HARDWARE-CONSCIOUS DATA PROCESSING SYSTEMS

Holger Pirk

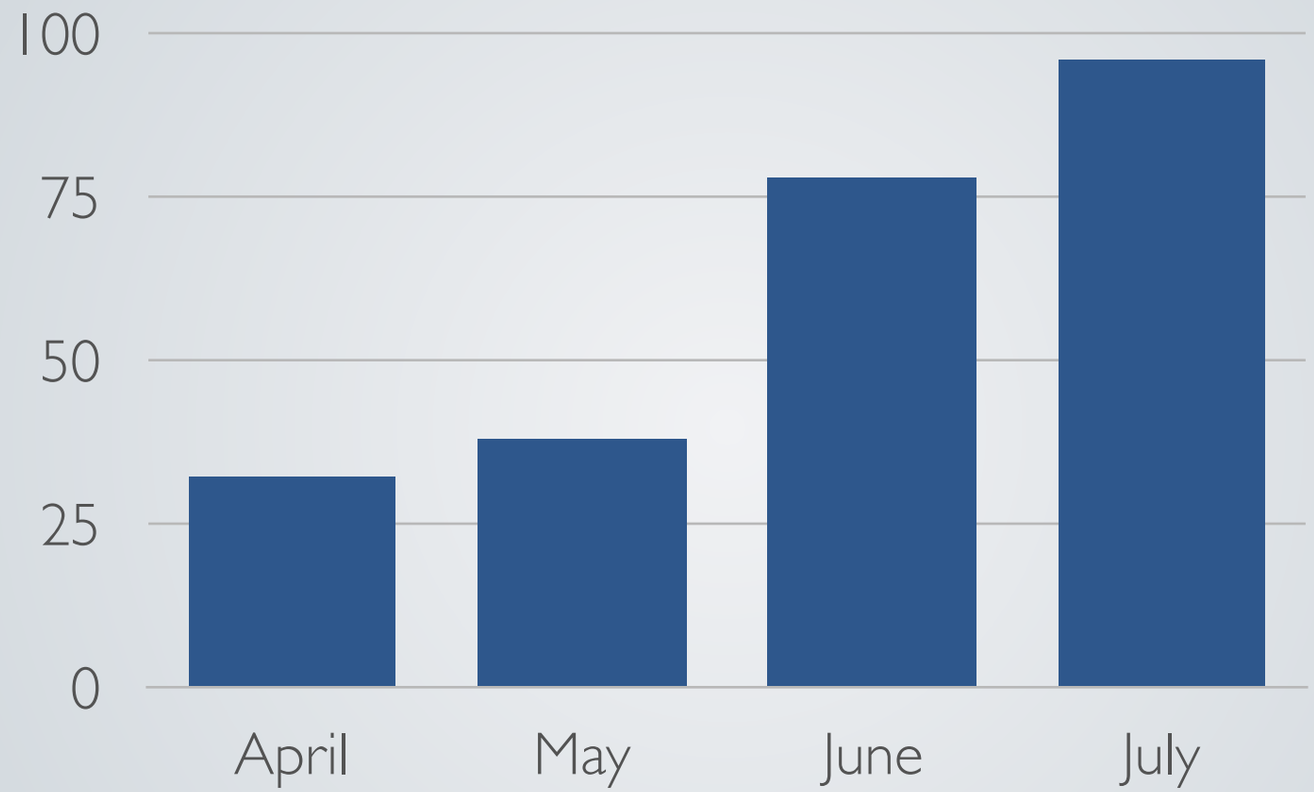
http://doc.ic.ac.uk/~hlgr

Imperial College London

Microsoft

Data Processing Performance
- A Case Study

Data Processing Performance - A Case Study



≈ select sum(sales), … where country = US … group by month, …

Data Processing Performance - A Case Study

TPC-H Query 1

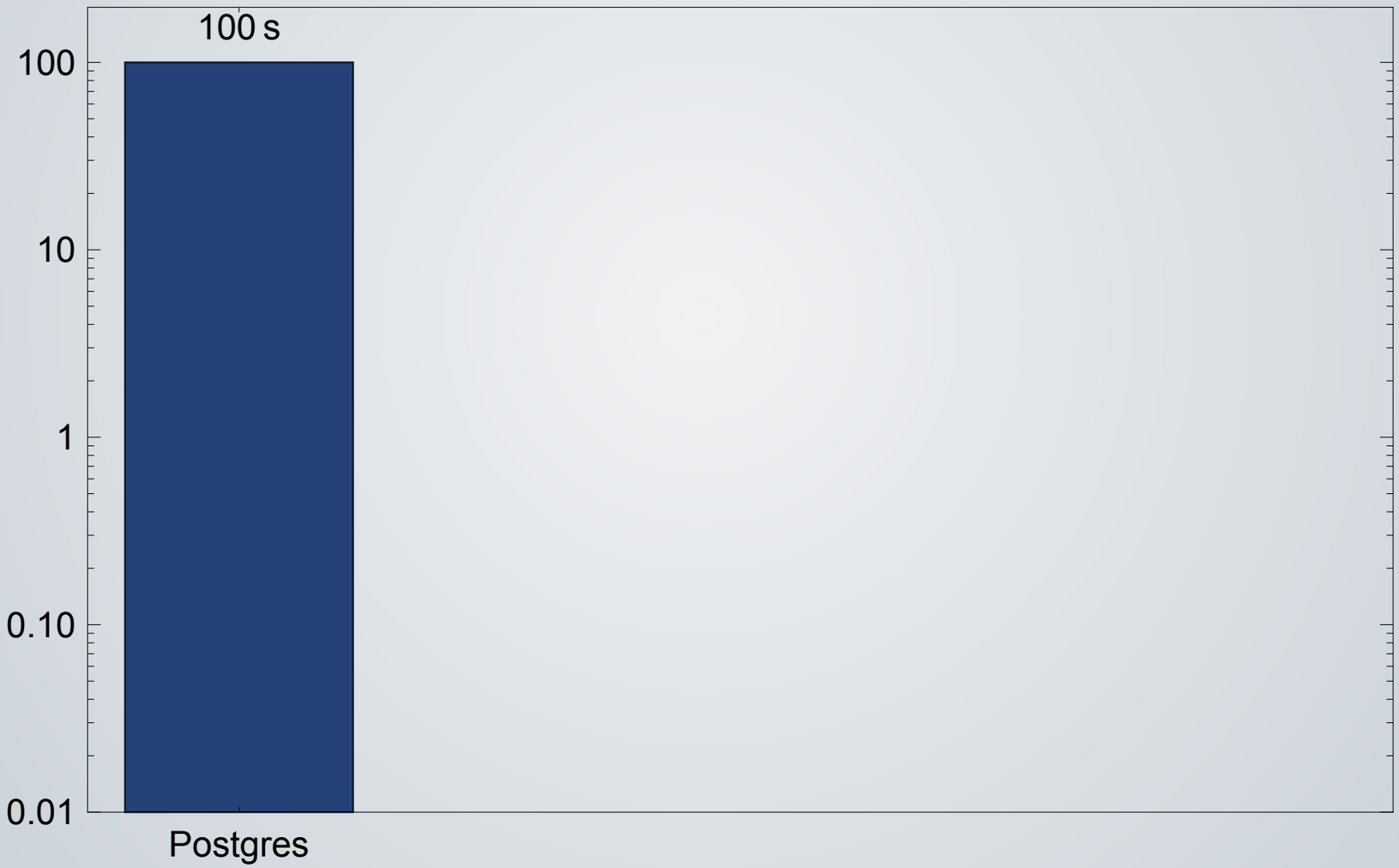
≈ select sum(sales), … where country = US … group by month, …

(Roughly 10 GB of Data)

Data Processing Performance - A Case Study

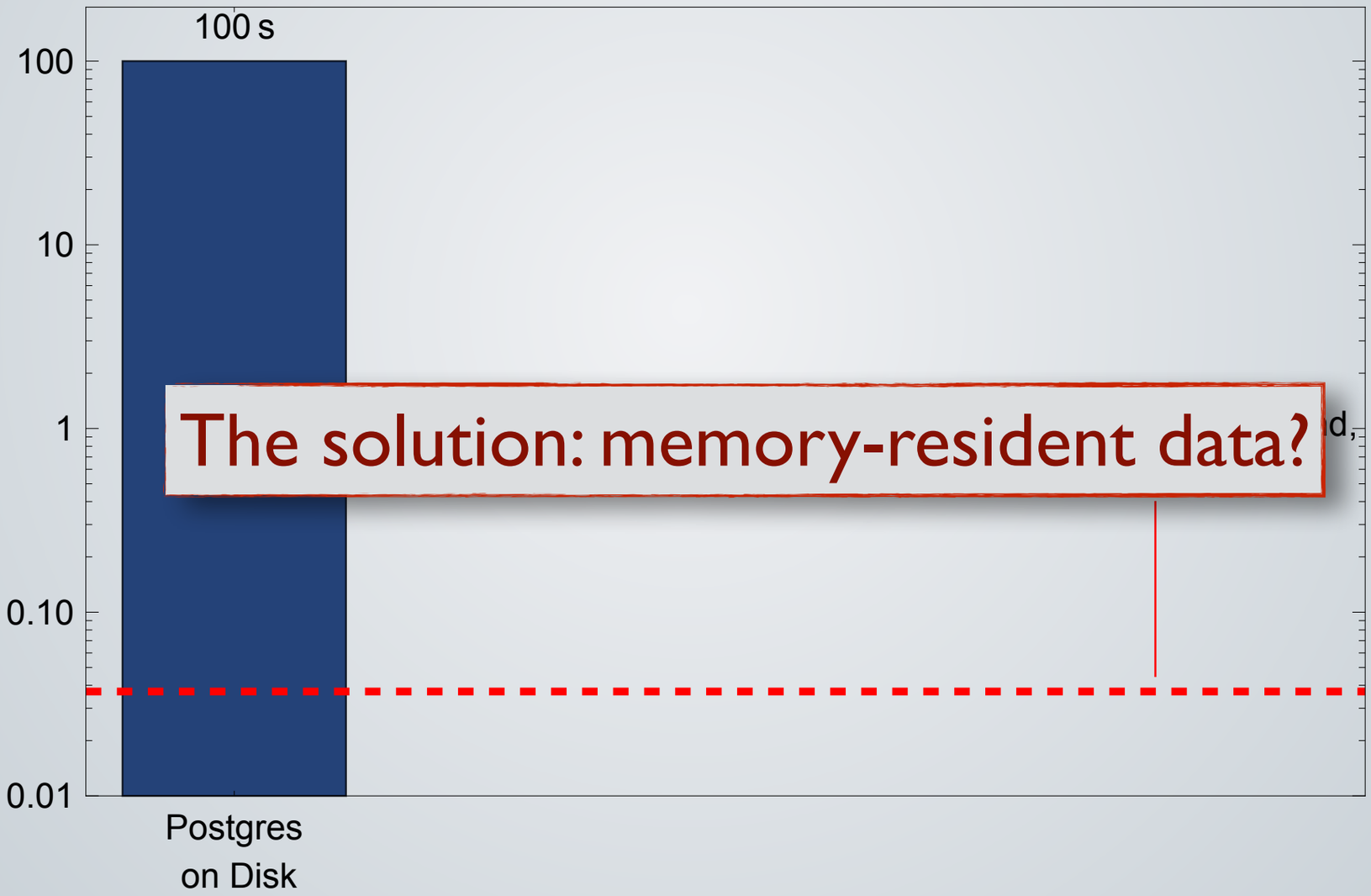
select sum(a) where b=6 group by c

Data Processing Performance - A Case Study



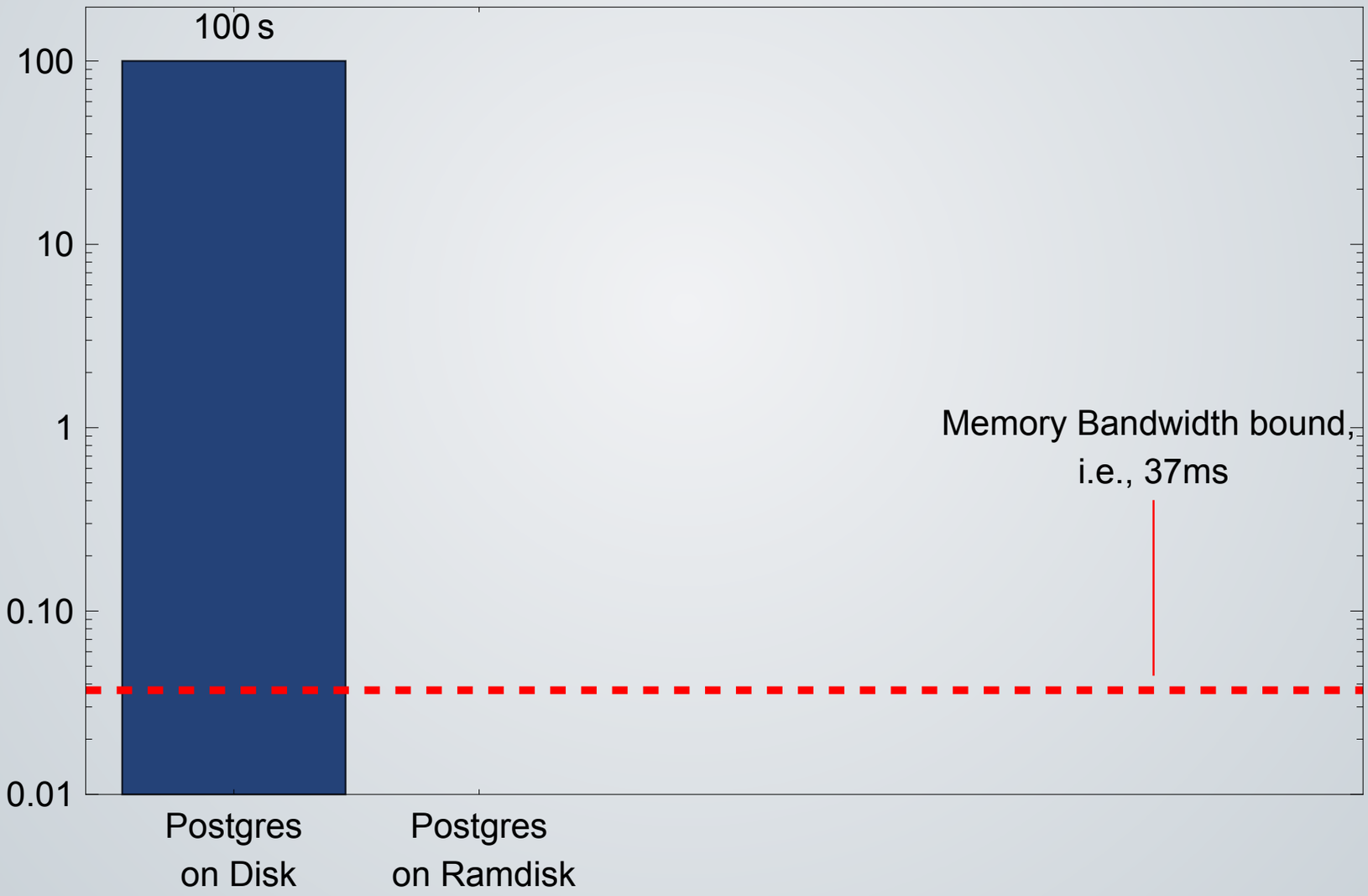
select sum(a) where b=6 group by c

Data Processing Performance - A Case Study

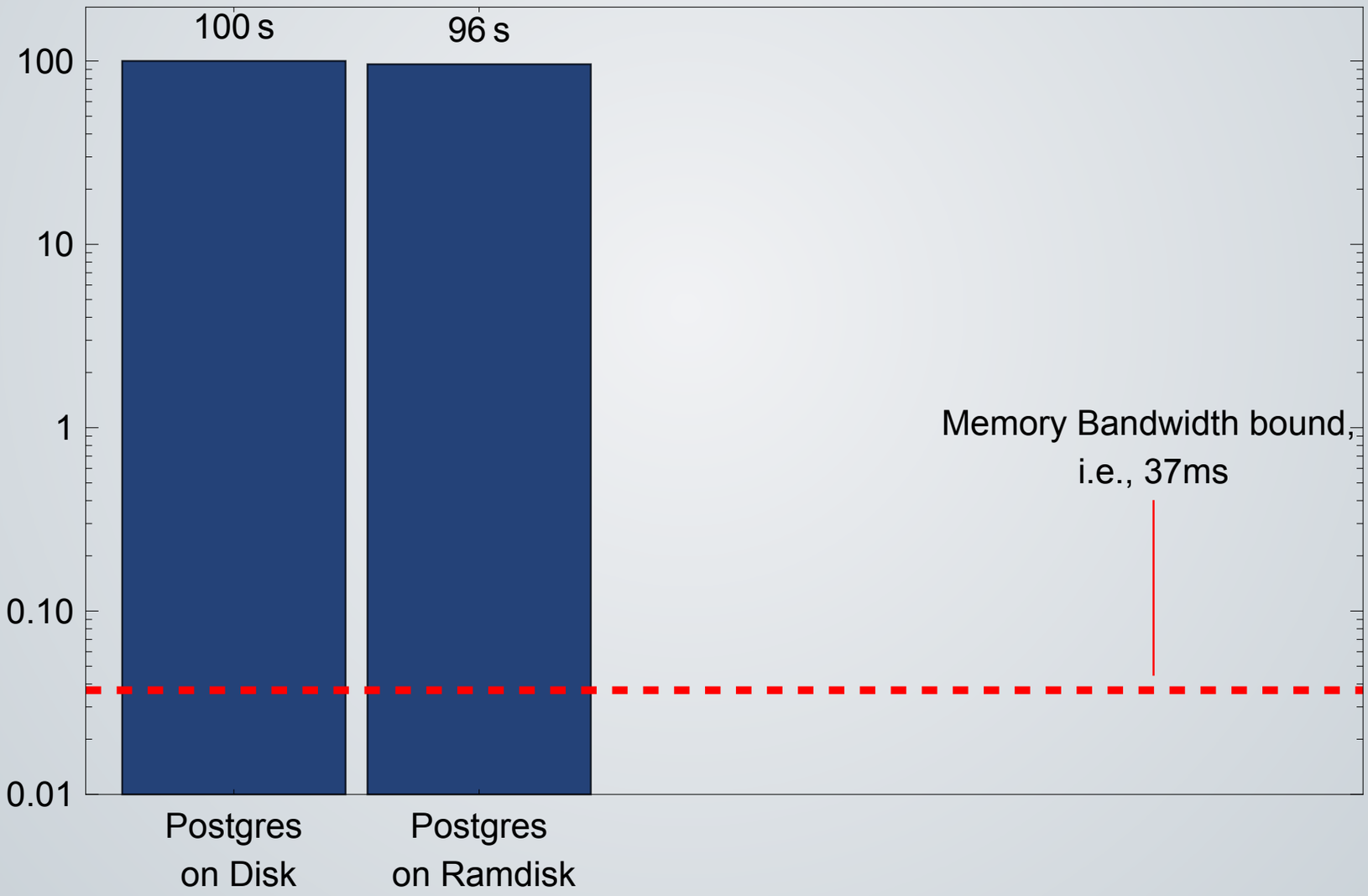


select sum(a) where b>6 group by c

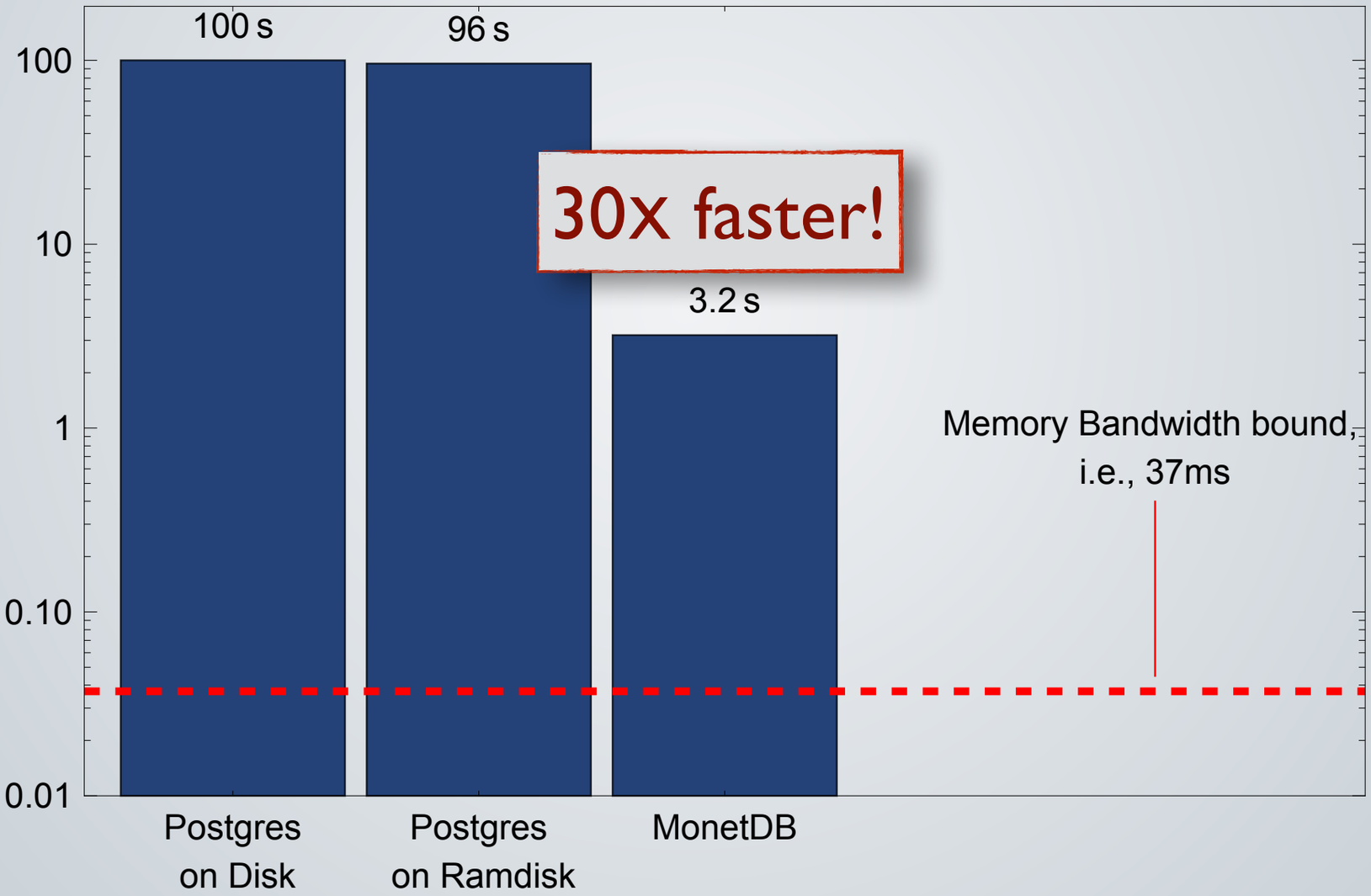
Data Processing Performance - A Case Study



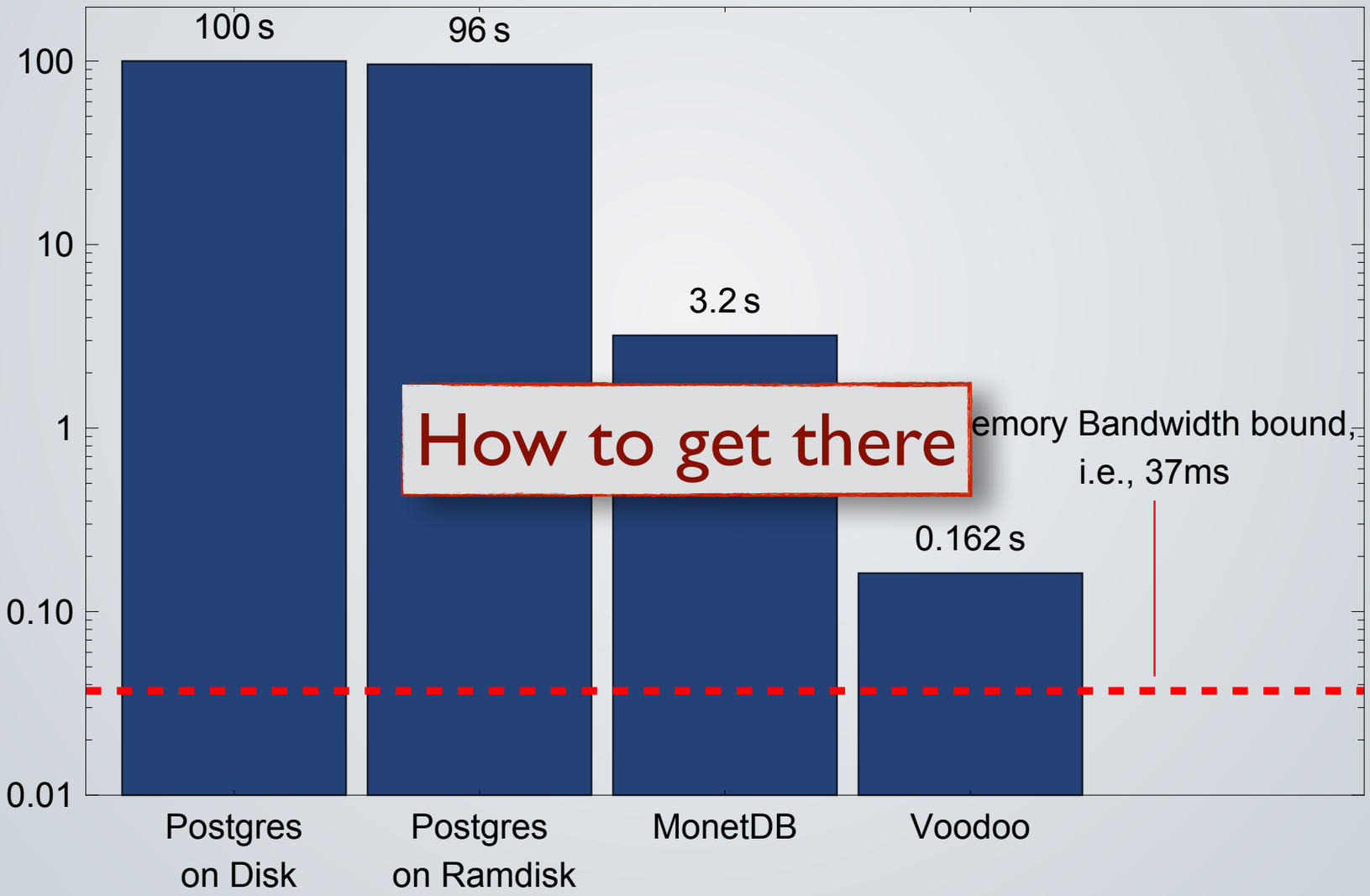
Data Processing Performance - A Case Study



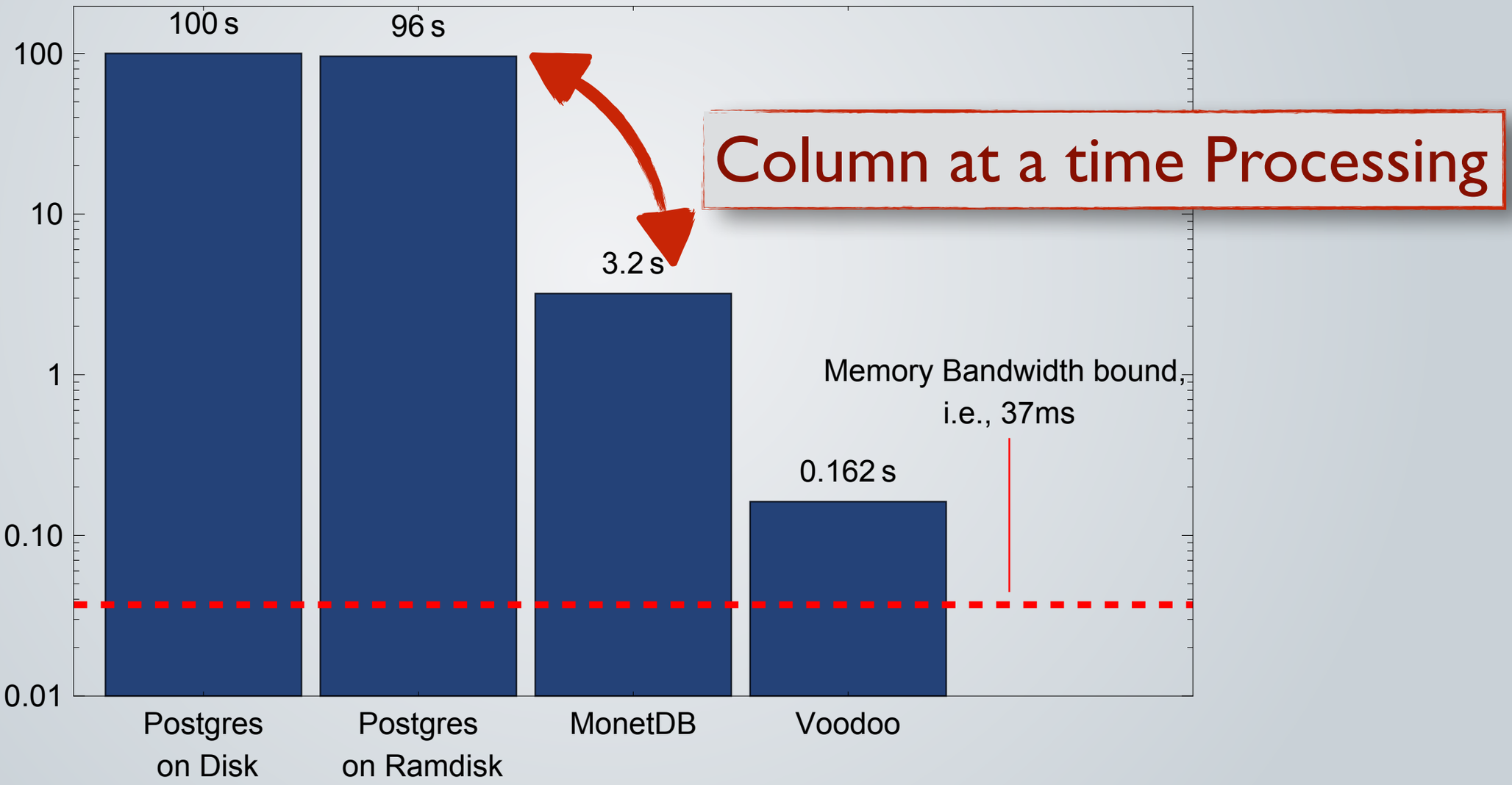
Data Processing Performance - A Case Study



