Data Processing on Modern Hardware

Jana Giceva

Lecture 3: Cache awareness

for query execution models

Cache awareness for query execution models

Processing models

The ***processing model*** of a database defines ***how the system executes the query plan***.

The four main approaches are:

- ***Iterator*** model (volcano, tuple-at-a-time)
- ***Materialization*** model (operator-at-a-time, column-at-a-time)
- ***Vectorization*** model (vector-at-a-time, batch, block-wise)
- ***Pushing tuples up*** model

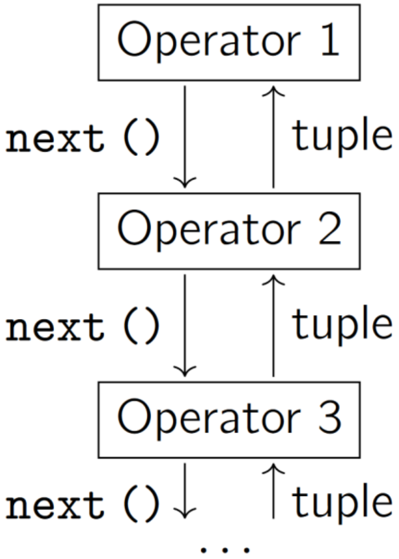
There are different trade-offs depending on the workload type and the underlying hardware.

→ cf. Database Systems on Modern CPU Architectures (chapter 5)

Iterator model

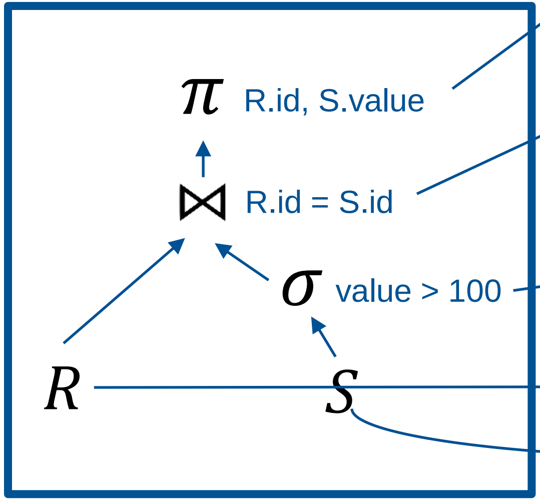
Most classical systems implement the ***Volcano iterator model:***

- Operators request tuples from their input using `next()`
  - On each invocation, the operator returns either a single tuple or `null` if there are no more tuples
- Data is processed ***tuple-at-a-time*** in a ***pipelined*** fashion
  - Also called the Volcano or pipeline model
- Each operator keeps its own ***state***



Iterator model – Example

```
SELECT R.id, S.cdate
FROM   R JOIN S
ON     R.id = S.id
WHERE  S.value > 100
```



```
for t in child.Next():
  emit(projection(t))
```

```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

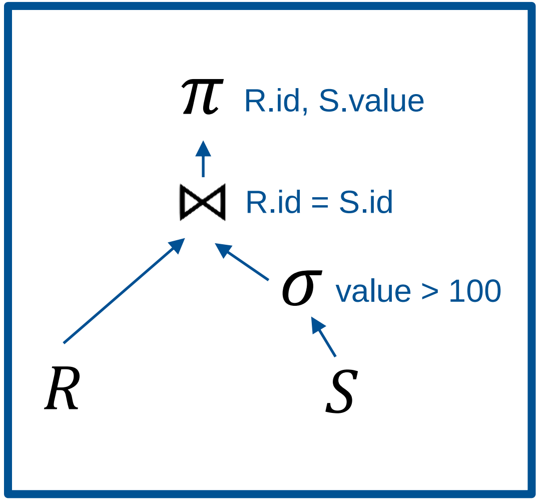
```
for t in child.Next():
  if evalPred(t): emit(t)
```

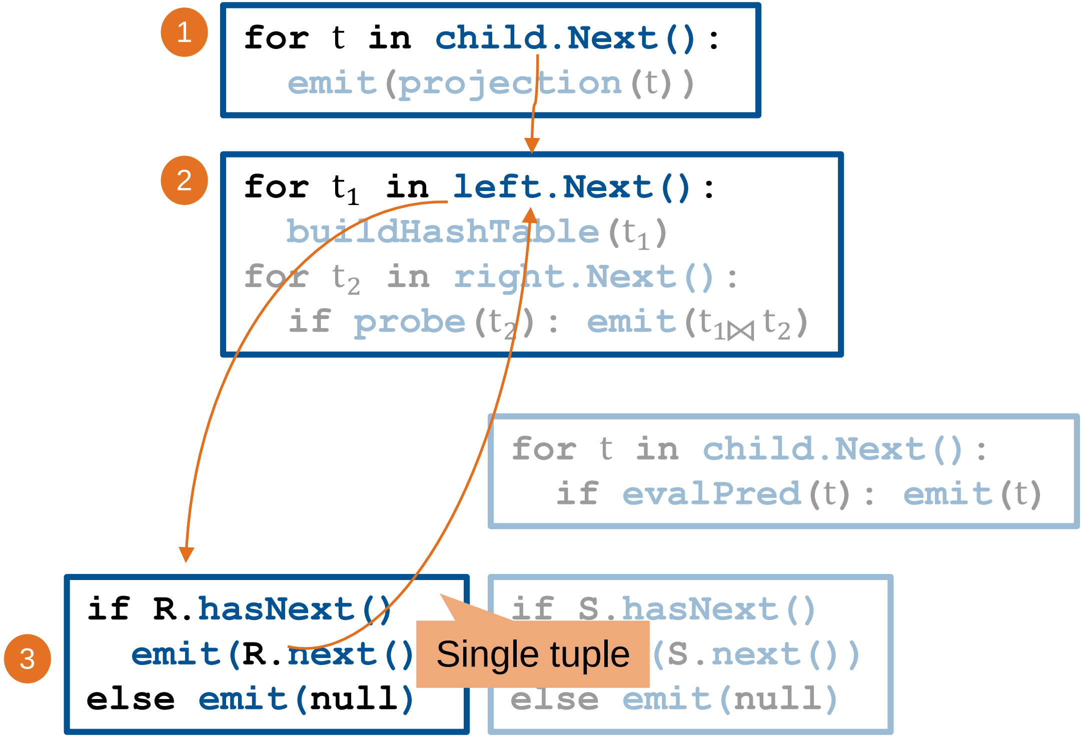
```
if R.hasNext()
  emit(R.next())
else emit(null)
```

```
if S.hasNext()
  emit(S.next())
else emit(null)
```

Iterator model – Example

```
SELECT R.id, S.cdate
FROM   R JOIN S
ON     R.id = S.id
WHERE  S.value > 100
```





1

```
for t in child.Next():
  emit(projection(t))
```

2

```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

```
for t in child.Next():
  if evalPred(t): emit(t)
```

3

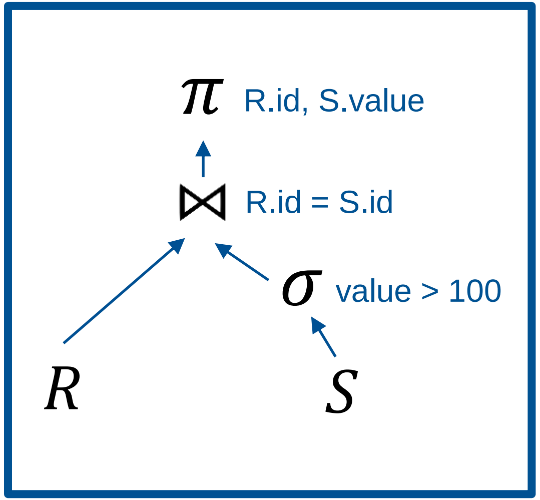
```
if R.hasNext()
  emit(R.next())
else emit(null)
```

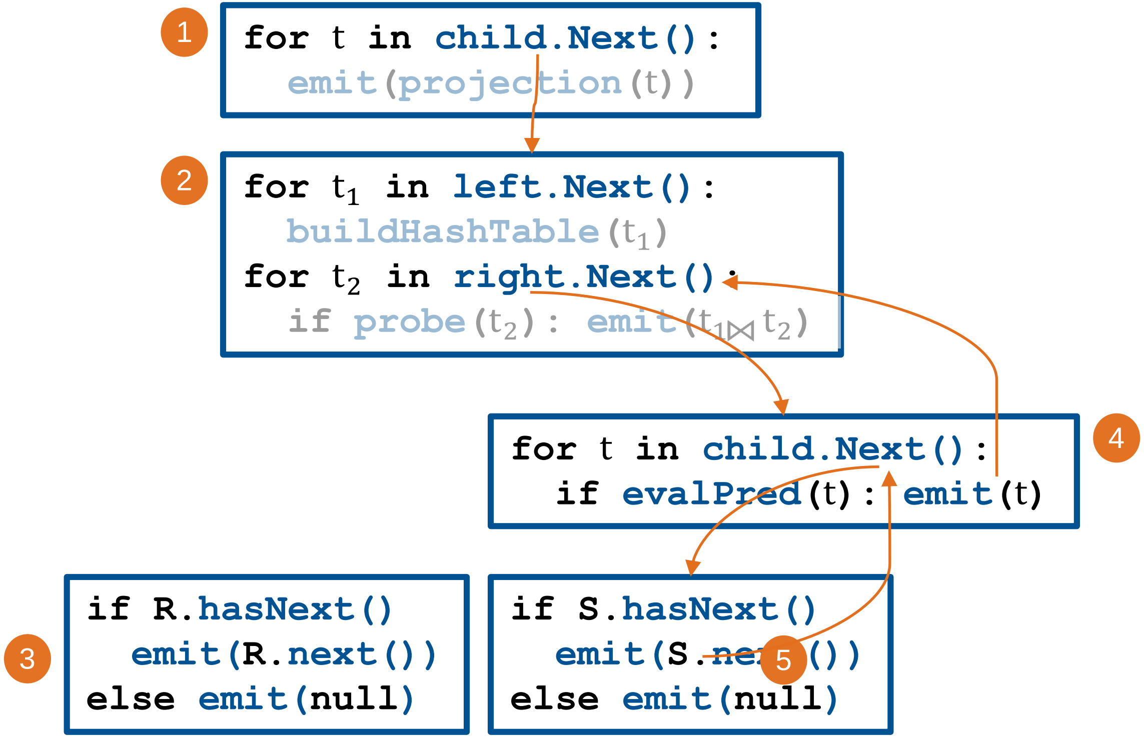
```
if S.hasNext()
  emit(S.next())
else emit(null)
```

