Parallel Query Evaluation System. IEEE Trans. on Knowl. and Data Eng. 6, 1 (February 1994), 120–135.

火山模型 V1.0 优点

- 每个运算符之间的代数计算是相互独立的，并且运算符可以伴随查询关系的变化出现在查询计划树的任意位置，这使得运算符的算法实现变得简单并且富有拓展性。

- 数据以行的形式在运算符之间流动，只要没有sort之类破坏流水性的运算出现，每个运算符仅需要很少的资源就可以很好的运行起来，非常的节省内存资源。

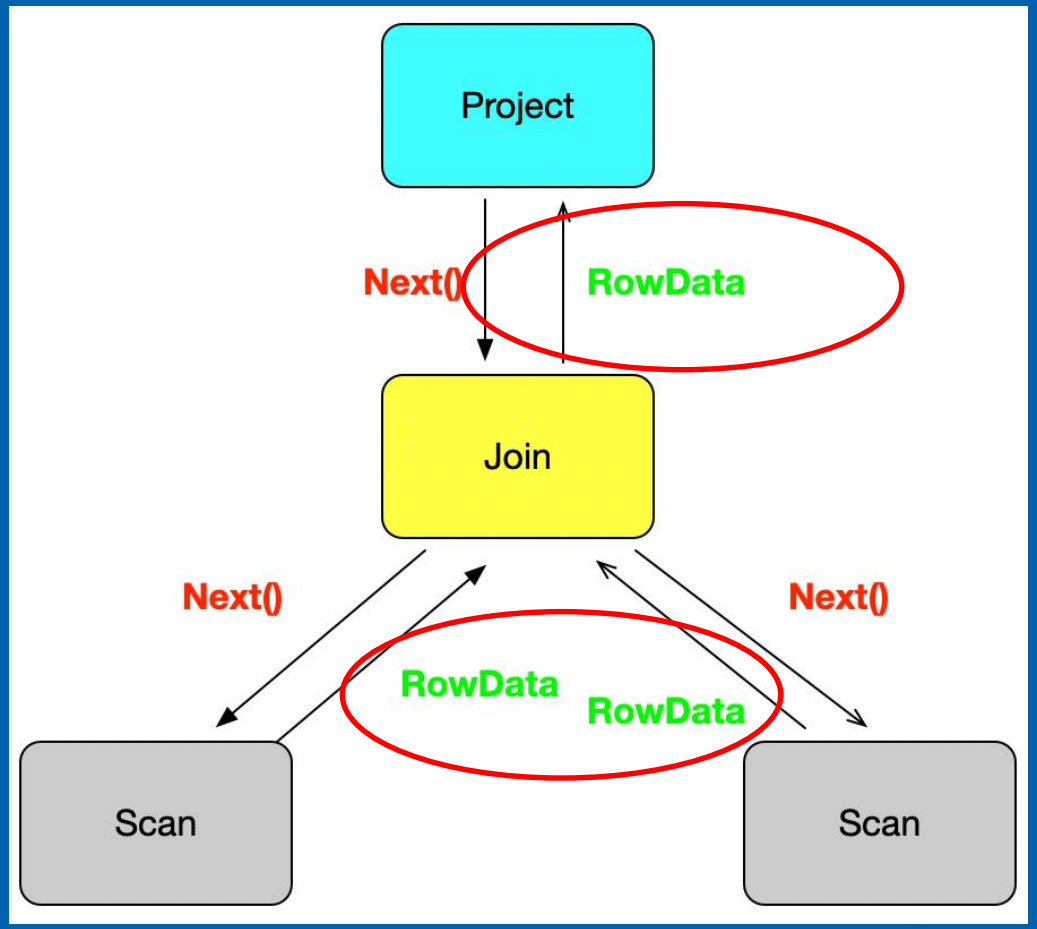
火山模型 V1.0 痛点

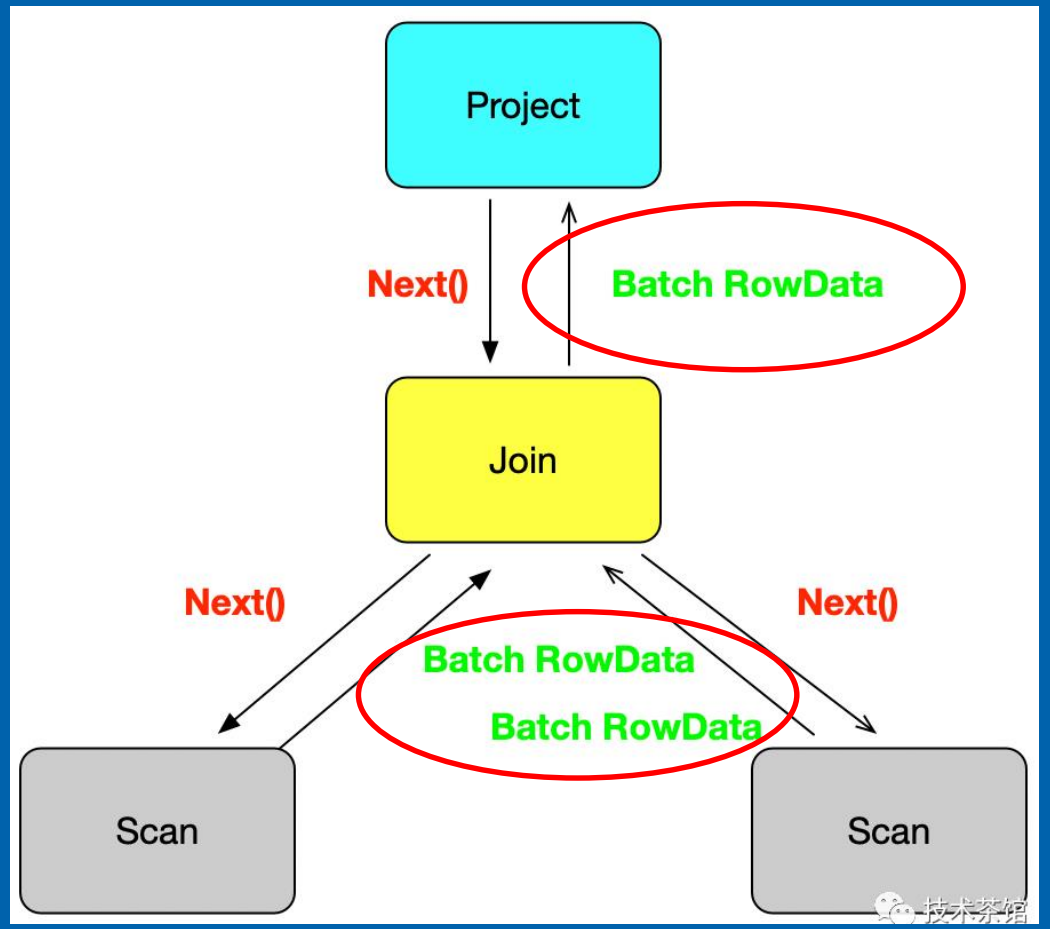
- 虚函数的开销 CPU利用率低下

数据以行为单位进行处理，不利于CPU cache 发挥作用。且每处理一行需要调用多次 next() 函数，而 next() 为虚函数，开销大。

每次都是计算一个 tuple（Tuple-at-a-time），这样会造成多次调用 next ，也就是造成大量的虚函数调用，这样会造成 CPU 的利用率不高

火山模型 V2.0 批处理-向量化执行



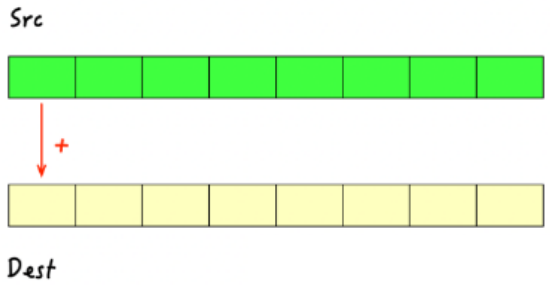


火山模型 V2.0 优势

- 大大减少火山模型中的虚函数调用数量
- 以块为处理单位数据，提供了cache命中率
- 多行并发处理，契合了CPU乱序执行与并发执行的特性
