Efficiently Compiling Efficient Query Plans for Modern Hardware

By Kyle Imrie

Background on the Paper and the Author

Who is the author?

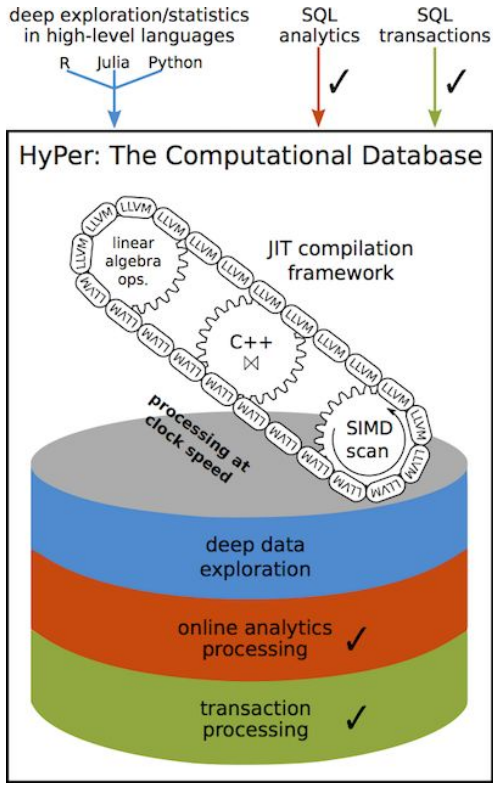
- Paper was published by Thomas Neumann in June of 2011
- Professor of Query Optimization and Database Systems at the Technical University of Munich

TUM

Technische Universität München

What else did he accomplish?

- In April of 2011, he presented at the 27th International IEEE Conference on Data Engineering in Germany
- At this presentation, he introduced HyPer, a hybrid OLTP/OLAP database management system
- The ideas covered in this presentation are built into HyPer


deep exploration/statistics in high-level languages
R
Julia
Python
SQL analytics
✓
SQL transactions
✓
HyPer: The Computational Database
LLVM
linear algebra ops.
JIT compilation framework
C++
⋈
SIMD scan
processing at clock speed
deep data exploration
online analytics processing
✓
transaction processing
✓

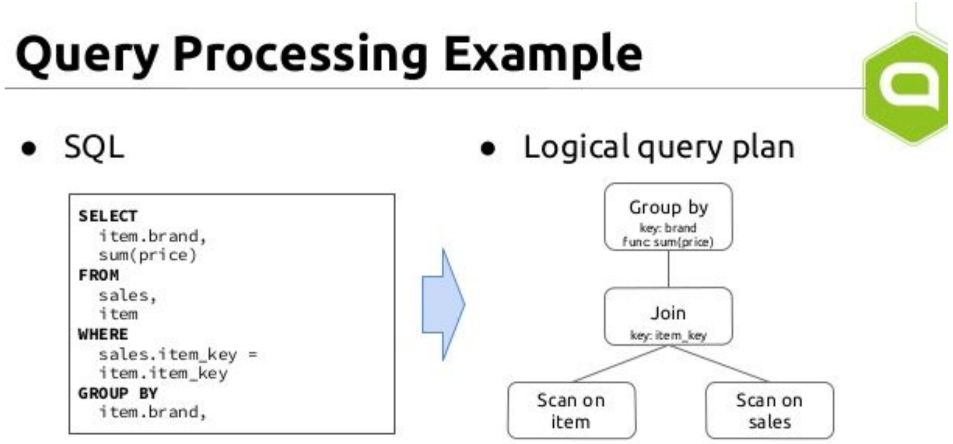
What are the main takeaways of this paper?

- What are the weaknesses of the Volcano style of query optimization?
- How does the HyPer DBMS tackle these issues?
- How is it all implemented and how well does it perform?

Weaknesses of the Volcano model of query optimization

What is the Volcano model?

When you are telling SQL to run a query against your database, what is happening behind the scenes? Your declarative statements are translated into physical algebra and executed repeatedly over a stream of data tuples.



