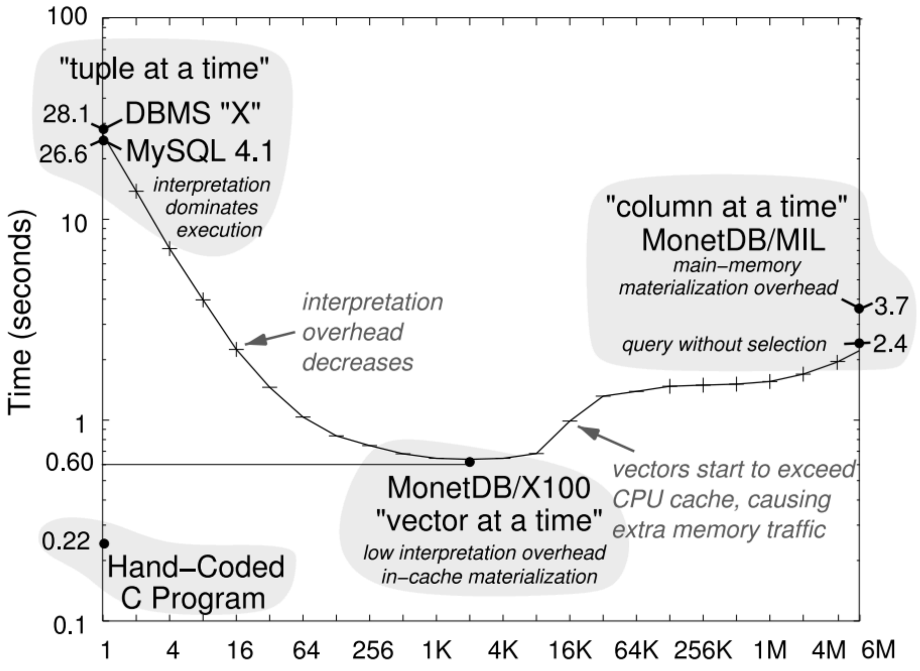
Efficiently Compiling Efficient Query Plans for Modern Hardware