Single tuple

Iterator model – Example

```
SELECT R.id, S.cdate
FROM   R JOIN S
ON     R.id = S.id
WHERE  S.value > 100
```





```
for t in child.Next():
  emit(projection(t))
```

```
for $t_1$ in left.Next():
  buildHashTable($t_1$)
for $t_2$ in right.Next():
  if probe($t_2$): emit($t_1 \bowtie t_2$)
```

```
for t in child.Next():
  if evalPred(t): emit(t)
```

```
if R.hasNext()
  emit(R.next())
else emit(null)
```

```
if S.hasNext()
  emit(S.next())
else emit(null)
```

Iterator model

This is used in almost every RDBMS.

- Allows for tuple pipelining.
- Some operators must block until their children emit all their tuples:
  - Joins, subqueries, sort, group-by, etc.

Implications on cache usage efficiency:

- All operators in a plan run ***tightly interleaved***
  - Their ***combined*** instruction ***footprint*** may be ***large***
  - Many ***instruction cache misses***
- Operators constantly call each other's functionality
  - Results in a big ***function call overhead***
- The combined ***state of the operators*** may be too large to fit into caches
  - *e.g.*, hash tables, cursors, partial aggregates
  - Results in many ***data cache misses***

Example: TPC-H on MySQL

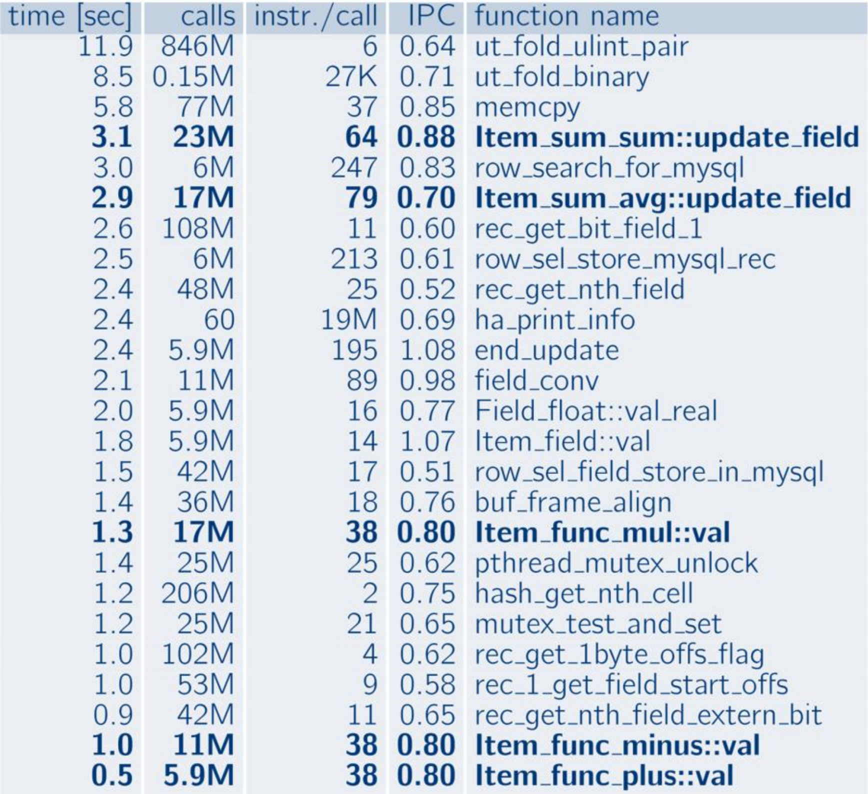
Example: Query Q1 from the TPC-H benchmark on MySQL

```
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
```

- **Scan query** with **arithmetics** on **aggregated tuples** without a join

Results taken from MonetDB/X100: Hyper-Pipelining Query Execution *CIDR 2005*

Show results from executing the query



time [sec]	calls	instr./call	IPC	function name
11.9	846M	6	0.64	ut_fold_ulint_pair
8.5	0.15M	27K	0.71	ut_fold_binary
5.8	77M	37	0.85	memcpy
3.1	**23M**	**64**	**0.88**	**Item_sum_sum::update_field**
3.0	6M	247	0.83	row_search_for_mysql
2.9	**17M**	**79**	**0.70**	**Item_sum_avg::update_field**
2.6	108M	11	0.60	rec_get_bit_field_1
2.5	6M	213	0.61	row_sel_store_mysql_rec
2.4	48M	25	0.52	rec_get_nth_field
2.4	60	19M	0.69	ha_print_info
2.4	5.9M	195	1.08	end_update
2.1	11M	89	0.98	field_conv
2.0	5.9M	16	0.77	Field_float::val_real
1.8	5.9M	14	1.07	Item_field::val
1.5	42M	17	0.51	row_sel_field_store_in_mysql
1.4	36M	18	0.76	buf_frame_align
1.3	**17M**	**38**	**0.80**	**Item_func_mul::val**
1.4	25M	25	0.62	pthread_mutex_unlock
1.2	206M	2	0.75	hash_get_nth_cell
1.2	25M	21	0.65	mutex_test_and_set
1.0	102M	4	0.62	rec_get_1byte_offs_flag
1.0	53M	9	0.58	rec_1_get_field_start_offs
0.9	42M	11	0.65	rec_get_nth_field_extern_bit
1.0	**11M**	**38**	**0.80**	**Item_func_minus::val**
0.5	**5.9M**	**38**	**0.80**	**Item_func_plus::val**

Each call only processes a ***single tuple*** → ***millions of calls***

Only ***10% of the time*** spent on actual query task.

Very low ***instructions-per-cycle*** (IPC) ratio.

Further observations

Much time spent on field access (*e.g.*, **`rec_get_nth_field()`**).

- Row-store → polymorphic operators.

Single-tuple functions are hard to optimize (by compiler):

- Low IPC ratio – empty pipelines make the CPU stall
- Optimization across functions not possible (or **very** difficult)
- Function call overhead is high
- Vector instructions (SIMD) are hardly applicable

Example:

- Let’s consider the **`Item_func_plus::val`** function from the previous table
- $\frac{38 \text{ instr.}}{0.8 \,{}^{\text{instr.}}/_{\text{cycle}}} = 48 \text{ cycles}$ vs. 3 instructions for load/add/store assembly
- One explanation for this high cost is the absence of *loop pipelining,* dependent instructions → 20 cycles
- High cost of a function (routine) call (~ 20 cycles) that cannot be amortized

Materialization model

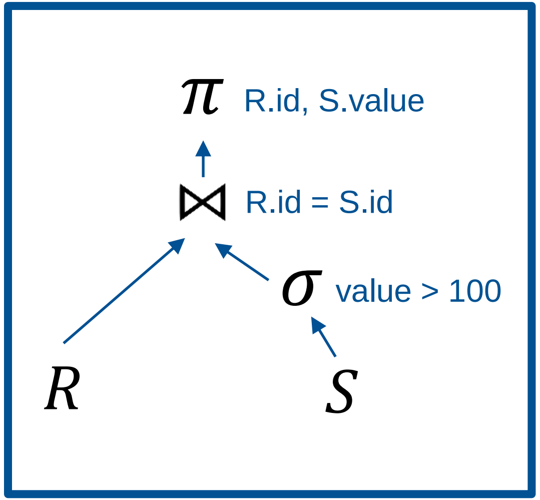
Each operator processes its input all at once and then stores its output all at once (in one buffer)