Query Processing Example

- SQL
- Logical query plan

```
SELECT
  item.brand,
  sum(price)
FROM
  sales,
  item
WHERE
  sales.item_key =
  item.item_key
GROUP BY
  item.brand,
```

Components of a Query Plan

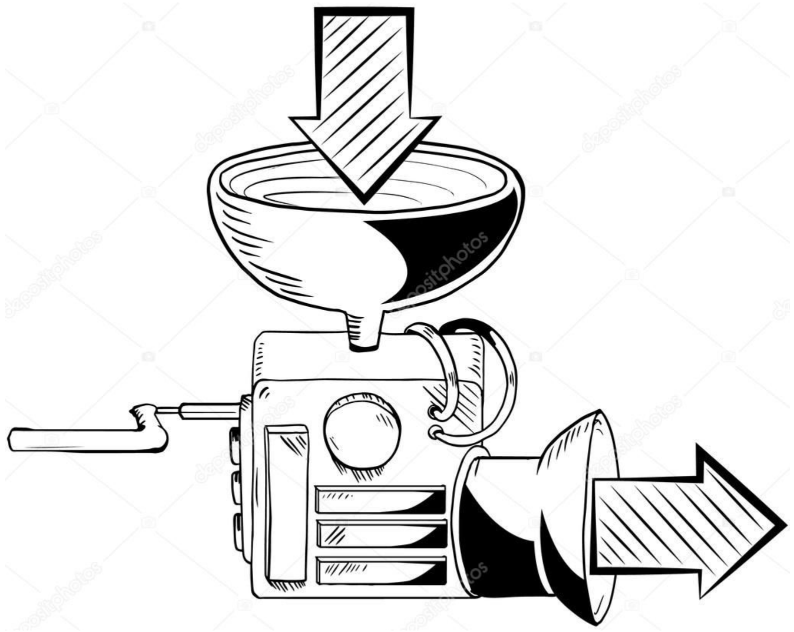
Picture each step of your logical plan as a function/method in some programming language, with a single data tuple as its input. Everytime you call this method, it does some work and outputs its results and seeks to the next tuple in the data stream.

In the iterative model, where you can have many intermediary steps as well as one final step, these functions are called ***millions*** of times. One of the aims of the author was to reduce the CPU consumption spent on these calls.

Other Issues

What if we operate on multiple tuples of data at once?

Vectorization is a step in the right direction, but it incurs the cost of having to load your data into memory (removes ability to pipeline together different operators into one)



How does the paper solve these issues?

Three Main Points: The Secret Sauce of HyPer

- Processing is data centric, not operator centric
- Emphasize data and code locality
- Queries are compiled into native machine code

Moving from Operator Model to Data Model

In the iterative model, the operators would pull up the data tuples one at a time, constantly cycling through the extremely-high throughput CPU cache memory.

The system described in the paper seeks to keep as much data in the CPU registers for as long as possible.

In this way, the direction of data flow is ***reversed*** compared to the iterator model (the data is ***pushed to*** the operators rather than ***pulled from***.

What does Data/Code Locality Mean?

Strong Data Locality => Data is stored close to where it will be processed, i.e in CPU registers

Weak Data Locality => Data is most often found in main memory or on disk

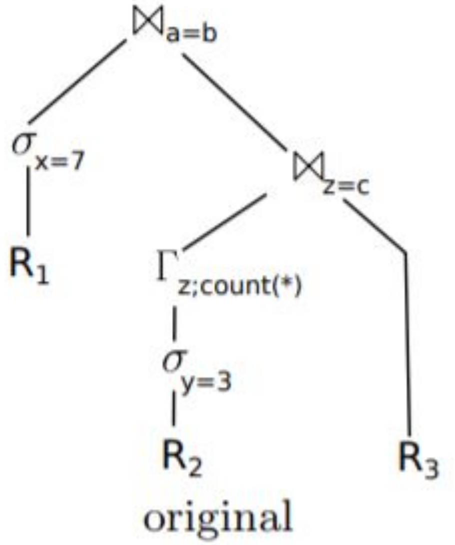
Strong Code Locality => Data is frequently being pushed in volume to the operators, allowing the operators to work longer without fetching more data

Weak Code Locality => Each time an operator iterates over a tuple, time is spent seeking to the next data tuple from memory