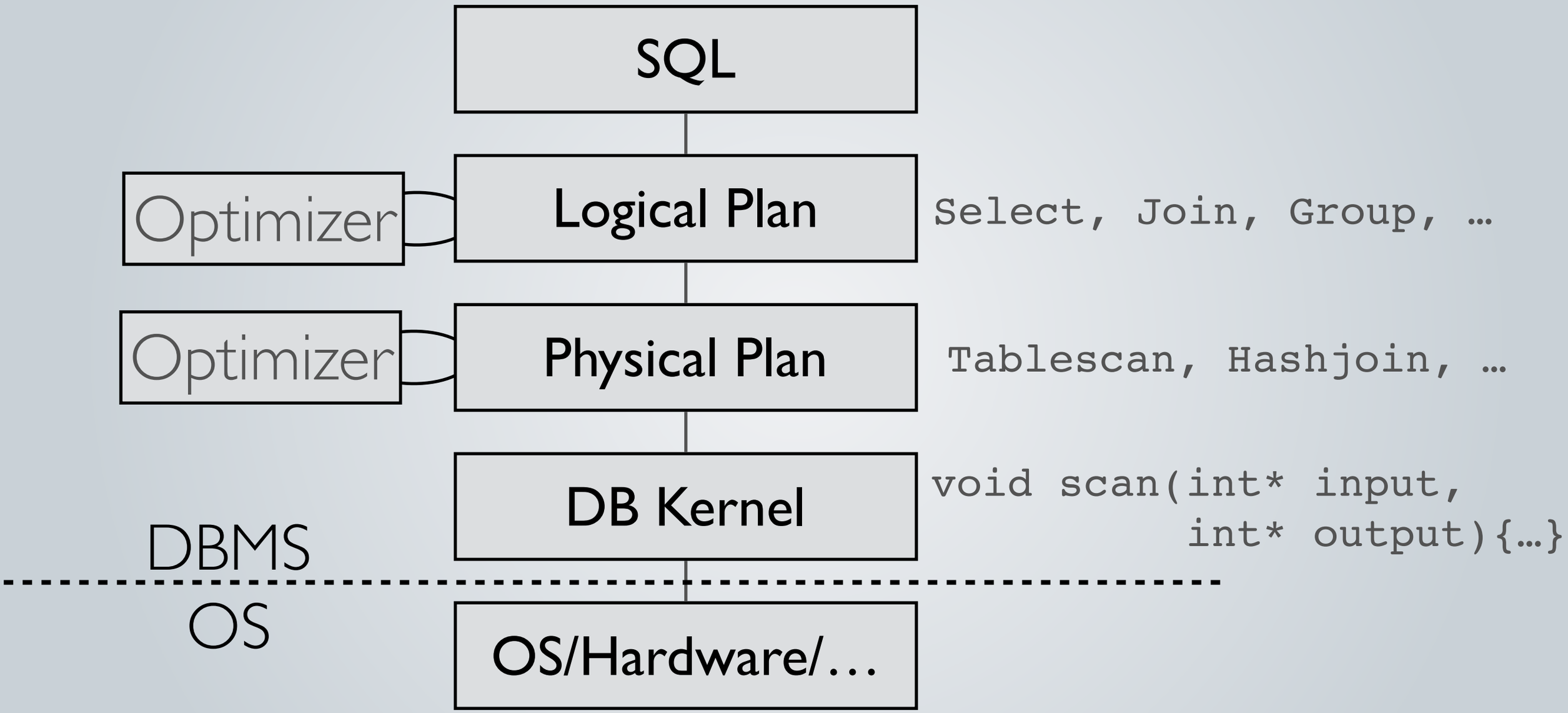
Data Processing Performance - A Case Study



Data Processing Performance - A Case Study



Classic database architecture



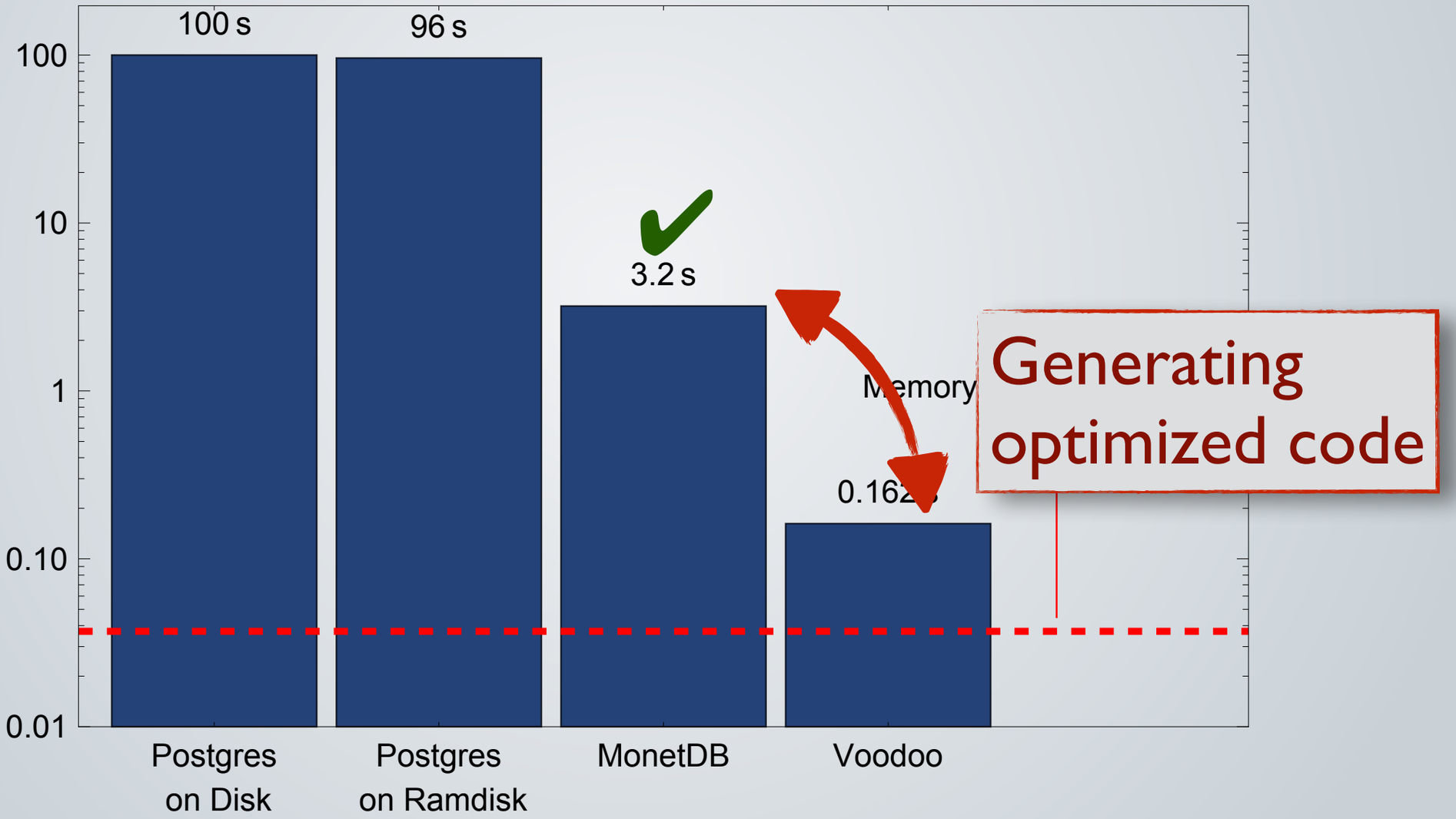
Column at a time Processing

- Postgres is a *tuple at a time* kernel
  - High interpretation overhead
  - Hard to parallelize

Ramdisk does not impact performance

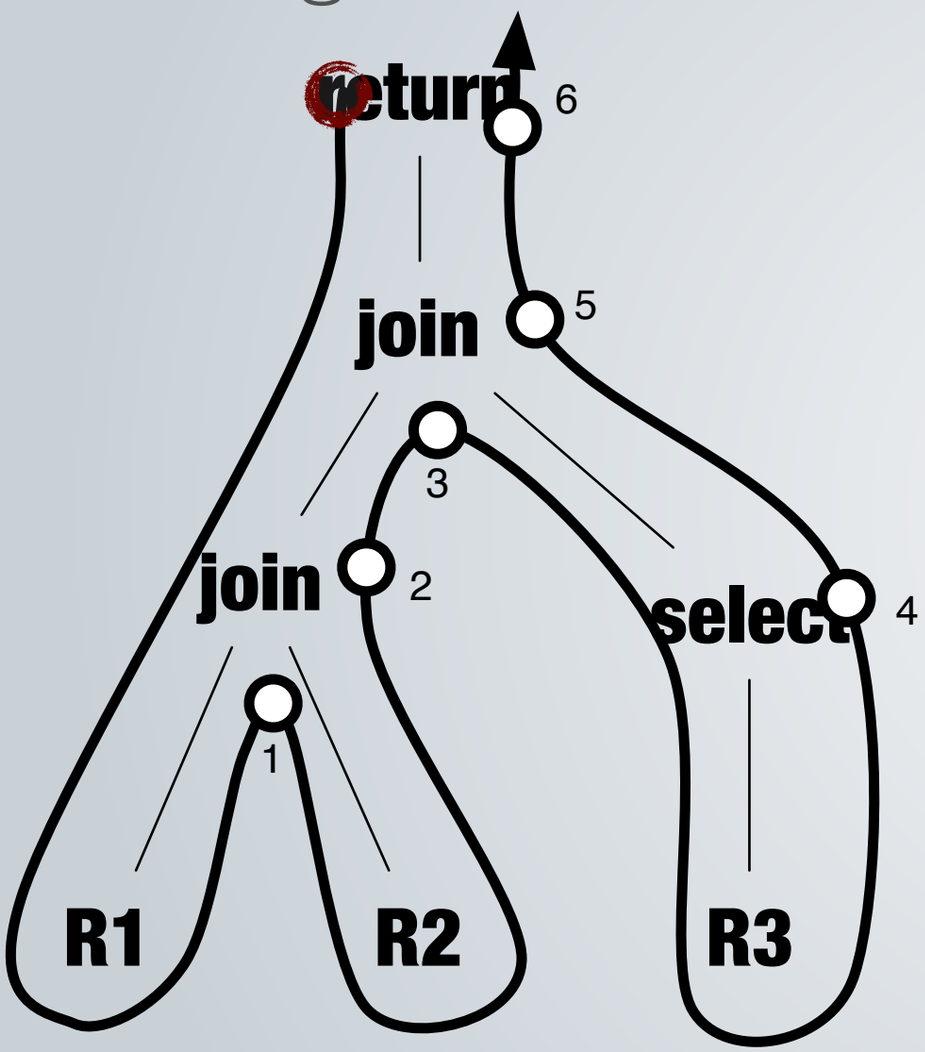
- MonetDB is a *column at a time* kernel
  - Overhead amortized over many tuples/values
  - Easy to parallelize

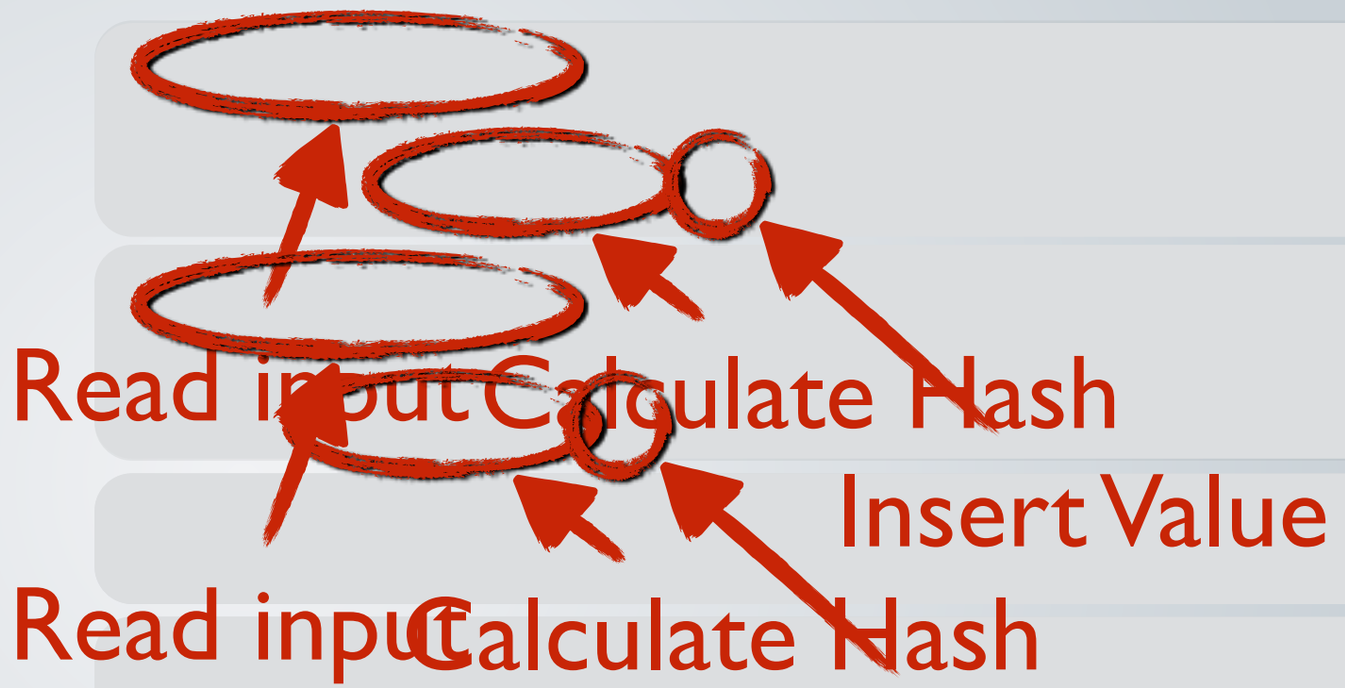
…back to our study



Query Compilers

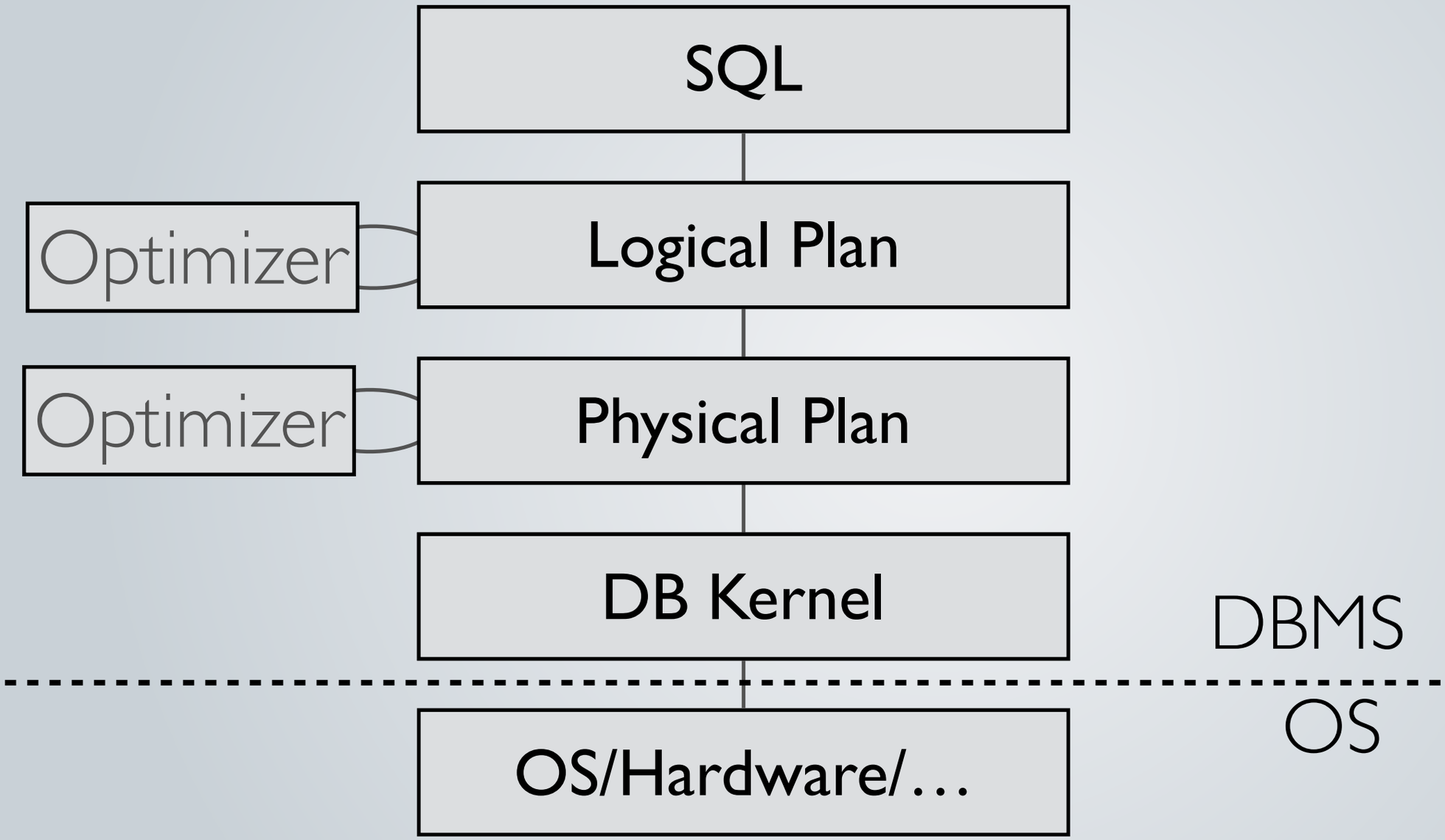
Logical Plan → Executable Program



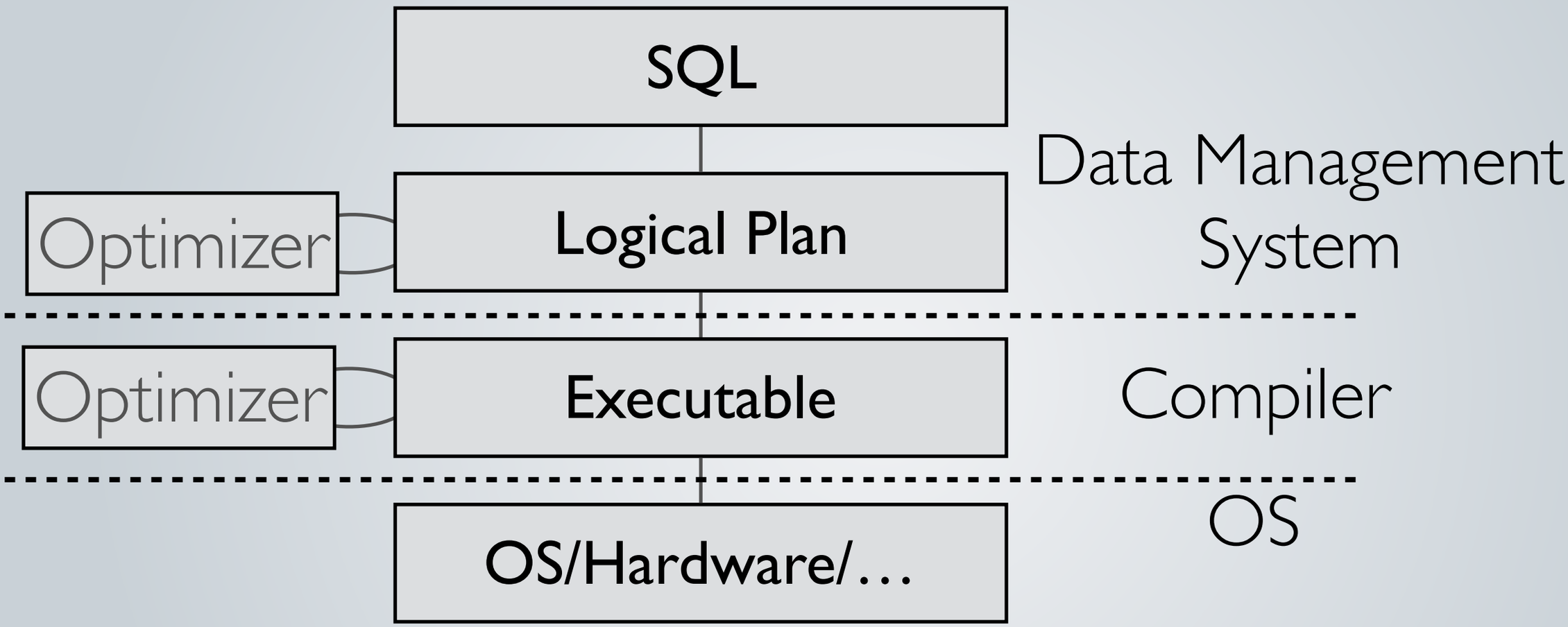


4x Performance improvement: 3.2s to .7s

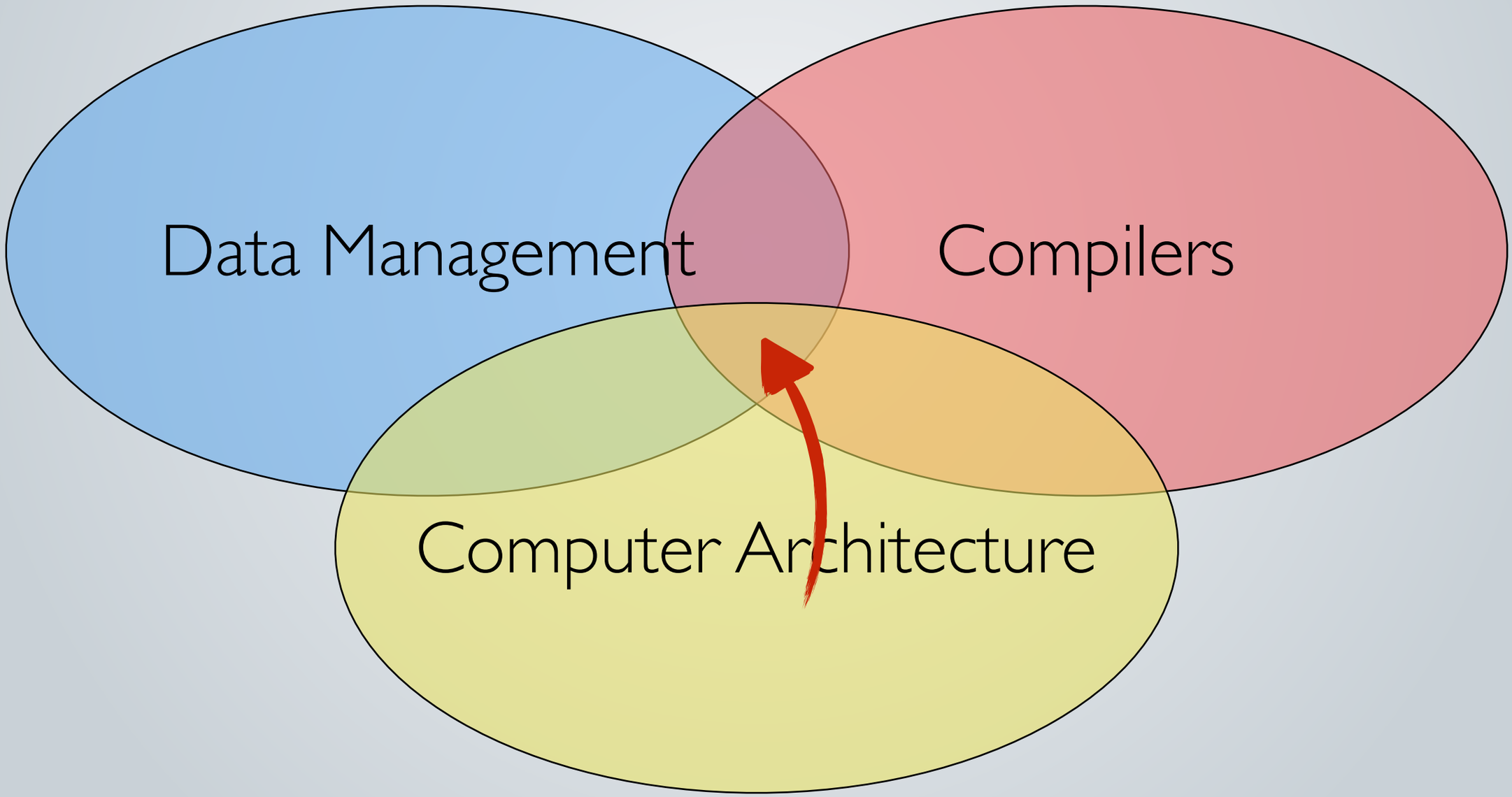
Query Compiler Architecture



Query Compiler Architecture



Database Performance Engineering



The Performance Engineering Deluge

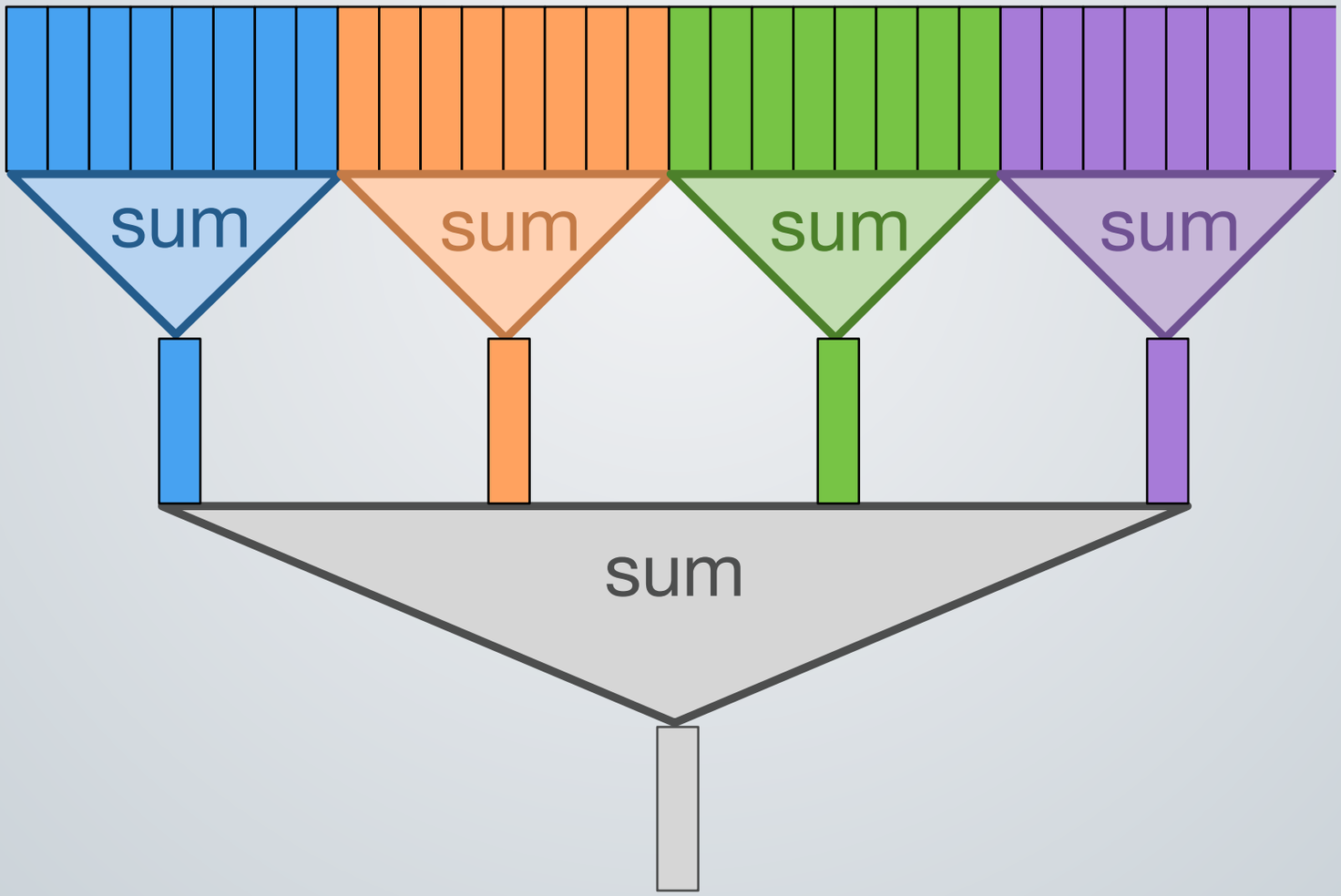
Branch-Free Selections, Radix-Joins, Vectorized Processing,
Architecture-Conscious Hashing, SIMD-Parallel Processing,
Bitwise Processing, NUMA-Aware Processing,
Superscalar (De)compression, Instruction-Cache Aware
Processing, Multicore-Parallelism, Co-Processing, …

Multicore vs. SIMD

Multicore-Parallelism

SIMD-Parallel Processing

A data processing example



Multicore-Parallelism using TBB

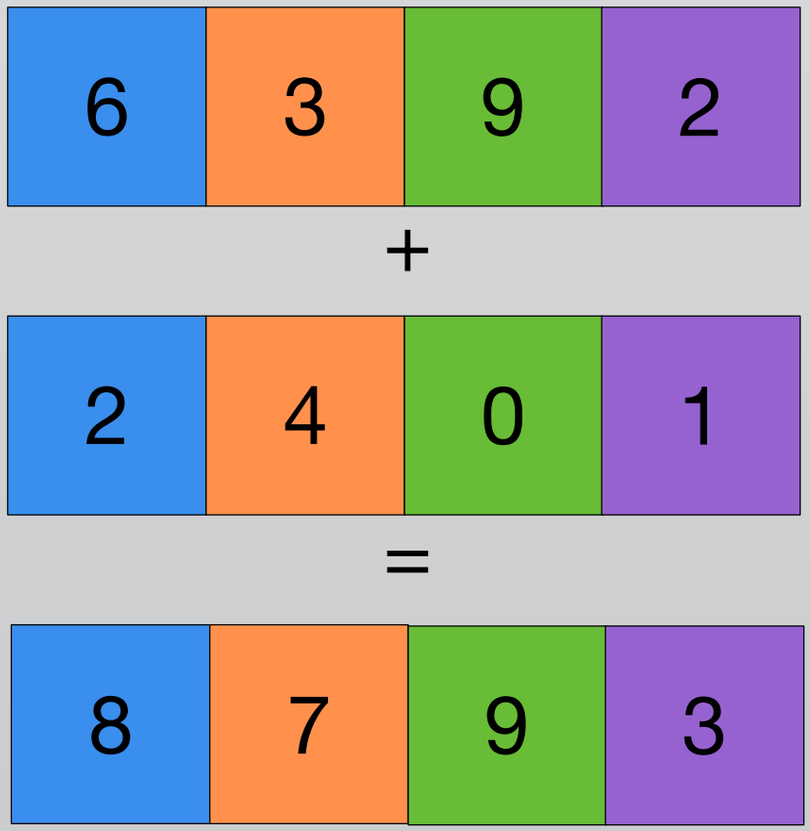
```
1  auto input = load("input");
2  auto totalssum =
3   parallel_deterministic_reduce(
4     blocked_range<size_t>(0, input.size,
5                           input.size / 1024)
6   0,  [&input](auto& range, auto partsum) {
7           for(size_t i = range.begin();
8               i < range.end(); i++) {
9        partsum += input.elements[i].constant;
10      }
11      return partsum;
12    },
13  [](auto s1, auto s2) { return s1 + s2; });
```