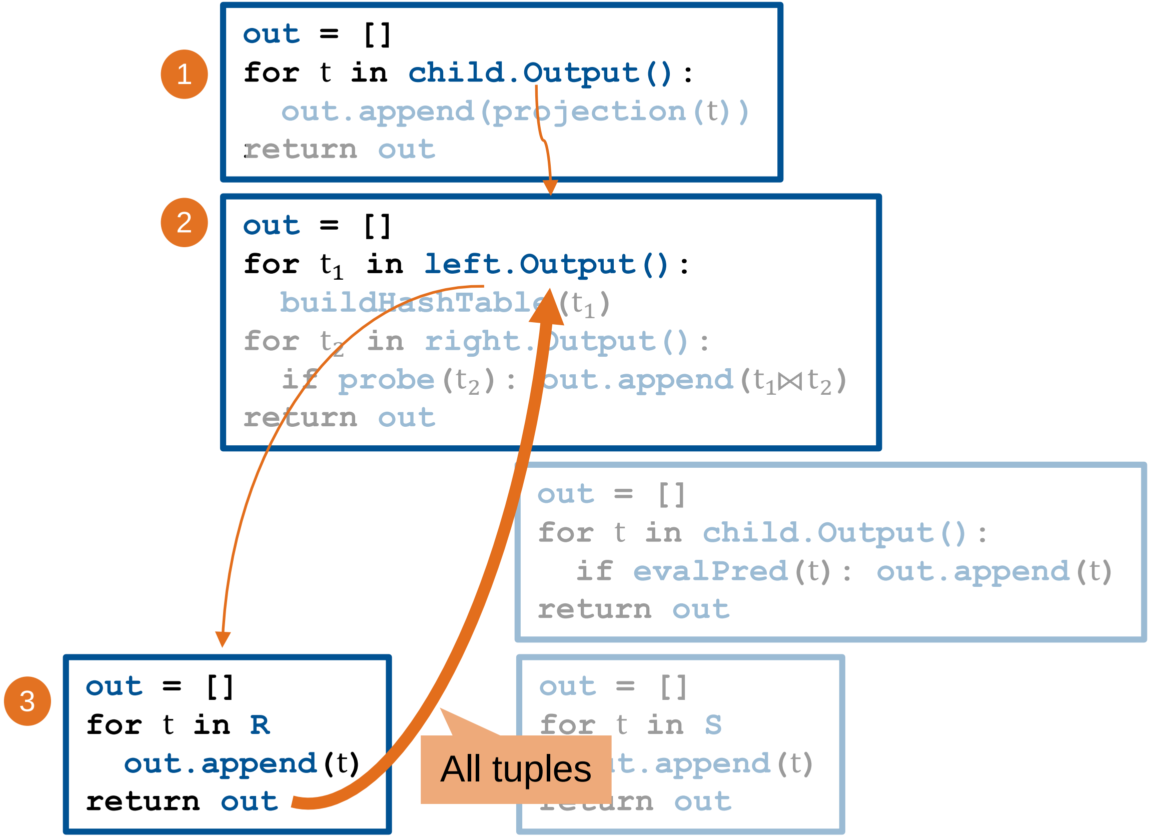
- Operators consume and produce ***full columns*** (or tables).
- Each (sub-)result is ***fully materialized*** (in memory)
- ***No*** pipelining (rather a sequence of statements)
- Each operator runs exactly once.

The output can be either a whole tuple (row-store) or subsets of columns (column-store).

Materialization model – Example

```
SELECT R.id, S.cdate
FROM   R JOIN S
ON     R.id = S.id
WHERE  S.value > 100
```





1

```
out = []
for t in child.Output():
  out.append(projection(t))
return out
```

2

```
out = []
for $t_1$ in left.Output():
  buildHashTable($t_1$)
for $t_2$ in right.Output():
  if probe($t_2$): out.append($t_1$⋈$t_2$)
return out
```

```
out = []
for t in child.Output():
  if evalPred(t): out.append(t)
return out
```

3

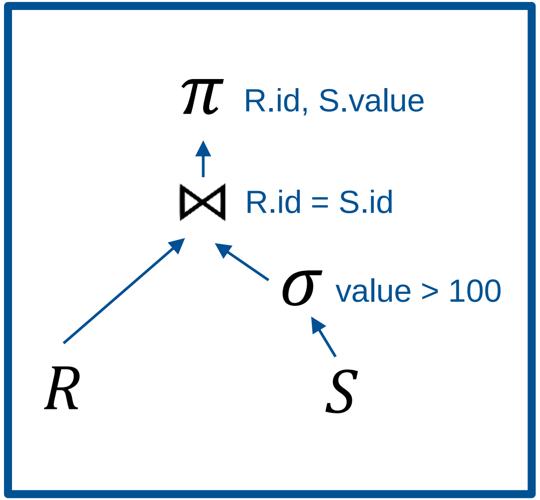
```
out = []
for t in R
  out.append(t)
return out
```

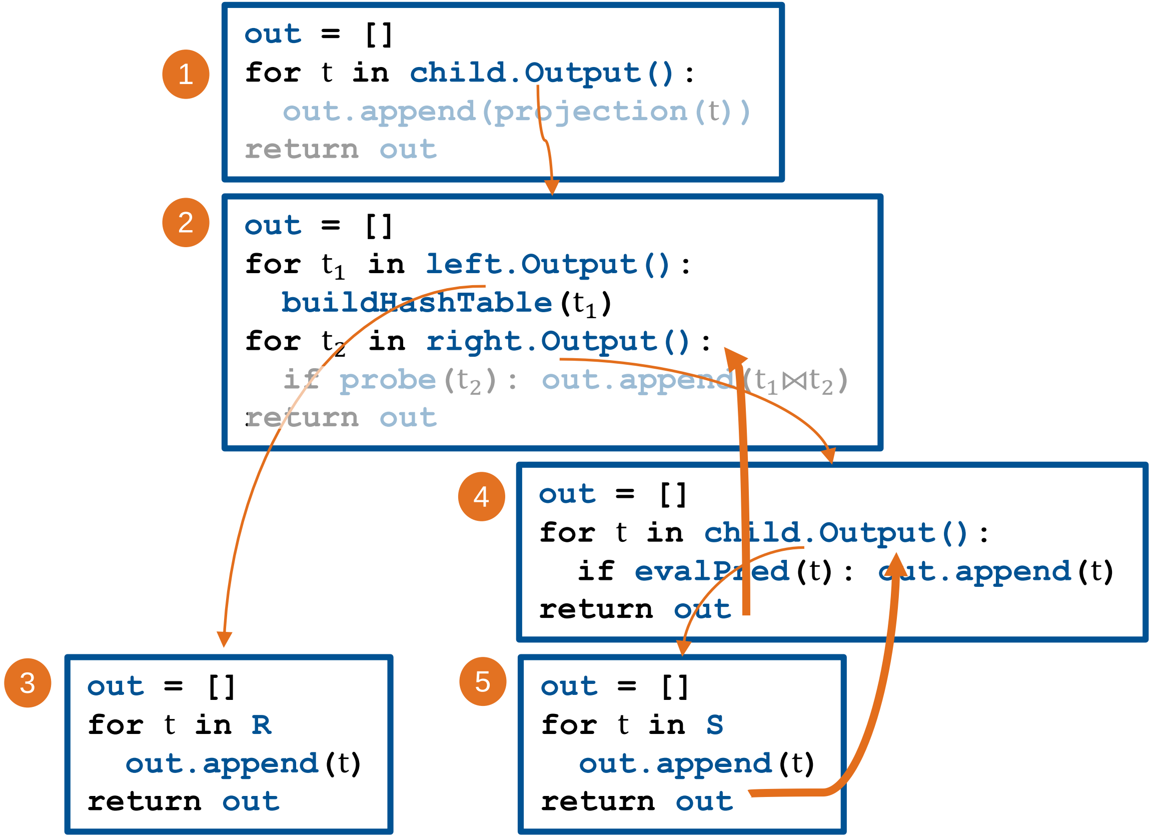
```
out = []
for t in S
  out.append(t)
return out
```

All tuples

Materialization model – Example

```
SELECT R.id, S.cdate
FROM   R JOIN S
ON     R.id = S.id
WHERE  S.value > 100
```





1
```
out = []
for t in child.Output():
    out.append(projection(t))
return out
```

2
```
out = []
for $t_1$ in left.Output():
    buildHashTable($t_1$)
for $t_2$ in right.Output():
    if probe($t_2$): out.append($t_1 \bowtie t_2$)
return out
```

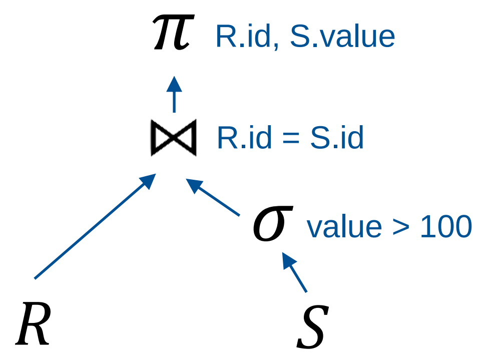
3
```
out = []
for t in R
    out.append(t)
return out
```