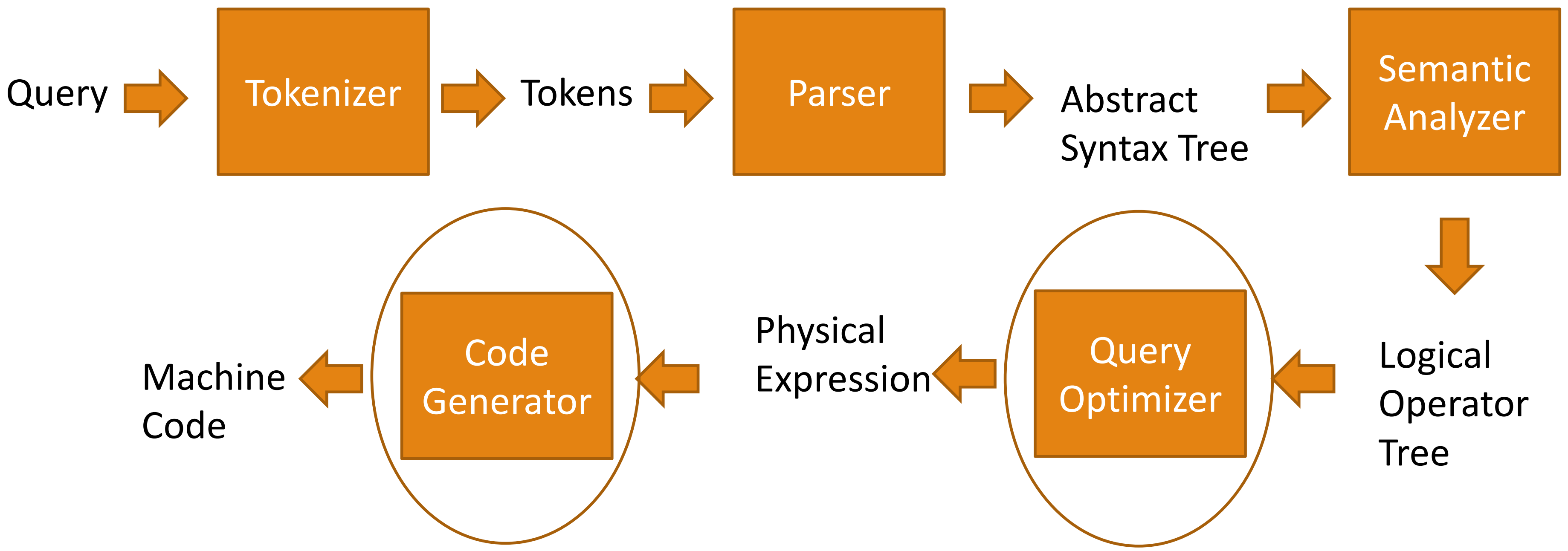
PRESENTED BY RUOCHEN

Motivation

- ❑ Disk I/O is no longer the bottleneck
- ❑ Query performance is determined by CPU cost



Compiler of Query



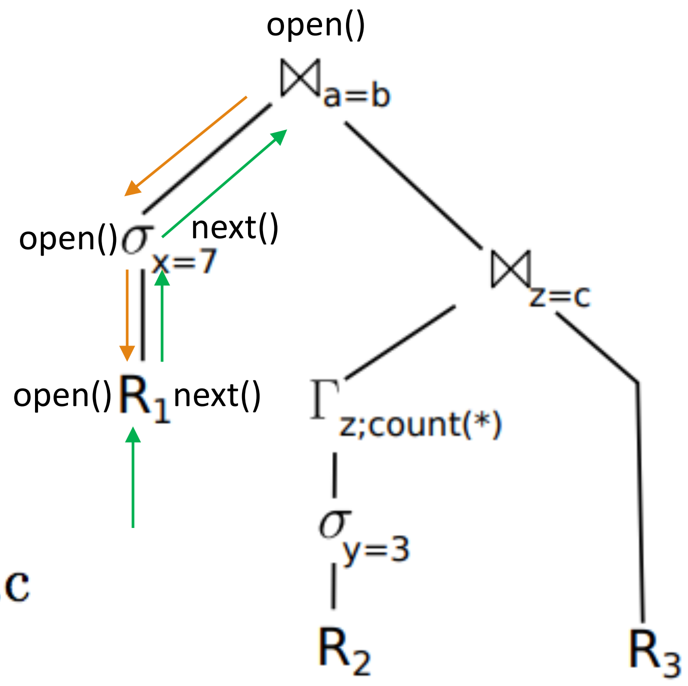
Query Optimizer

- How does Iterator Model do
  - Three virtual functions
    - open: initialize
    - next: produce a record as output
    - close: clean up
  - Data is pulled by operators recursively (functions)

Query Optimizer

- Example

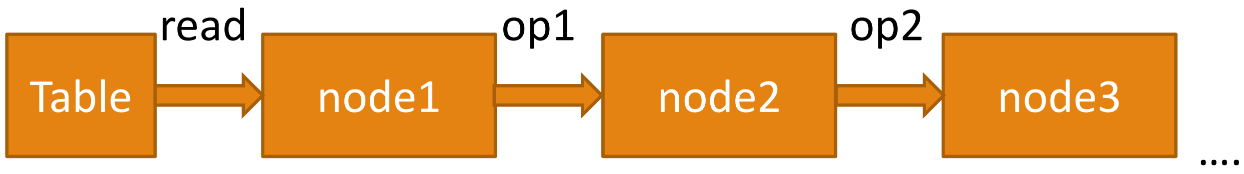
```
select    *
from      R1,R3,
          (select    R2.z,count(*)
          from       R2
          where      R2.y=3
          group by R2.z) R2
where     R1.x=7 and R1.a=R3.b and R2.z=R3.c
```