- Parallel Reduce
- Block-Range
- Per-Partition Lambda
- Global Lambda

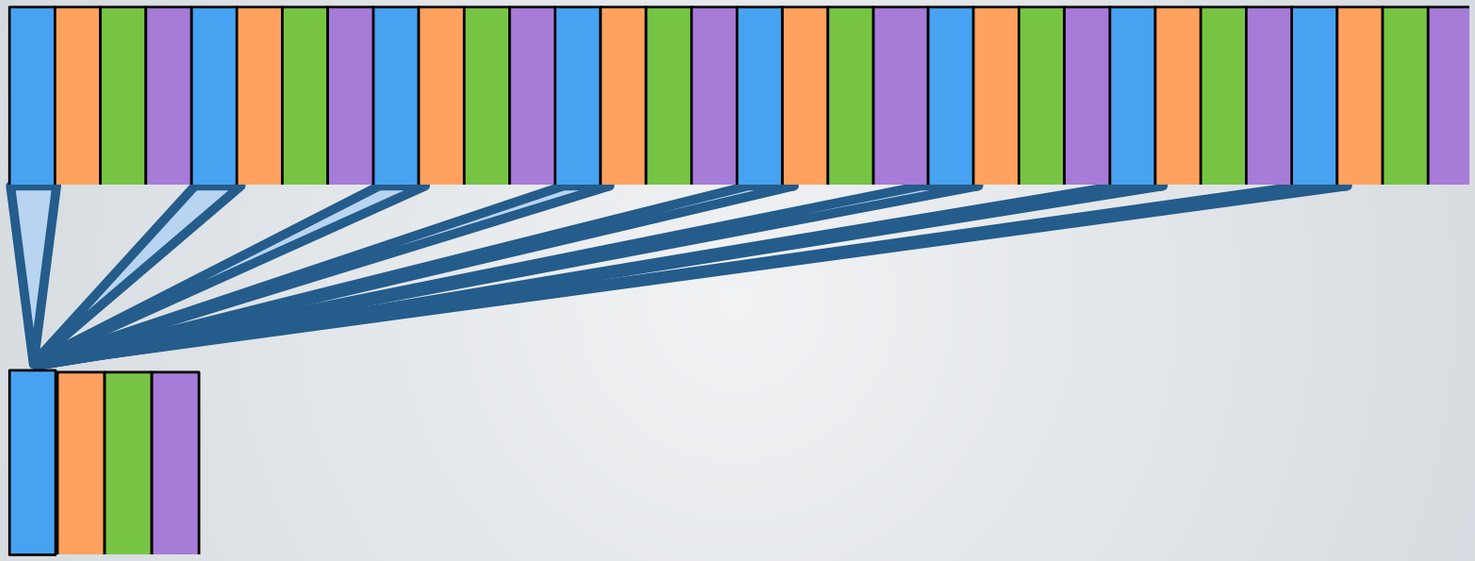
Multicore vs. SIMD

Single Instruction Multiple Data parallelism

Vectorized Add:



A data processing example using SIMD



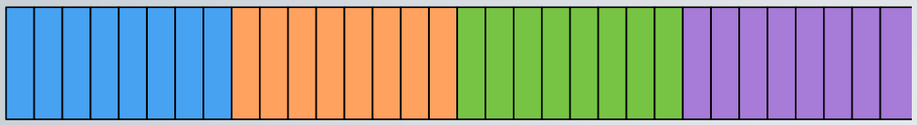
A data processing example using SIMD

```
auto input = load("input");
typedef int v4i __attribute__((vector_size(16)));
auto vSize = (sizeof(v4i) / sizeof(int));
v4i sums = {};
for(size_t i = 0; i < input.size / vSize; i++) {
 sums += ((v4i*)input.elements)[i];
}
int* scalarSums = (int*)&sums;
auto totalsum = 0l;
for(size_t i = 0; i < 4; i++) {
 totalsum += scalarSums[i];
}
```

- SIMD Datatypes
- Loop Bound Adaption
- Array Cast
- Sequential Reduction

Multicore vs. SIMD

Multicore-Parallelism



- Parallel Reduce
- Block-Range
- Per-Partition Lambda
- Global Lambda

SIMD-Parallel Processing

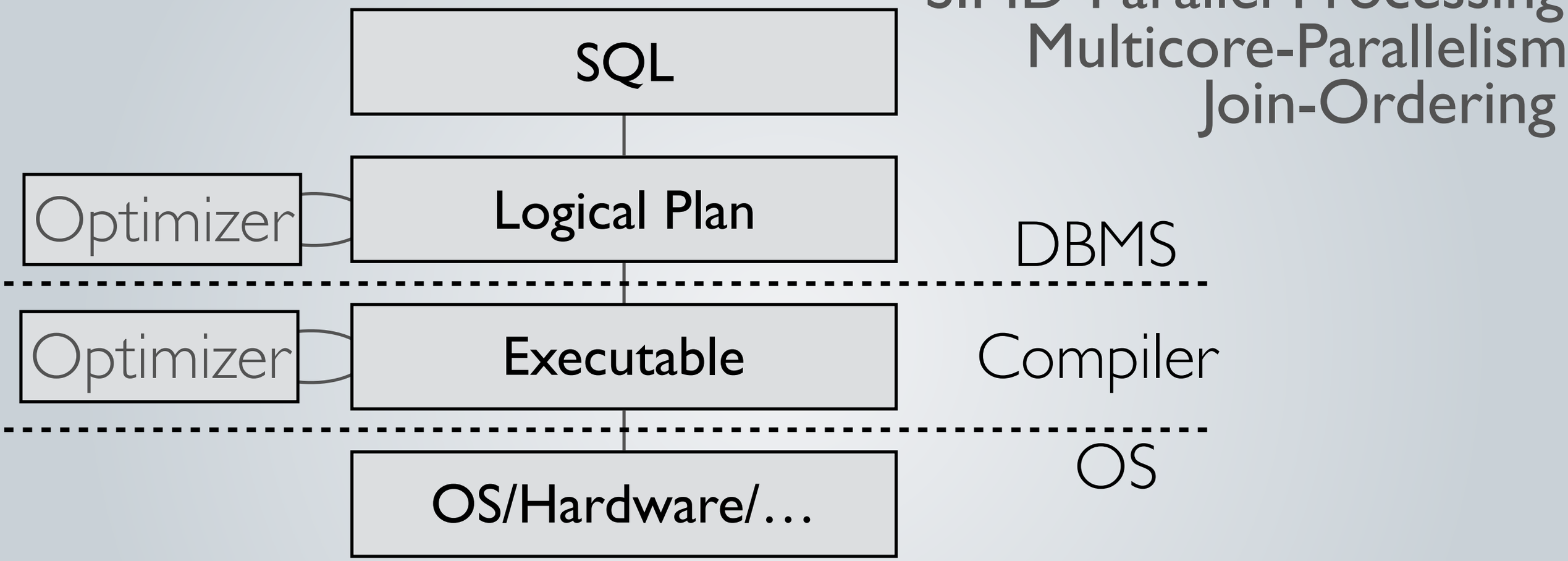
- SIMD Datatypes
- Loop Bound Adaption
- Array Cast
- Sequential Reduction

Particularly problematic when generating code

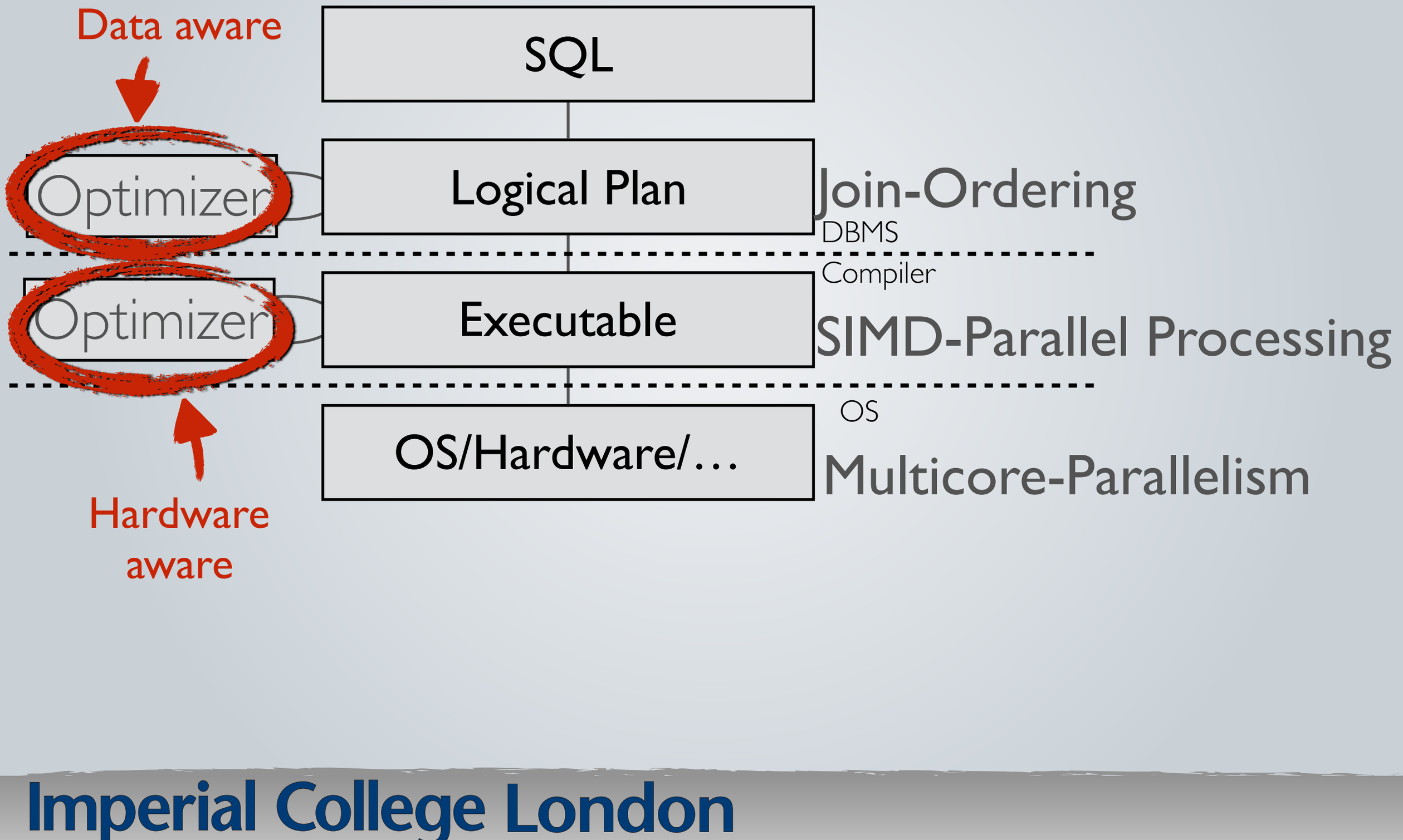
Query Compiler Architecture

SIMD-Parallel Processing
Multicore-Parallelism
Join-Ordering

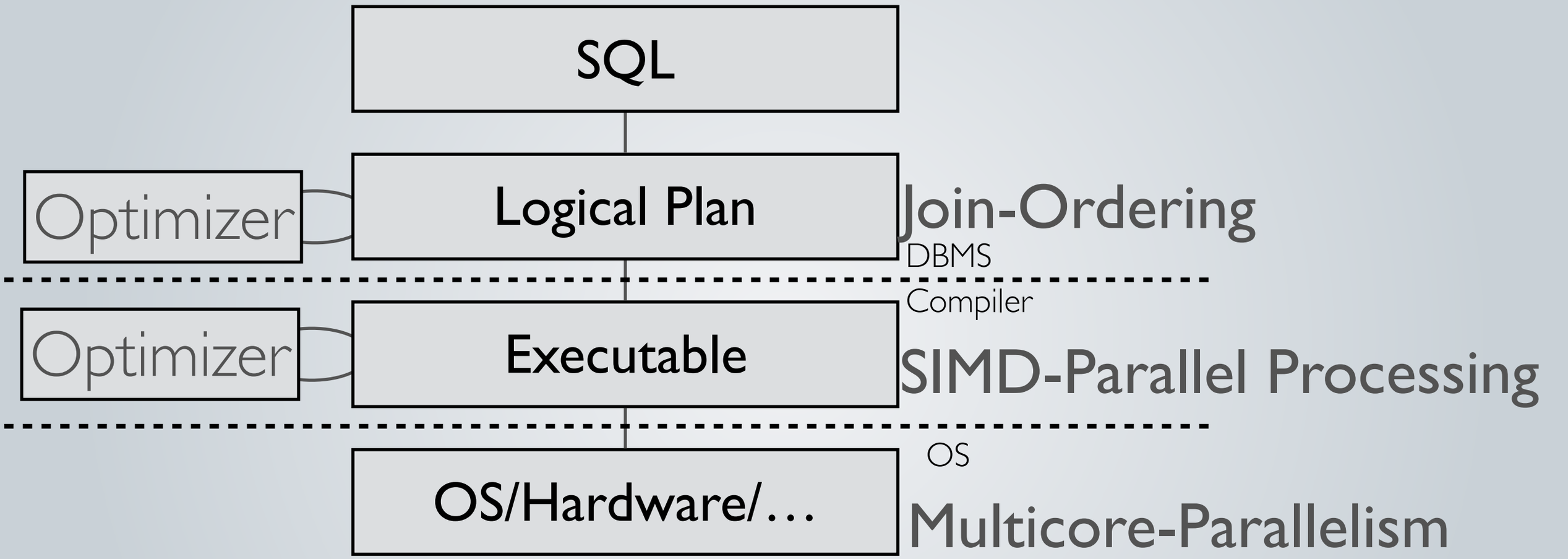
Query Compiler Architecture



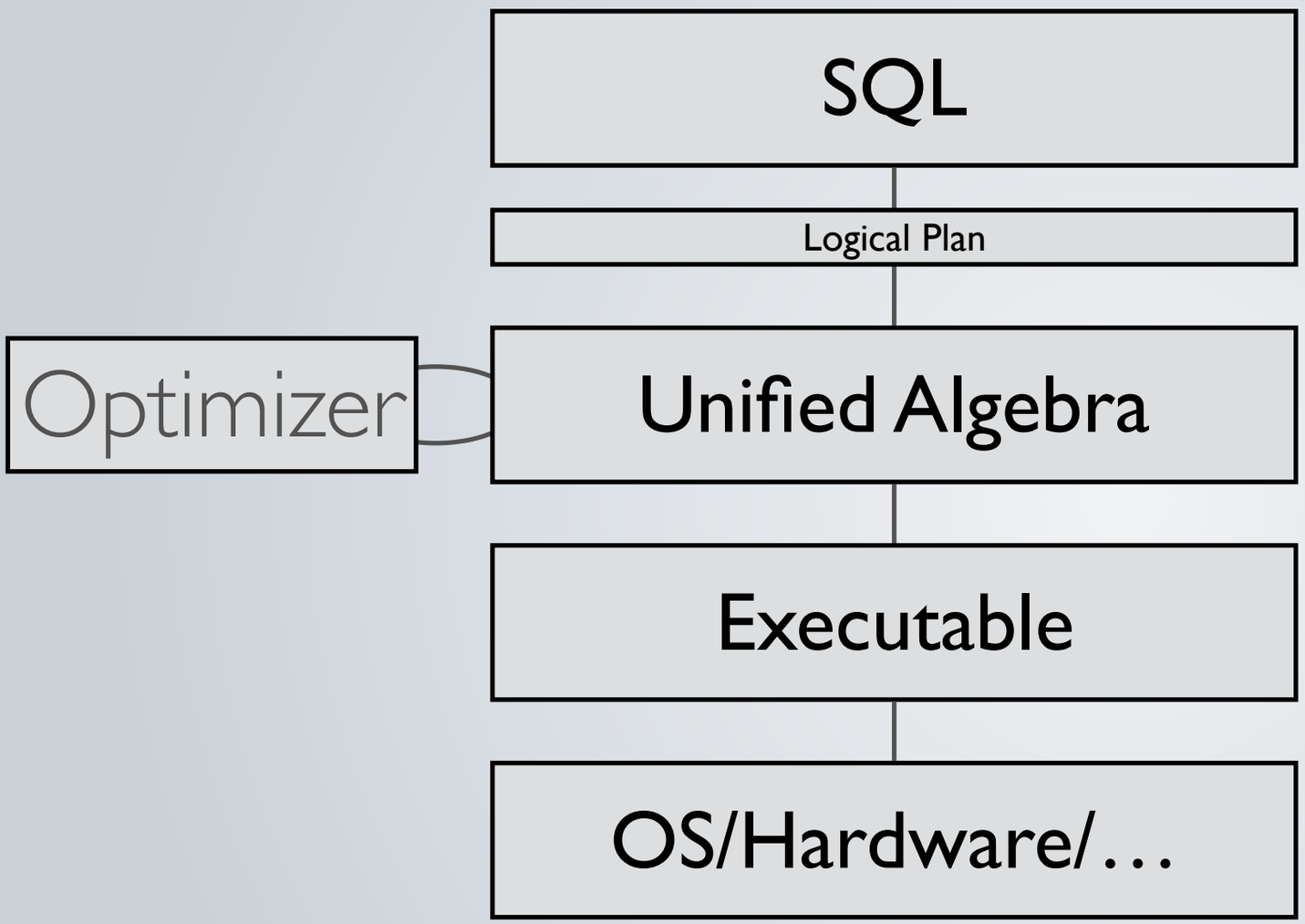
Query Compiler Architecture



What if…


SQL
Optimizer
Logical Plan
Join-Ordering
DBMS
Compiler
Optimizer
Executable
SIMD-Parallel Processing
OS
OS/Hardware/…
Multicore-Parallelism

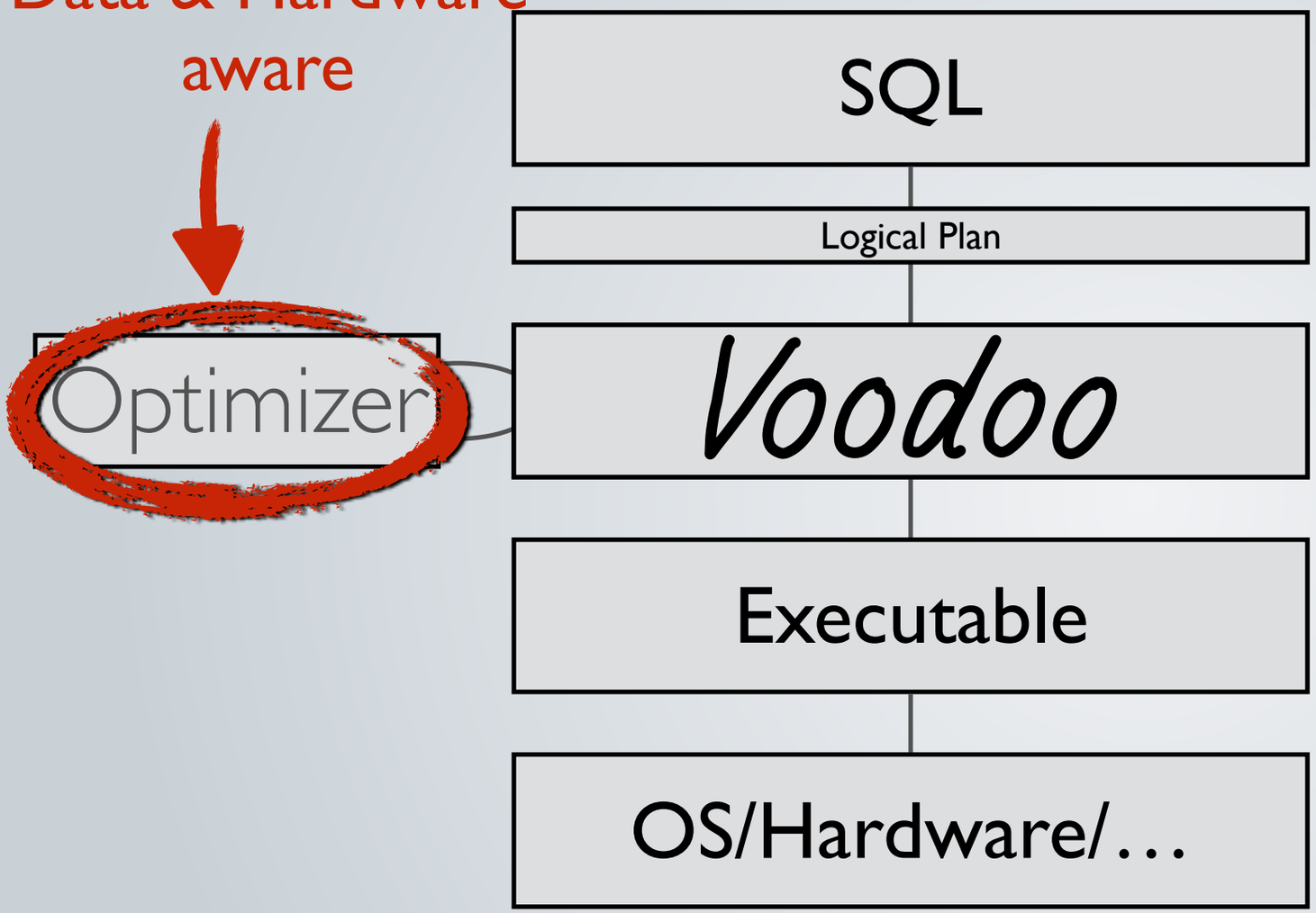
What if…



SIMD-Parallel Processing
Multicore-Parallelism
Join-Ordering

What if we had an intermediate algebra

Data & Hardware aware



SIMD-Parallel Processing
Multicore-Parallelism
Join-Ordering

Voodoo

A Vector Algebra

A portable high-performance database kernel

A platform for database performance engineering

[VLDB 2016/17]

Imperial College London

The Voodoo Vector Algebra

Imperial College London

Design goals

- Fast and expressive like C
- Optimizable like relational algebra
- Portable to different devices
- (Enable serendipitous discovery of new optimizations)

Design goals

- Fast and expressive like C
  - All tuning decisions are explicit in the program
  - Focused on data processing
- Optimizable like relational algebra
  - Dataflow ⇒ Optimization are graph transformation rules
- Portable to different devices
  - Least common denominator data model and operator set

Data model: the least common denominator of targeted hardware

- struct {int id;
  struct {
  int x;
  int y
  } position;
  } poi[n]

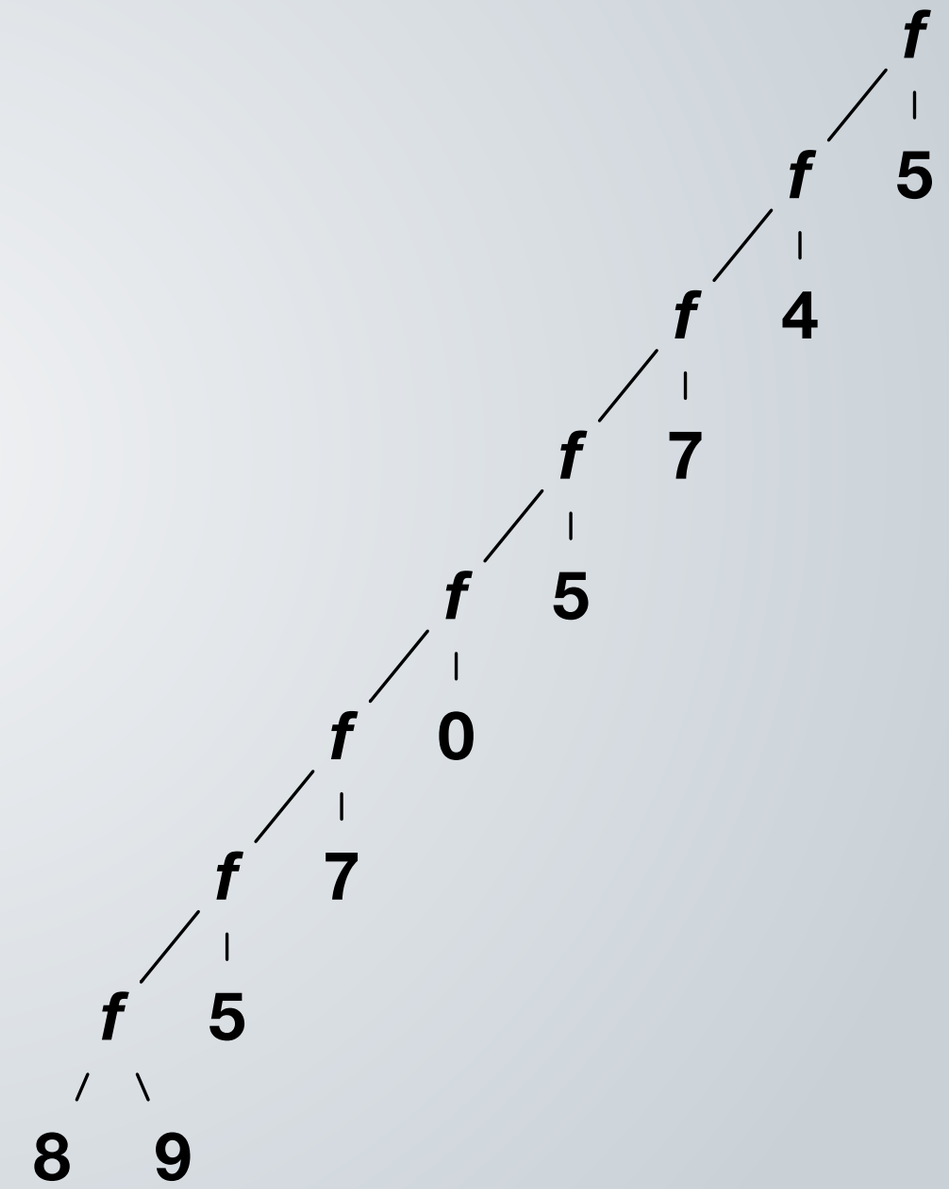
id	pos.x	pos.y
15	7	9
21	4	5
11	10	14
92	7	1
78	45	12
65	0	12

All Voodoo operators are parallel, some are controlled

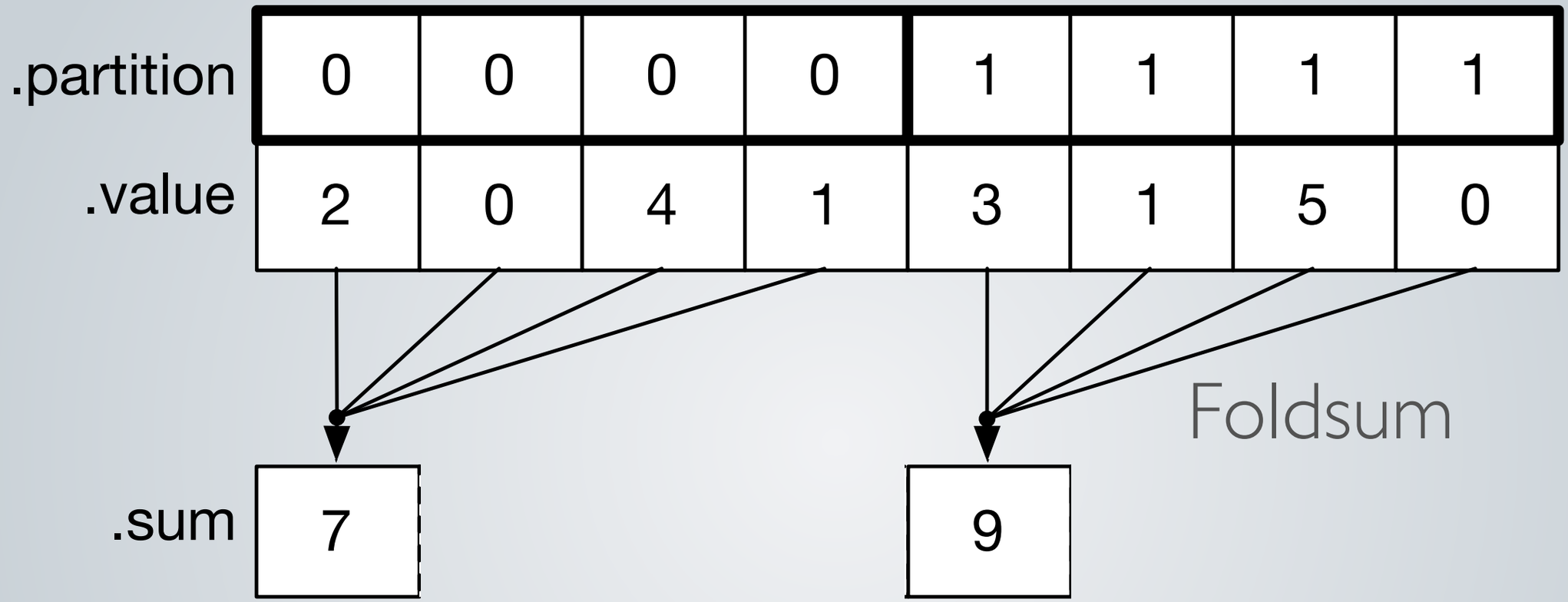
- Map-like (Fully Data Parallel):
  - `Project, Zip, Arithmetic, Logical, Bit Operations, Gather (i.e., z = x[y]), Cross`
- **Controlled:**
  - `FoldSelect, FoldSum, FoldMax, FoldScan, Scatter (i.e, x[y] = z), …`

A short revision on fold

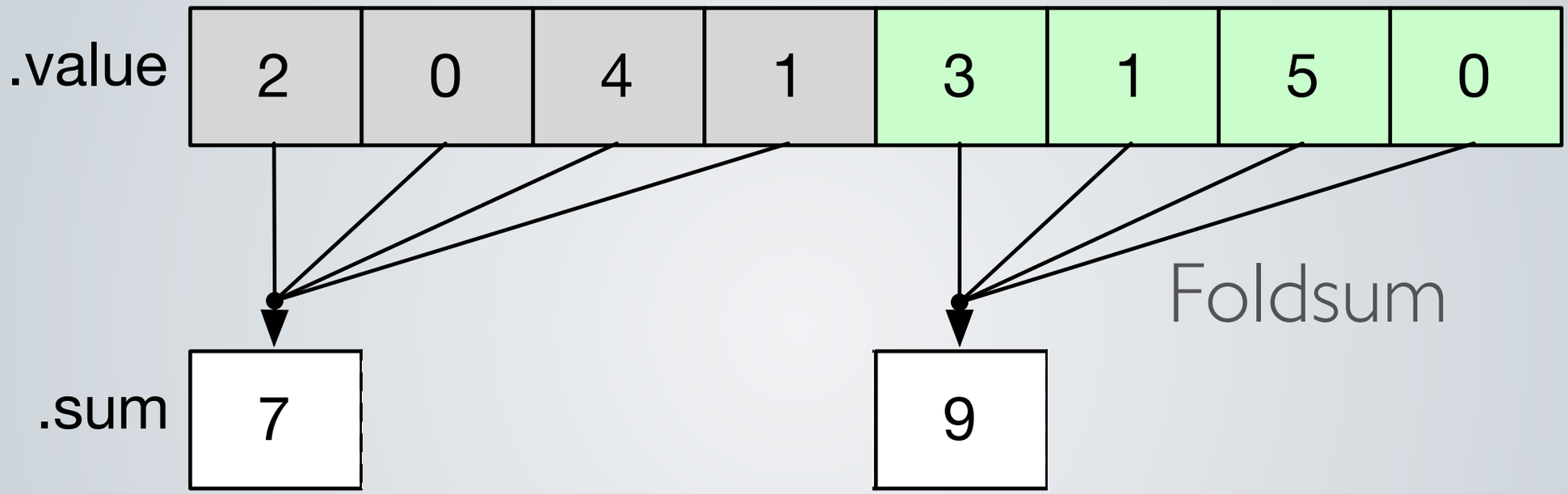
- Fold(f, [8,9,5,7,0,5,7,4,5])