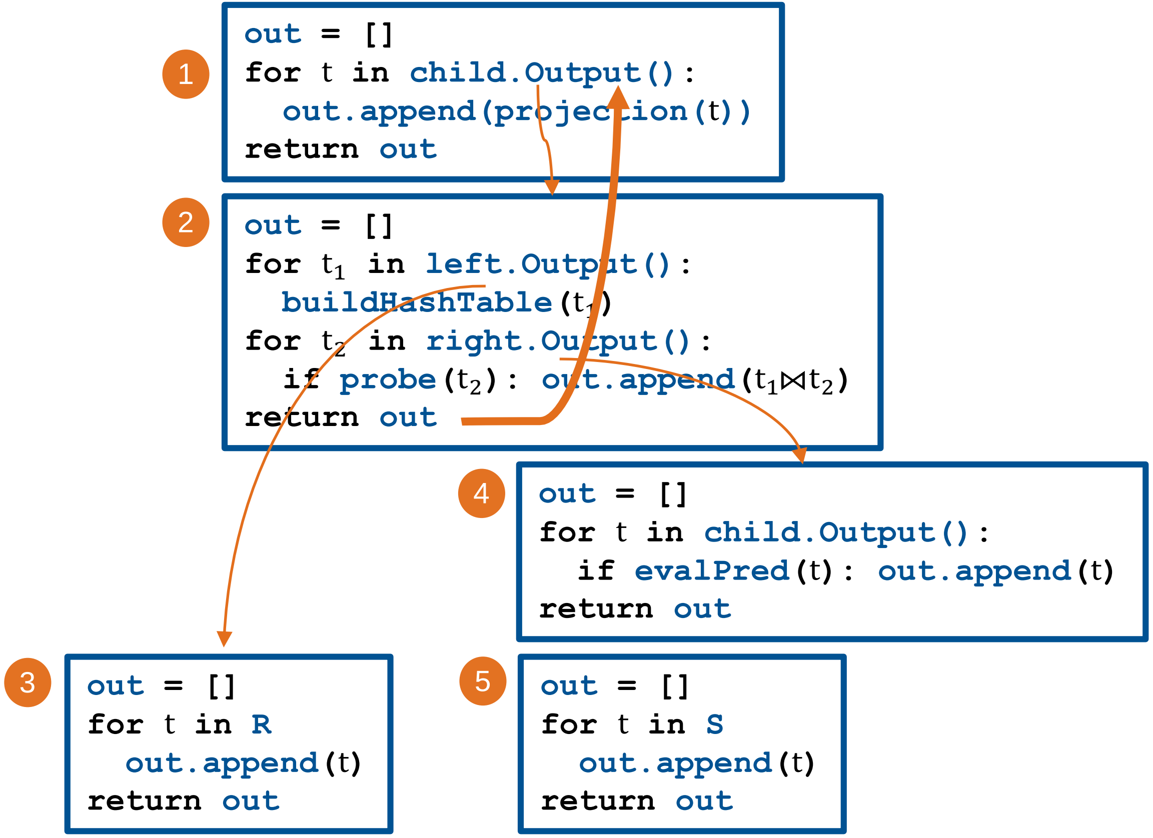
4
```
out = []
for t in child.Output():
    if evalPred(t): out.append(t)
return out
```

5
```
out = []
for t in S
    out.append(t)
return out
```

Materialization model – Example

```
SELECT R.id, S.cdate
FROM   R JOIN S
ON     R.id = S.id
WHERE  S.value > 100
```





1

```
out = []
for t in child.Output():
  out.append(projection(t))
return out
```

2

```
out = []
for $t_1$ in left.Output():
  buildHashTable($t_1$)
for $t_2$ in right.Output():
  if probe($t_2$): out.append($t_1 \bowtie t_2$)
return out
```

3

```
out = []
for t in R
  out.append(t)
return out
```

4

```
out = []
for t in child.Output():
  if evalPred(t): out.append(t)
return out
```

5

```
out = []
for t in S
  out.append(t)
return out
```

Materialization model – analysis

Much fewer number of function calls

Due to such operator-at-a-time processing, its tight loops

- Conveniently ***fit into instruction caches***
- Can be optimized effectively by modern compilers
  - ***Loop unrolling***
  - ***Vectorization*** (use of SIMD instructions)
- Can leverage modern CPU features (***hardware prefetching***)
- Far less expensive function calls are now ***out of the critical code path***

Iterator vs. Materialization model

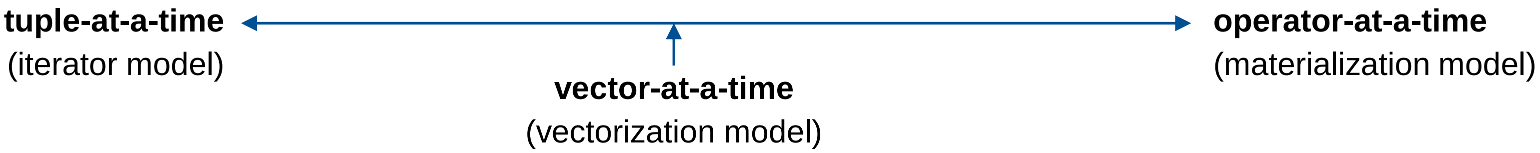
The materialization (operator-at-a-time) model is a two-edged sword:

- Cache-efficient with respect to ***code*** and ***operator state***
- Tight loops, optimizable code

But, each operator reads in and out everything, so

- Data won’t fully fit into the cache:
  - Repeated scans will fetch data ***from memory*** over and over
  - Strategy falls apart when ***intermediate*** (materialized) ***results no*** longer ***fit*** in ***memory/caches***

Can we aim for the middle-ground between the two extremes?



tuple-at-a-time
(iterator model)

vector-at-a-time
(vectorization model)

operator-at-a-time
(materialization model)

Vectorization model

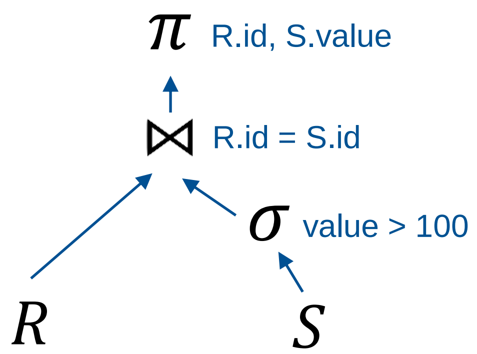
Idea: use volcano-style iteration

But for each `next()` call return a ***batch of tuples*** instead of a single tuple

- Vector in MonetDB/X100 terminology
- The operator’s internal loop processes multiple tuples at a time
- The size of the batch can vary based on the hardware and query properties

Vectorization model – Example

```
SELECT R.id, S.cdate
FROM   R JOIN S
ON     R.id = S.id
WHERE  S.value > 100
```



1

```
out = []
for t in child.Output():
  out.append(projection(t))
if |out|>n: emit(out)
```

2

```
out = []
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.append(t1⋈t2)
if |out|>n: emit(out)
```

3

```
out = []
while R.hasNext() & |out|<n
  out.append(R.next())
emit(out)
```

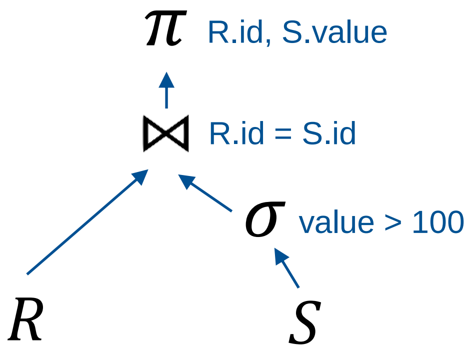
```
out = []
for t in child.Output():
  if evalPred(t): out.append(t)
if |out|>n: emit(out)
```

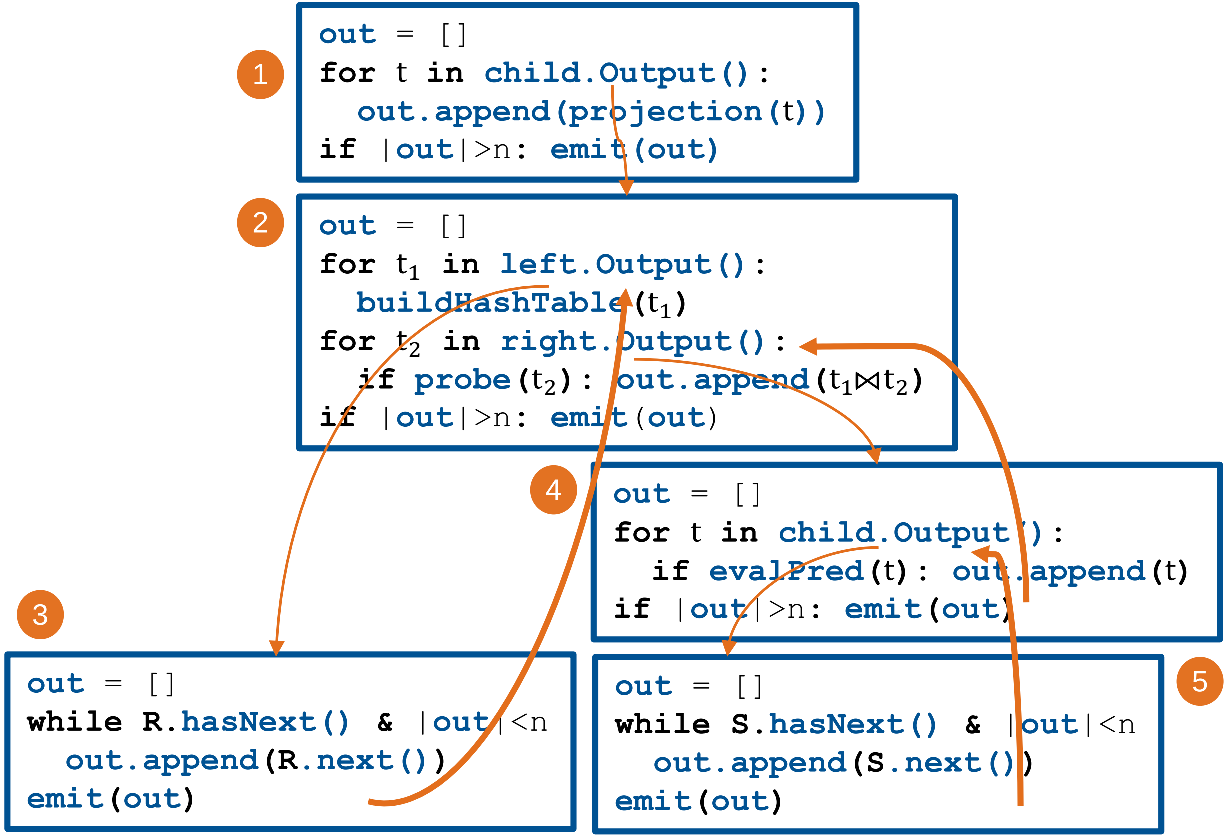
vector of tuples

```
while S.hasNext() & |out|<n
  out.append(S.next())
emit(out)
```