- 同时处理多行数据，使SIMD有了用武之地

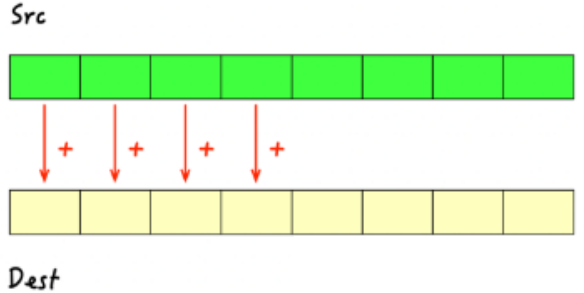
向量化计算就是将一个循环处理一个数组的时候每次处理 1 个数据共处理 N 次，转化为向量化——每次同时处理 8 个数据共处理 N/8 次，其中依赖的技术就是 SIMD（Single Instruction Multiple Data，单指令流多数据流），SIMD 可以在一条 CPU 指令上处理 2、4、8 或者更多份的数据

```
void plus(uint32_t *dest,
          uint32_t *src,
          size_t n) {
    for i in 0..n {
        dest[i] += src[i];
    }
}
```



普通数组加法运算

```
void plus(uint32_t *dest,
          uint32_t *src,
          size_t n) {
    while ... {
        _mm_add_epi32(&dest[i], &src[i]);
        i += 4;
    }
}
```

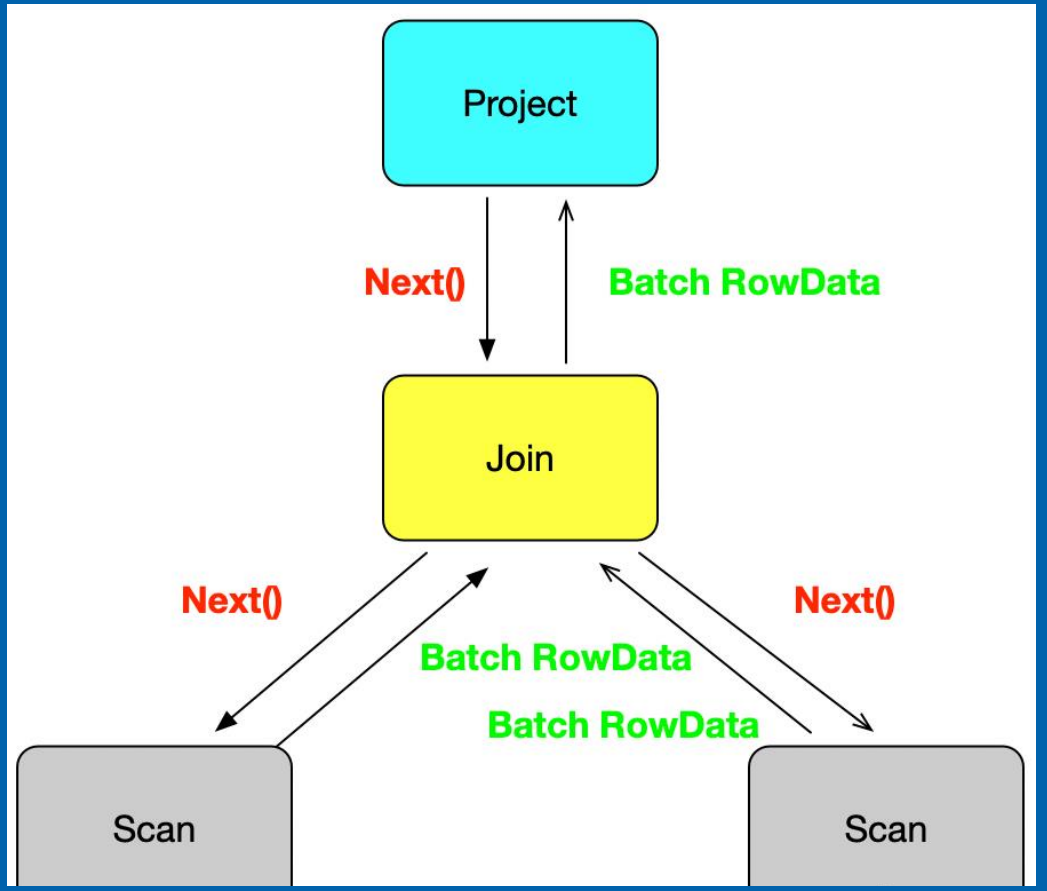


SIMD加法运算

火山模型 V2.0 问题

- 遗憾的是，无论如何选择Batch的大小，虚函数的调用成为了火山模型永远的痛；即使Batch分的已经很大，虚函数的问题依旧存在，CPU的效率始终无法得到更高的提升