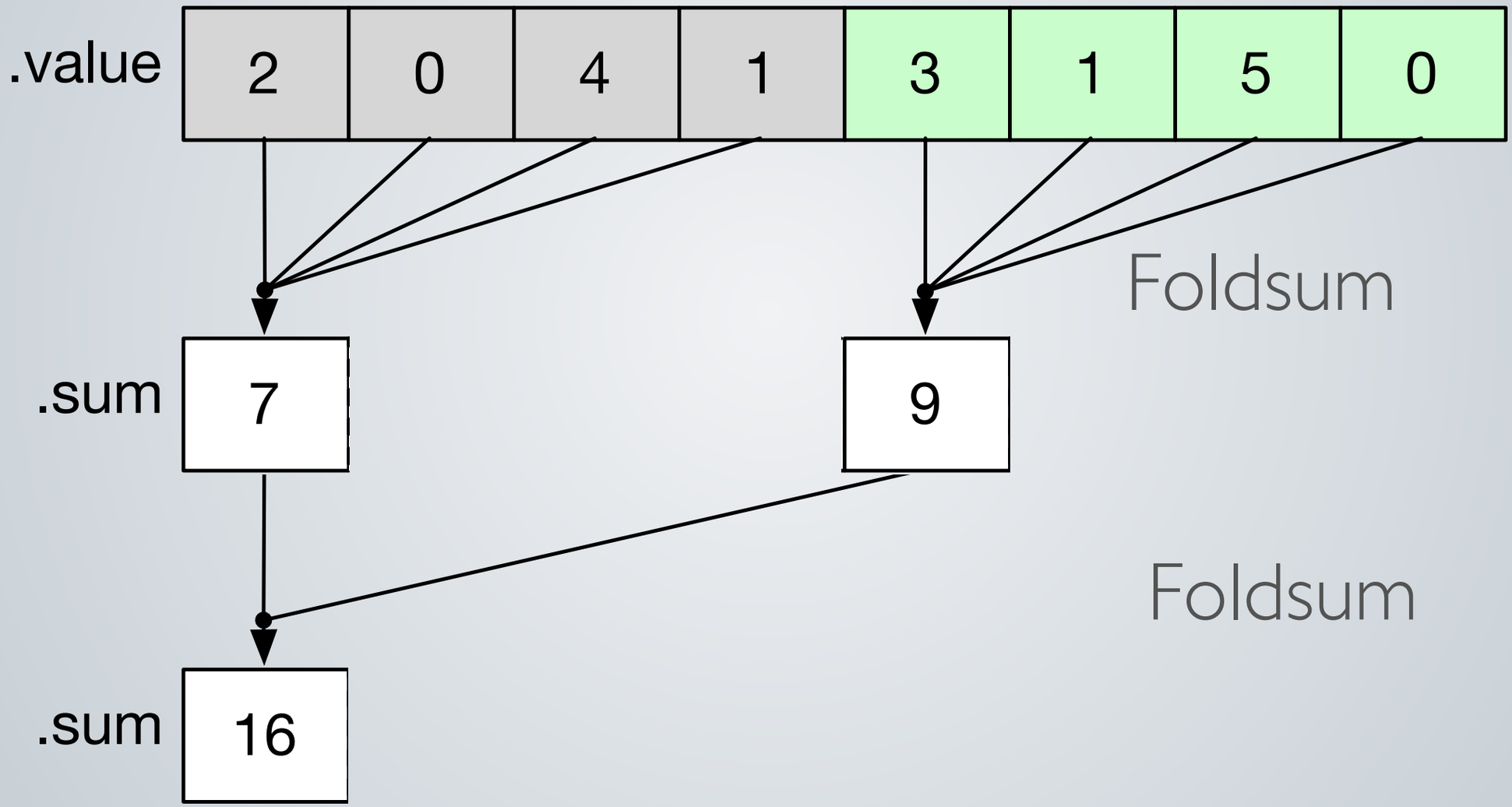
Control: explicit partition assignment



Controlled folding - declarative & tunable



Controlled folding - declarative & tunable



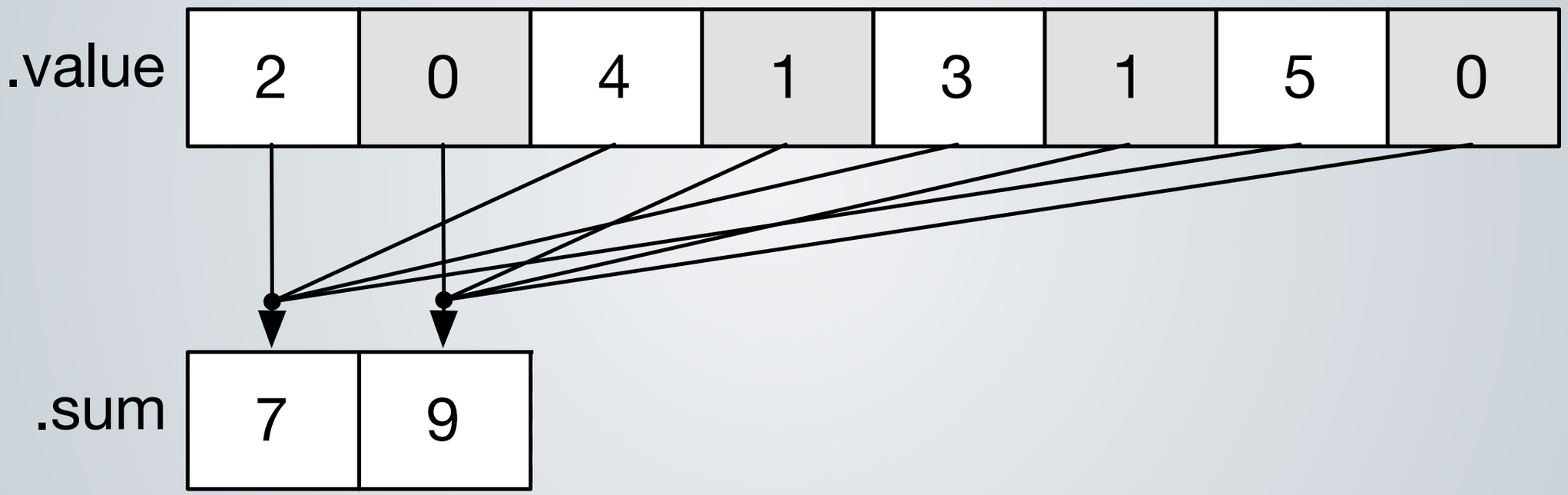
Creating Control Vectors

0	1	2	3	4	5	6	7	range(data)
÷	4	4	4	4	4	4	4	4 constant(data, 4)
=	0	0	0	0	1	1	1	1 control vector

Partitioned aggregation in Voodoo is controlled

```
1  input := Load("input") // Single Column: val
2  ids := Range(input)
3  partitionSize := Constant(4)
4  partitionIDs := Divide(ids, partitionSize)
5  positions := Partition(partitionIDs)
6  inputWPart := Zip(input, partition)
7  partInput := Scatter(inputWPart, positions)
8  pSum := FoldSum(partInput.val, partInput.partition)
9  totalSum := FoldSum(pSum)
```

Lanewise folding abstracts SIMD parallelism…



Creating Control Vectors

0	1	2	3	4	5	6	7	range(data)

mod	2	2	2	2	2	2	2	2	constant(data, 2)

=	0	1	0	1	0	1	0	1	partition IDs

…and looks almost the same as multicore-parallelism

```
1  input := Load("input") // Single Column: val
2  ids := Range(input)
3  partitionSize := Constant(1024)
4  partitionIDs := Divide(ids, partitionSize)
5  positions := Partition(partitionIDs)
6  inputWPart := Zip(input, partition)
7  partInput := Scatter(inputWPart, positions)
8  pSum := FoldSum(partInput.val
9  totalSum := FoldSum(pSum)
```

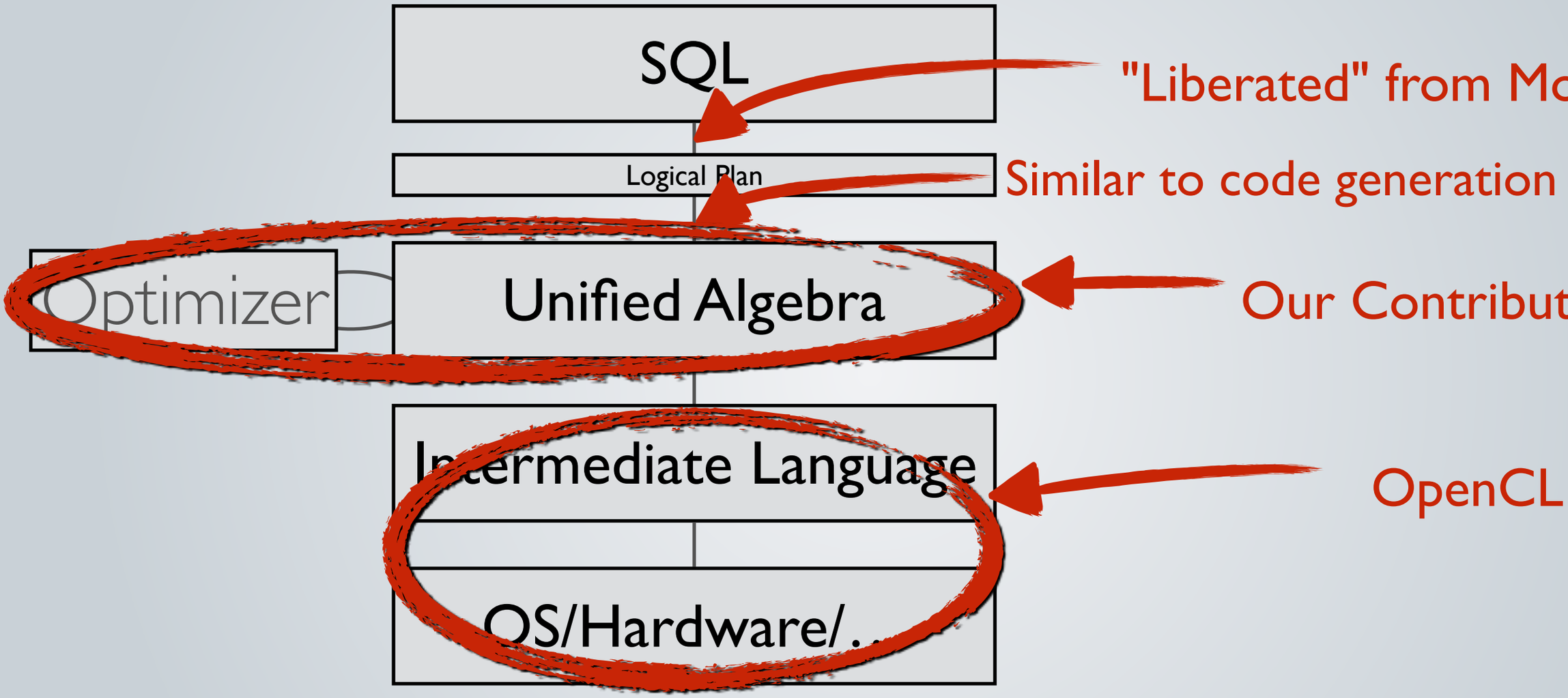
backend performs simple static analysis

```
1  input := Load("input") // Single Column: val
2  ids := Range(input)
3  laneCount := Constant(2)
4  partitionIDs := Modulo(ids, laneCount)
5  positions := Partition(partitionIDs)
6  inputWPart := Zip(input, partition)
7  partInput := Scatter(inputWPart, positions)
8  pSum := FoldSum(partInput.val, partInput.partition)
9  totalSum := FoldSum(pSum)
```

A portable high-performance database kernel

Imperial College London

The Voodoo query processing system

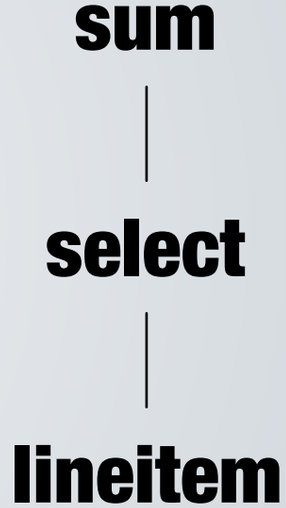


Let's extend our example

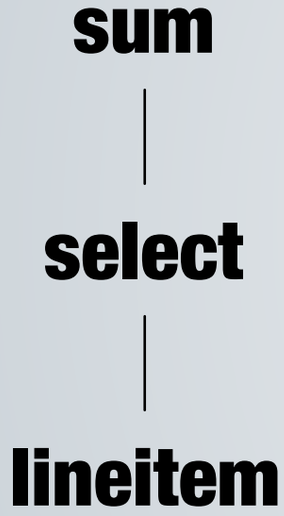
```
SELECT SUM(l_quantity)
FROM lineitem
WHERE l_shipdate> 5
```

MonetDB generates a logical plan…

```
SELECT SUM(l_quantity)
FROM lineitem
WHERE l_shipdate> 5
```

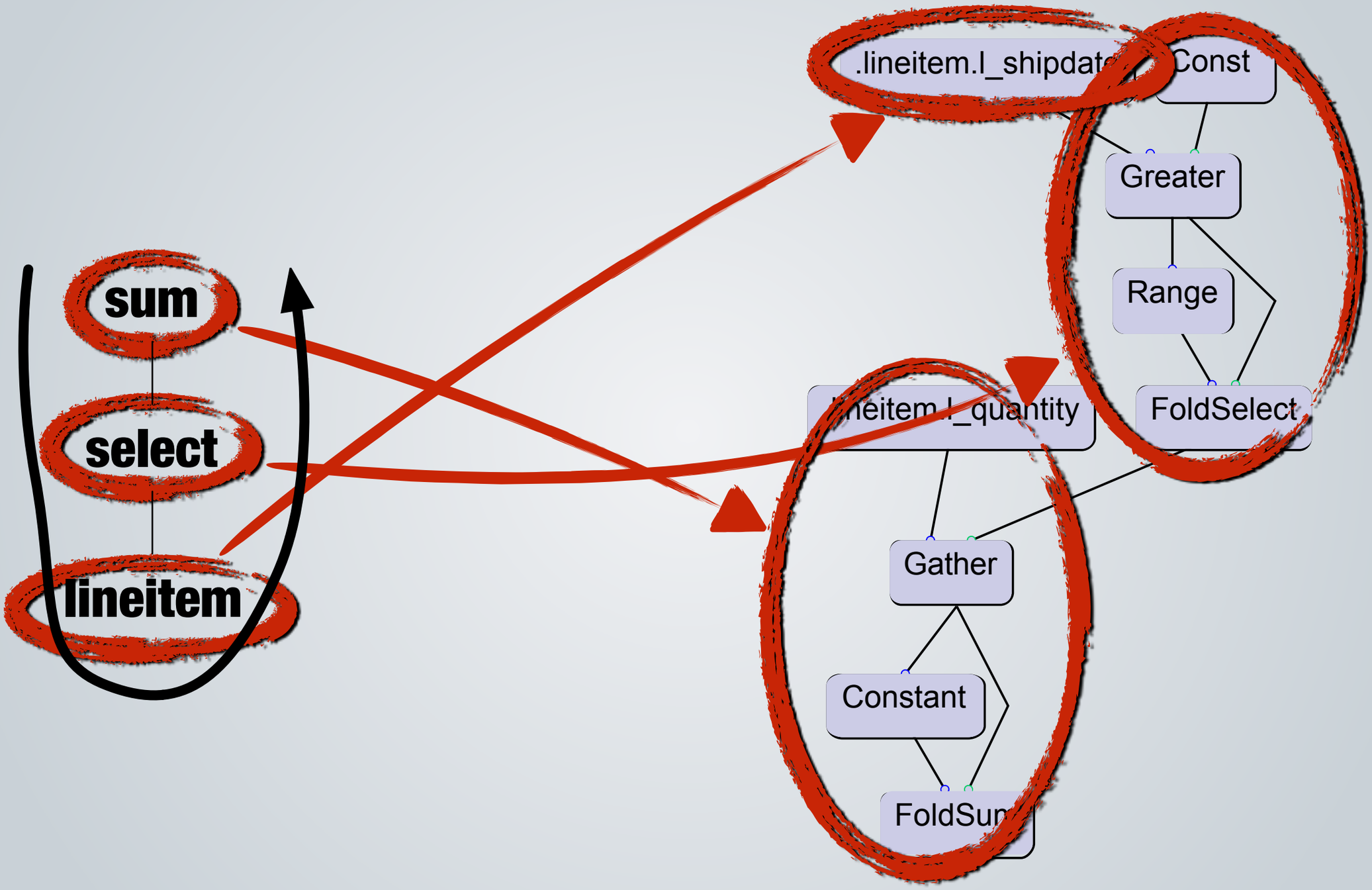


…we compile the logical plan to Voodoo…

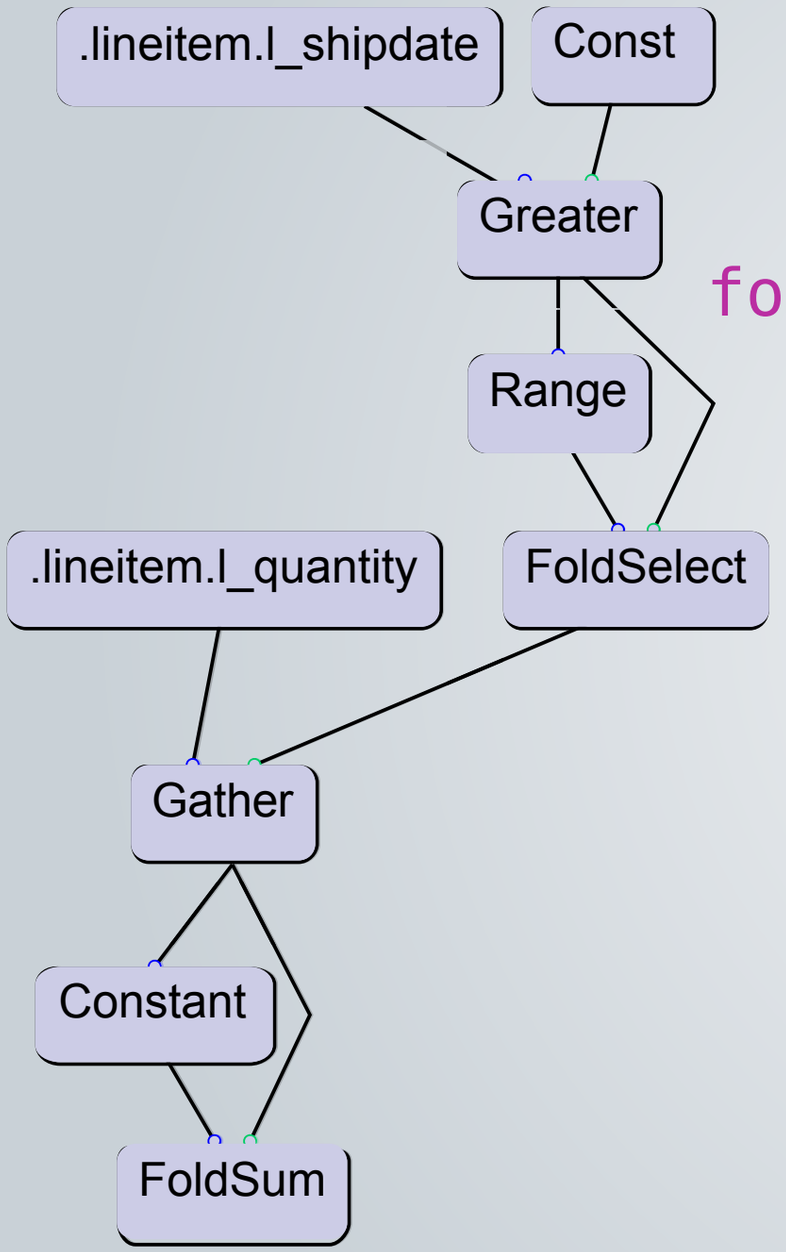



```
tmp1 = Load(.lineitem.l_quantity)
tmp2 = Load(.lineitem.l_shipdate)
tmp3.val = Range(728659,tmp2,0)
tmp4.val = Greater(tmp2,.l_shipdate,tmp3,.val)
tmp5.val = Range(0,tmp4,1)
tmp6 = Zip(.fold,tmp5,.val,.value,tmp4,.val)
tmp7.val = FoldSelect(tmp6,.fold,.value)
tmp8 = Gather(tmp1,tmp7,.val)
tmp14.val = Range(0,tmp8,0)
tmp15 = Zip(.fold,tmp14,.val,.value,tmp8,.val)
tmp16.L1 = FoldSum(tmp15,.fold,.value)
Return(tmp16)
```

. . . i.e., a dataflow program

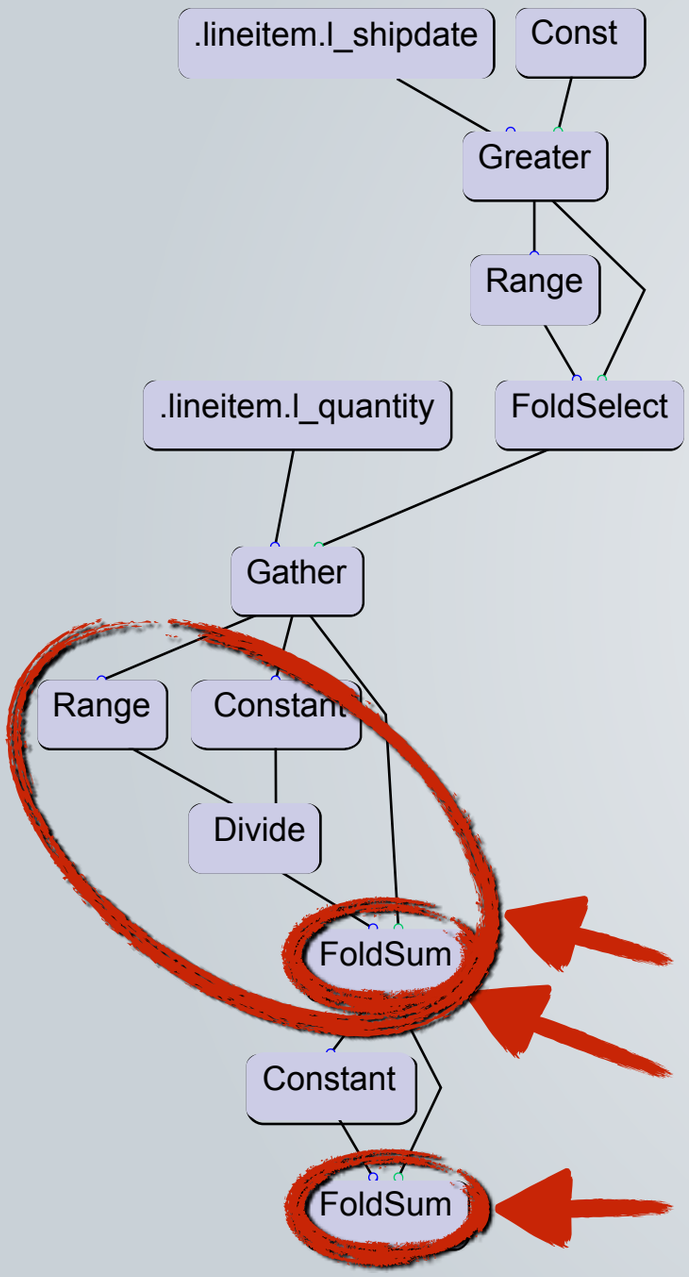


Generating "undergraduate" C from Voodoo




```
for(size_t i = 0; i < sizeof(l_shipdate); i++)
   if(l_shipdate[i] > 5)
      result += l_quantity[i]
```

The “graduate" version



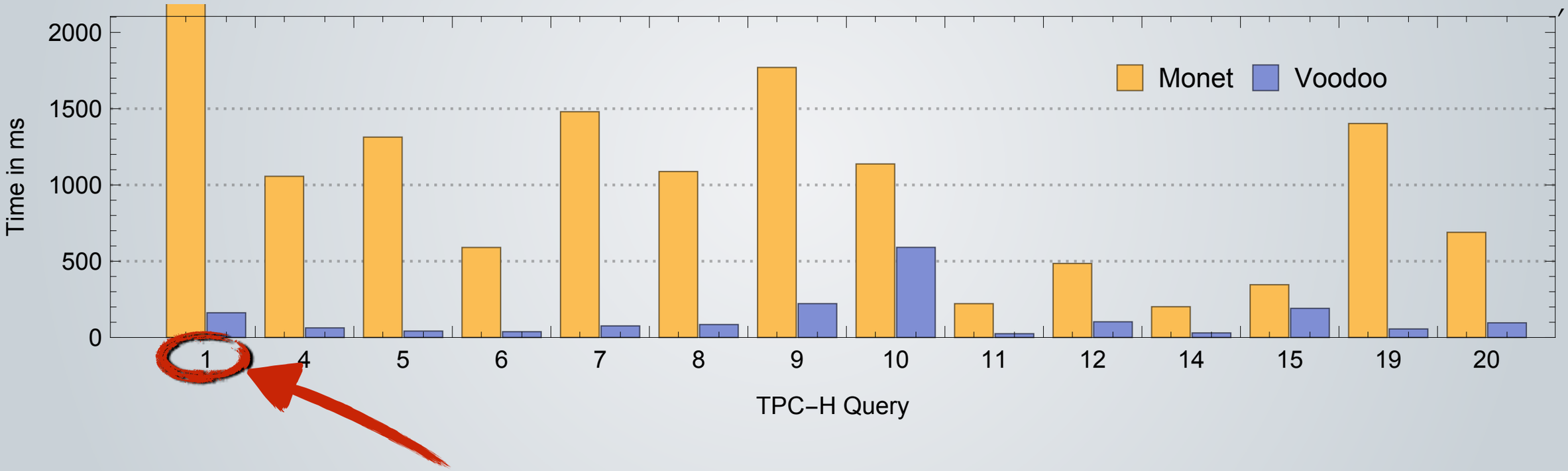

```
extern size_t inputSize;

void fragment1(int* tmp, int* s_date, int* quantity) {
  for(size_t i = 0; i < grainsize; i++)
    if(s_date[pId * grainsize + i] > 5)
      out[pId] += quantity[pId * grainsize + i]
}

void fragment2(int* out, int* tmp) {
  for(size_t i = 0; i < inputSize / grainsize; i++)
    output[0] += tmp[i]
}
```

Multicore-partitioned fold

control-incompatible folds

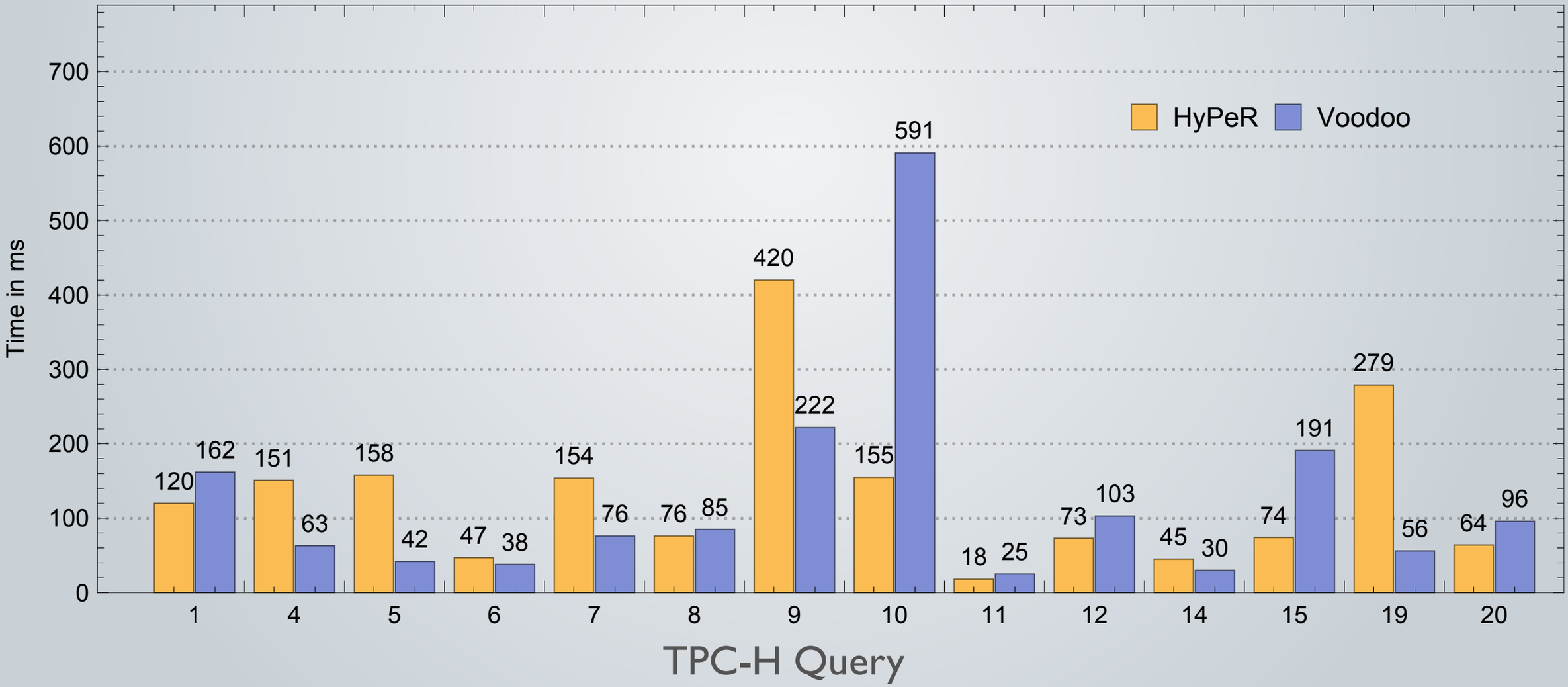
→ hierarchical aggregation

Voodoo outperforms MonetDB (TPC-H subset, SF10)



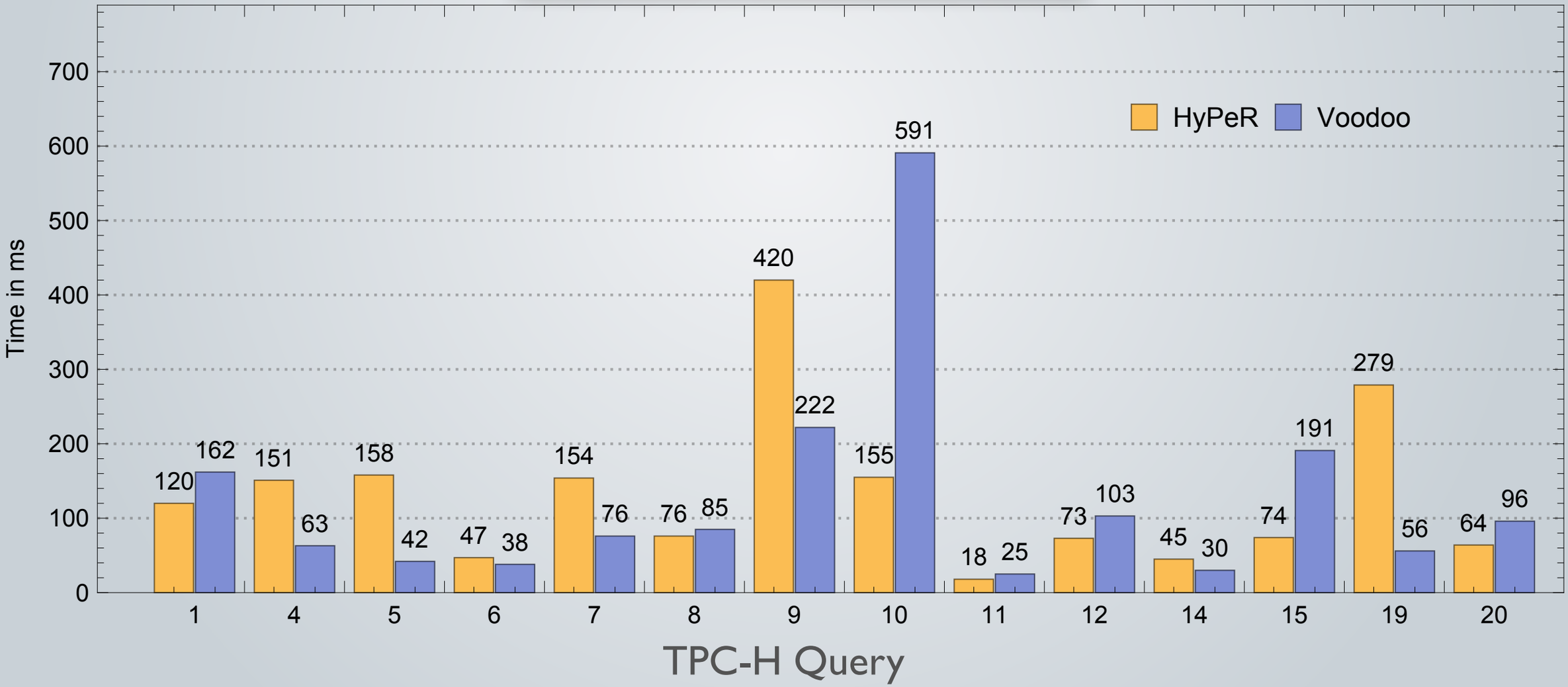
an old acquaintance

Voodoo is competitive with Hyper on CPU (selected TPC-H queries, SF10)