Vectorization model – Example

```
SELECT R.id, S.cdate
FROM   R JOIN S
ON     R.id = S.id
WHERE  S.value > 100
```





1

```
out = []
for t in child.Output():
  out.append(projection(t))
if |out|>n: emit(out)
```

2

```
out = []
for $t_1$ in left.Output():
  buildHashTable($t_1$)
for $t_2$ in right.Output():
  if probe($t_2$): out.append($t_1 \bowtie t_2$)
if |out|>n: emit(out)
```

3

```
out = []
while R.hasNext() & |out|<n
  out.append(R.next())
emit(out)
```

4

```
out = []
for t in child.Output():
  if evalPred(t): out.append(t)
if |out|>n: emit(out)
```

5

```
out = []
while S.hasNext() & |out|<n
  out.append(S.next())
emit(out)
```

Vectorization model – analysis

Uses the best of both worlds (iterator and materialization models):

- Reduces the number of invocations per operator
- Allows for operators to use vectorized (SIMD) instructions to process batches of tuples

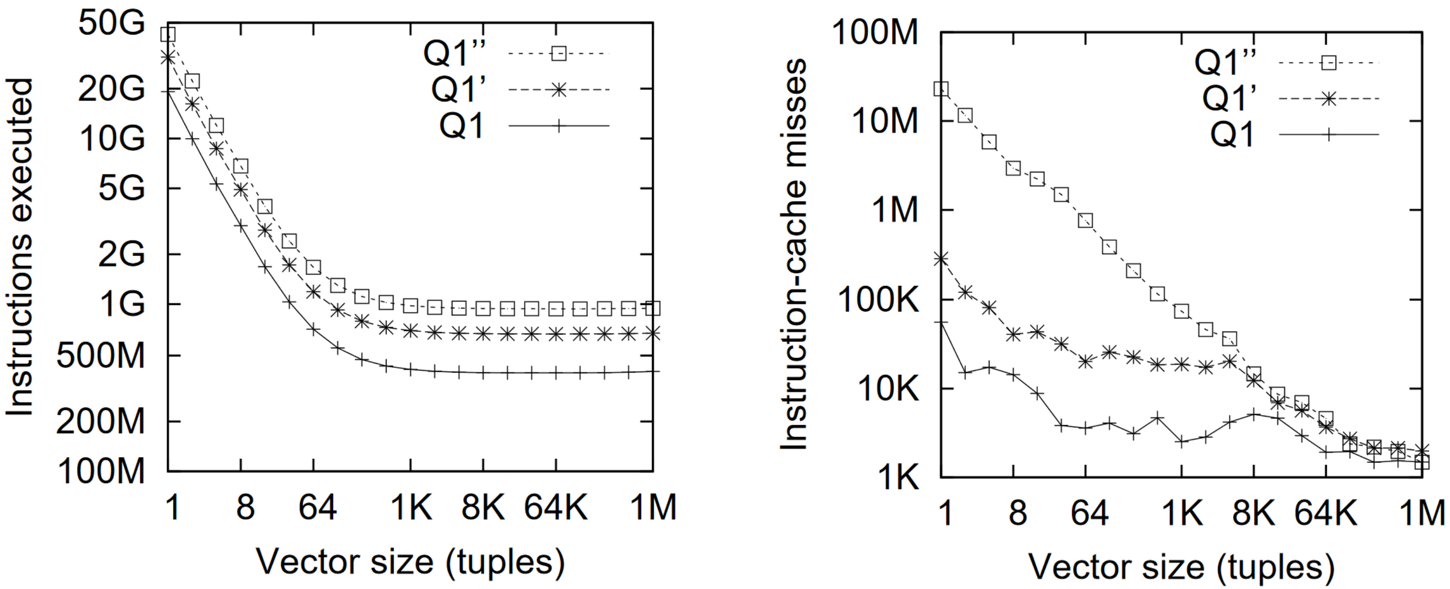
Imperative to choose a vector size that is:

- ***Large enough*** to amortize the iteration overhead (*e.g.*, function calls, instruction cache misses, etc),
- ***Small enough*** to not thrash data caches

Will there be such a vector size?

- Or will caches be thrashed long before iteration overhead is compensated?

Vector size ↔ instruction cache effectiveness

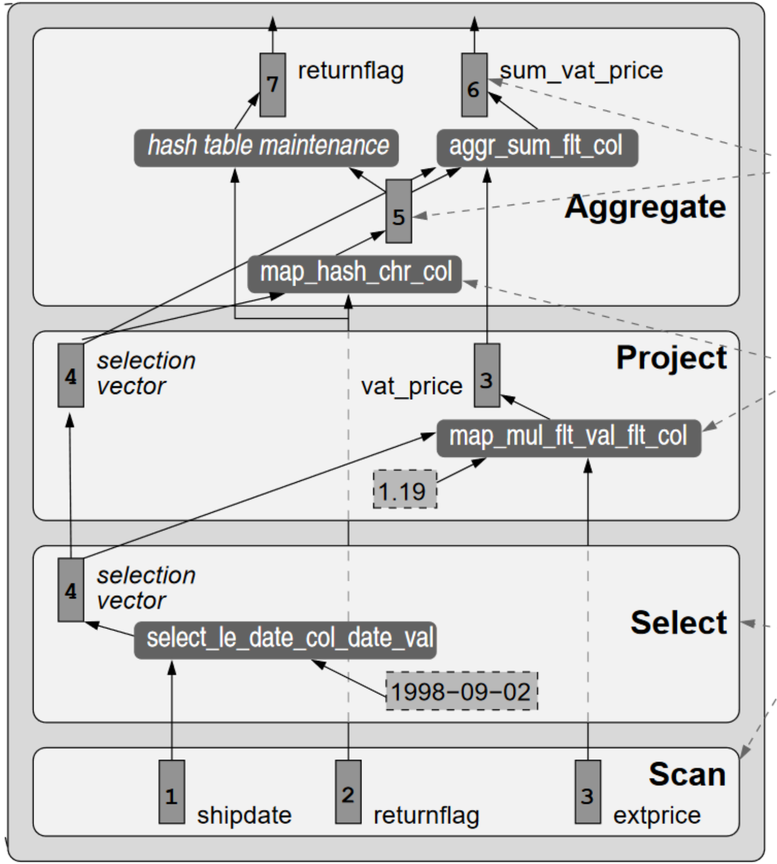


src: M. Zukowski, Balancing Vectorized Query Execution with Bandwidth Optimized Storage, PhD thesis, CWI Amsterdam, 2009

Observations:

- Vectorized execution quickly compensates for iteration overhead
- 1000 tuples should conveniently fit into caches

Vectorized execution in MonetDB/X100



vectors
contain multiple values of a single attribute

primitives
process entire vectors at a time

operators
process sets of tuples represented as aligned vectors

src: M. Zukowski, Balancing Vectorized Query Execution with Bandwidth Optimized Storage, PhD thesis, CWI Amsterdam, 2009

Vectorized execution in SQL Server 11

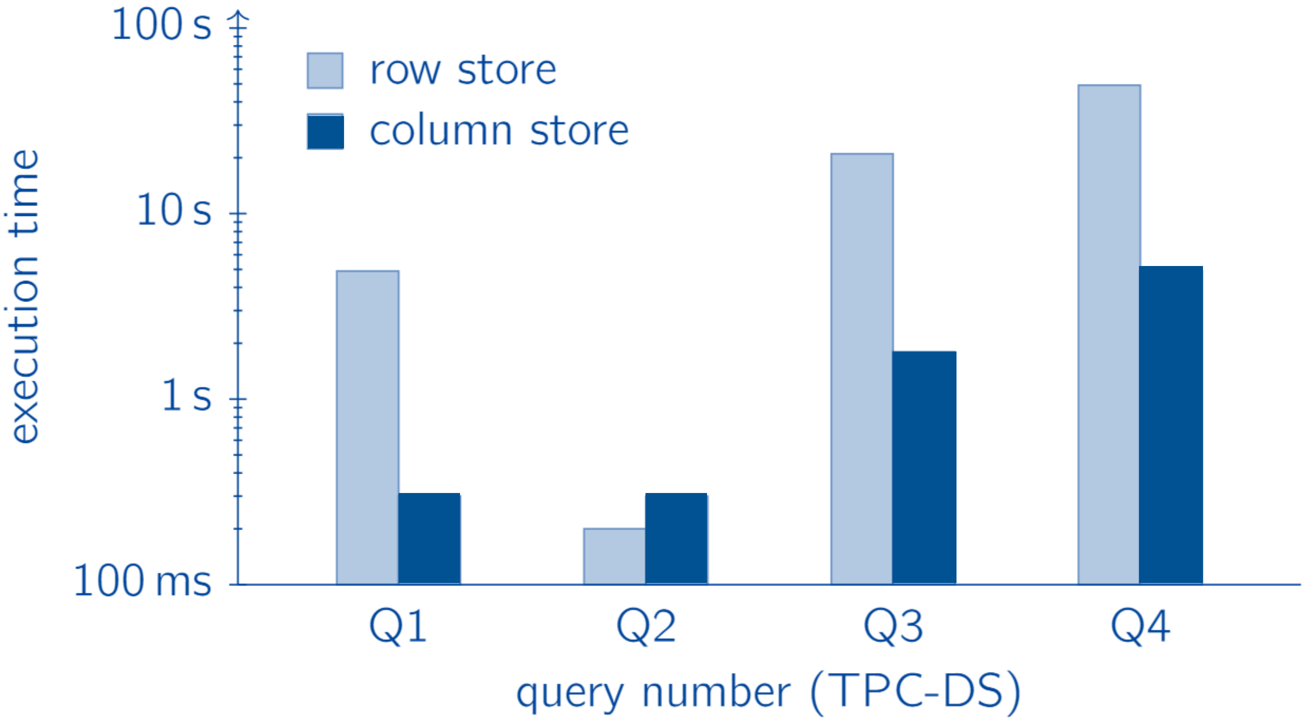
Microsoft SQL Server supports vectorized (“batched” in MS jargon) execution since version 11.