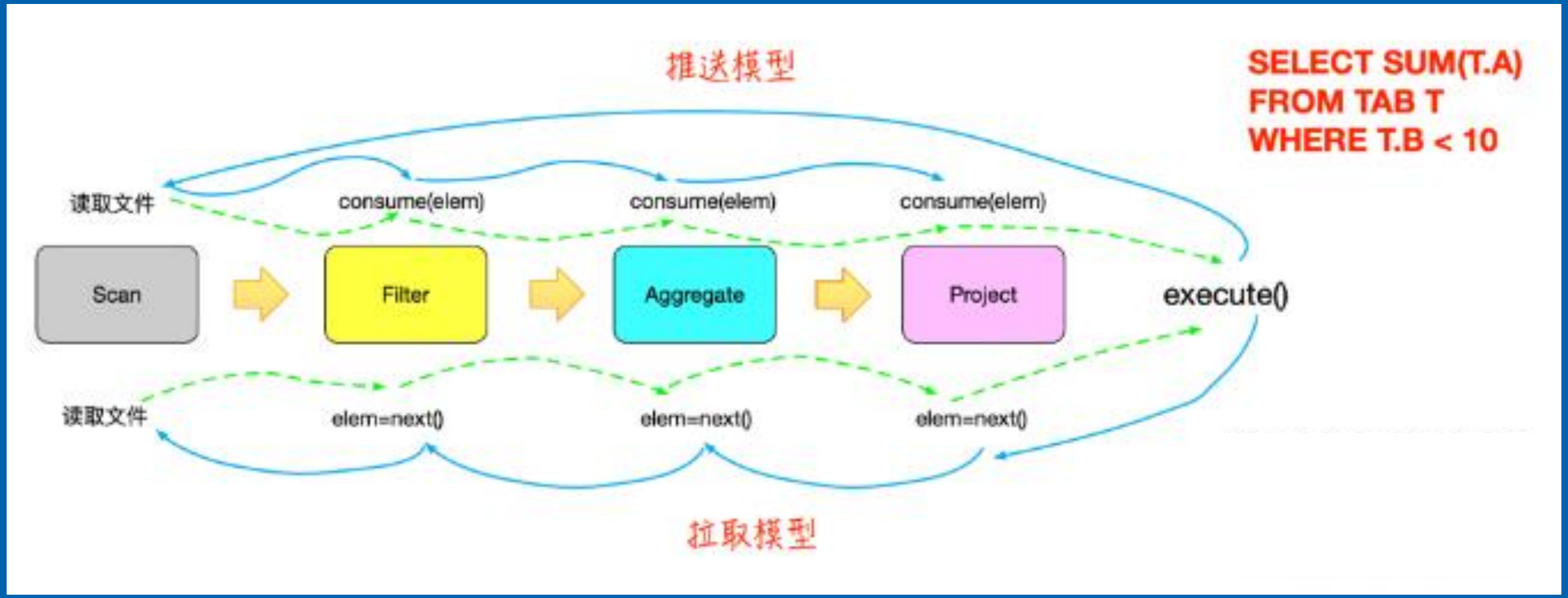
如何解决虚函数的问题成为了火山模型与生俱来的痛点



执行引擎的演化之路

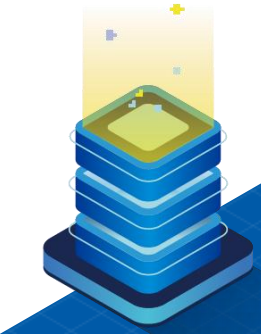
Kersten T， Leis V， Kemper A , et al. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask[J]. Proceedings of the VLDB Endowment, 2018, 11(13):2209-2222.

拉取模型&推送模型



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.

目 录



相关论文

引入编译执行

构建并行框架

优化编译执行

Efficiently Compiling Efficient Query Plans for Modern Hardware

Thomas Neumann
Technische Universität München
Munich, Germany

Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age

Viktor Leis* Peter Boncz† Alfons Kemper* Thomas Neumann*
* Technische Universität München † CWI

Adaptive Execution of Compiled Queries

André Kohn, Viktor Leis, Thomas Neumann
Technische Universität München

Thomas Neumann （托马斯·诺伊曼）博士

从事**数据库系统**研究 重点关注**查询优化**（计算高效查询策略）、**查询处理**（高效查询执行）

Efficiently Compiling Efficient Query Plans for Modern Hardware
pipeline-breaker切分

为了追求最优性能，需要考虑的角度就是最大化data/code的locality，因此提出了pipeline-breaker的概念

使用了与火山模型相反的数据流方向，不再pull数据，而是将数据"push"到下一个算子，这个push一直进行直到下一个pipeline-breaker

寻找最优物化点，在两个物化点之间的所有逻辑可以编译到同一个代码片段中，这样每个tuple，在加载之后，流水线的完成所有可能的处理，再保存到目标pipeline breaker中，每个tuple都是如此，这样就使得数据最大程度留在运算当中，最小化memory access的次数

```
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

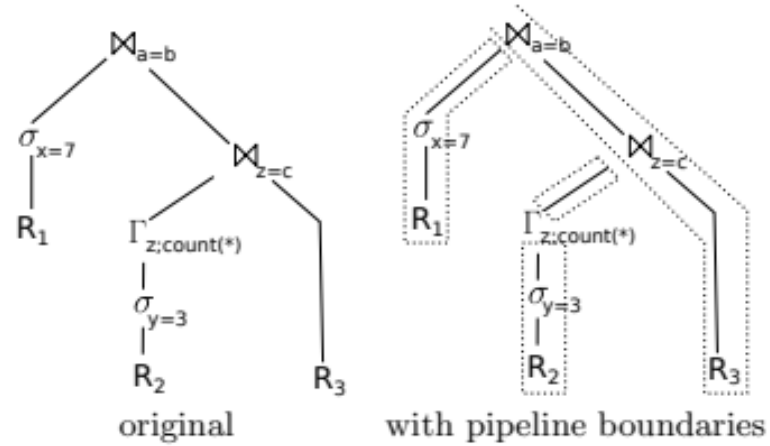


Figure 3: Example Execution Plan for Figure 2

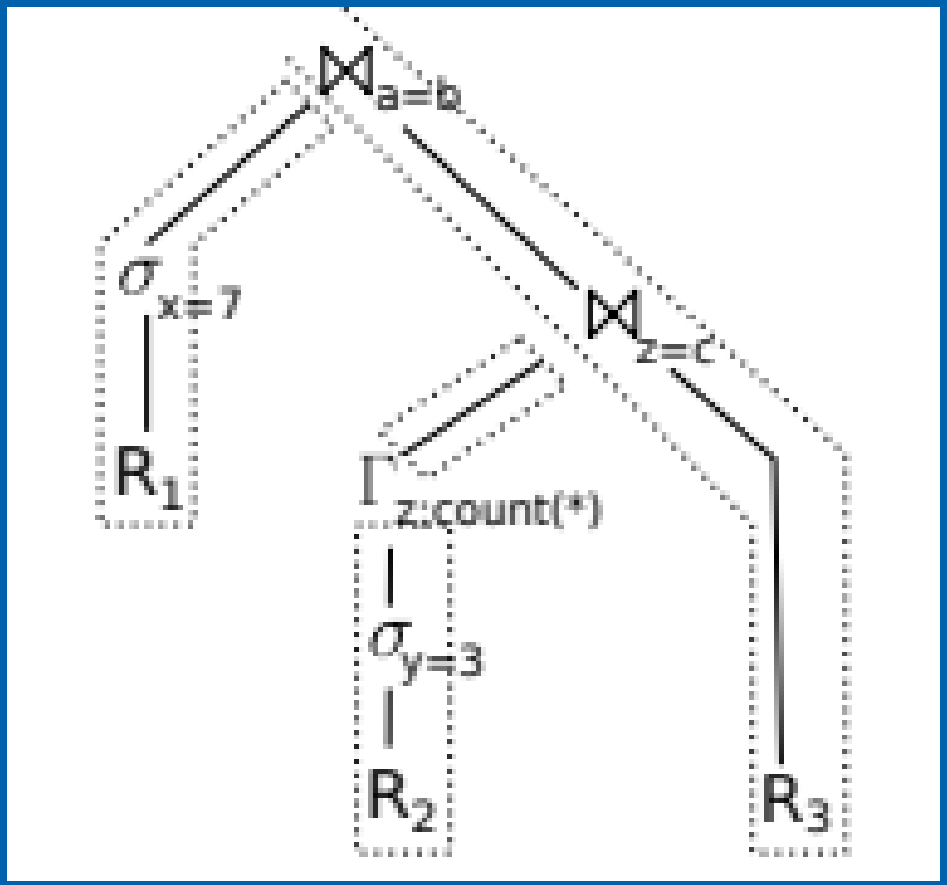
Neumann, Thomas. Efficiently Compiling Efficient Query Plans for Modern Hardware[J]. Proceedings of the VLDB Endowment, 2012, 4(9):539-550.

每个Operator会根据规则拆分为两个代码块，一块对应Produce()，一块对应consume()

代码生成的时候就可以根据这个规则生成代码

Produce()函数负责产生结果tuple

Consume()函数负责具体的tuple处理逻辑