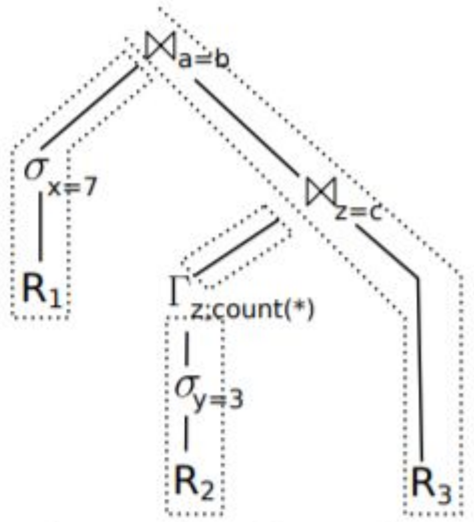
Example Query: Iterator Model

```
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



Example Query: Data-Centric Model



with pipeline boundaries

```
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 $\bowtie_{z=c}$
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 $t_1 \circ t_2 \circ t_3$
```

How do we build these plans?

Going from Algebra to Machine Code

When it comes to compiling your SQL query, the model introduced in this paper is quite similar to the iterator model:

- SQL query is parsed
- Query is translated into physical algebra
- Algebraic expression is optimized
- Instead of this expression being passed as an operator, we compile it into an imperative program (see prev example)

How well does it perform?

Inner Workings of the HyPer Compiler

Initial experiments in full C++ compilation at runtime led to full seconds of compiling time.

Instead, they compiled machine code using the Low Level Virtual Machine compiler framework, using C++ for data structure management. The C++ is pre-compiled; only the LLVM code is compiled at runtime.

Total compilation time in milliseconds.

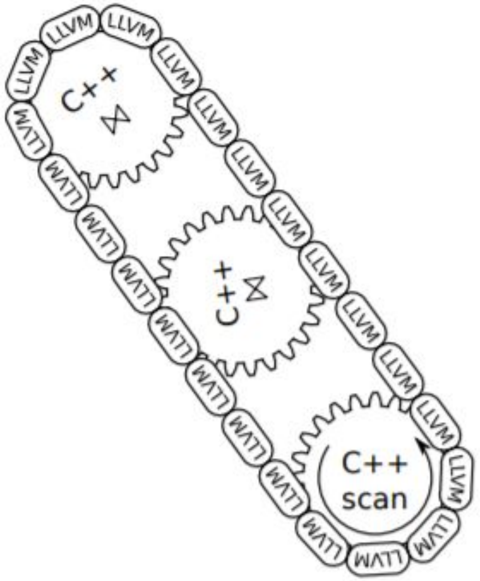
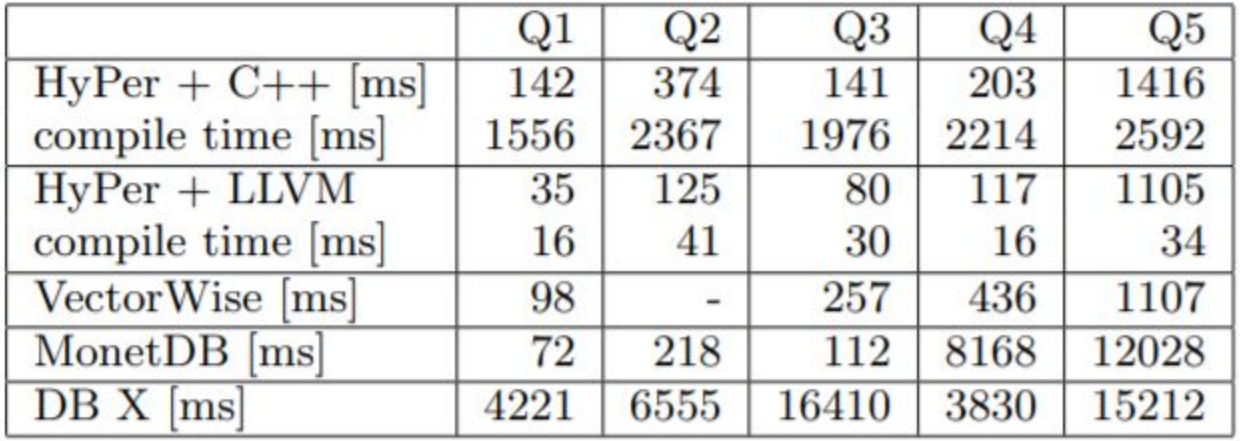


Figure 6: Interaction of LLVM and C++

The TPC-H Benchmarks



	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

Conclusions

- While the iterator model offers a simple interface, it comes with performance costs
- **Putting data first**: By reversing the data flow in our query plans, we can achieve great results
- The C++/LLVM framework can produce machine code that rivals hand-crafted solutions

Questions?