- Storage via new **column-wise index** (with compression and prefetching improvements)
- New operators with **batch-at-a-time processing**
- Typical pattern:
  - Scan, pre-filter, project, aggregate data early in the plan using **batch operators**
  - **row operators** may be needed to finish the operation
- Good for scan-intensive workloads (OLAP), **not** for point queries (OLTP workloads)
- Internally, the optimizer treats batch processing as new **physical property** (like being sorted) to combine operators in a proper way.

src: Per-Ake Larson *et al*. SQL Server Column Store Indexes. *SIGMOD 2011*

SQL Server: Performance

Performance impact (TPC-DS, scale factor 100, ~ 100GB)

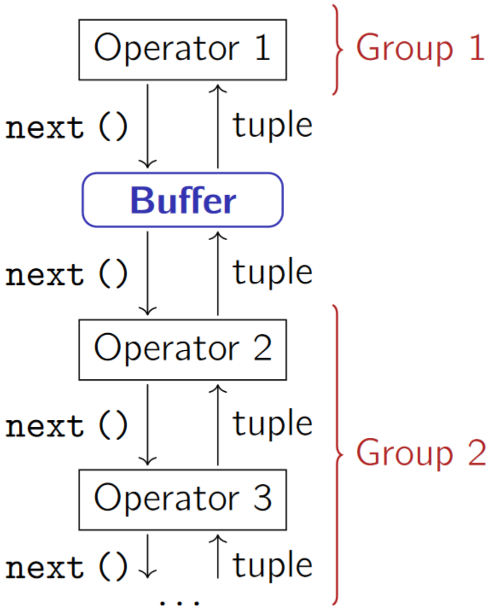


src: Per-Ake Larson *et al*. SQL Server Column Store Indexes. *SIGMOD 2011*

Vectorized execution in PostgreSQL

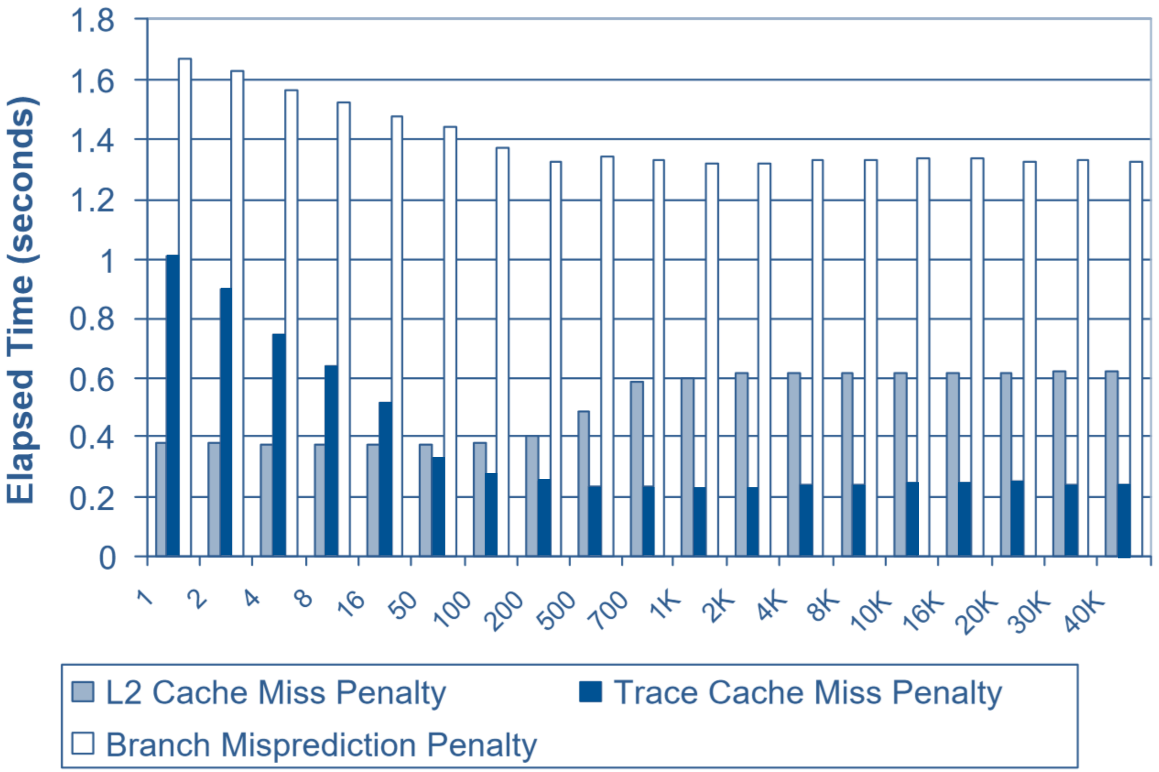
- Organize query plan into **execution groups**
- Add **buffer operator** between execution groups
- The buffer operator provides tuple-at-a-time interface to the outside, but **batches up** tuples internally.
- Similar to the example we covered previously

```
function: next()
// Read a batch of input tuples if buffer is empty
if empty and !end-of-tuples then
  while !full do
    append child.next() to buffer
    if end-of-tuples then
      break;
// Return tuples from buffer
return next tuple in buffer;
```



src: Zhou and Ross. Buffering Database Operations for Enhanced Instruction Cache Performance. SIGMOD 2004

Buffer operators in PostgreSQL



src: Zhou and Ross. Buffering Database Operations for Enhanced Instruction Cache Performance. SIGMOD 2004

Comparison of processing models

Overview of the discussed execution models

Execution model	iterator (tuple)	materialization (operator)	vectorization (vector)
query plans	simple	complex	simple
instruction cache utilization	poor	extremely good	very good
function calls	many	extremely few	very few
attribute access	complex	direct	direct
most time spent on	interpretation	processing	processing
CPU utilization	poor	good	very good
compiler optimizations	limited	applicable	applicable
materialization overhead	very cheap	expensive	cheap
scalability	good	limited	good

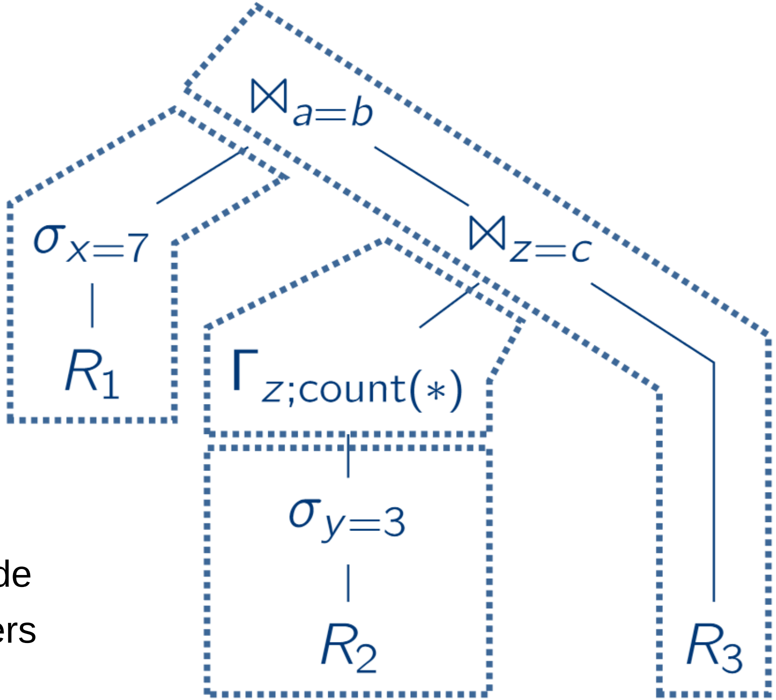
src: M. Zukowski, Balancing Vectorized Query Execution with Bandwidth Optimized Storage, PhD thesis, CWI Amsterdam, 2009

Pushing the envelope further with query compilation

- Database operators tend to be extremely simple:
  - Selection, arithmetic, etc.
  - Even operators like hash probes are quite simple

- Even a ***cache access can*** incur a noticeable ***cost***
  - Keep tuples ***in registers*** between operators?