```
[Join a=b].produce
  -> [Selection x=7].produce
    -> [Scan R1].produce    /* 获取R1中的数据 */
      ->  [Selection x=7].consume /* 执行过滤操作 */
         -> [Join a=b].consume  /* 获取left side数据，放入hash table */
  ->  [Join c=z].produce
     ...
```

Neumann, Thomas. Efficiently Compiling Efficient Query Plans for Modern Hardware[J]. Proceedings of the VLDB Endowment, 2012, 4(9):539-550.

Efficiently Compiling Efficient Query Plans for Modern Hardware
代码生成

考虑到动态编译的效率，HyPer中使用了LLVM来生成可以跨平台的IR code，为了降低难度，使用了LLVM提供的API来生成IR code而不是完全手动编写，然后在运行中通过JIT编译器将IR code编译为针对运行平台的机器码

这个代码片段是针对示例中的scan -> selection -> aggregation这个pipeline所生成的汇编代码，用红框标注的是对C++的数据结构的访问(hash table)和函数调用(@_ZN12Hash...)

```
define internal void @scanConsumer(%8* %executionState, %Fragment_R2* %data) {
body:
  ...
  %columnPtr = getelementptr inbounds %Fragment_R2* %data, i32 0, i32 0
  %column = load i32** %columnPtr, align 8
  %columnPtr2 = getelementptr inbounds %Fragment_R2* %data, i32 0, i32 1
  %column2 = load i32** %columnPtr2, align 8
  ... (loop over tuples, currently at %id, contains label %cont17)
  %yPtr = getelementptr i32* %column, i64 %id
  %y = load i32* %yPtr, align 4
  %cond = icmp eq i32 %y, 3
  br i1 %cond, label %then, label %cont17
then:
  %zPtr = getelementptr i32* %column2, i64 %id
  %z = load i32* %zPtr, align 4
  %hash = urem i32 %z, %hashTableSize
  %hashSlot = getelementptr %"HashGroupify::Entry"** %hashTable, i32 %hash
  %hashIter = load %"HashGroupify::Entry"** %hashSlot, align 8
  %cond2 = icmp eq %"HashGroupify::Entry"* %hashIter, null
  br i1 %cond, label %loop20, label %else26
  ... (check if the group already exists, starts with label %loop20)
else26:
  %cond3 = icmp le i32 %spaceRemaining, i32 8
  br i1 %cond, label %then28, label %else47
  ... (create a new group, starts with label %then28)
else47:
  %ptr = call i8* @_ZN12HashGroupify15storeInputTupleEmj
          (%"HashGroupify"* %1, i32 hash, i32 8)
  ... (more loop logic)
}
```

1. locate tuples in memory
2. loop over all tuples
3. filter $y = 3$
4. hash z
5. lookup in hash table (C++ data structure)
6. not found, check space
7. full, call C++ to allocate mem or spill

Figure 7: LLVM fragment for the first steps of the query $\Gamma_{z;count(*)}(\sigma_{y=3}(R_2))$

Neumann, Thomas. Efficiently Compiling Efficient Query Plans for Modern Hardware[J]. Proceedings of the VLDB Endowment, 2012, 4(9):539-550.

总结

文章描述了在查询执行中应用动态编译技术所能带来的好处，并描述了HyPer中这种以**数据为中心 + 推送**的执行模型所基于的基本思想：最大化pipeline执行中数据维持在CPU的时间，从而最小化memory access的次数与代价

