Efficiently Computing Efficient Query Plans for Modern Hardware

Ashkan Alinejad

Ideas behind NewSQL

- provide the same scalable performance of NoSQL for OLTP read-write workloads.
- maintaining ACID guarantees for transactions.

Contribution

Proposing a novel compilation strategy that differs in a way that:

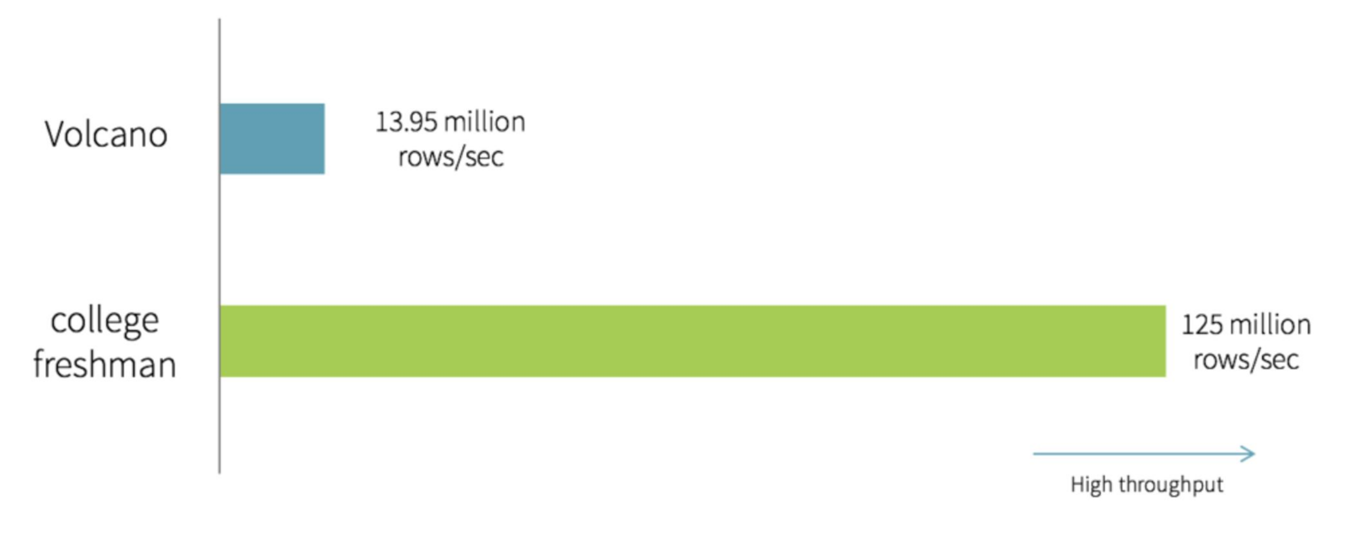
- Processing is data centric not operator centric.
- Data is pushed toward the operator.
- Queries are compiled into machine code.

Volcano-style processing

- A query is translated into an algebraic plan.
- Traditional way to execute them is the iterator model.
- Every algebraic operator produces a tuple stream.
- Allows for iterating over it by repeatedly calling *next()* function.

What was wrong with it?

What if we ask a college freshman to implement a query?



What was wrong with it?

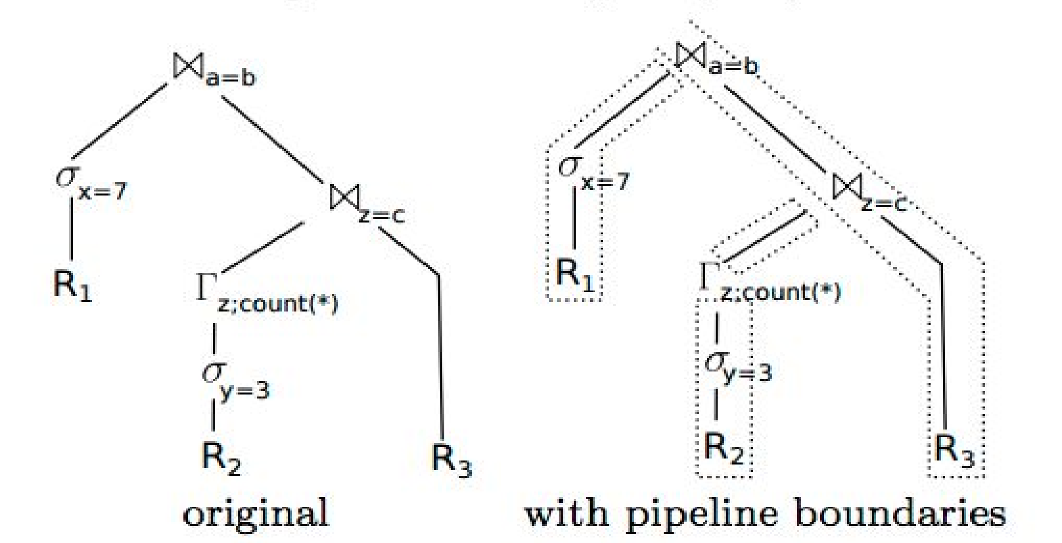
Comes from the time when query processing was dominated by I/O.