- ***Pipeline breakers***
  - Tuples must be moved out of CPU registers on input side
  - Try to keep data in registers in-between pipeline breakers



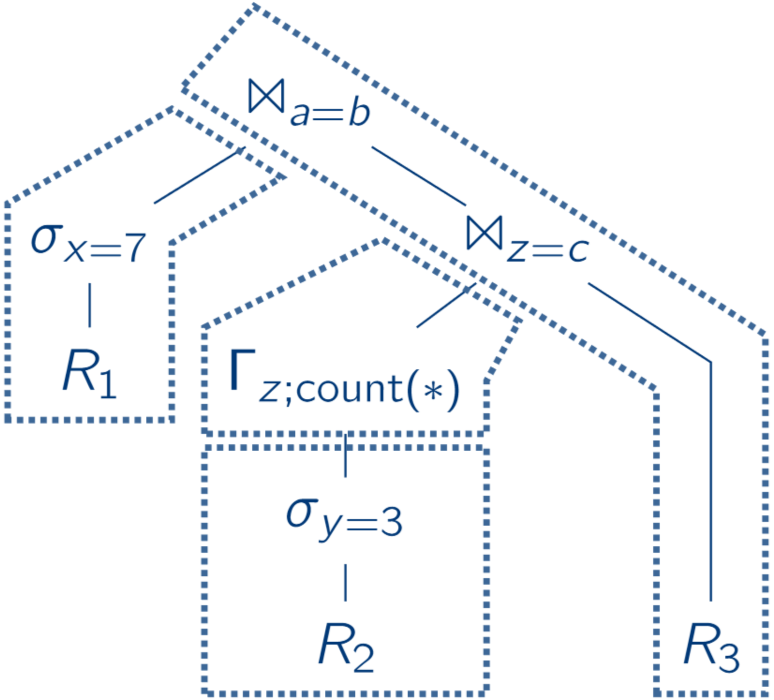
Pushing tuples up execution model

- ***HyPer***
  - ***Fuse all*** adjacent non-blocking ***operators*** of a query pipeline in a ***single, tight loop***
  - ***Compile*** query plans into ***machine code***

- ***Trick:***
  - ***Push-based*** (rather than pull-based) model
  - each code block ***consumes*** from one pipeline breaker and ***pushes*** into next
  - Carefully ***keep data in registers*** within one code block

Pushing tuples up execution model

Example:



```
// initialize memory of $\bowtie_{a=b}$, $\bowtie_{c=z}$, and $\Gamma_z$
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 $\Gamma_z$
for each tuple $t$ in $\Gamma_z$
  // materialize $t$ in hash table of $\bowtie_{c=z}$
for each tuple $t_3$ in $R_3$
  for each match $t_2$ in $\bowtie_{c=z}[t_3.c]$
    for each match $t_1$ in $\bowtie_{a=b}[t_3.b]$
      output $t_1 \circ t_2 \circ t_3$
```

Code blocks don’t quite match operator boundaries, *e.g.*, build/probe parts of hash join

Operator-centric → ***Data-centric*** execution

src: Thomas Neumann, Viktor Leis. Compiling Database Queries into Machine Code. Bulleting of the IEEE Computer Society Technical Committee on Data Engineering 2014

HyPer – compiled query execution

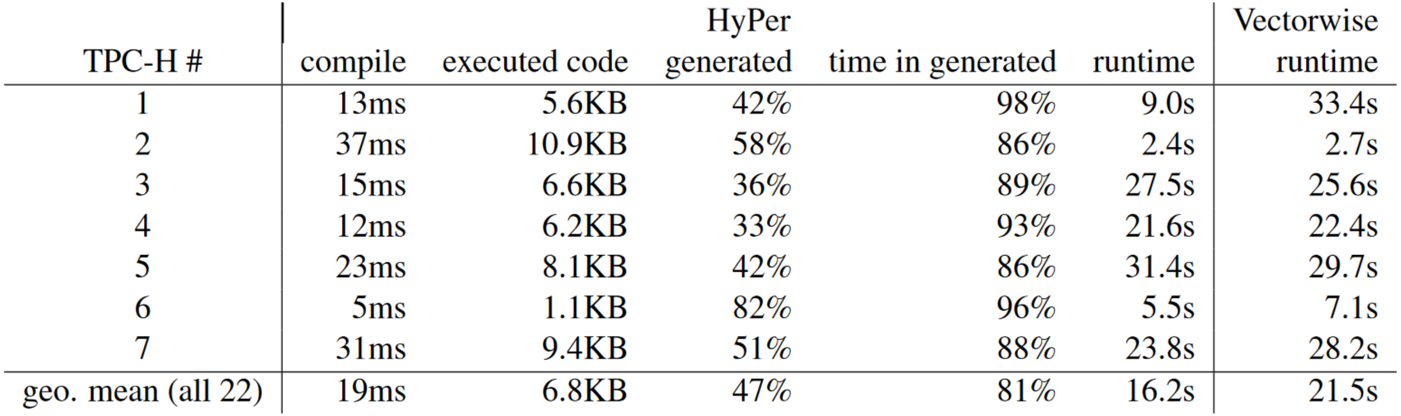
- Use ***LLVM*** (Low Level Virtual Machine) to generate code instead of C++
  - assembly-style code
  - ***platform-independent***
  - automatic ***register assignment***
  - strong ***type checking***

- Mix generated code with pre-compiled libraries
  - *e.g.*, memory management, error handling
  - Written in C++ anyway, no need to re-compile at runtime

Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. PVLDB, 4(9). 2011

HyPer performance

- Showing both compilation and execution time
- Also the size of the generated machine code, the fraction of the machine code that was generated at runtime using LLVM, and the fraction of the time spent in the generated code



			HyPer			Vectorwise
TPC-H #	compile	executed code	generated	time in generated	runtime	runtime
1	13ms	5.6KB	42%	98%	9.0s	33.4s
2	37ms	10.9KB	58%	86%	2.4s	2.7s
3	15ms	6.6KB	36%	89%	27.5s	25.6s
4	12ms	6.2KB	33%	93%	21.6s	22.4s
5	23ms	8.1KB	42%	86%	31.4s	29.7s
6	5ms	1.1KB	82%	96%	5.5s	7.1s
7	31ms	9.4KB	51%	88%	23.8s	28.2s
geo. mean (all 22)	19ms	6.8KB	47%	81%	16.2s	21.5s

- ***Insights:*** most of the critical code path is generated, the compilation time is quite low, and performance is often better than Vectorwise.

src: Thomas Neumann, Viktor Leis. Compiling Database Queries into Machine Code.
Bulleting of the IEEE Computer Society Technical Committee on Data Engineering 2014

Compiling query plans in other systems

- HyPer was a pioneering system implementing the data-centric code generation

- Other systems that compile queries (not necessarily with LLVM) are:
  - Microsoft's Hekaton
    - compiles stored procedures into native code using C as intermediary language
  - Cloudera Impala
    - LLVM JIT compilation for predicate evaluation and record parsing
  - MemSQL
    - High-level imperative DSL → second language of opcodes → LLVM IR → native code,
  - Apache Spark
    - WHERE clause expression trees → Scala AST → JVM bytecode,
  - PostgreSQL added support for JIT compilation in 2018
    - Automatically compiles Postgres' back-end C code into LLVM C++ code to remove iterator calls
  - etc.

src: Andy Pavlo – https://15721.courses.cs.cmu.edu/spring2020/slides/14-compilation.pdf

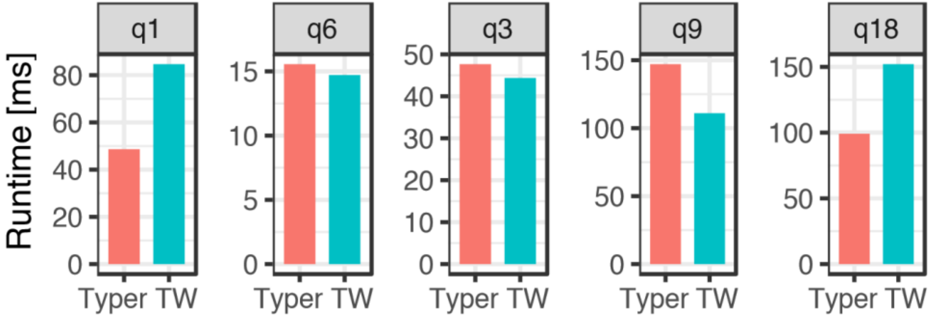
Compiled vs. vectorized query execution

- Single test system compares the fundamental properties of the execution model of a ***compilation-based pushing tuples up engine (Typer)*** and a ***vectorization based engine (Tectorwise).***

- Selected OLAP queries from TPC-H with SF 1
  - 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)

There is ***no clear winner***:

- Typer is faster for Q1 and Q18
- TW is better for Q6, Q3 and Q9

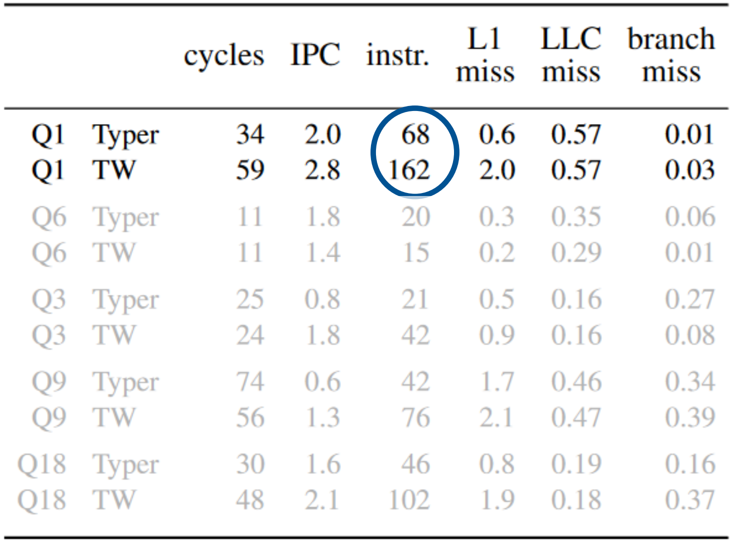


Understanding the results, requires more ***in-depth analysis*** on the ***CPU/cache characteristics***.