Neumann, Thomas. Efficiently Compiling Efficient Query Plans for Modern Hardware[J]. Proceedings of the VLDB Endowment, 2012, 4(9):539-550.

总结

优化方向：

1. **批处理问题：**如果将push one tuple at a time改变为一次一批tuple，可以利用现代CPU的SIMD指令 + 寄存器实现更高效的处理，在LLVM中，有原生的类型(vector）对SIMD的支持
2. **并行问题：**每个tuple做一条pipeline，无法利用多个tuple的loop-pipelining来最大化现代CPU的并行能力
3. **编译器效率问题：**查询编译本身的启动成本还是较高的，即使是LLVM大概也要在ms级别（取决于优化等级），Neumann后续的研究提出了一种自适应的动态编译策略，用来解决这个问题

Neumann, Thomas. Efficiently Compiling Efficient Query Plans for Modern Hardware[J]. Proceedings of the VLDB Endowment, 2012, 4(9):539-550.

Morsel-Driven Parallelism:
A NUMA-Aware Query Evaluation Framework for the Many-Core Age

本文介绍了一种基于小块数据（morsel）来驱动的数据库**并行查询框架**

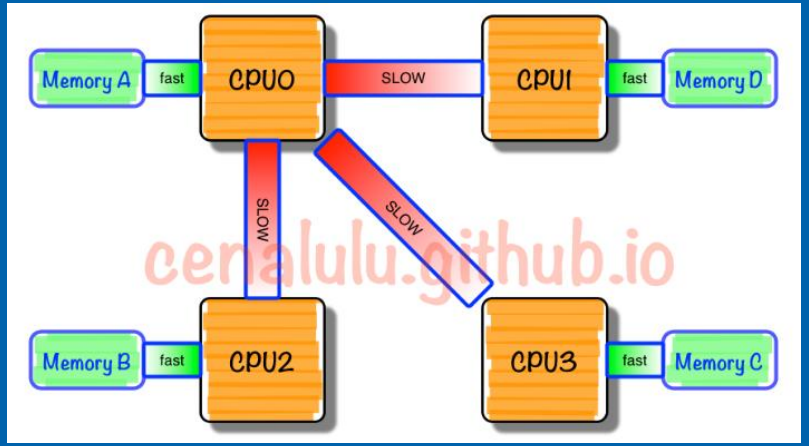
解决在多核心（NUMA环境）下的，分析型查询（OLAP）的并行查询问题

Leis V，Boncz P，Kemper A，et al. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age[M]. ACM, 2014.

➢ NUMA环境

NUMA是一种关于多个CPU如何访问内存的架构模型

现在的CPU基本都是NUMA架构，Linux内核2.5开始支持NUMA



NUMA架构表现为，一个物理CPU（一般包含多个逻辑CPU或者说多个核心）构成一个node，这个node不仅包括CPU，还包括一组内存插槽，也就是说一个物理CPU以及一块内存构成了一个node。每个CPU可以访问自己node下的内存，也可以访问其他node的内存，但是访问速度是不一样的，本地的node更快

Leis V，Boncz P，Kemper A，et al. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age[M]. ACM, 2014.

➢ 并行执行调度方案

自适应的数据驱动的调度执行方案：系统使用固定数量的线程池，各个查询的执行过程中的数据，被切分为细粒度的单元(morsel)，然后结合对其进行处理的operator pipeline，封装为task，交给worker thread pool去执行

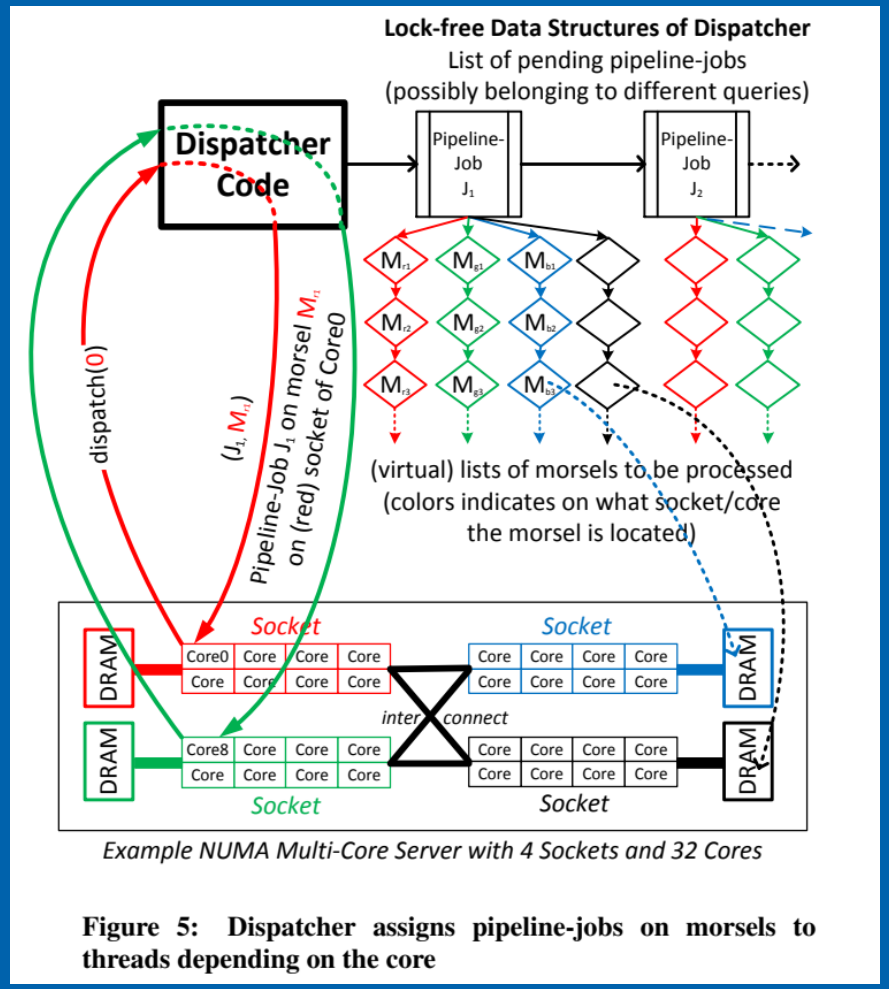


Figure 5: Dispatcher assigns pipeline-jobs on morsels to threads depending on the core

Leis V , Boncz P , Kemper A , et al. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age[M]. ACM, 2014.

➢ 结构测试方案&结果：

4-socket Nehalem EX (Intel Xeon X7560 at 2.3GHz)

4-socket Sandy Bridge EP
(Intel Xeon E5-4650L at 2.6GHz-3.1GHz).

both systems have 32 cores, 64 hardware threads, and almost the same amount of cache

system	geo. mean	sum	scal.
HyPer	0.45s	15.3s	28.1×
Vectorwise	2.84s	93.4s	9.3×
Vectorwise, full-disclosure settings	1.19s	41.2s	8.4×

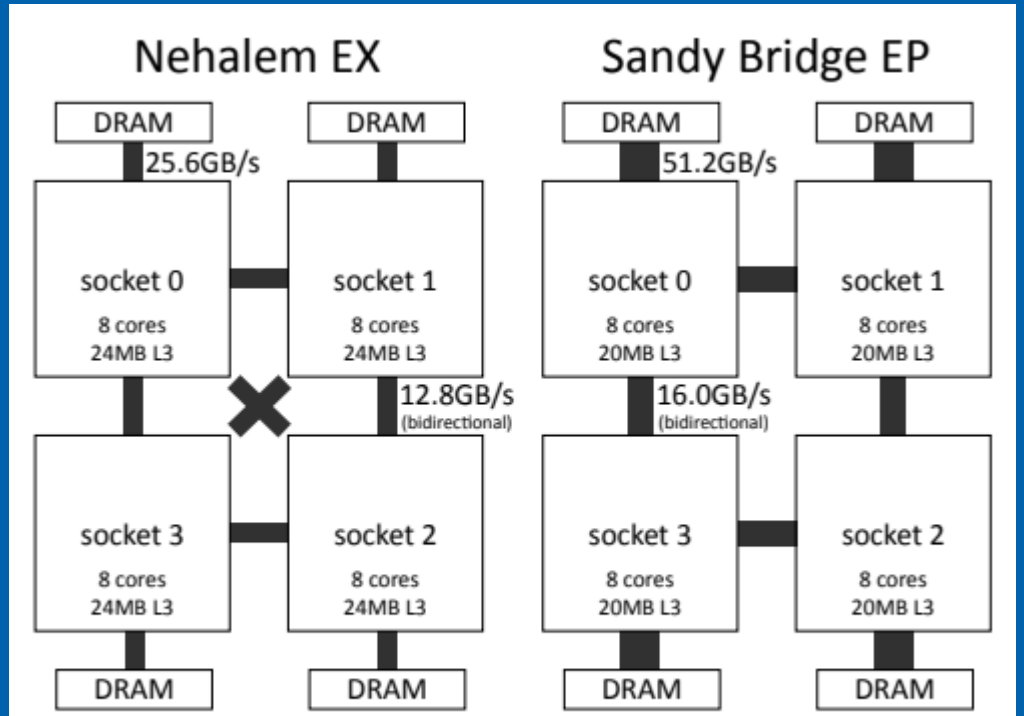


Figure 10: NUMA topologies, theoretical bandwidth

Leis V , Boncz P , Kemper A , et al. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age[M]. ACM, 2014.

Morsel-Driven Parallelism:
A NUMA-Aware Query Evaluation Framework for the Many-Core Age

西安电子科技大学 XIDIAN UNIVERSITY

➢ 实验结果：

在两个不同多核架构上，进行TPC-H的22个SQL语句查询

所有的在HyPer数据库查询都是在3s内运行结束的

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
time [s]	0.21	0.10	0.63	0.30	0.84	0.14	0.56	0.29	2.44	0.61	0.10	0.33	2.32	0.33	0.33	0.81	0.40	1.66	0.68	0.18	0.74	0.47
scal. [×]	39.4	17.8	18.6	26.9	28.0	42.8	25.3	33.3	21.5	21.0	27.4	41.8	16.5	15.6	20.5	11.0	34.0	29.1	29.6	33.7	26.4	8.4