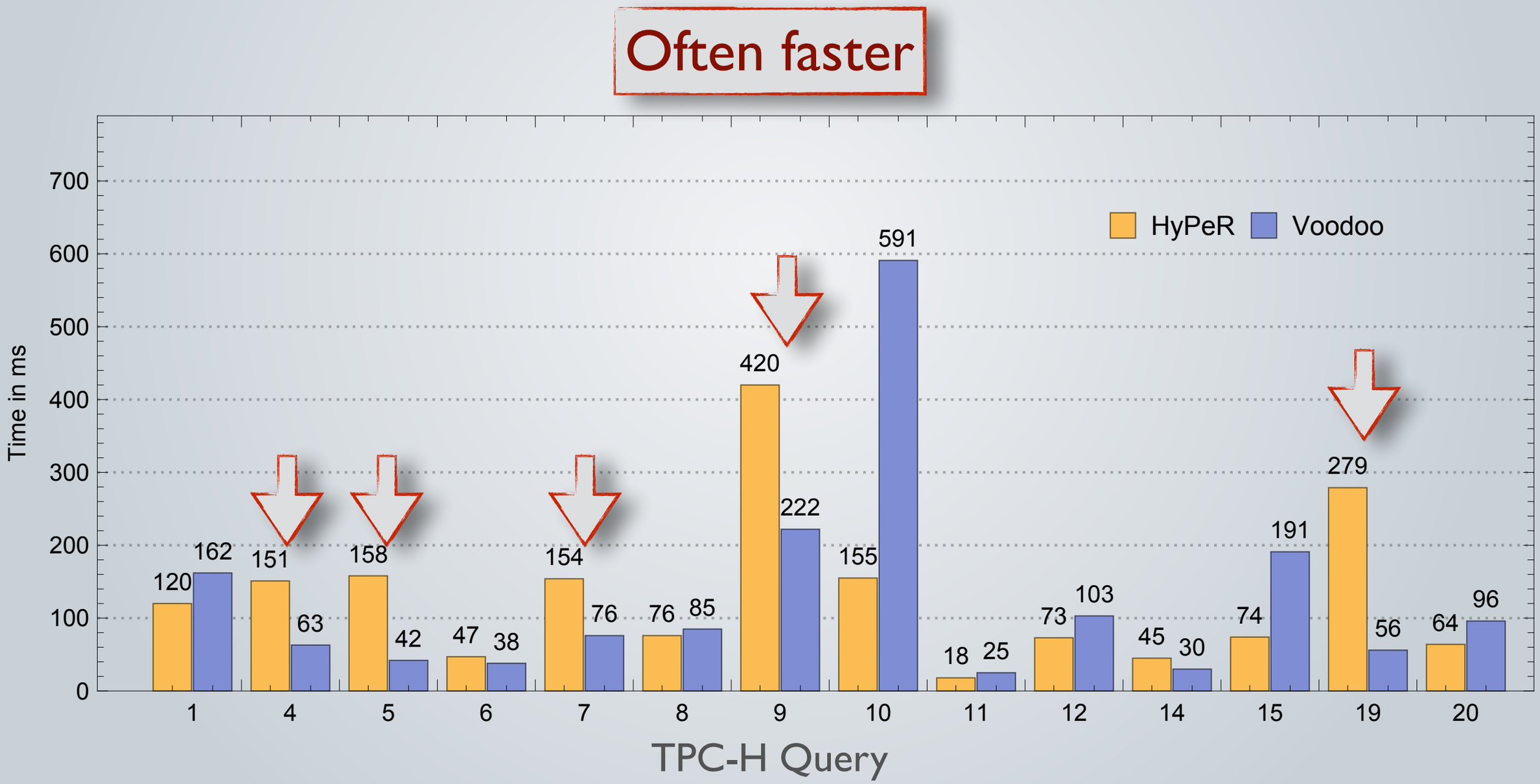


Voodoo is competitive with Hyper on CPU (selected TPC-H queries, SF10)

Generally Competitive

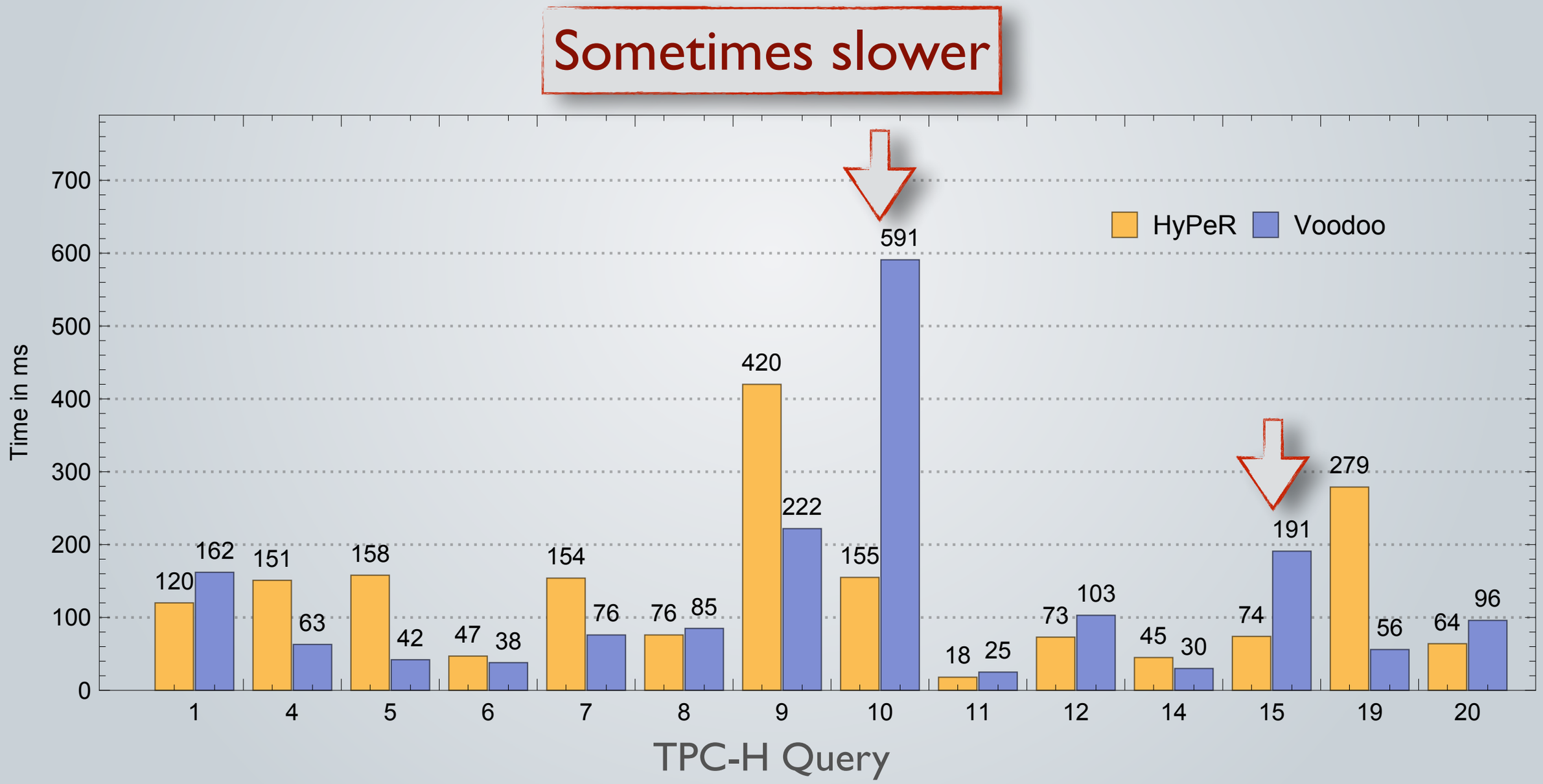


Voodoo is competitive with Hyper on CPU (selected TPC-H queries, SF10)



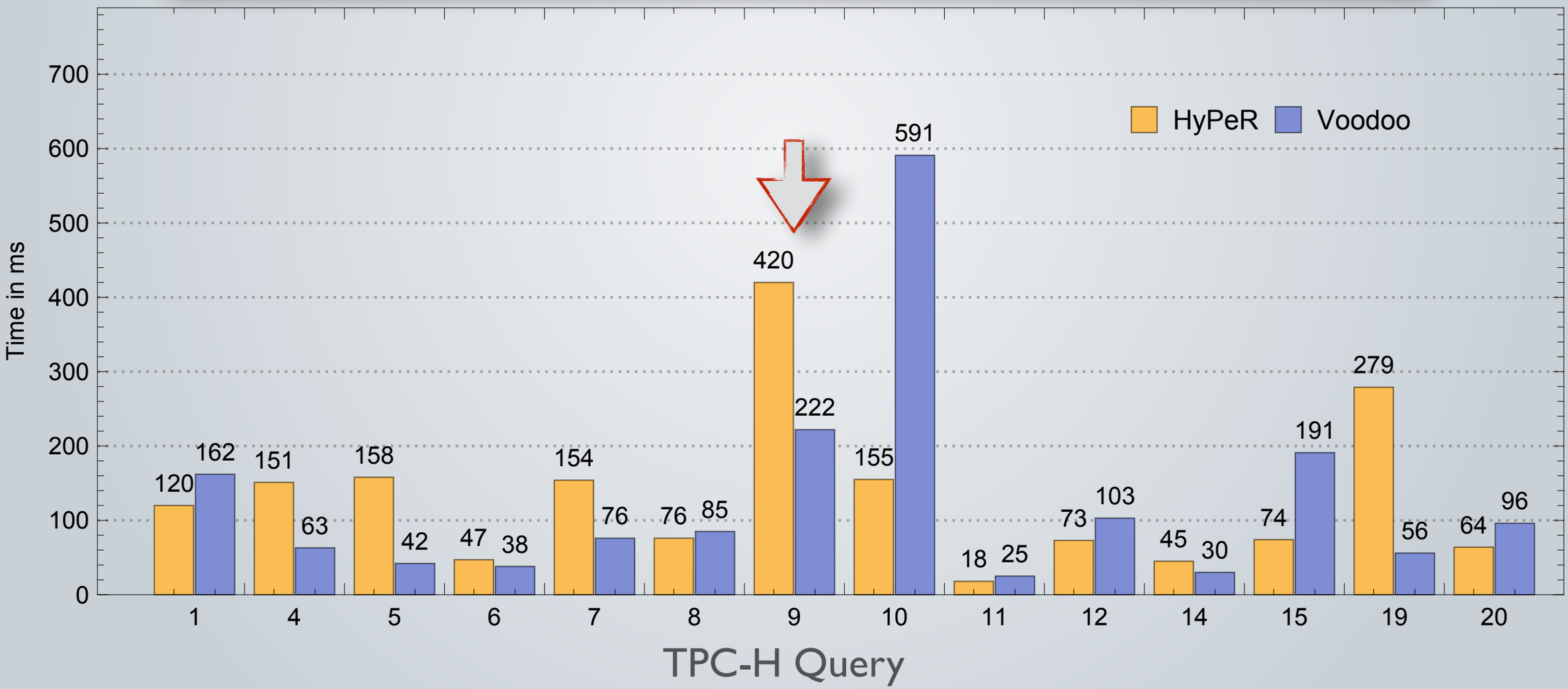
Imperial College London

Voodoo is competitive with Hyper on CPU (selected TPC-H queries, SF10)



Voodoo is competitive with Hyper on CPU (selected TPC-H queries, SF10)

Q9 is an interesting case where tuning is important



VOODOO AS A TUNING TOOL

Selective Foreign-Key Joins

join

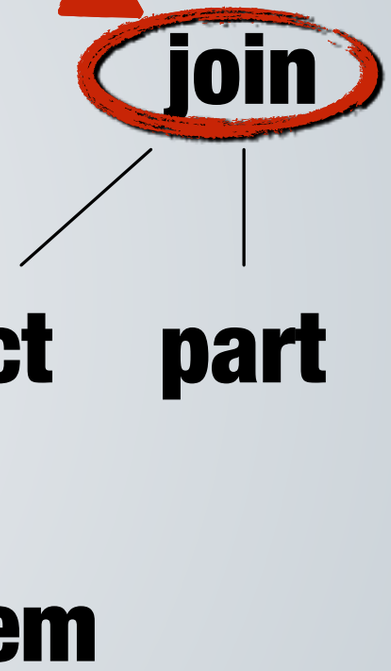
```
select sum(part.price)
from lineitem, part
where discount > $x
and lineitem.partkey_fk = part.partkey_pk
```

select items that were sold at a discount greater than x
look up their price from the parts table
and sum it

Selective Foreign-Key Joins

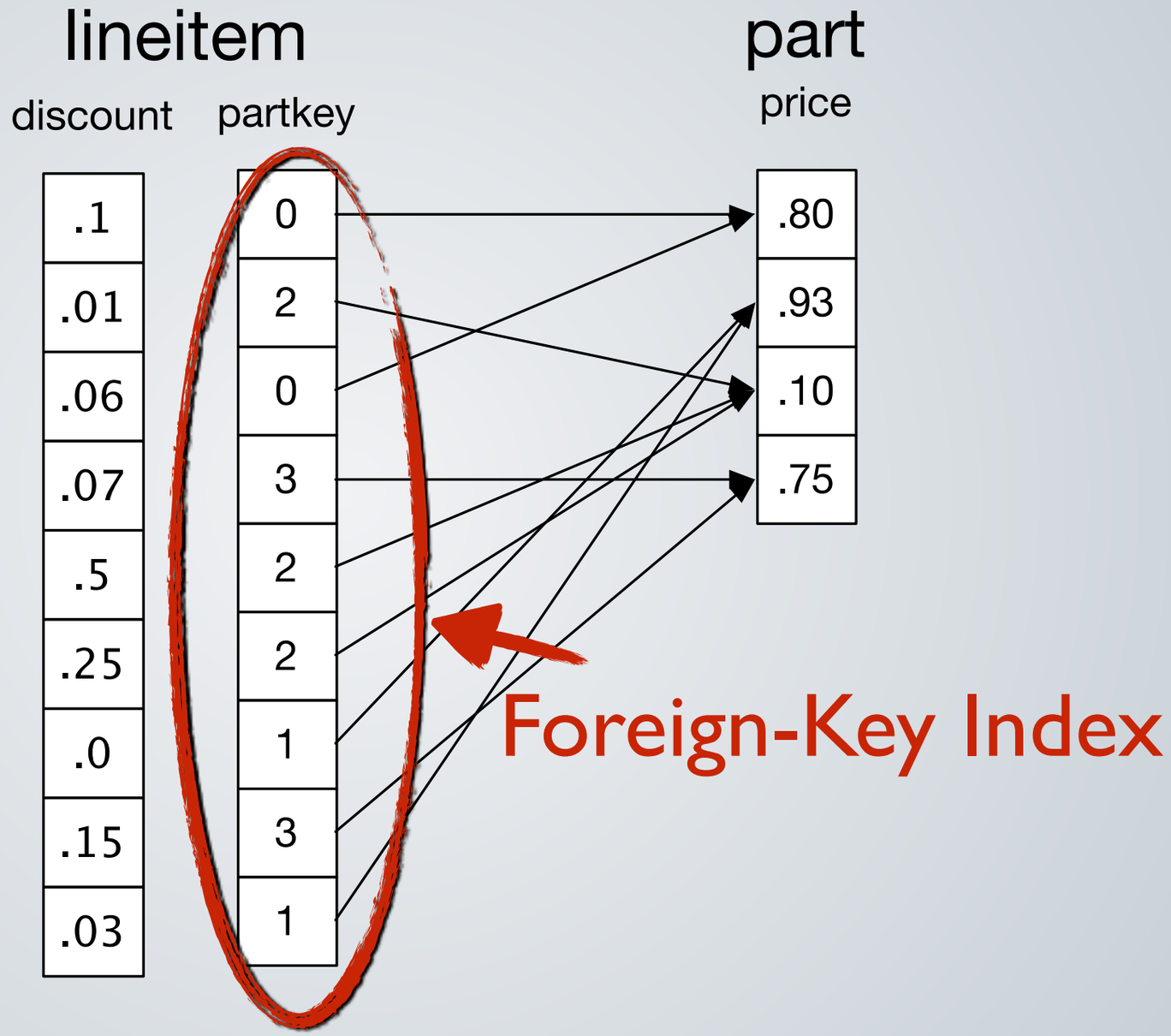
lookup/Foreign-Key Join

```
select sum(part.price)
from lineitem, part
where discount > $x
and lineitem.partkey_fk
  = part.partkey_pk
```



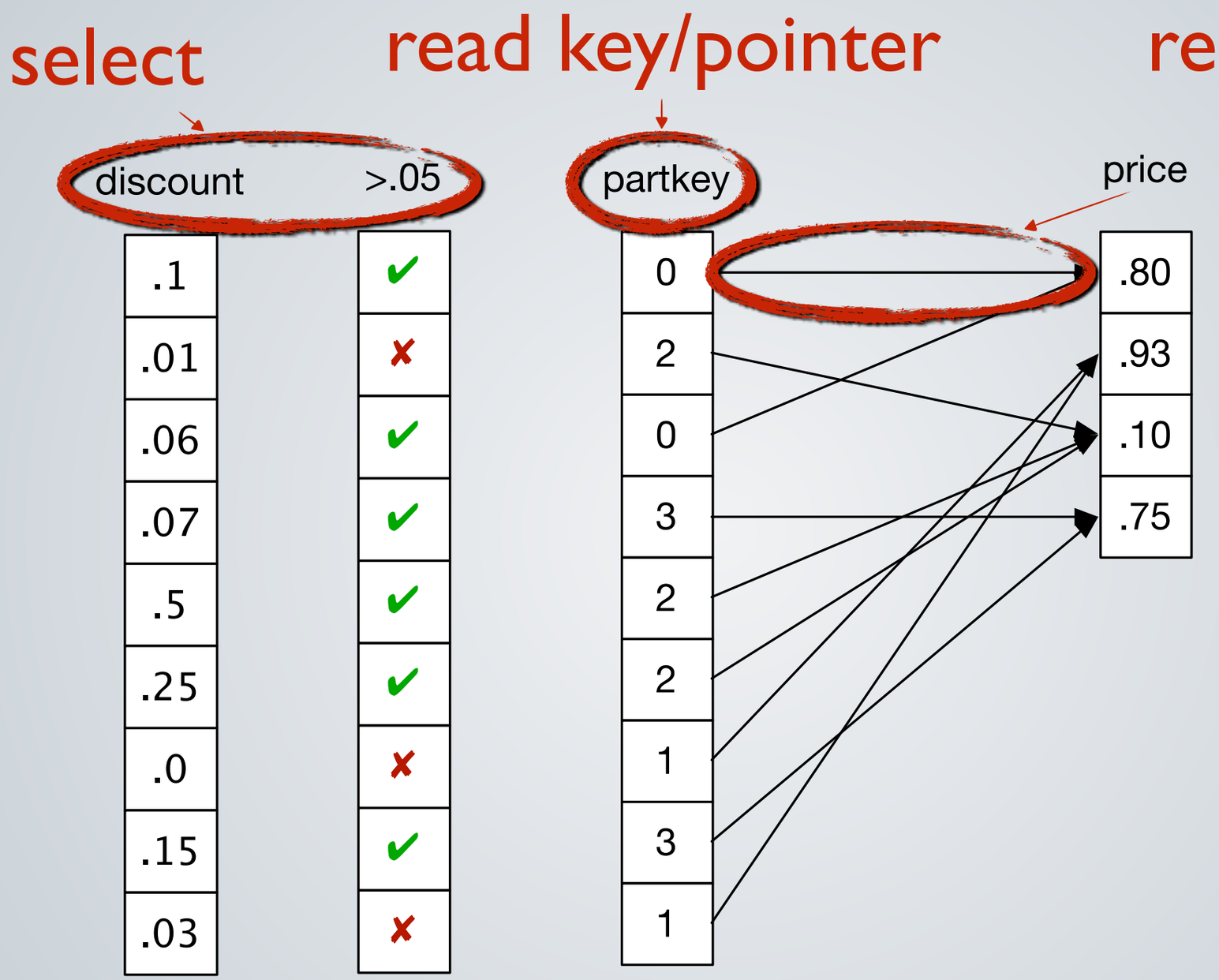
primary key, i.e., unique

Selective Foreign-Key Joins



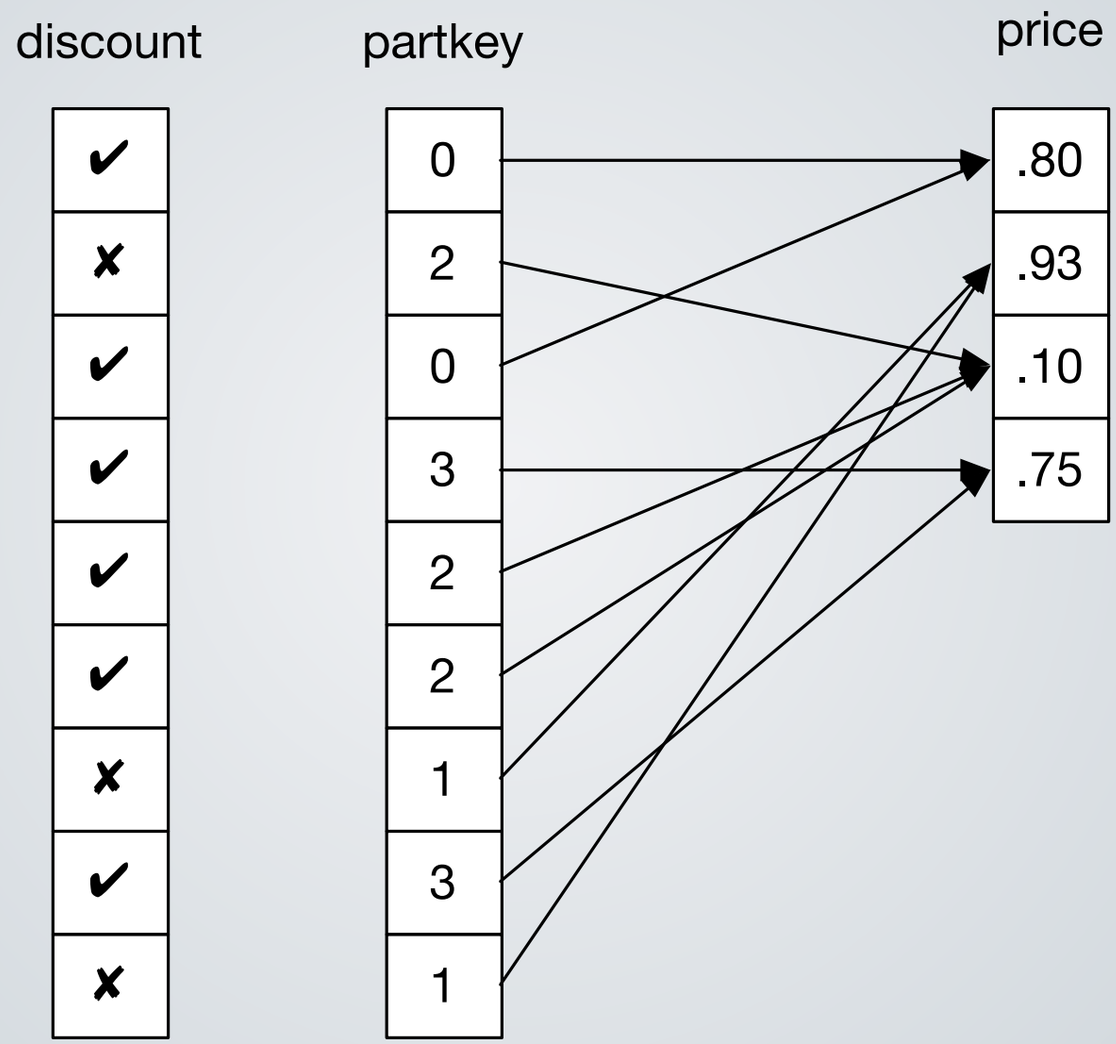

```
select sum(price)
from lineitem, part
where discount > $1
and lineitem.partkey = part.partkey
```

Selective Foreign-Key Joins



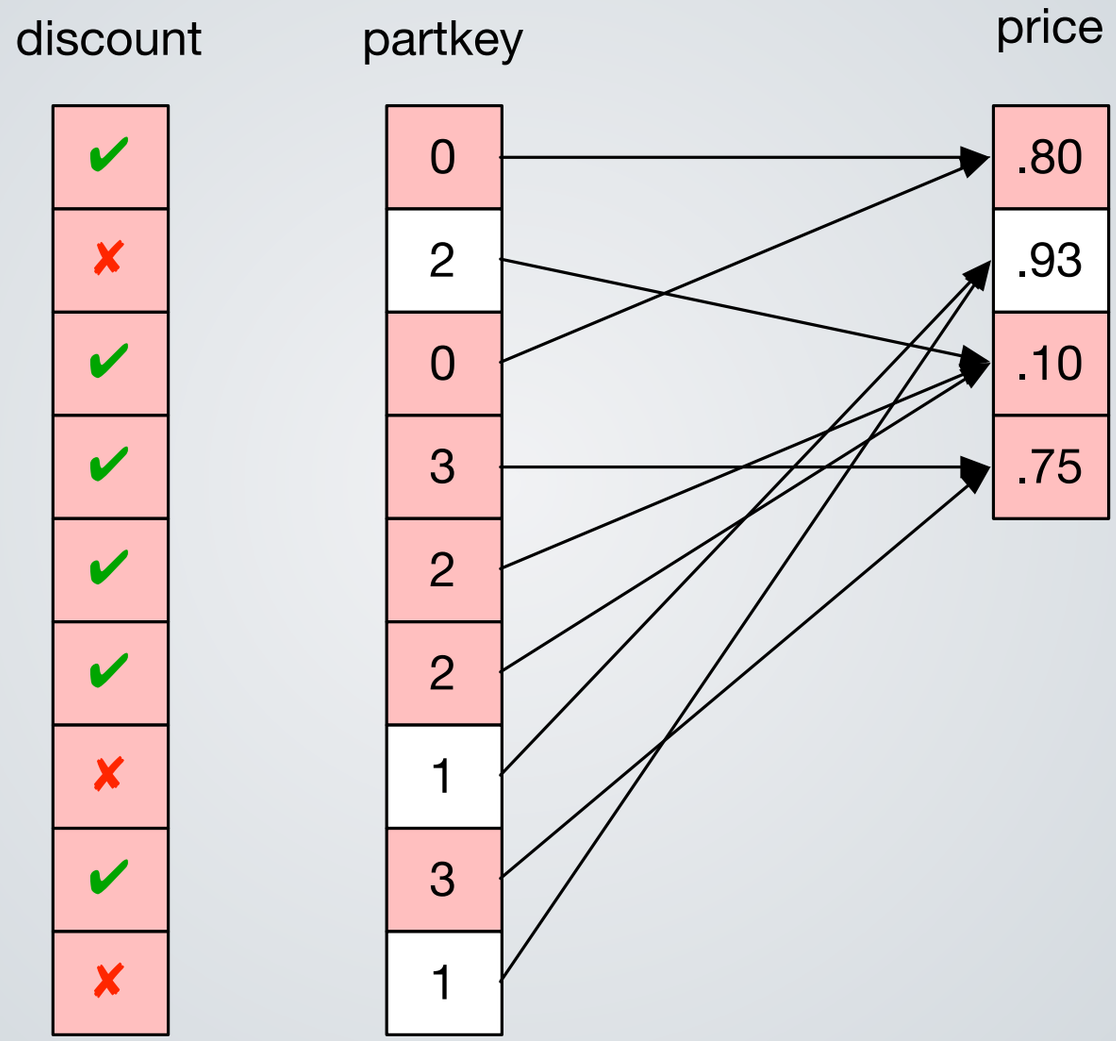

```
select sum(price)
from lineitem, part
where discount > $1
and lineitem.partkey = part.partkey
```

Selective Foreign-Key Joins




```
select sum(price)
from lineitem, part
where discount > $1
and lineitem.partkey = part.partkey
```

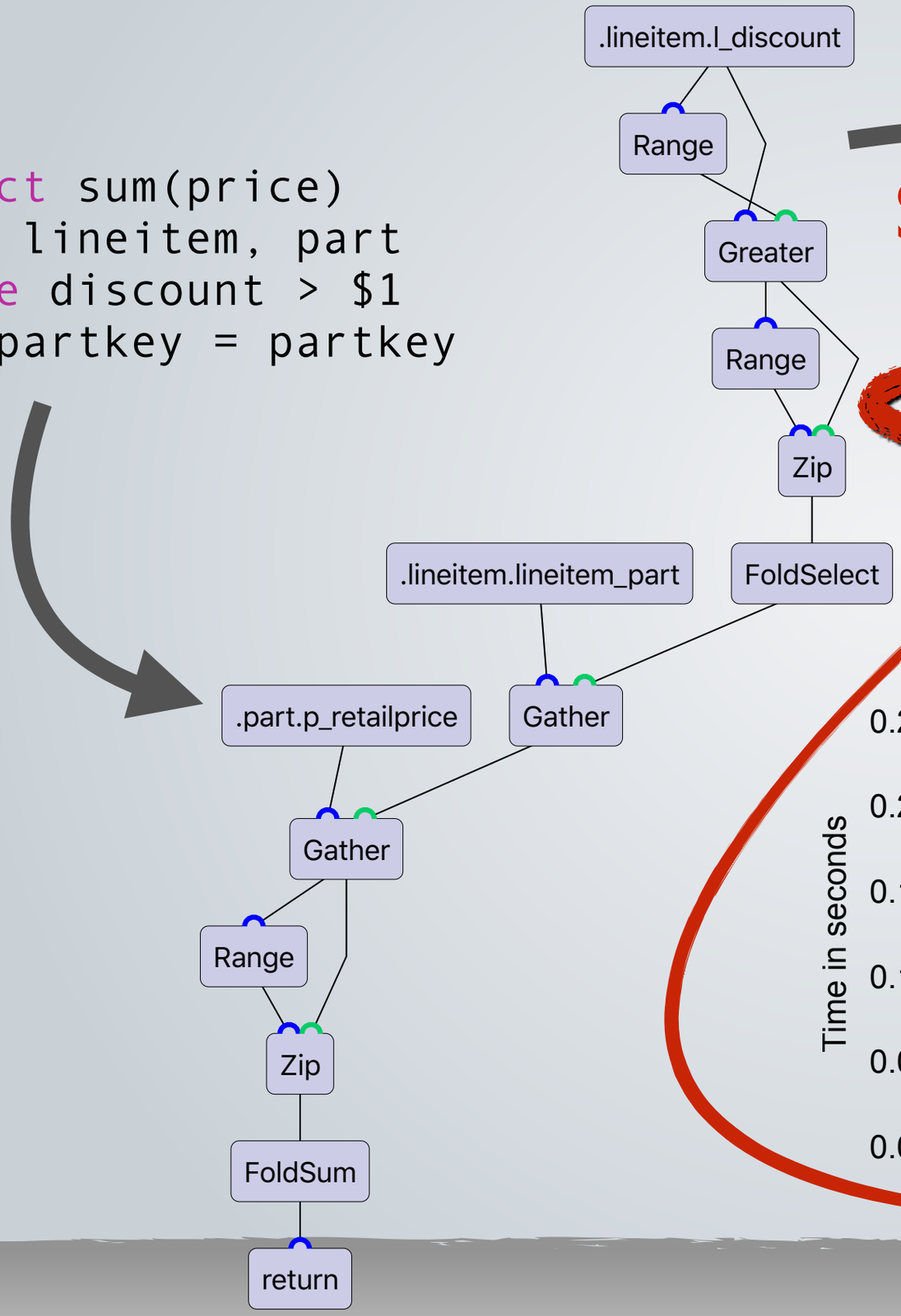
Selective Foreign-Key Joins




```
select sum(price)
from lineitem, part
where discount > $1
and lineitem.partkey = part.partkey
```