Timo Kersten et al. *Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask.* VLDB 2018

Compiled vs. vectorized query execution

- Micro-architectural analysis for the TPC-H queries, main observations:
  - TW executes significantly more instructions (up to 2.4x) and has more L1-d cache misses (up to 3.3x)



		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

Query 1: dominated by fixed point arithmetic operations and a cheap in-cache aggregation.

TW intermediate results must be materialized, which is as expensive as the computation itself.

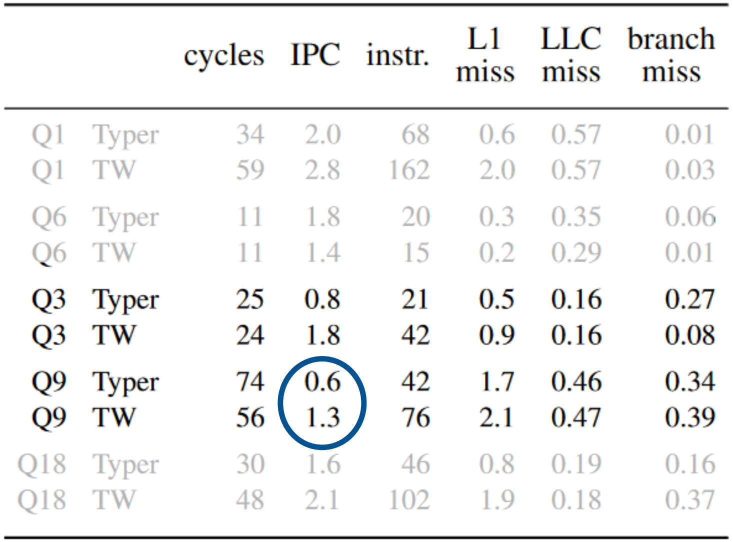
Typer can often keep intermediate results in CPU registers and perform the same operations with less instructions

- **Typer** is more efficient for **computational** queries that can hold intermediate results in CPU registers and have few cache misses.

Timo Kersten et al. *Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask.* VLDB 2018

Compiled vs. vectorized query execution

- Micro-architectural analysis for the TPC-H queries, main observations:
  – Typer’s complex loops induce more memory stalls and branch misses



		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

Q3 and Q9’s performance is determined by the efficiency of hash table probing.

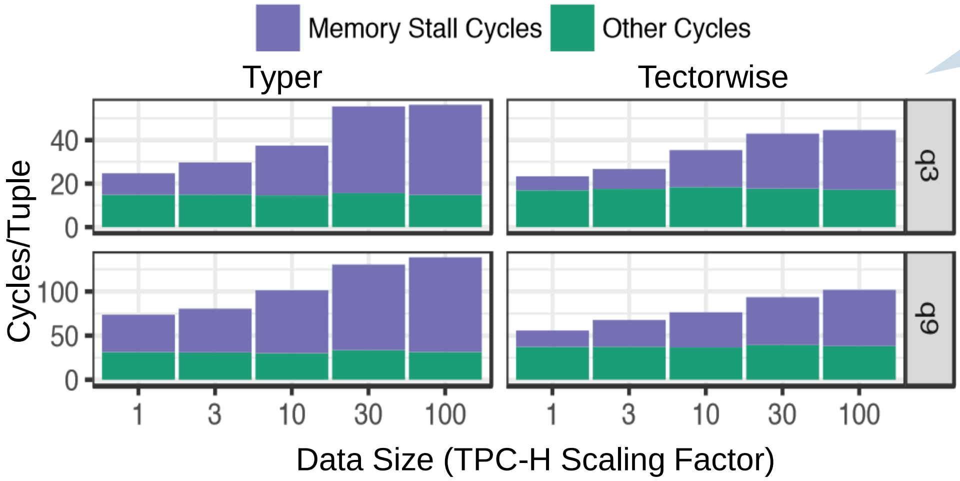
TW’s simple loops for hash probing allow for generating many outstanding memory loads.

- Observation: **TW** (vectorization) is better at **hiding** cache-miss **latencies**.

Timo Kersten et al. *Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask.* VLDB 2018

Compiled vs. vectorized query execution

- Micro-architectural analysis for the TPC-H queries, main observations:
  - Typer’s complex loops induce more memory stalls and branch misses



Typer’s performance suffers due to memory stalls more than TW’s.

Both systems observe more CPUs spent on memory stalls as data size increases.

- TW (vectorization) is better at hiding cache-miss latencies.

Timo Kersten et al. *Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask.* VLDB 2018

Compiled vs. vectorized query execution

Qualitative analysis and comparison

Execution model	Compilation-based engines	Vectorized engines
Computation	More efficient code	Less efficient (data not in registers)
Parallel data access	Less good at hiding memory latencies	Can generate more parallel memory requests
SIMD vectorization	Good (but, more complex to implement)	Very good (better match to leverage SIMD)
Parallelization	Good	Good
OLTP	Can generate fast stored procedures	Slower
Language support	Easy to write portable code	More complex
Compile time	Has overhead	Primitives are pre-compiled
Profiling	More complex for performance debugging	Naturally can attribute profiling to primitives
Adaptivity	Possible by switching from interpretation to compilation	Possible to swap execution primitives mid-flight
Debugging	Runtime and compile time co-exist	Simple (like every c++)

Timo Kersten et al. *Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask.* VLDB 2018

Relaxed Operator Fusion

- ***Idea: Relax*** the pipeline breakers to create ***mini-batches***
  - Good for operators that can be vectorized
  - Leverage software prefetching to hide memory stalls
  - Use data-level parallelism (e.g., using SIMD)

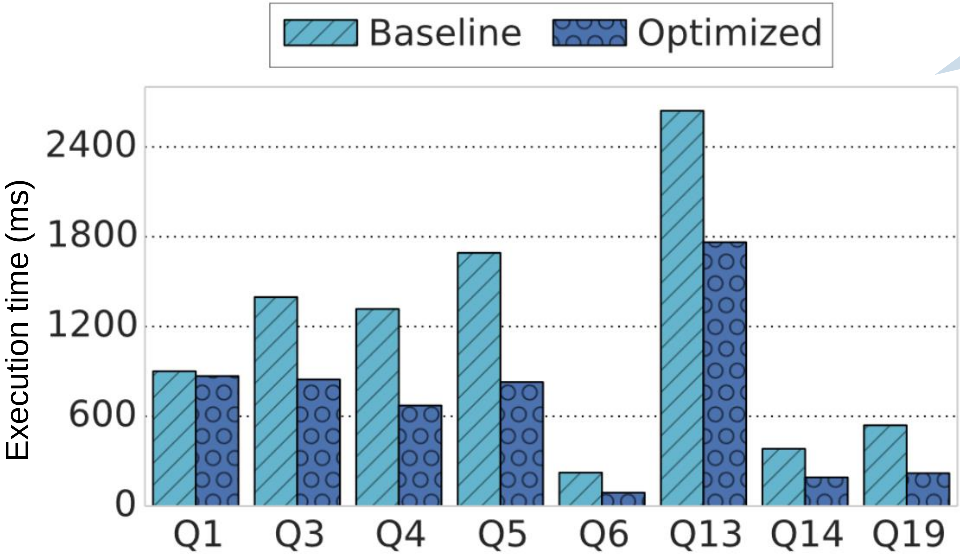
- Implemented in the Peloton DMBS (2017)

- If the query optimizer can correctly decide when to break a pipeline, this approach can be faster than both standard models (vectorized vs compiled).

Menon et al. *Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At last.* VLDB 2018

Results of Relaxed Operator Fusion

- Performance comparison of baseline (compiled query execution) vs. optimized (relaxed operator fusion)
  - For selected TPC-H queries, with scaling factor 10 (~ 10GB data)



Q1 only marginal improvement. Time is spent primarily on selection and aggregation.

The big performance improvements for Q3 is for the memory-intensive operators, the joins, mainly due to pre-fetching and SIMD.

Q13 gets 34% improvement solely due to efficient pre-fetching.

Menon et al. *Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At last.* VLDB 2018

Next lecture

- Memory-intensive operations (e.g., join and aggregation) deserve attention on their own.
- How can we optimize their performance for the memory sub-system?
- Next week we will look into how we can optimize the hash joins using techniques such as:
  - “blocking” or partitioning
  - software-based prefetching
  - software write-combining
  - non-temporal writes
  - etc.

References

- Various papers cross-referenced in the slides
  - Boncz *et al*. *MonetDB/X100: Hyper-Pipelining Query Execution* CIDR 2005
  - *Zukowski Balancing Vectorized Query Execution with Bandwidth Optimized Storage*, PhD thesis, CWI Amsterdam 2009
  - Zhou *et al. Buffering Database Operations for Enhanced Instruction Cache Performance*. SIGMOD 2004
  - Larson *et al*. *SQL Server Column Store Indexes.* SIGMOD 2011
  - Neumann. *Efficiently Compiling Efficient Query Plans for Modern Hardware.* VLDB 2011
  - Neumann, Leis. *Compiling Database Queries into Machine Code*. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 2014
  - Kersten *et al*. *Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask.* VLDB 2018
  - Menon et al. *Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At last.* VLDB 2018

- Lecture: *Database Systems on Modern CPU Architectures* by Prof. Thomas Neumann (TUM)
- Lecture: *Data Processing on Modern Hardware* by Prof. Jens Teubner (TU Dortmund, past ETH)
- Lecture: *Advanced Databases* by Prof. Andy Pavlo (CMU)

- Check out the code from Timo Kersten and play around with the TPC-H queries from Typer and Tectorwise (TW):
  - https://github.com/TimoKersten/db-engine-paradigms