Table 2: TPC-H (scale factor 100) performance on Sandy Bridge EP

	HyPer				[%]		Vectorwise				[%]	
TPC-H #	time [s]	scal. [×]	rd. [GB/s]	wr.	remote QPI		time [s]	scal. [×]	rd. [GB/s]	wr.	remote QPI	
1	0.28	32.4	82.6	0.2	1	40	1.13	30.2	12.5	0.5	74	7
2	0.08	22.3	25.1	0.5	15	17	0.63	4.6	8.7	3.6	55	6
3	0.66	24.7	48.1	4.4	25	34	3.83	7.3	13.5	4.6	76	9
4	0.38	21.6	45.8	2.5	15	32	2.73	9.1	17.5	6.5	68	11
5	0.97	21.3	36.8	5.0	29	30	4.52	7.0	27.8	13.1	80	24
6	0.17	27.5	80.0	0.1	4	43	0.48	17.8	21.5	0.5	75	10
7	0.53	32.4	43.2	4.2	39	38	3.75	8.1	19.5	7.9	70	14
8	0.35	31.2	34.9	2.4	15	24	4.46	7.7	10.9	6.7	39	7
9	2.14	32.0	34.3	5.5	48	32	11.42	7.9	18.4	7.7	63	10
10	0.60	20.0	26.7	5.2	37	24	6.46	5.7	12.1	5.7	55	10
11	0.09	37.1	21.8	2.5	25	16	0.67	3.9	6.0	2.1	57	3
12	0.22	42.0	64.5	1.7	5	34	6.65	6.9	12.3	4.7	61	9
13	1.95	40.0	21.8	10.3	54	25	6.23	11.4	46.6	13.3	74	37
14	0.19	24.8	43.0	6.6	29	34	2.42	7.3	13.7	4.7	60	8
15	0.44	19.8	23.5	3.5	34	21	1.63	7.2	16.8	6.0	62	10
16	0.78	17.3	14.3	2.7	62	16	1.64	8.8	24.9	8.4	53	12
17	0.44	30.5	19.1	0.5	13	13	0.84	15.0	16.2	2.9	69	7
18	2.78	24.0	24.5	12.5	40	25	14.94	6.5	26.3	8.7	66	13
19	0.88	29.5	42.5	3.9	17	27	2.87	8.8	7.4	1.4	79	5
20	0.18	33.4	45.1	0.9	5	23	1.94	9.2	12.6	1.2	74	6
21	0.91	28.0	40.7	4.1	16	29	12.00	9.1	18.2	6.1	67	9
22	0.30	25.7	35.5	1.3	75	38	3.14	4.3	7.0	2.4	66	4

Table 1: TPC-H (scale factor 100) statistics on Nehalem EX

Leis V , Boncz P , Kemper A , et al. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age[M]. ACM, 2014.

➢ 总结：

构建了数据库系统中的自适应的NUMA框架使之更好的适应现代CPU的并行处理能力

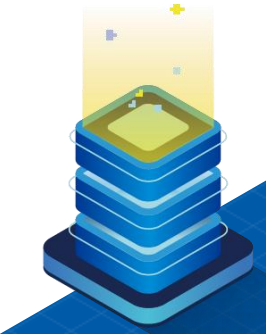
并优化了其中的动态调度策略使之更趋向于NUMA本地执行（NUMA-local）以提高效率

这解决了上一篇文章遗留的一个问题（并行性）

现在还需要解决编译执行本身的**编译效率问题**

Leis V，Boncz P，Kemper A，et al. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age[M]. ACM, 2014.

目 录



- 原始数据库编译执行

  查询编译的启动本身成本是相对较高的

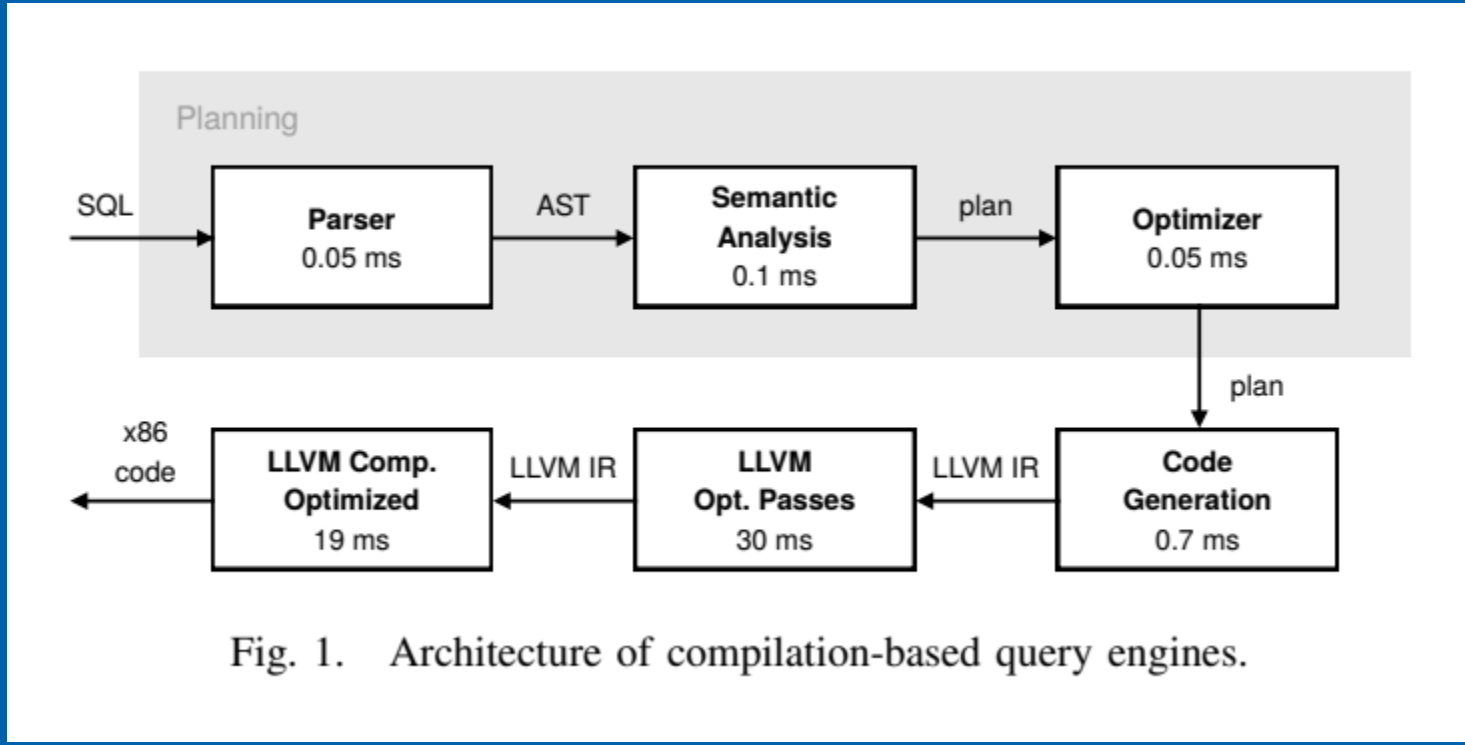


Fig. 1. Architecture of compilation-based query engines.

Kohn A , Leis V , Neumann T . Adaptive Execution of Compiled Queries[C]// 2018 IEEE 34th International Conference on Data Engineering (ICDE). IEEE, 2018.

- 数据库自适应编译执行

  解决查询编译的启动本身成本是相对较高的问题

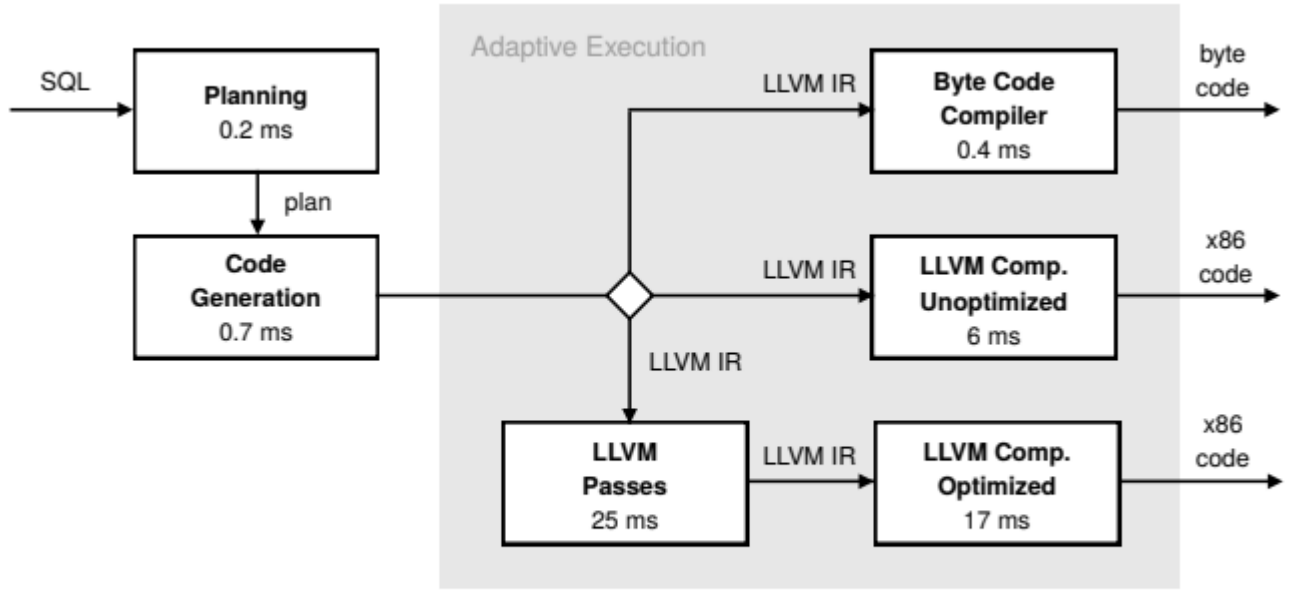


Fig. 3. Execution modes and their compilation times.

基于LLVM bytecode解析执行（高效）

基于LLVM IR做无优化的编译

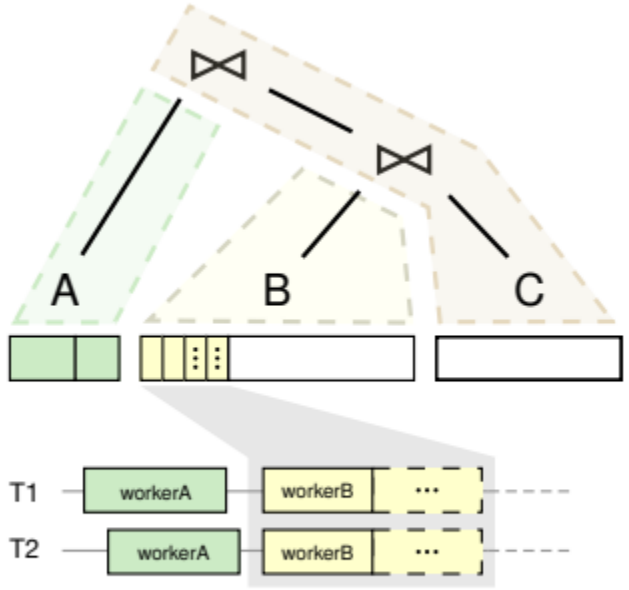
生成machine code

基于LLVM IR做有优化的编译

生成更高效的machine code

Kohn A , Leis V , Neumann T . Adaptive Execution of Compiled Queries[C]// 2018 IEEE 34th International Conference on Data Engineering (ICDE). IEEE, 2018.

➢ 自适应编译执行实现方式：