Branching Foreign-Key Joins

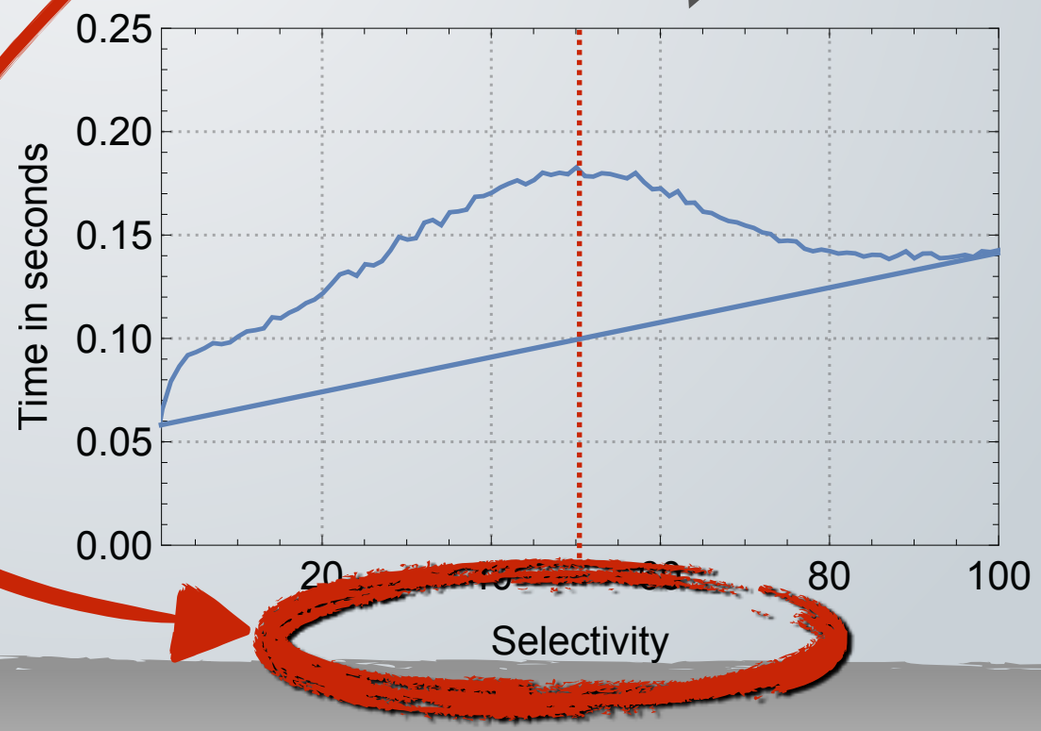
```
select sum(price)
from lineitem, part
where discount > $1
and partkey = partkey
```



Selection

Key-Lookup

```
for(size_t i = 0; i < lineitemsize; i++)
  if(discount[i] > $1
    [partkey[i]];
```



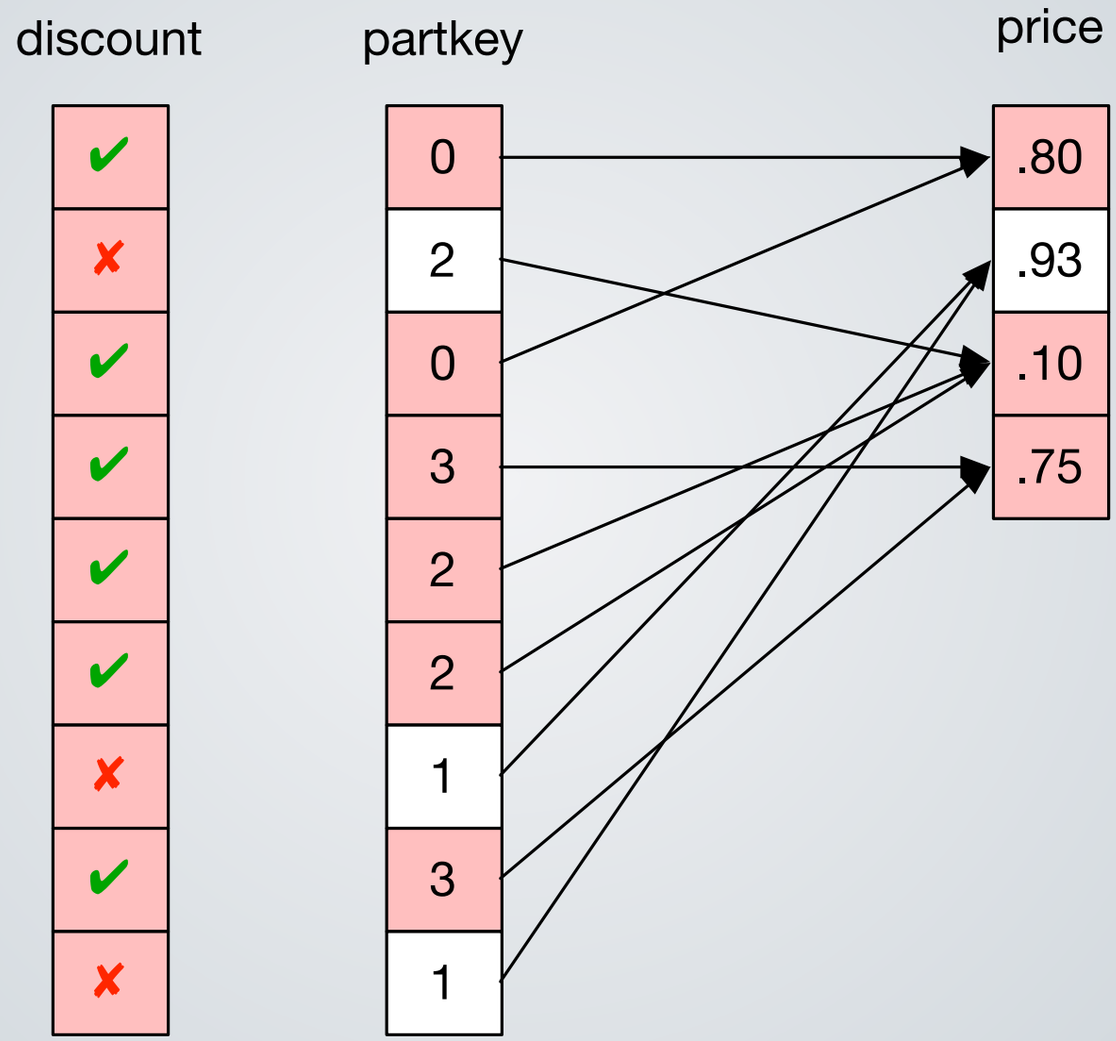
Predicated Foreign-Key Joins

```
for(size_t i = 0; i < lineitemsize; i++)
  if(discount[i] > $1)
    result += price[partkey[i]];
```

Predicated Foreign-Key Joins

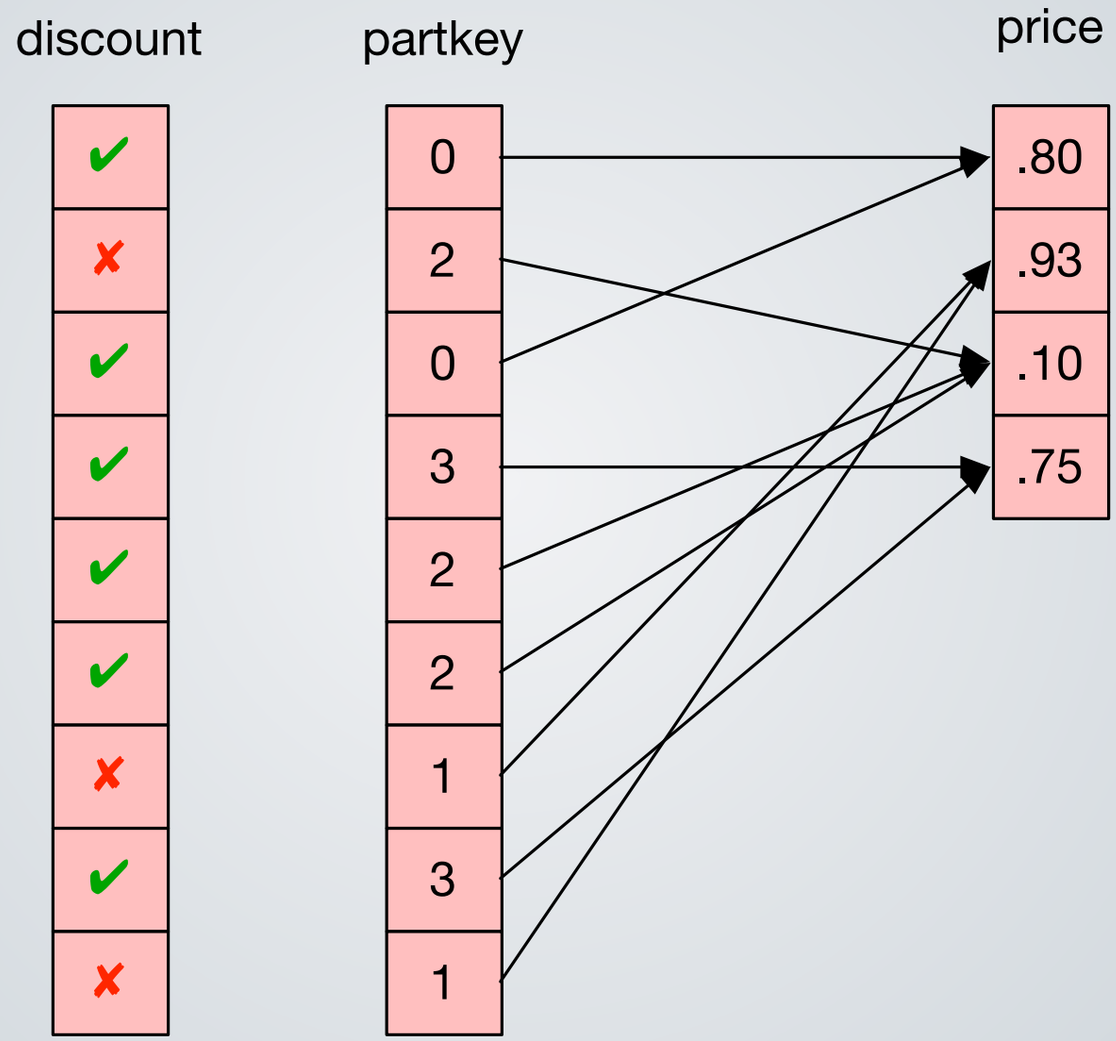
```
for(size_t i = 0; i < lineitemsize; i++)
    result += (discount[i] > $1) *
              price[partkey[i]];
```

Selective Foreign-Key Joins



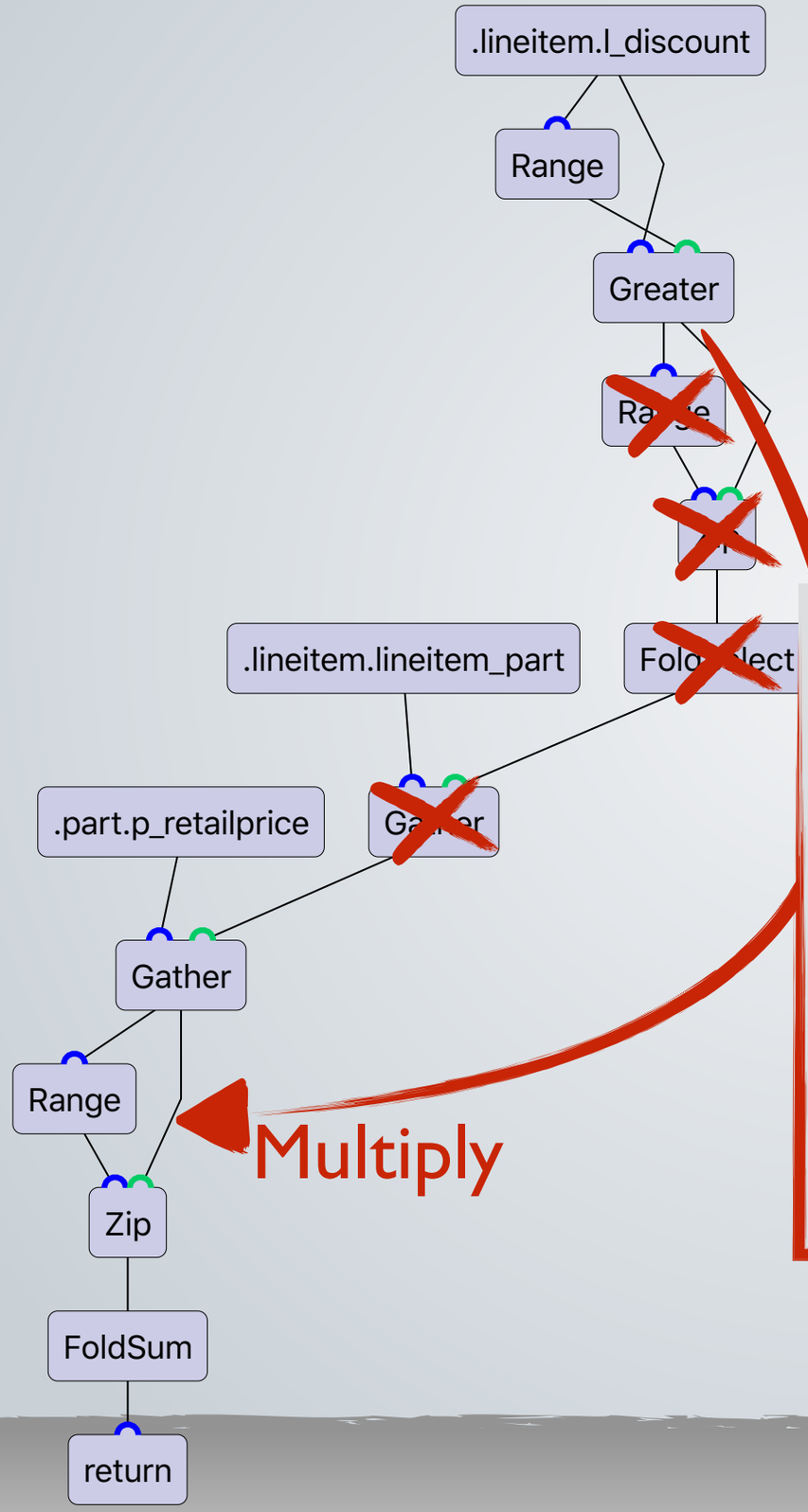

```
select sum(price)
from lineitem, part
where discount > $1
and lineitem.partkey = part.partkey
```

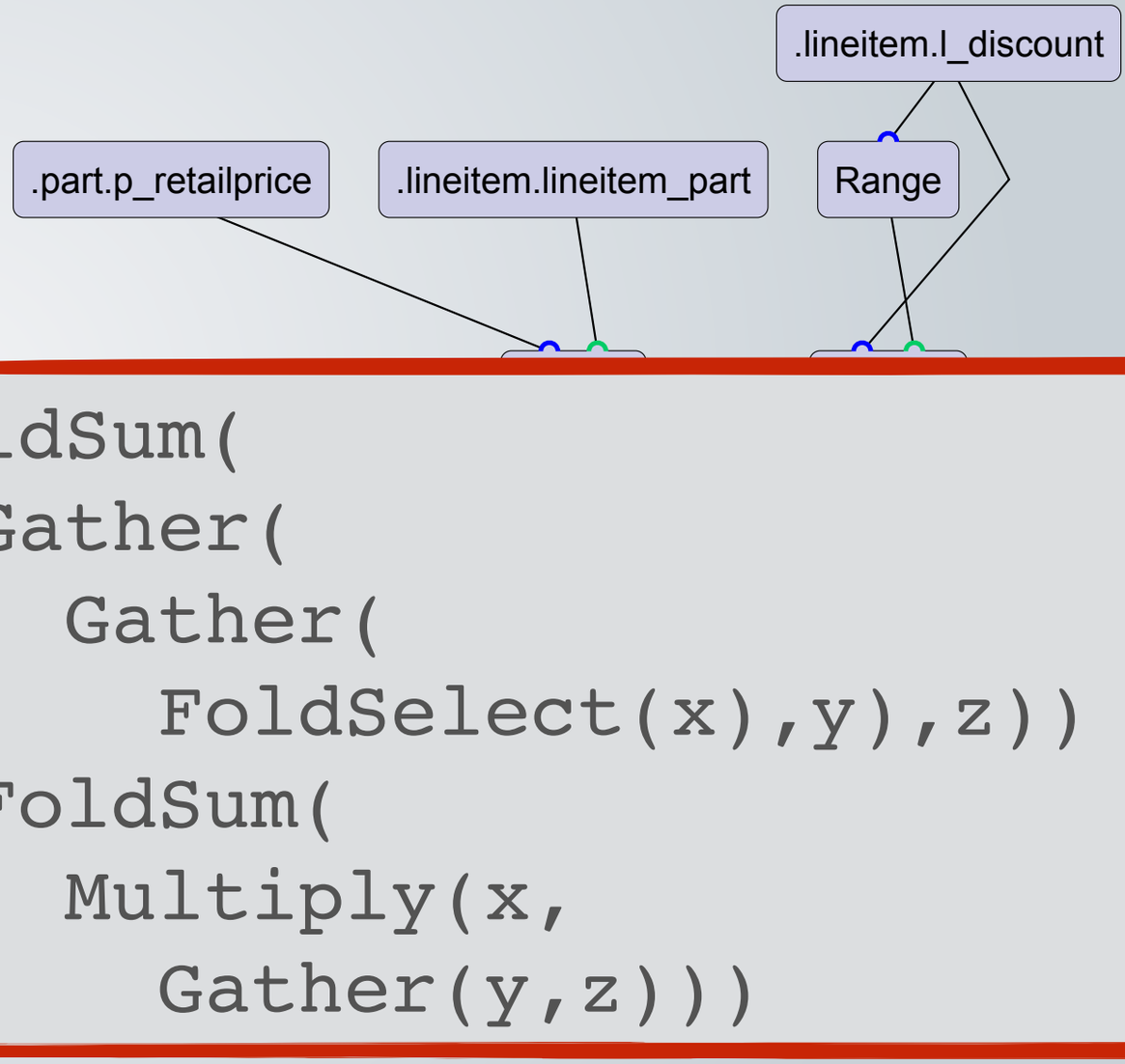
Predicated Foreign-Key Joins




```
select sum(price)
from lineitem, part
where discount > $1
and lineitem.partkey = part.partkey
```

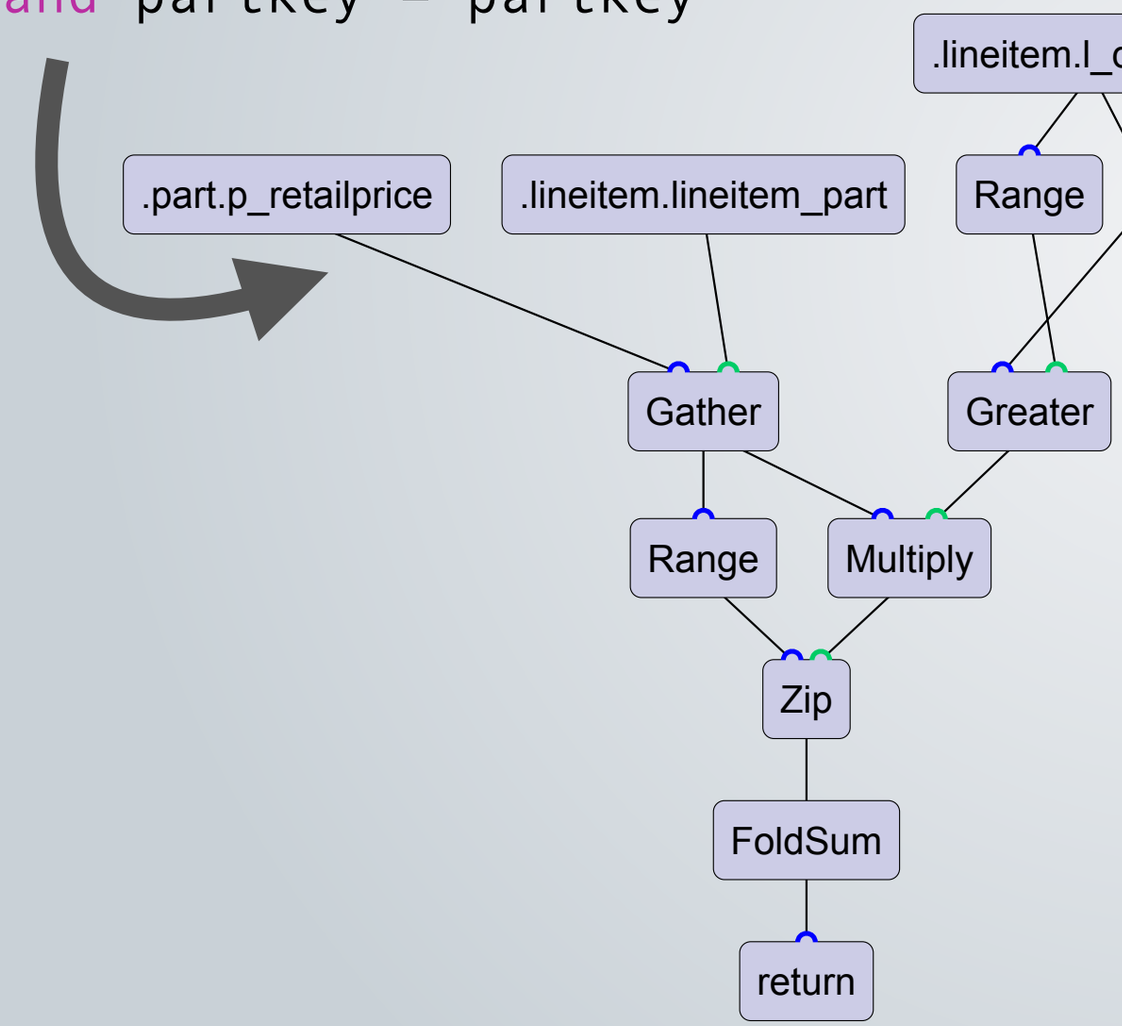
Predicated Foreign-Key Joins





Selective Foreign-Key Joins

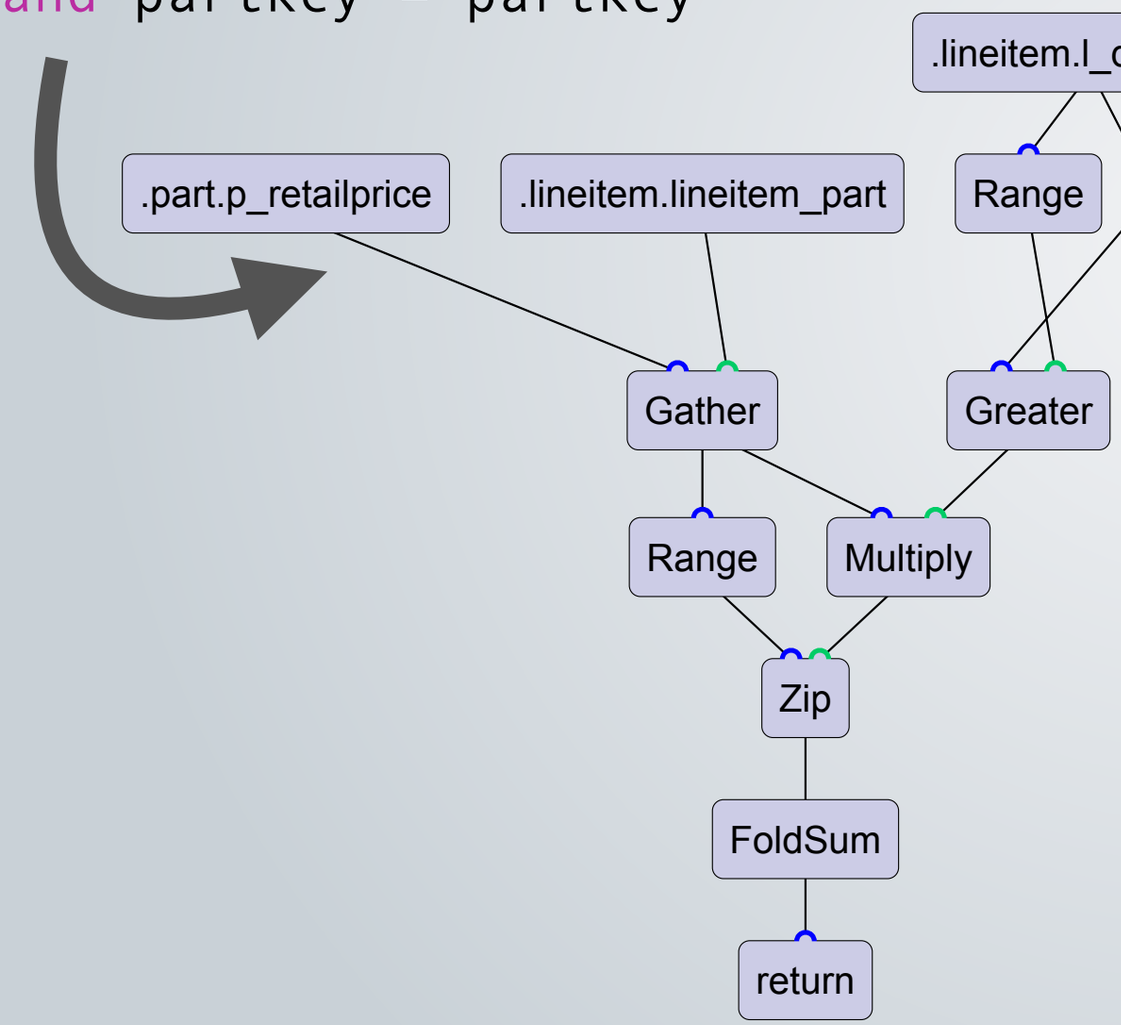
```
select sum(price)
from lineitem, part
where discount > $1
and partkey = partkey
```



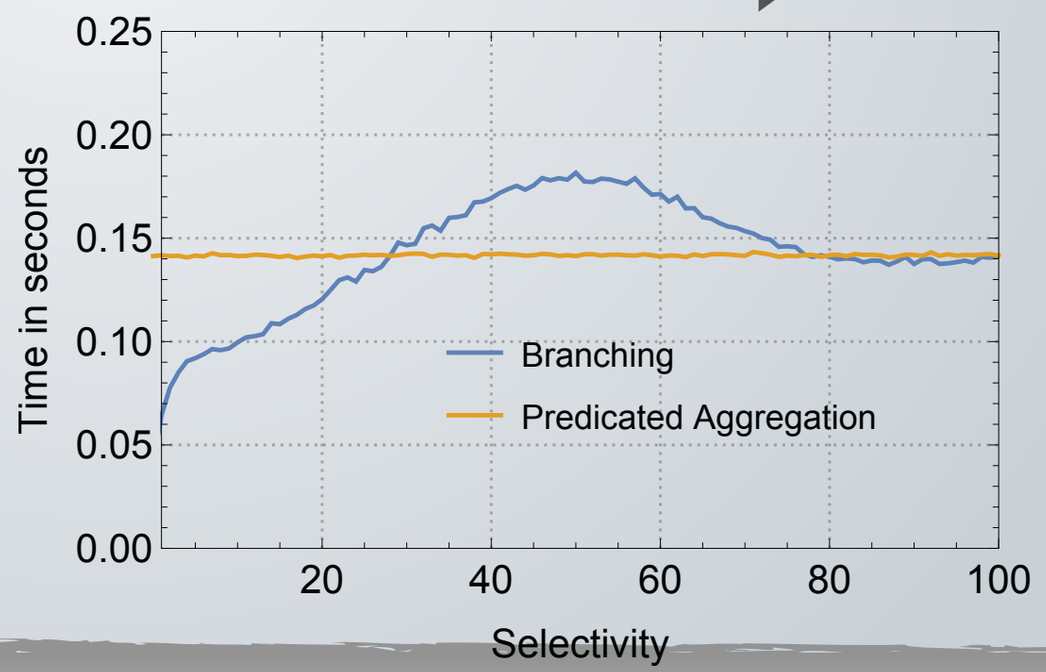

```
for(size_t i = 0; i < lineitemsize; i++)
  result += (price[i] > $1) *
            p_discount[partkey[i]];
```

Selective Foreign-Key Joins

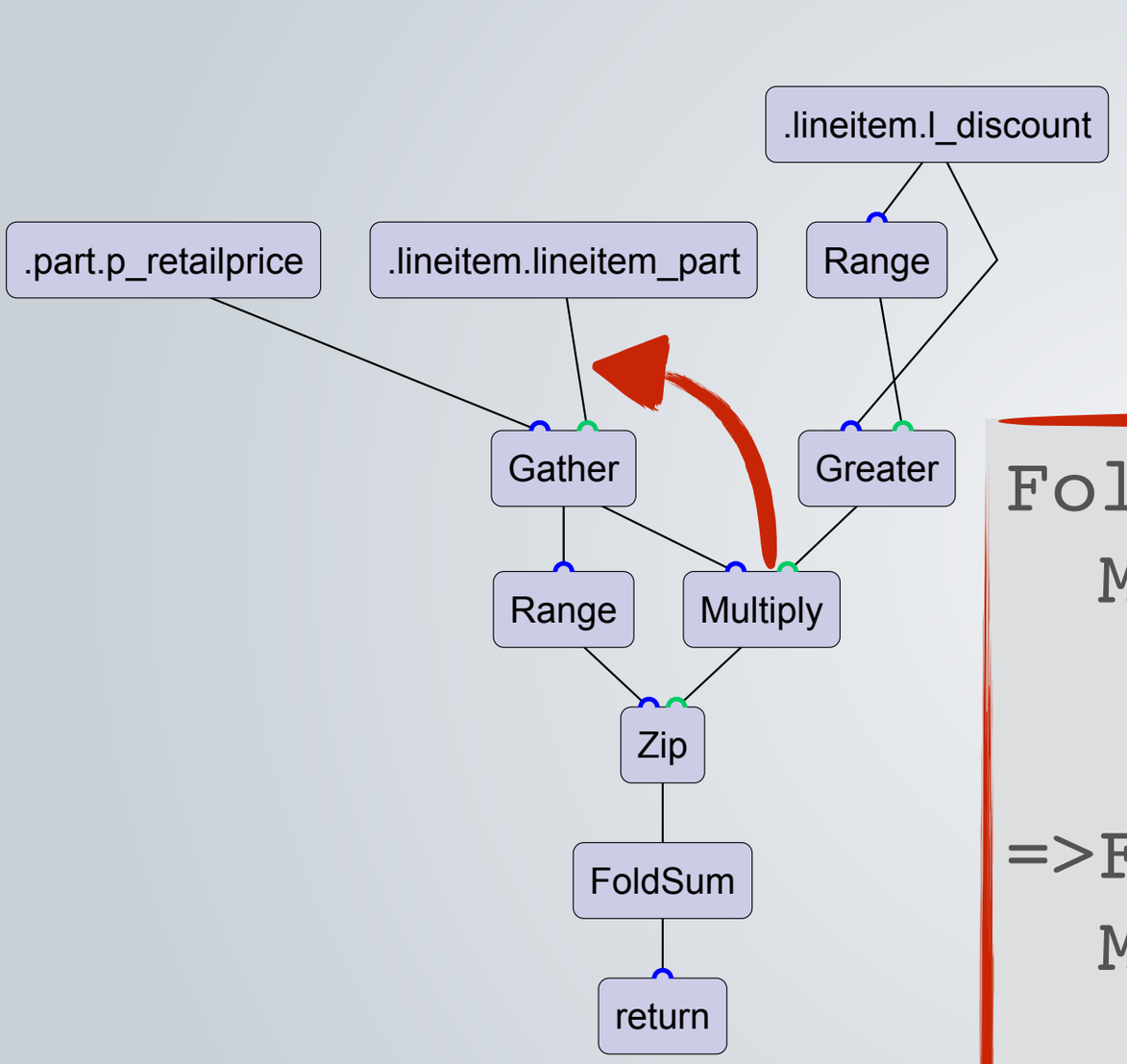
```
select sum(price)
from lineitem, part
where discount > $1
and partkey = partkey
```

