Efficiently Compiling Efficient Query Plans for Modern Hardware

Thomas Neumann

Technische Universität München

August 30, 2011

Motivation

Most DBMS offer a *declarative* query interface

- the user specifies the only desired result
- the exact evaluation mechanism is up the the DBMS
- for relational DBMS: SQL

For execution, the DBMS needs a more imperative representation

- usually some variant of relational algebra
- describes the the real execution steps
- set oriented, but otherwise quite imperative

How to evaluate such an execution plan? How to generate code?

Motivation (2)

The classical evaluation strategy is the **iterator model**
(sometimes called Volcano Model, but actually much older [Lorie 74])

- each algebraic operator produces a *tuple stream*
- a consumer can *iterate* over its input streams
- interface: open/next/close
- each *next* call produces a new tuple
- all operators offer the same interface, implementation is opaque

Motivation (3)

Very popular strategy, but not optimal for modern DBMS

- **millions** of virtual function calls
- control flow constantly changes between operators
- branch prediction and cache locality suffer
- was fine when I/O dominated everything
- but today large parts of data are in main memory
- CPU costs become an issue

Motivation (4)

Some DBMS therefore switched to **blockwise** processing

- like the iterator model, but return a few hundred tuples at a time

Advantages:

- amortizes the costs of *next* calls
- good locality
- one loop will process many tuples
- reduces branching, allows for vectorization

Disadvantages:

- pipelining not (easily) possible
- additional memory reads/writes

Example

```
Tuple[] Select::next()
  tuples=input.next()
  if (!tuples)
    return tuples
  writer=0
  for (i=0;i!=tuples.length;++i)
    tuples[writer]=tuples[i]
    writer+=(checkPred[tuples[i]])
  tuples.length=writer
  return tuples
```

Data-Centric Query Execution

Why does the iterator model (and its variants) use the operator structure for execution?

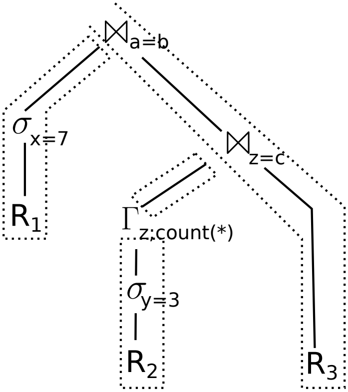
- it is convenient, and feels natural
- the operator structure is there anyway
- but otherwise the operators only describe the data flow
- in particular operator boundaries are somewhat arbitrary

What we really want is **data centric** query execution

- data should be read/written as rarely as possible
- data should be kept in CPU registers as much as possible
- the code should center around the data,
  not the data move according to the code
- increase locality, reduce branching

Data-Centric Query Execution (2)

Example plan with visible pipeline boundaries:



- data is always taken out of a pipeline breaker and materialized into the next
- operators in between are passed through
- the relevant chunks are the pipeline fragments
- instead of iterating, we can push up the pipeline

Data-Centric Query Execution (3)

Corresponding code fragments:

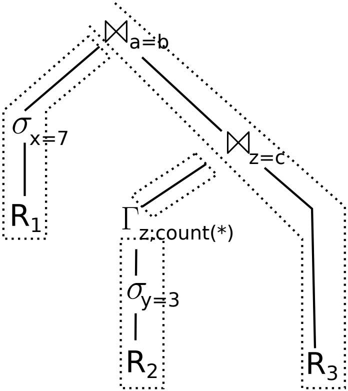
initialize memory of $\bowtie_{a=b}$, $\bowtie_{c=z}$, and Γ_z

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 $t_1 \circ t_2 \circ t_3$



Data-Centric Query Execution (4)

Basic strategy:

1. the producing operator loops over all materialized tuples
2. the current tuple is loaded into CPU registers
3. all pipelining ancestor operators are applied
4. the tuple is materialized into the next pipeline breaker

- tries to maximize code and data locality
- a tight loops performs a number of operations
- memory access in minimized
- operator boundaries are blurred
- code centers on the data, not the operators

Code Generation

The algebraic expression is translated into query fragments.

Each operator has two interfaces:

1. produce
   - asks the operator to produce tuples and push it into
2. consume
   - which accepts the tuple and pushes it further up

Note: only a mental model!

- the functions are not really called
- they only exist conceptually during code generation

Code Generation (2)

A simple translation scheme:

$\bowtie$.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)

Code Generation (3)

How can we evaluate the data-centric query fragments?

- interpretation is simple but unattractive
  - adds a lot of branching
  - no access to CPU registers, many memory accesses
  - can be more expensive than the iterator model itself!
- compilation into machine code is very attractive
  - real inlining, no additional branches
  - evaluation can be "near optimal" (i.e., everything in CPU registers)
  - execution is extremely fast

But how? System R suffered from lack of portability.

Code Generation (4)

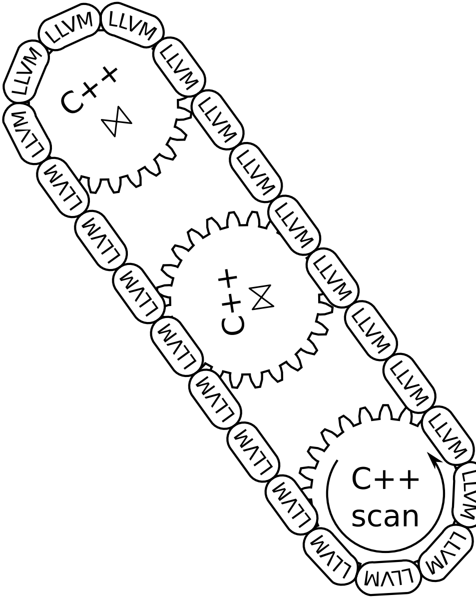
We tried two alternatives:

1. generate C++ code from the query
   - translate query into C++ code, compile, load as so
   - easy to understand
   - can directly interact with DBMS code
   - good performance, but compilation is really slow!
   - and code generation is surprisingly error prone
2. generate LLVM assembler code
   - portable, high-level assembler
   - optimizing compiler
   - much faster compilation time, good code quality
   - unbounded number of registers, strongly typed, many checks
   - initially daunting, but now much more pleasant then the C++ version

Code Generation (5)

Not everything needs to be LLVM code

- many complex code pieces remain unchanged
- e.g., spooling to disk
- much more reasonable to implement it in C++
- only the hot path is performance critical
- executed for millions of tuples, but relative simple
- implemented in LLVM code
- keeps the amount of runtime code down



Evaluation

We implemented this in our HyPer system

- initially we generated C++ code from code fragments
- then, switched to the data-centric LLVM code

Allows for comparisons C++ vs. LLVM

Compared it with other systems

- VectorWise, MonetDB, DB X
- TPC-C for OLTP (only HyPer)
- TPC-H queries adapted to TPC-C for OLAP

Evaluation (2)

OLTP results

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

- here queries are very simple, index structures etc. dominate
- therefore performance is similar
- but compile time differs greatly!
- unacceptable for interactive queries

Evaluation (3)

OLAP results

	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

- excellent performance
- compile time is low
- good cache locality, few branch misses (not shown here)

Conclusion

Data-centric query processing shows excellent performance

- minimizes number of memory accesses
- data can be kept in CPU registers
- increases locality, reduces branching

LLVM is an excellent tool for code generation

- fast, on-demand code generation for arbitrary queries
- good code quality
- portable and well maintained