```
queryStart():
  state = initQueryState()
  state.Ha = createHashTable()
  state.Hb = createHashTable()
  ...
  schedule( workerA )
  ...
  schedule( workerB )
  ...
  schedule( workerC )
```

```
schedule(worker):
  // assign morsels to threads
  ...
```

```
workerA(state, morsel):
  for i in morsel:
    state.Ha.insert(A[i])
```

```
workerB(state, morsel):
  for i in morsel:
    state.Hb.insert(B[i])
```

```
workerC(state, morsel):
  for i in morsel:
    for tb in state.Hb.lookup(C[i]):
      for ta in state.Ha.lookup(tb):
        ...
```

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.

Kohn A , Leis V , Neumann T . Adaptive Execution of Compiled Queries[C]// 2018 IEEE 34th International Conference on Data Engineering (ICDE). IEEE, 2018.

- 自适应编译执行的框架：

为了实现自适应编译，需要3个基本组件：

1. 高效的 ByteCode 解析器
2. Track 执行过程中的信息，从而动态决定是否切换，而不是提前利用optimizer的cost/card estimation来静态决定 （信息统计）
3. 根据推断动态切换执行模式 （切换规则、方式）

Kohn A，Leis V，Neumann T . Adaptive Execution of Compiled Queries[C]// 2018 IEEE 34th International Conference on Data Engineering (ICDE). IEEE, 2018.

➢ 自适应编译执行的执行模式

1. 所有query总是以解析模式开始，多worker threads并行执行
2. 在执行中通过收集+推断
3. 当判断后续采用compile方式更有益时，则使用一个worker thread开始异步编译过程，其他worker继续解析执行
4. 当编译完成时，后续所有worker都切换到compiled function继续执行

这种处理方式是以pipeline为单元的，并以morsel为监控+推断+切换的基本单位。

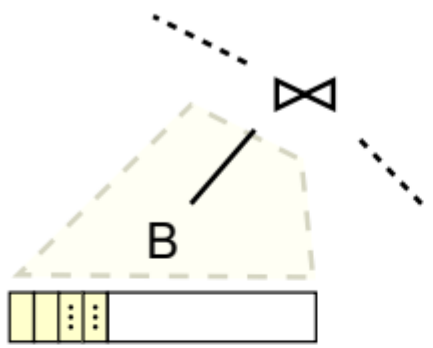
Kohn A，Leis V，Neumann T . Adaptive Execution of Compiled Queries[C]// 2018 IEEE 34th International Conference on Data Engineering (ICDE). IEEE, 2018.

➢ 自适应切换规则&方式：