Query Optimizer

❑ Weakness of Iterator Model

Operator centric → Moving data between registers frequently

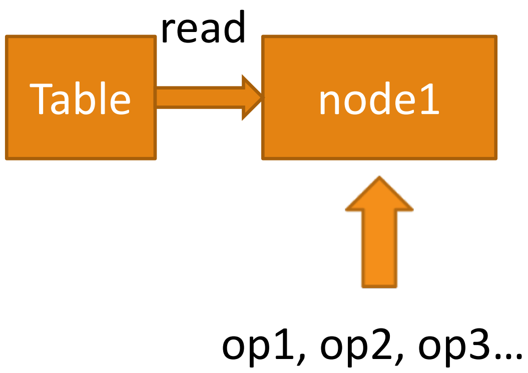


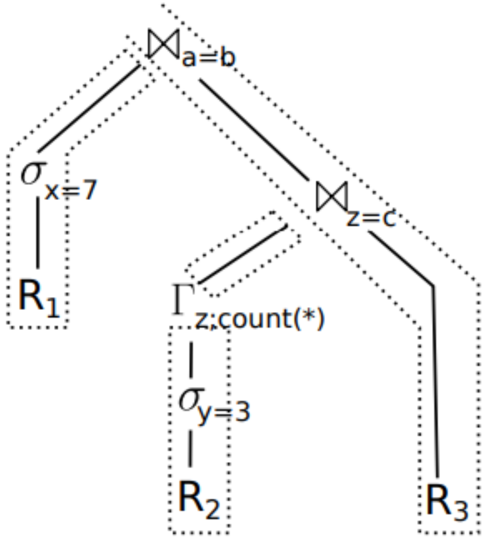
Using virtual functions → Not good for optimal code generation

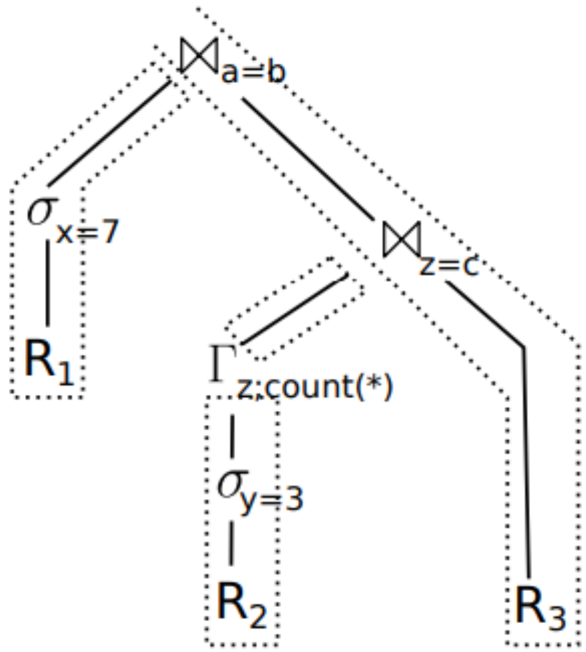
For example, node1 doesn't know the structure of node2, can't offer best structure of data

Query Optimizer

- Solution
  - Pipeline breaker: for a given input side if it takes an incoming tuple out of the CPU registers
  - Use general functions while exposing operator structures
    - produce()
    - consume(attribute, source)







⋈.produce	⋈.left.produce; ⋈.right.produce;

⋈.consume(a,s)	if (s==⋈.left)
	print “materialize tuple in hash table”;
	else
	print “for each match in hashtable[”
	+a.joinattr+“]”;
	⋈.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)