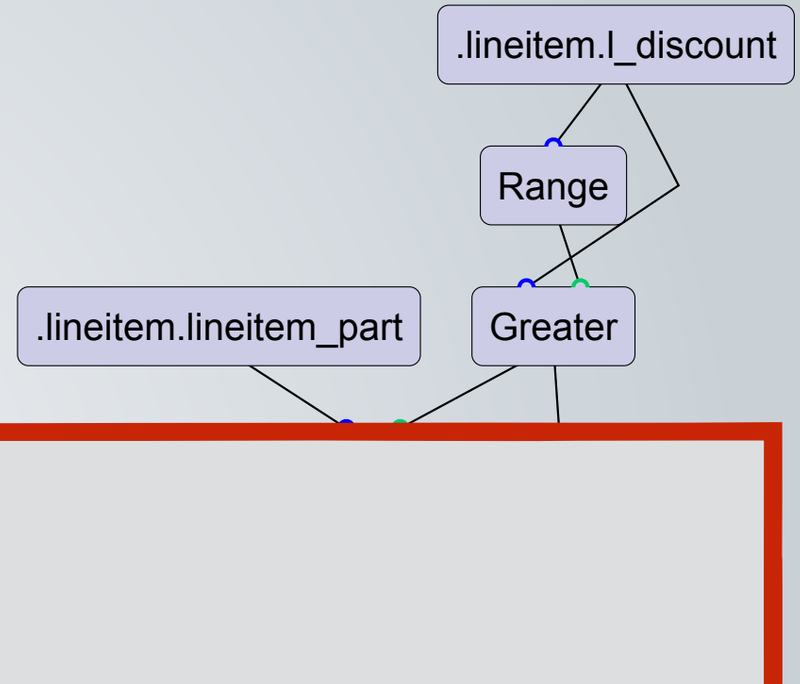



```
for(size_t i = 0; i < lineitemsize; i++)
  result += (price[i] > $1) *
            p_discount[partkey[i]];
```



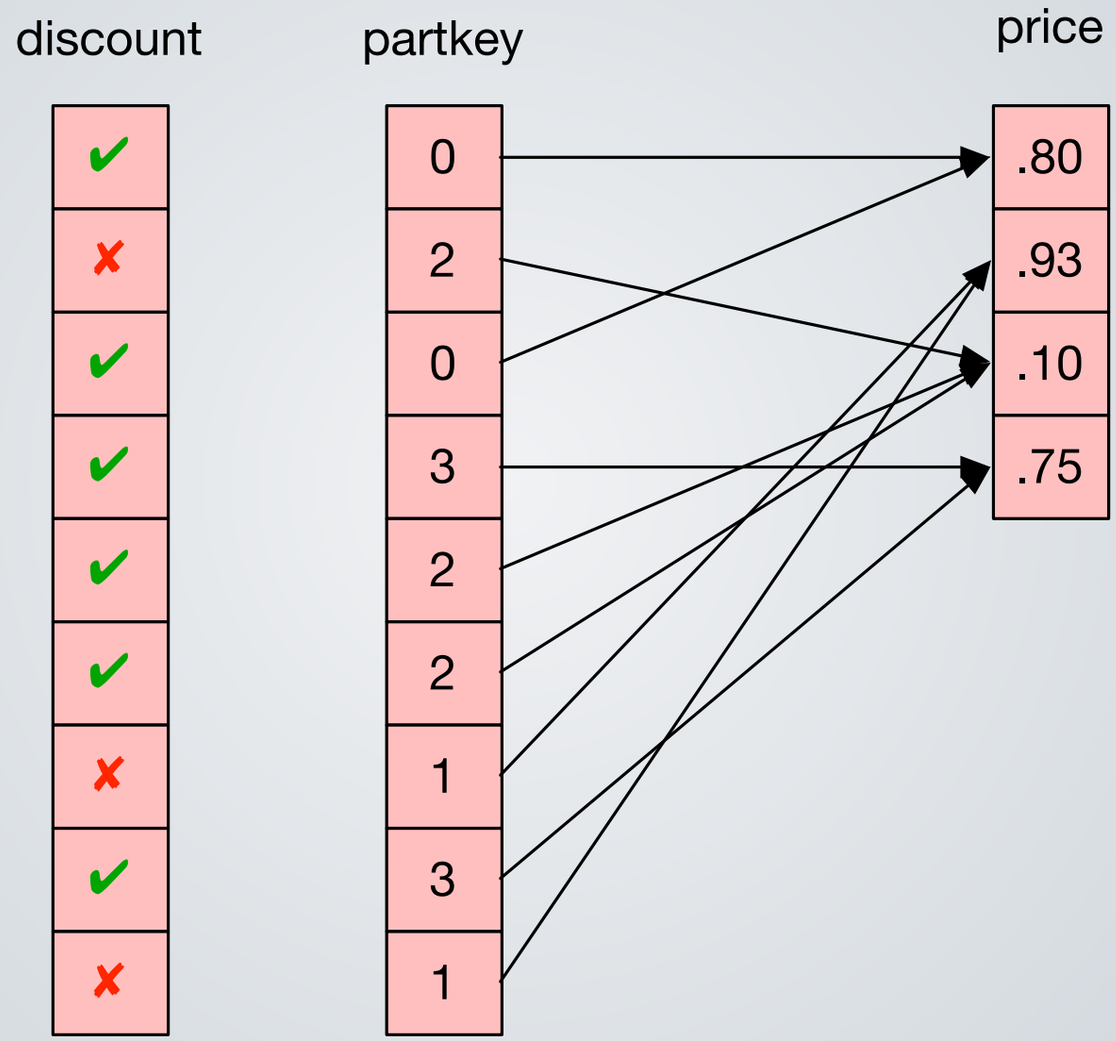
Double-predicated Foreign-Key Joins





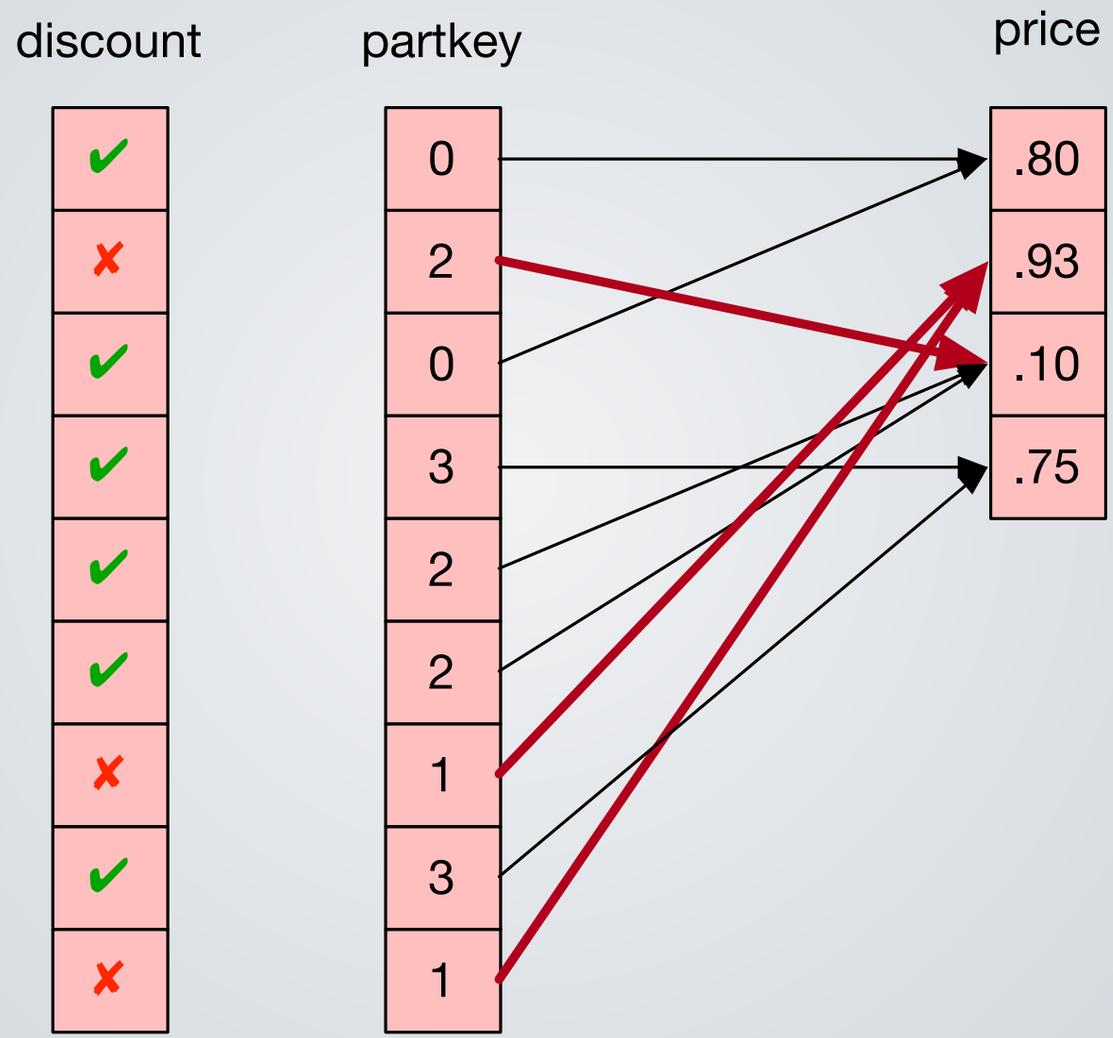

```
FoldSum(
  Multiply(
    Gather(
      x,y),z))
=>FoldSum(
  Multiply(
    Gather(
      Multiply(x,z),y),z))
```

Predicated Foreign-Key Joins



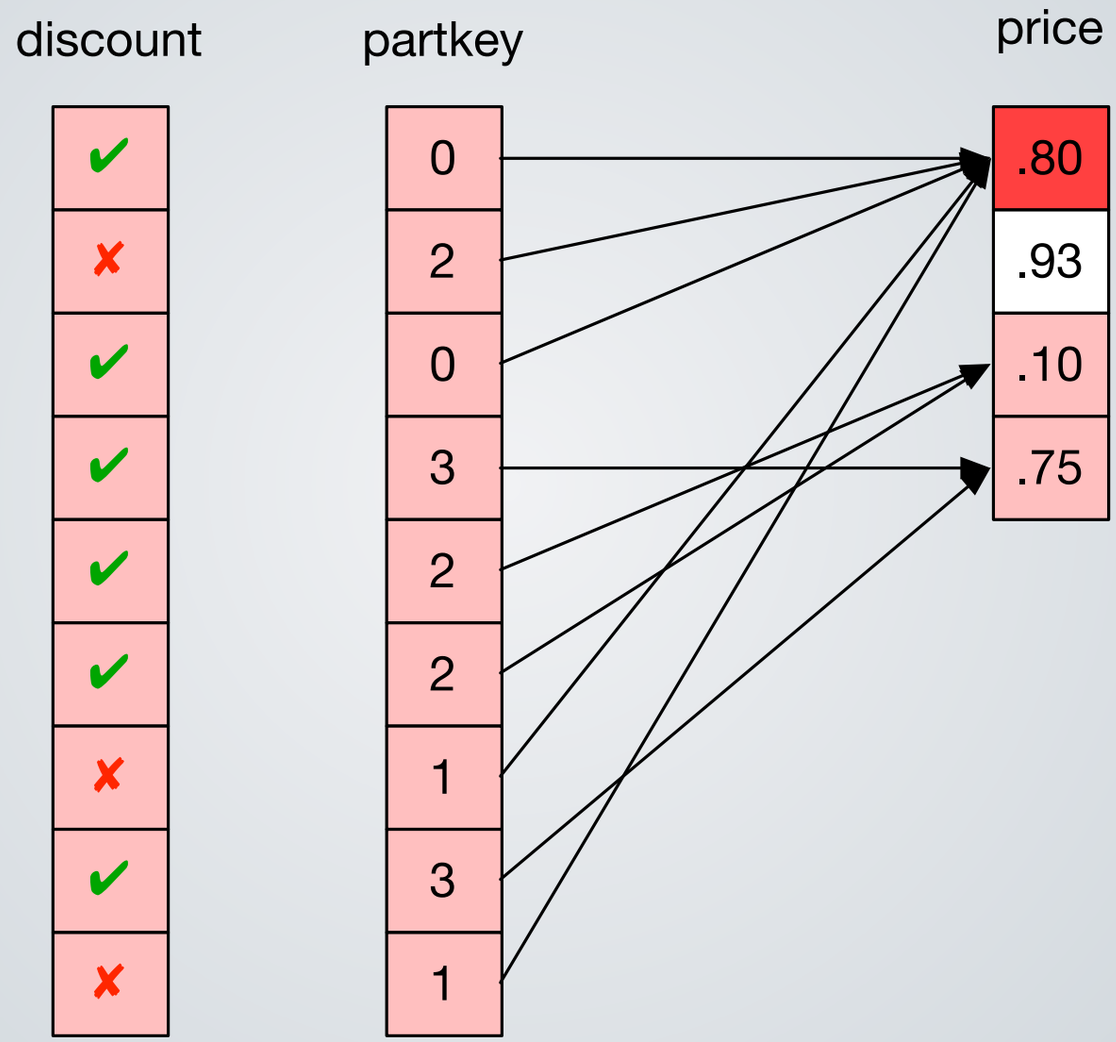

```
select sum(price)
from lineitem, part
where discount > $1
and lineitem.partkey = part.partkey
```

Predicated Foreign-Key Joins




```
select sum(price)
from lineitem, part
where discount > $1
and lineitem.partkey = part.partkey
```

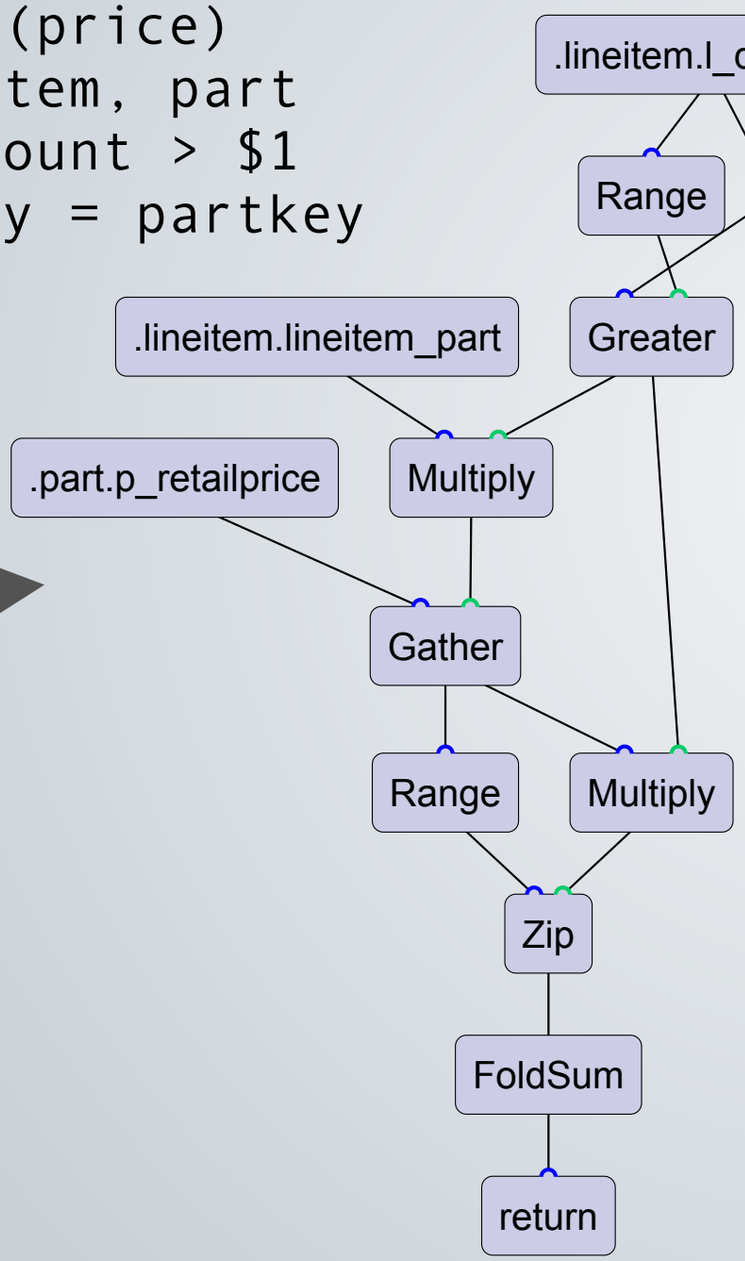
Double-predicated Foreign-Key Joins




```
select sum(price)
from lineitem, part
where discount > $1
and lineitem.partkey = part.partkey
```

Double-predicated Foreign-Key Joins

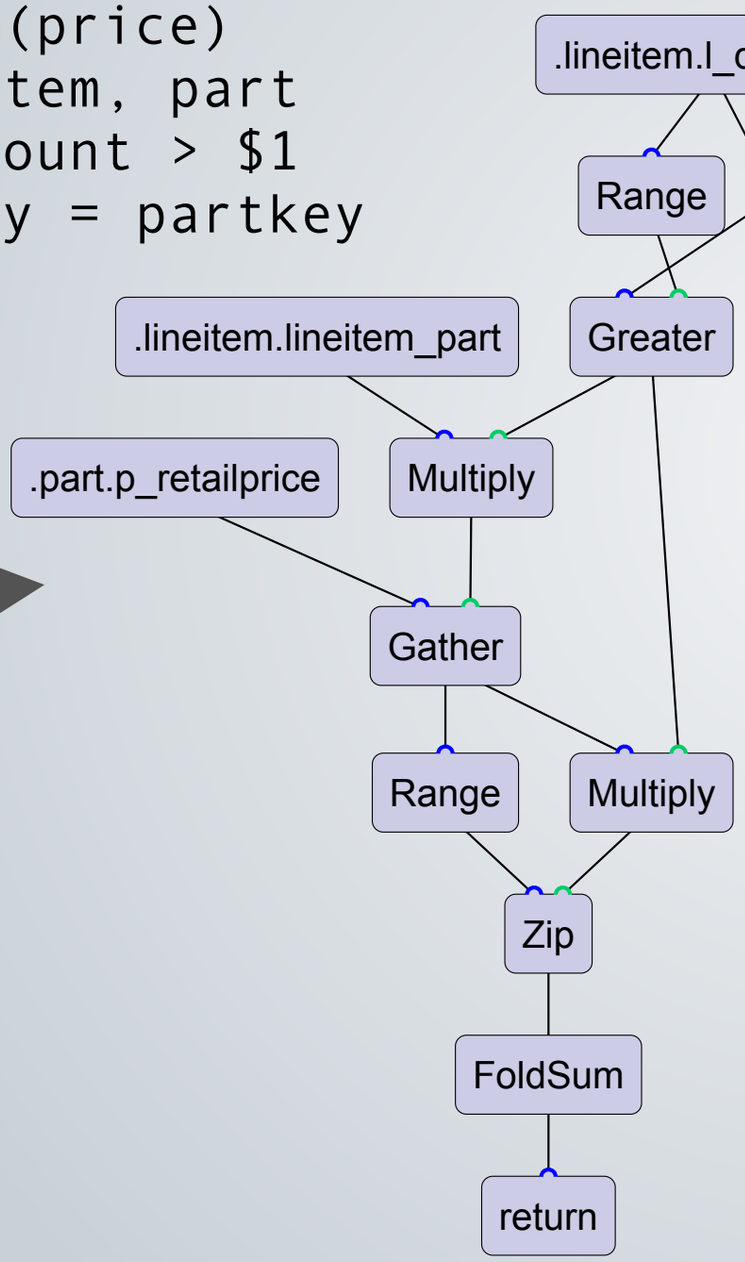
```
select sum(price)
from lineitem, part
where discount > $1
and partkey = partkey
```



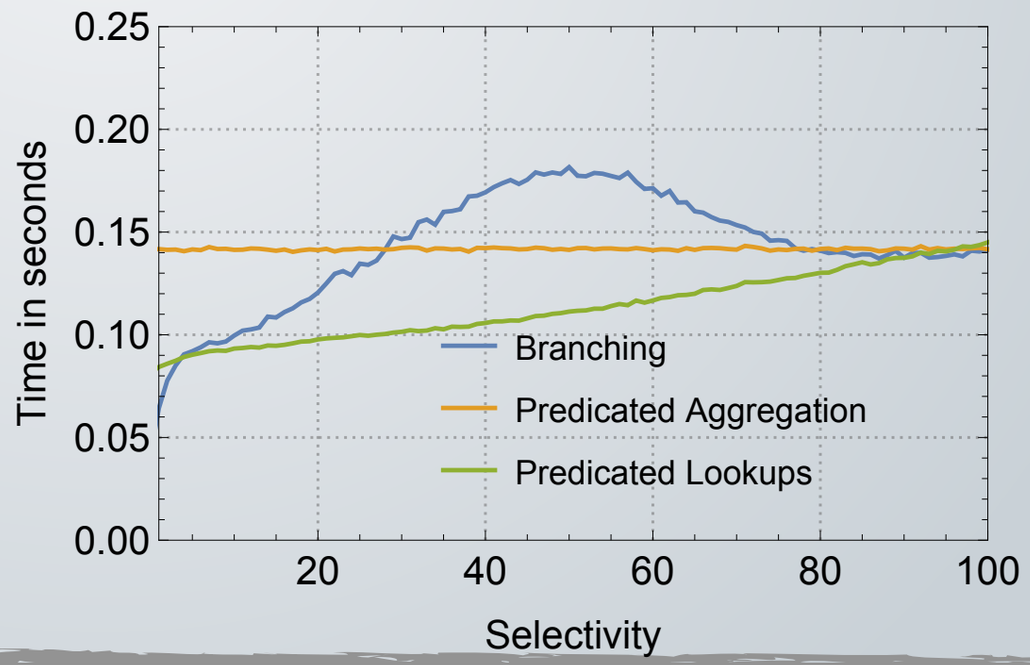

```
for(size_t i = 0; i < lineitemsize; i++)
  result += (discount[i] > $1) *
                price[partkey[i]];
```

Double-predicated Foreign-Key Joins

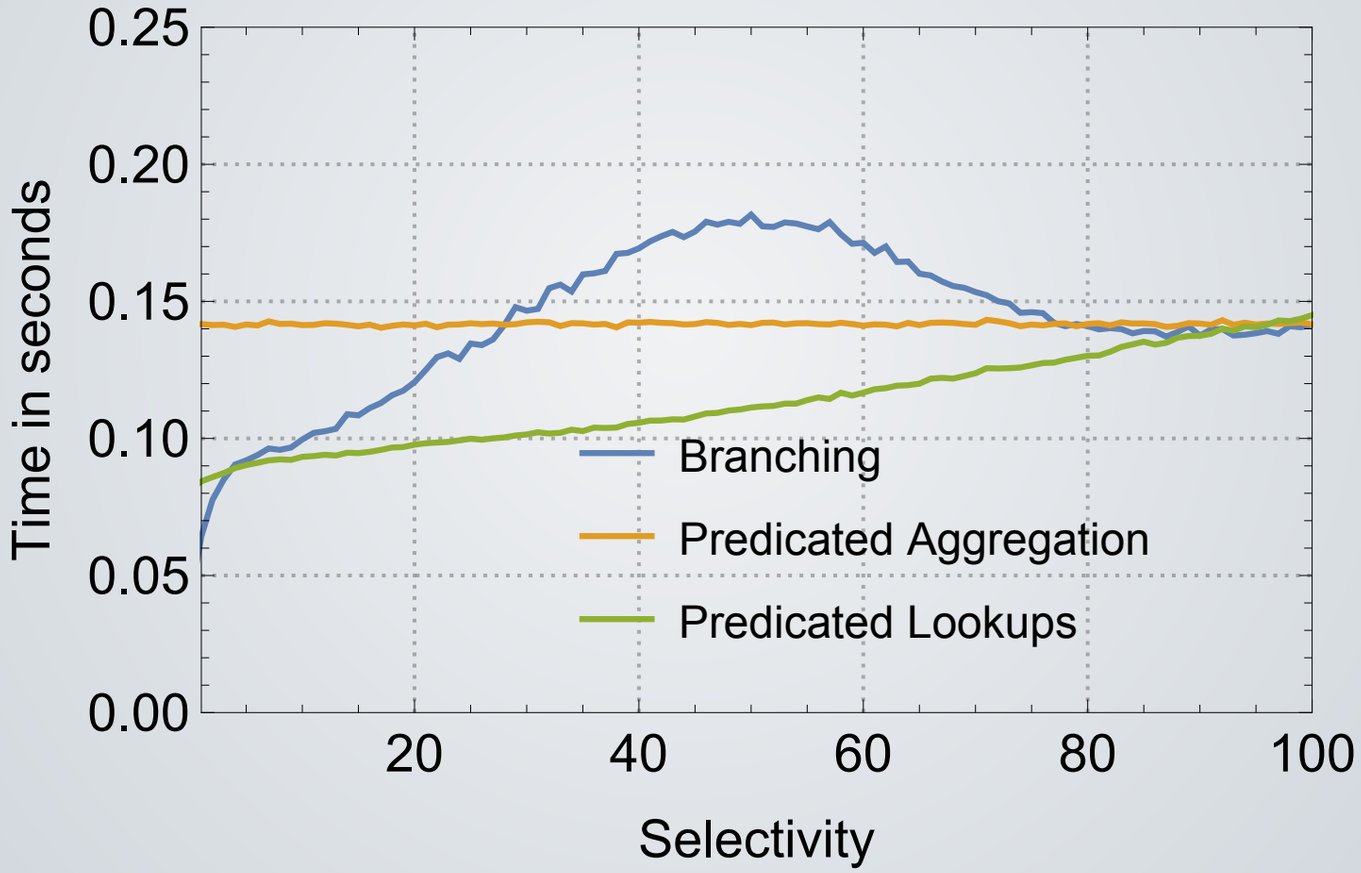
```
select sum(price)
from lineitem, part
where discount > $1
and partkey = partkey
```




```
for(size_t i = 0; i < lineitemsize; i++)
  result += (discount[i] > $1) *
            price[partkey[i] *
                  (discount[i] > $1)];
```

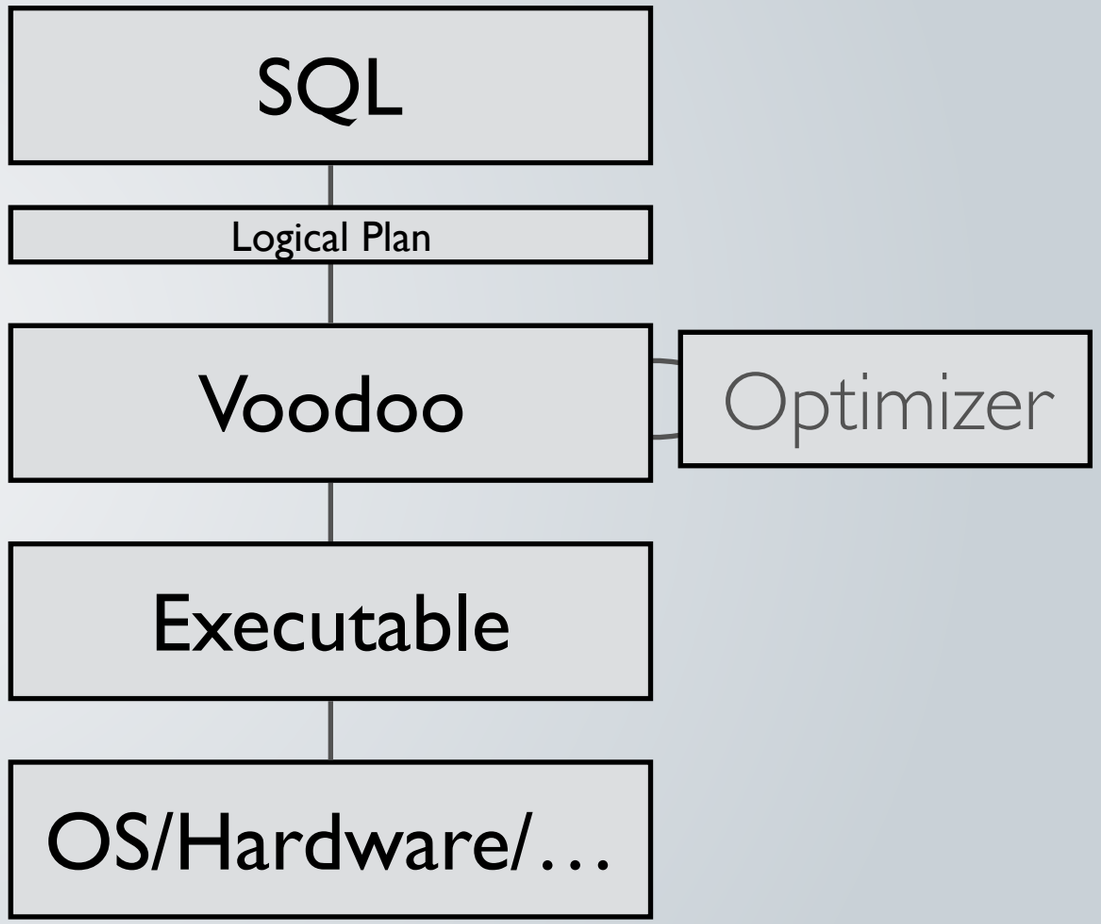


Double-predicated Foreign-Key Joins



Voodoo Wrap

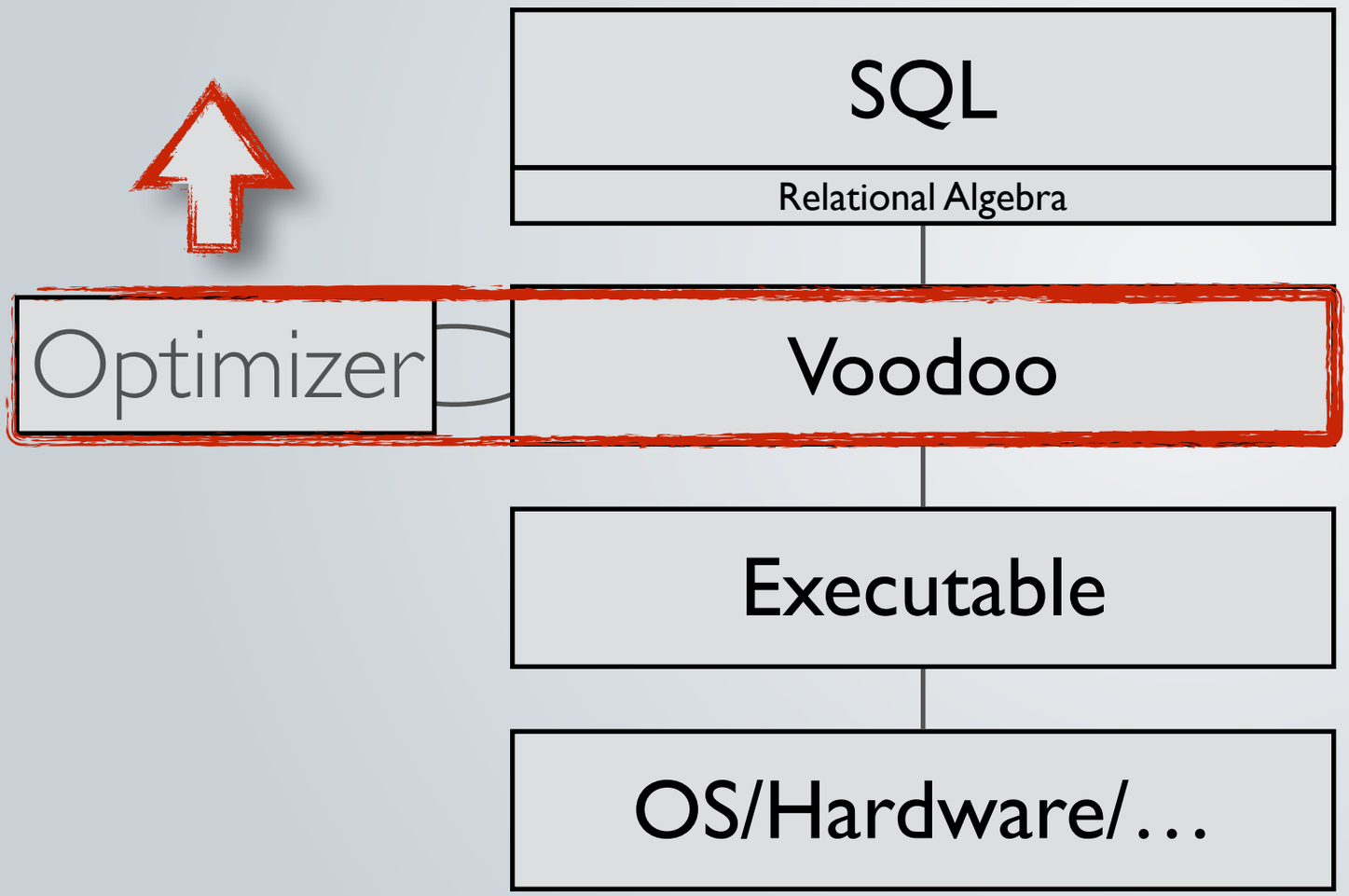
- Fast and expressive like C
- Optimizable like relational algebra
- Portable to different devices
- Enables serendipitous discovery of new optimizations