```
// 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.

r0/r1/r2是处理tuple的速率
c1/c2是编译时间
t0/t1/t2是计算的剩余执行时间




```
handleB.byteCode:
  0x00 load_i64 40 8 0
  0x14 load_i64 48 8 64
  0x28 icmp_ult_i64 56 40 48
  0x50 condbr 56 0x64 0xf0
  ...
```

```
handleB.fn:
  0x00 mov rax, [r12]
  0x04 mov rbx, [r12+8]
  0x08 cmp rax, rbx
  0x0b jnl 0xf00
  ...
```

```
dispatch( handleB , state):
  nextMorsel = grabMorsel()
  if (handleB.isCompiled()):
    handleB.fn(state, nextMorsel)
  else:
    VM.execute(handleB.byteCode, state, nextMorsel)
  // switch execution mode?
  choice = extrapolatePipelineDurations(...)
  if (choice != DoNothing):
    runAsync(λ -> handleB.fn = handleB.compile(choice))
```

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

Kohn A , Leis V , Neumann T . Adaptive Execution of Compiled Queries[C]// 2018 IEEE 34th International Conference on Data Engineering (ICDE). IEEE, 2018.

➢ 性能分析：

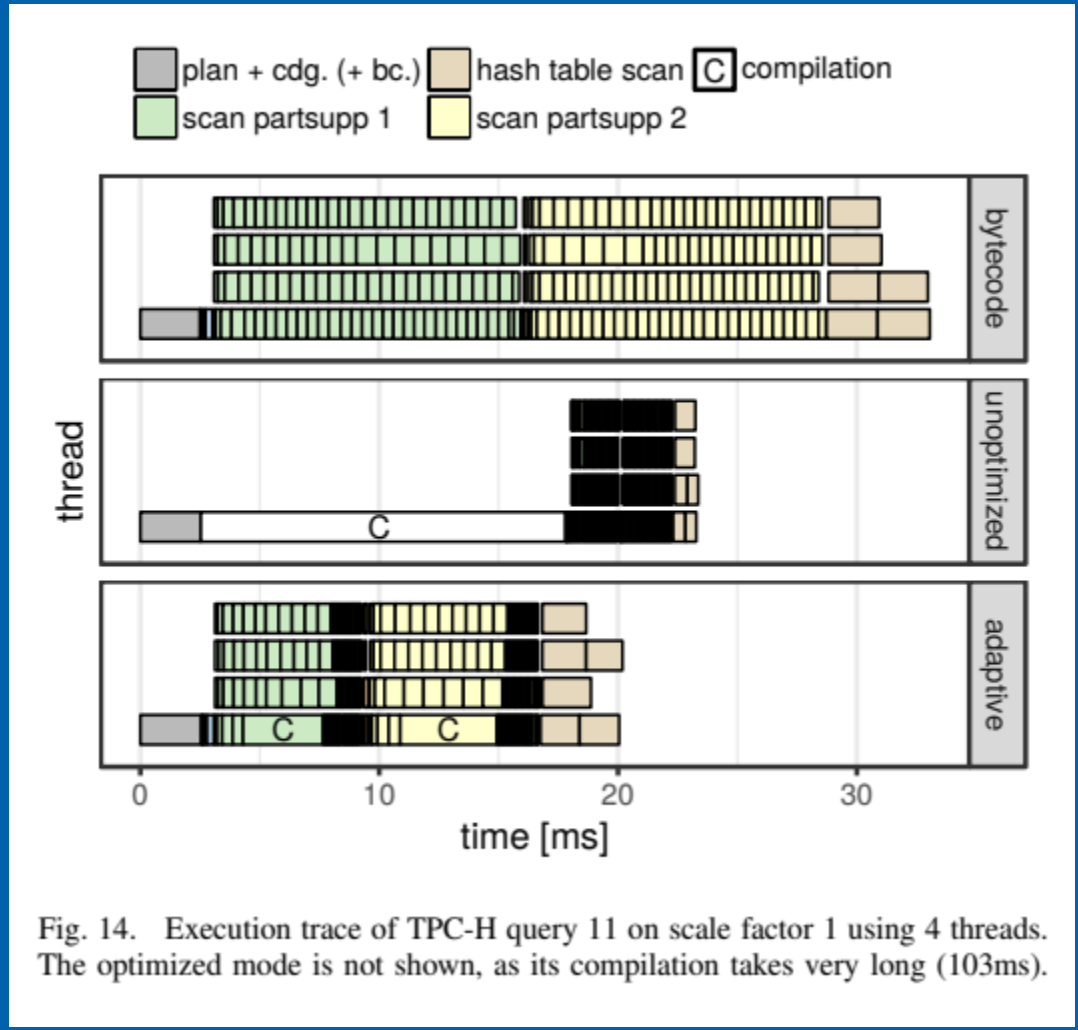
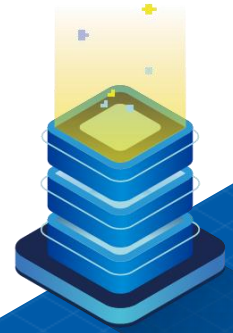


Fig. 14. Execution trace of TPC-H query 11 on scale factor 1 using 4 threads. The optimized mode is not shown, as its compilation takes very long (103ms).

Kohn A，Leis V，Neumann T . Adaptive Execution of Compiled Queries[C]// 2018 IEEE 34th International Conference on Data Engineering (ICDE). IEEE, 2018.

目 录



总结一下

- Efficiently Compiling Efficient Query Plans for Modern Hardware

  这篇文章是SQL编译查询很经典的一篇文章，主要讲解了HyPer数据库对于CodeGen的应用

- Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age

  这篇文章主要解决了现代CPU体系下并行执行的动态调度及执行问题

- Adaptive Execution of Compiled Queries

  这篇文章主要基于前两篇文章的编译执行体系下增加了自适应编译的执行框架，通过引入自适应编译执行的方式，将编译执行系统的实用性进一步提升

THANKS

关注我们 / 了解更多