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 $t_1 \circ t_2 \circ t_3$

Query Optimizer

Query Optimizer

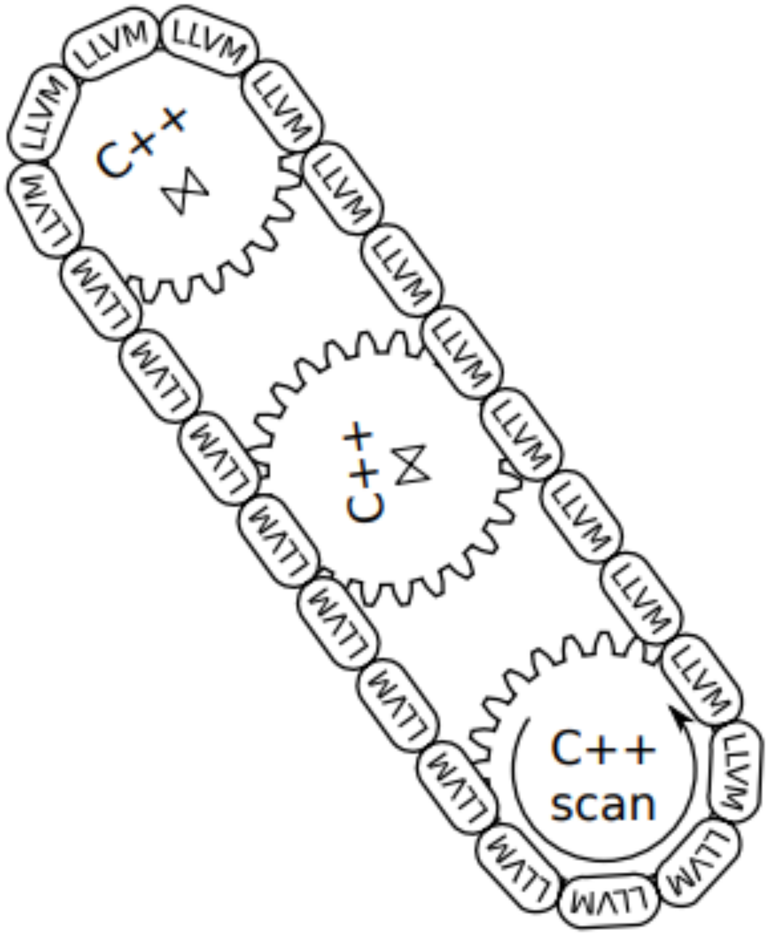
Data is pushed towards operator → Better code and data locality

Data centric not operator centric → Keep data in CPU registers as long as possible

Code Generator

	C++	LLVM
Advantages	Can directly access the data structure	Produces extreme fast machine code Easy for register allocation
Disadvantages	Compiling is slow Does not offer control over generated code	Tedious Much of database logic like index structure is written in C++

Mix LLVM and C++



Code Generator

- C++ as "cogwheels"
  - complex part (e.g. locating data structure)
  - "drive" the pipeline
  - pre-compiled (e.g. as shared library)
- LLVM as "chain"
  - combining "cogwheels"
  - dynamically generated

```
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 $\Gamma_{z;count(*)}(\sigma_{y=3}(R_2))$

Code Generator

Code Generator

- Lazy evaluation
  - try to load attributes as late as possible
- Branch prediction
  - guess which way a branch (e.g. if-else) will go
  - improve the flow in instruction pipeline

```
Entry* iter=hashTable[hash];
while (iter) {
  ... // inspect the entry
  iter=iter->next;
}
```

If hash entry exist
If reach the end

```
Entry* iter=hashTable[hash];
if (iter) do {
  ... // inspect the entry
  iter=iter->next;
} while (iter);
```

Improve hash table lookup by 20%

Contributions

- Query Optimizer
  - Data is pushed towards operator → Better code and data locality
  - Data centric not operator centric → Keep data in CPU registers as long as possible
- Code Generator
  - Using optimizing LLVM framework → Fast machine code

Evaluation

- Implemented on top of Hyper system (main memory database)
- Compare with MonetDB, Ingres VectorWise, DBX
- Dual Intel X5570 Quad-Core-CPU, 64G main memory, Red Hat 5.4
- gcc 4.5.2, LLVM 2.8
- TPC-CH benchmark

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

Table 1: OLTP Performance of Different Engines

	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

Table 2: OLAP Performance of Different Engines

System Comparison

	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

Code Quality

Questions?