1. Next function will be called for every single tuple.
2. Results in poor code locality.
3. Loop unrolling and SIMD.

The Query Compiler

- Different architecture for maximizing data and code locality
- *Pipeline breaker:* an operator which takes an incoming tuple out of CPU registers.
- *Fully Pipeline breaker:* An operator that materializes all incoming tuples.
- Either classical iterator model or the block-oriented execution models are ill-suited for keeping data in CPU registers.

```
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
```



Example execution plan

The Query Compiler

What's the solution ?

- Reverse the direction of data flow
- Instead of pulling tuples up, push them towards the consumer operators.
- Operators in between leave the tuples in CPU registers.

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

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

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 $\ltimes_{z=c}$

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

Compiled query

Compiling Algebraic Expressions

- The operator boundaries in query code are blurred.
- For binary pipeline breakers materializing an input tuple from the left will be very different from materializing an input tuple from the right.

Compiling Algebraic Expressions

Conceptually each operator offers two functions:

- *Produce()* : Asks operator to produce its result tuples
- *Consume(attribute, source)* : Uses the attribute to perform the operator's task.

$\bowtie$.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)

Query processing

Code Generation

- Up to now, we have pseudo-code.
- In practice we need machine code.
- First solution: Generating C++ code.
  - Could directly access the data structures
  - Optimizing C++ compiler was really slow.
  - C++ won’t let us control over the generated code.

Code Generation

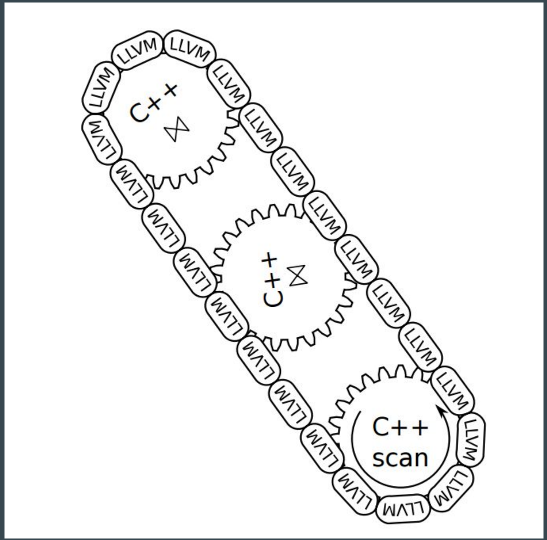
Instead of C++, They have used LLVM compiler.

- Hides the problem of register allocation.
- LLVM assembler is portable across machine architectures.
- LLVM can catch many bugs.
- And finally it’s faster than C++

Code Generation

But their code is not purely in LLVM assembler.

They have used C++ methods as it can be called directly from LLVM.



Advanced Parallelization techniques

Their initial implementation, without parallelization performs very well.

What if we could use parallelization techniques?

- Processing more than one tuple at a time!
    1. Allows for using SIMD instructions
    2. LLVM directly allow for modeling SIMD instructions.

Evaluation

Evaluation

- They have implemented on top of HyPer system which is designed as a hybrid OLTP and OLAP system.
- Run on MonetDB 1.36.5, Ingres VectorWise 1.0, DB X.
- Dual Intel X5570 Quad-Core-CPU
- 64GB main memory
- Red Hat Enterprise Linux 5.4
- gcc 4.5.2
- LLVM 2.8

System Comparison

	HyPer + C++	HyPer + LLVM
TPC-C [tps]	161,794	169,491
total compile time [s]	16.53	0.81

OLTP performance of different engines

System Comparison

	Q1	Q2	Q3	Q4	Q5
HyPer + C++ [ms]	142	374	141	203	1416
compile time [ms]	1556	2367	1976	2214	2592
HyPer + LLVM	35	125	80	117	1105
compile time [ms]	16	41	30	16	34
VectorWise [ms]	98	-	257	436	1107
MonetDB [ms]	72	218	112	8168	12028
DB X [ms]	4221	6555	16410	3830	15212

Code Quality

	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

Question ?