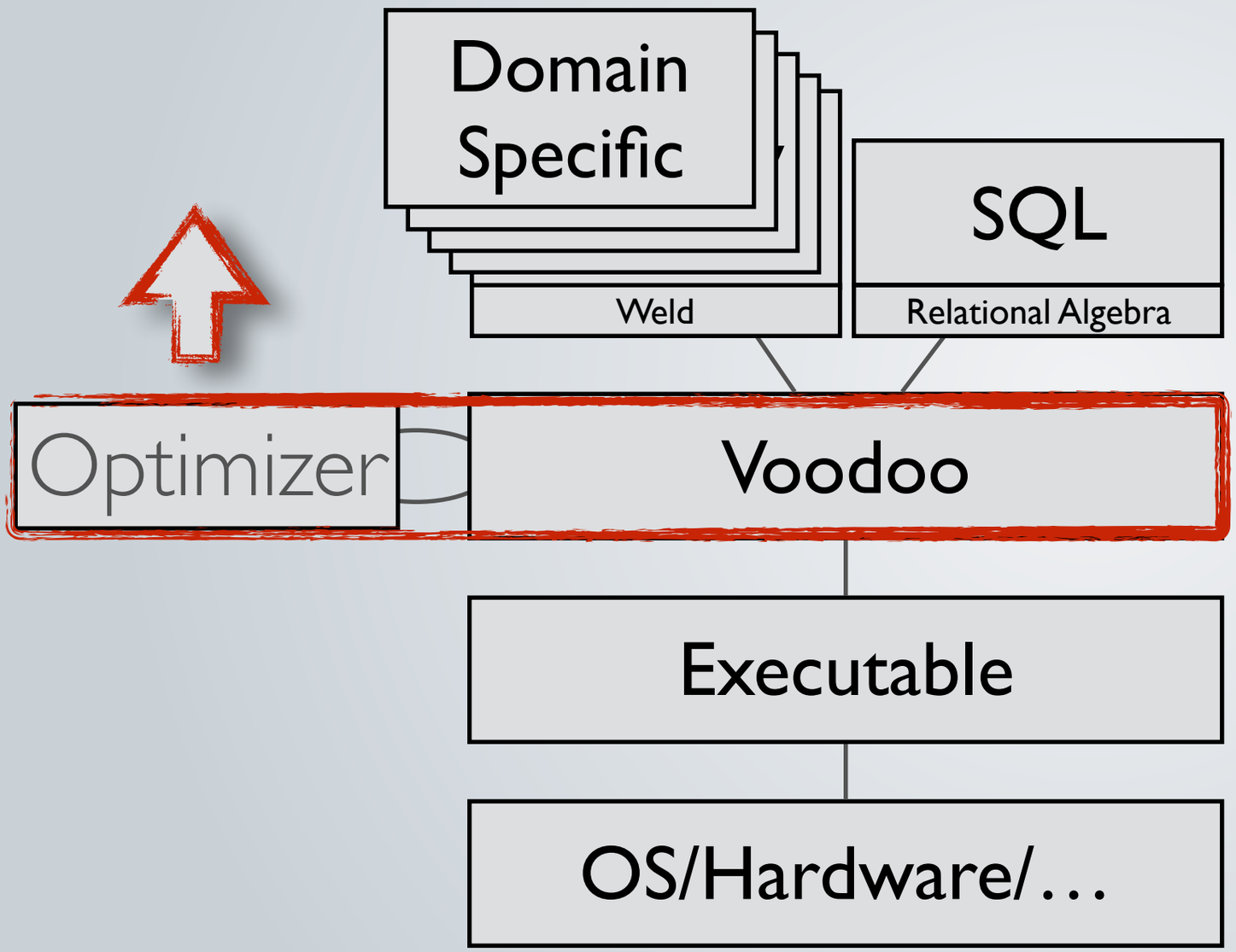
Quo Vadis

Imperial College London

FUTURE WORK

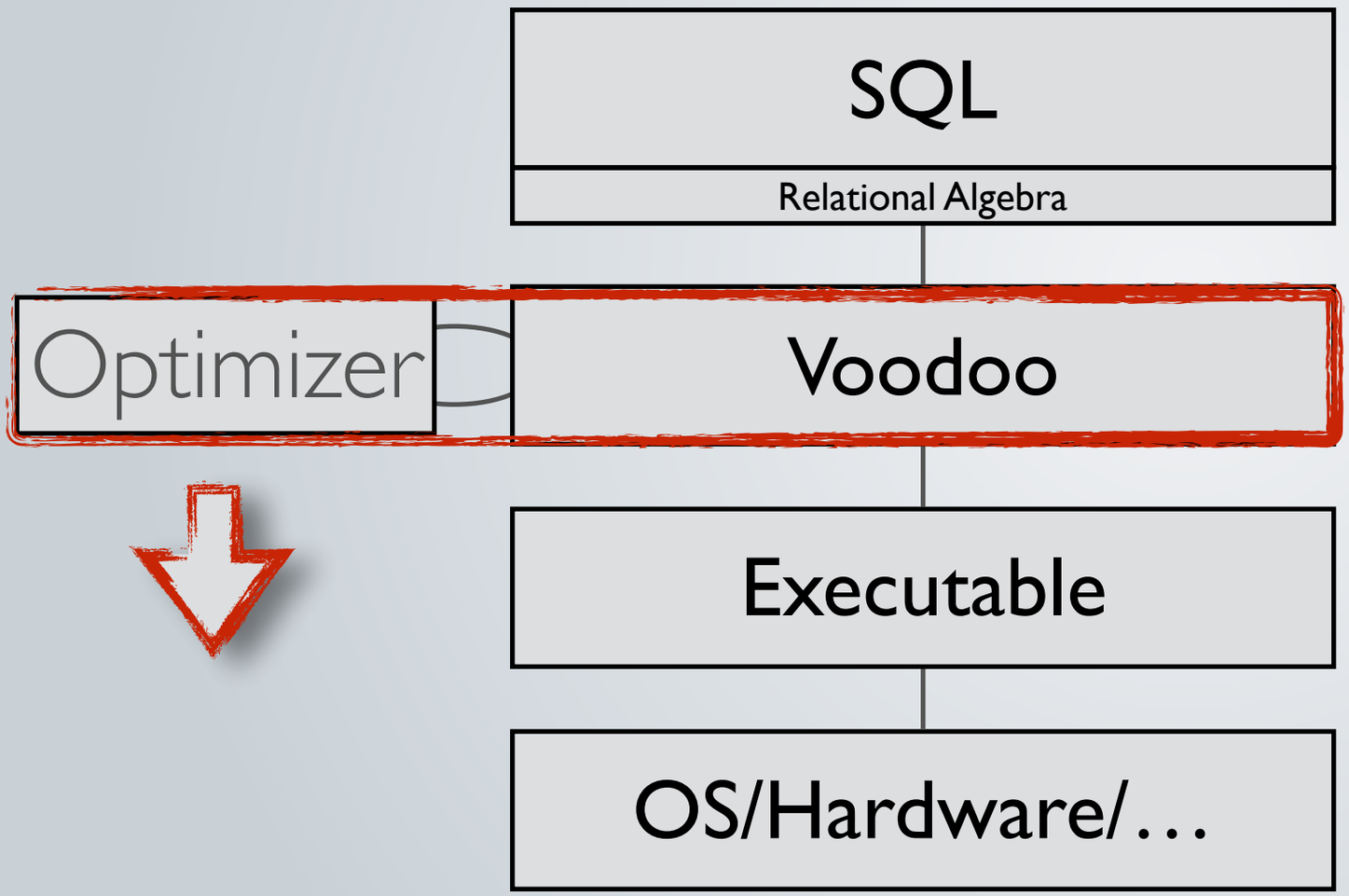

SQL
Relational Algebra
Optimizer
Voodoo
Executable
OS/Hardware/…

FUTURE WORK



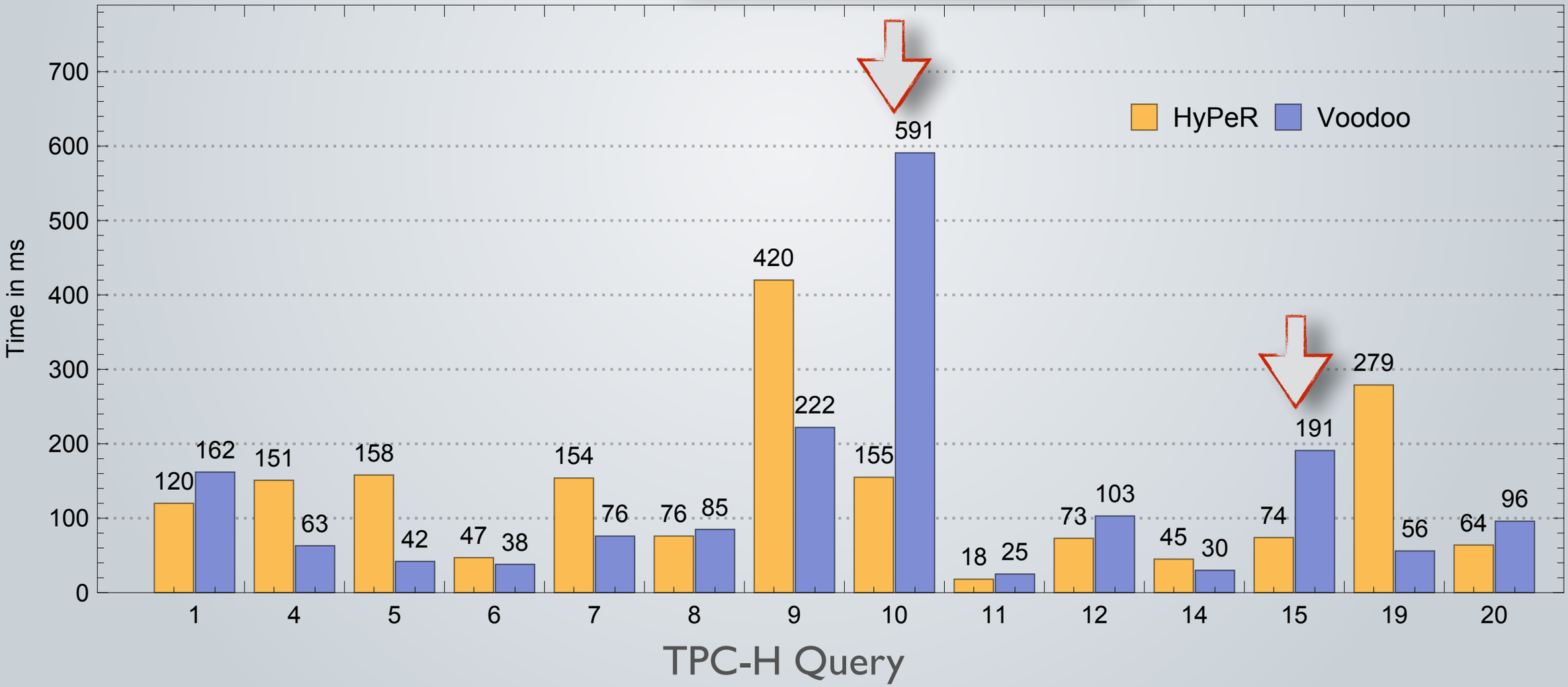
[VLDB 2018]

QUO VADIS


SQL
Relational Algebra
Optimizer
Voodoo
Executable
OS/Hardware/…

Plugging holes in the Design Space

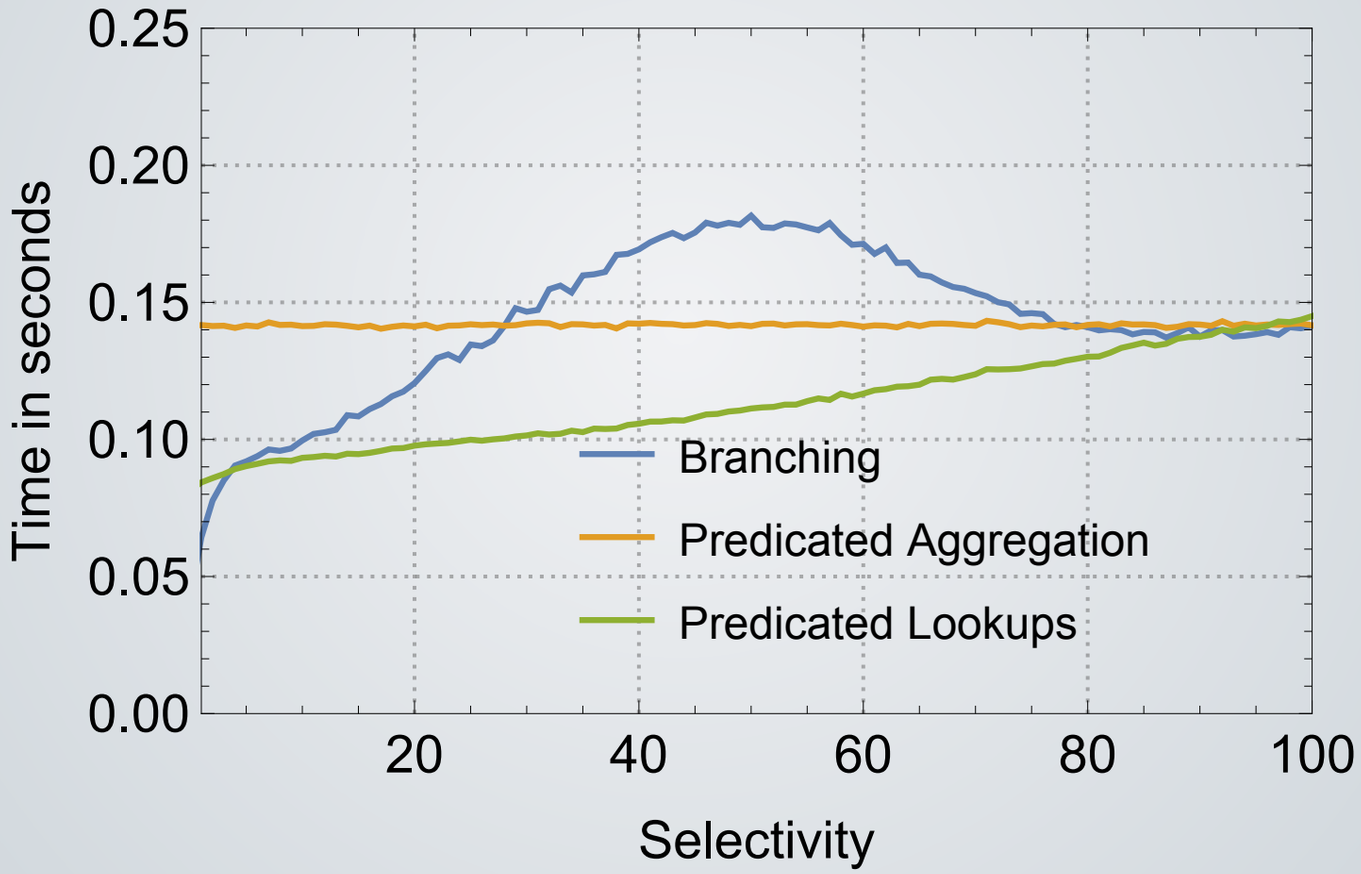
Remember these?



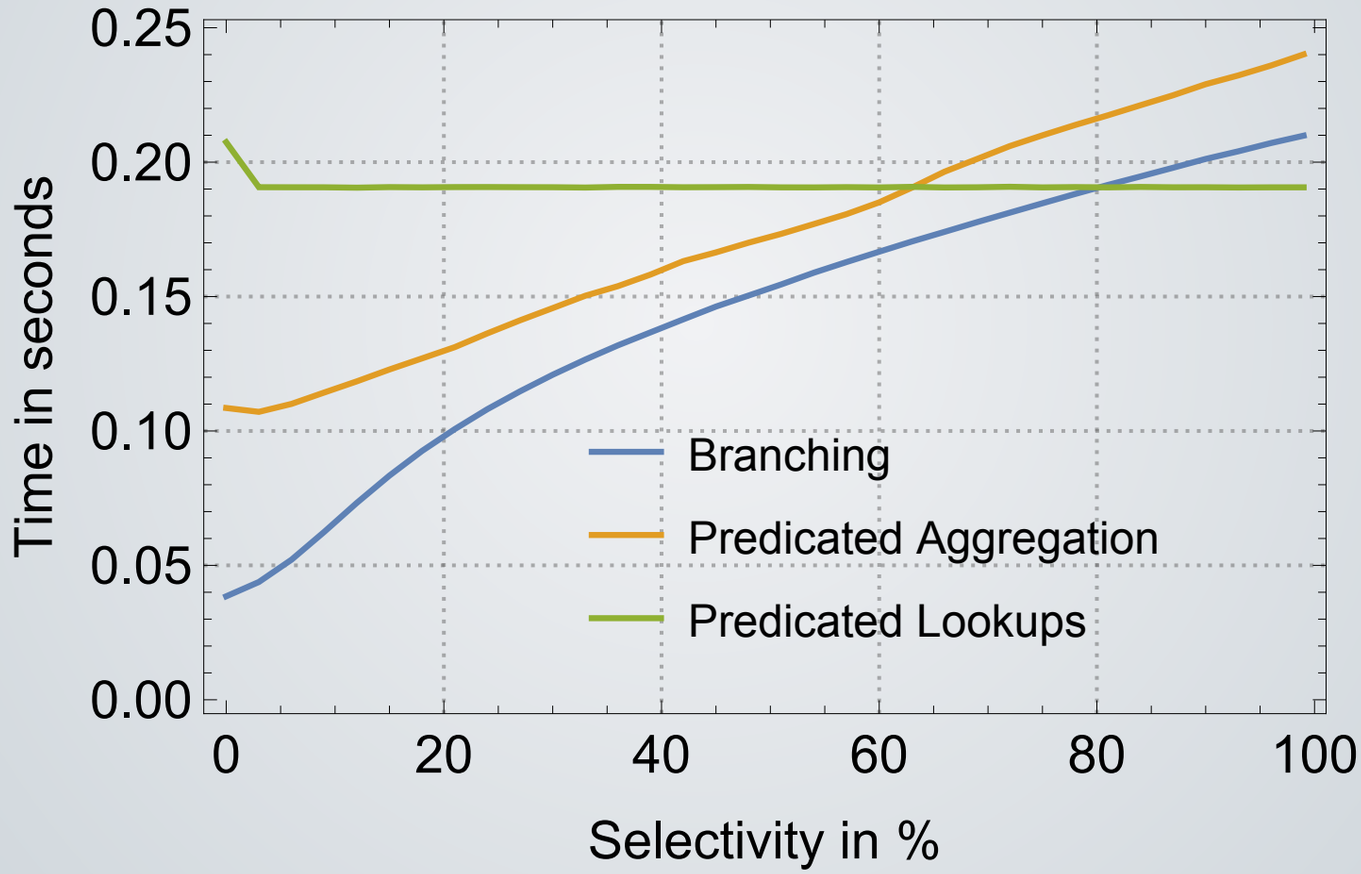
Plugging holes in the Design Space

- Massively Parallel Top-K [SIGMOD 2018]
- Incremental Window Computation [ongoing]
- Dynamic Load Balancing on GPUs [future]

Auto-Generating Databases



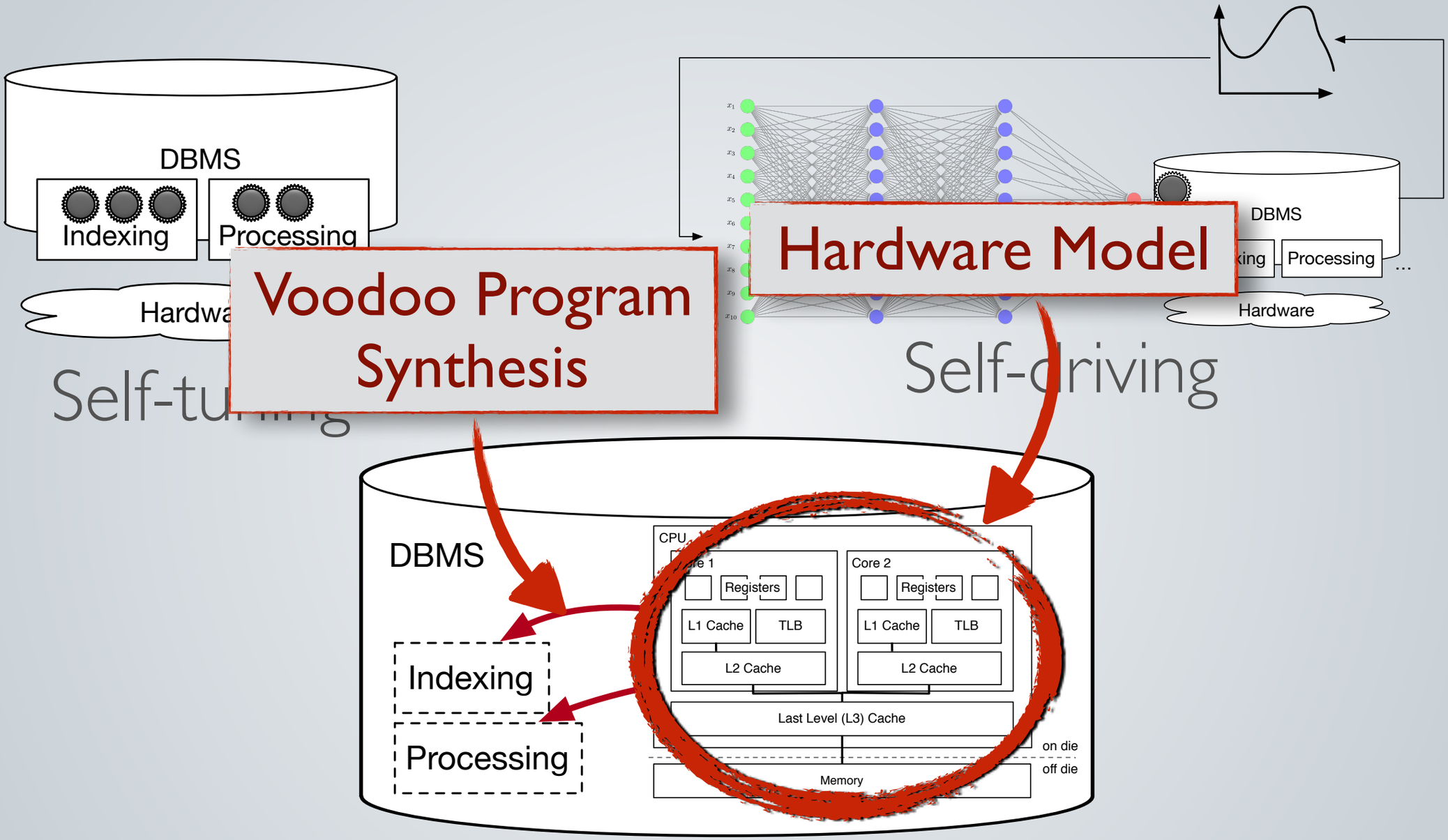
Auto-Generating Databases



Auto-Generating Databases

- Hardware-conscious modelling & tuning [ongoing]

Auto-Generating Databases



Self-tuning

Self-driving

Self-generating

Imperial College London

Auto-Generating Databases

- Hardware-conscious modelling & tuning [ongoing]
- Auto-Generating Databases [future]

Wrapping up

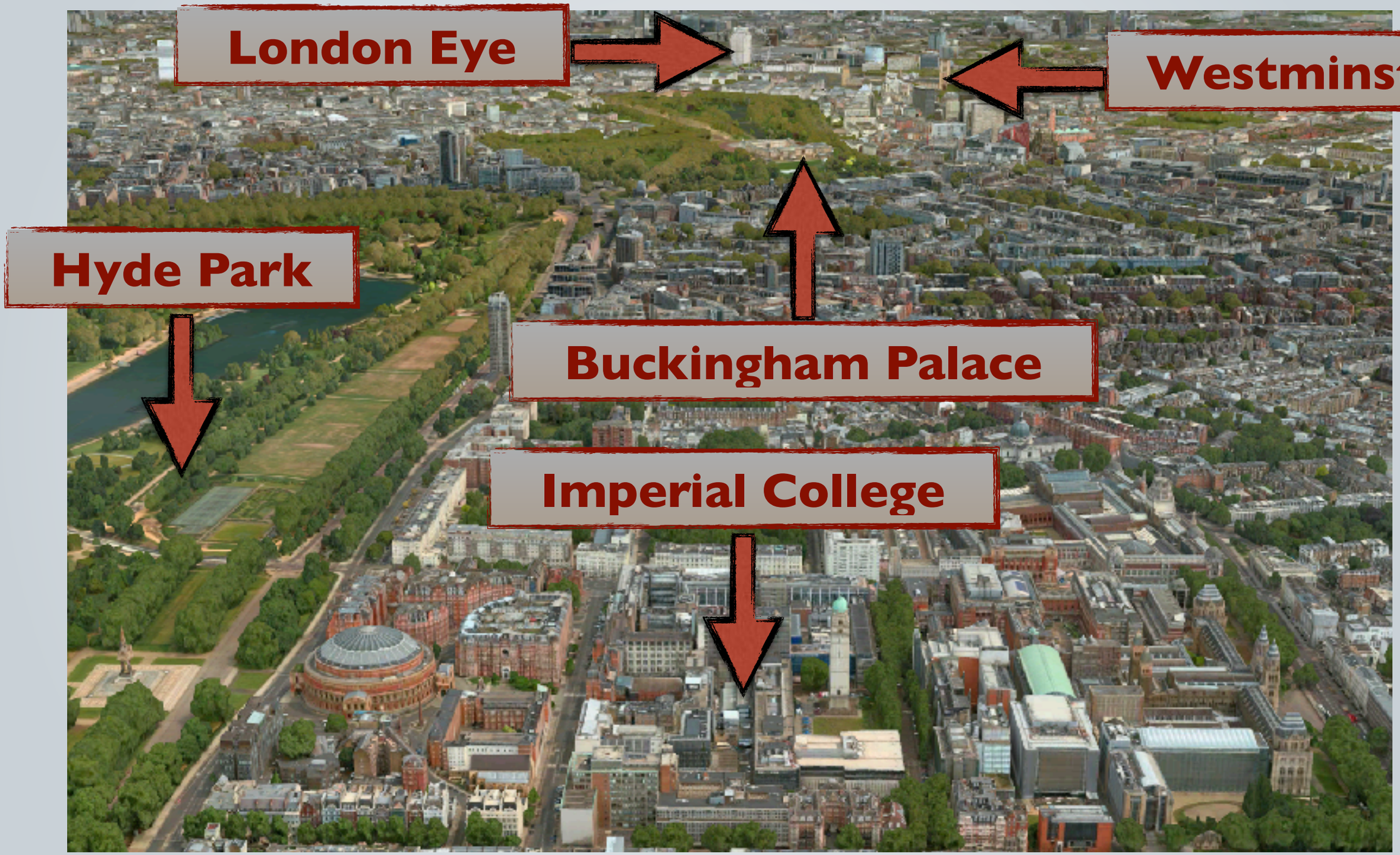
Thanks to Collaborators

Sam Madden
Mike Stonebraker
Anil Shanbhag
Malte Schwarzkopf

Matei Zaharia
Shoumik Palkar

Imperial College London

A new DB research hub — Visit us!



Imperial College London

Recruiting

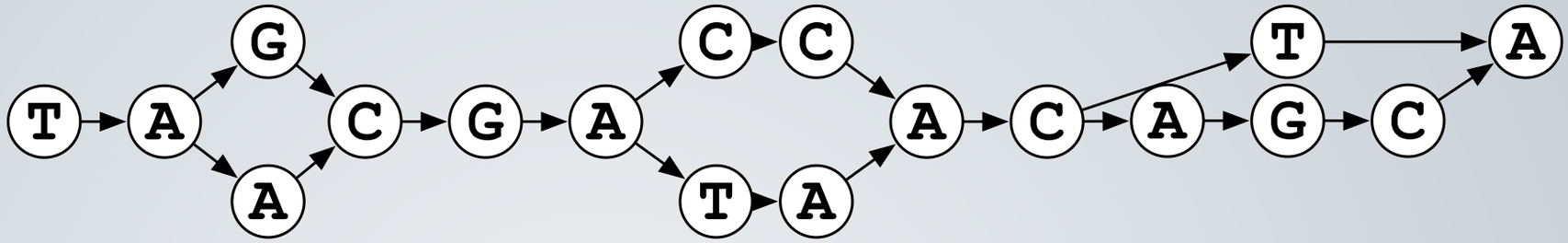
Imperial College London

Thank You

Backup slides

Domain-Conscious Database Design

Database:



Multiple sequence alignment

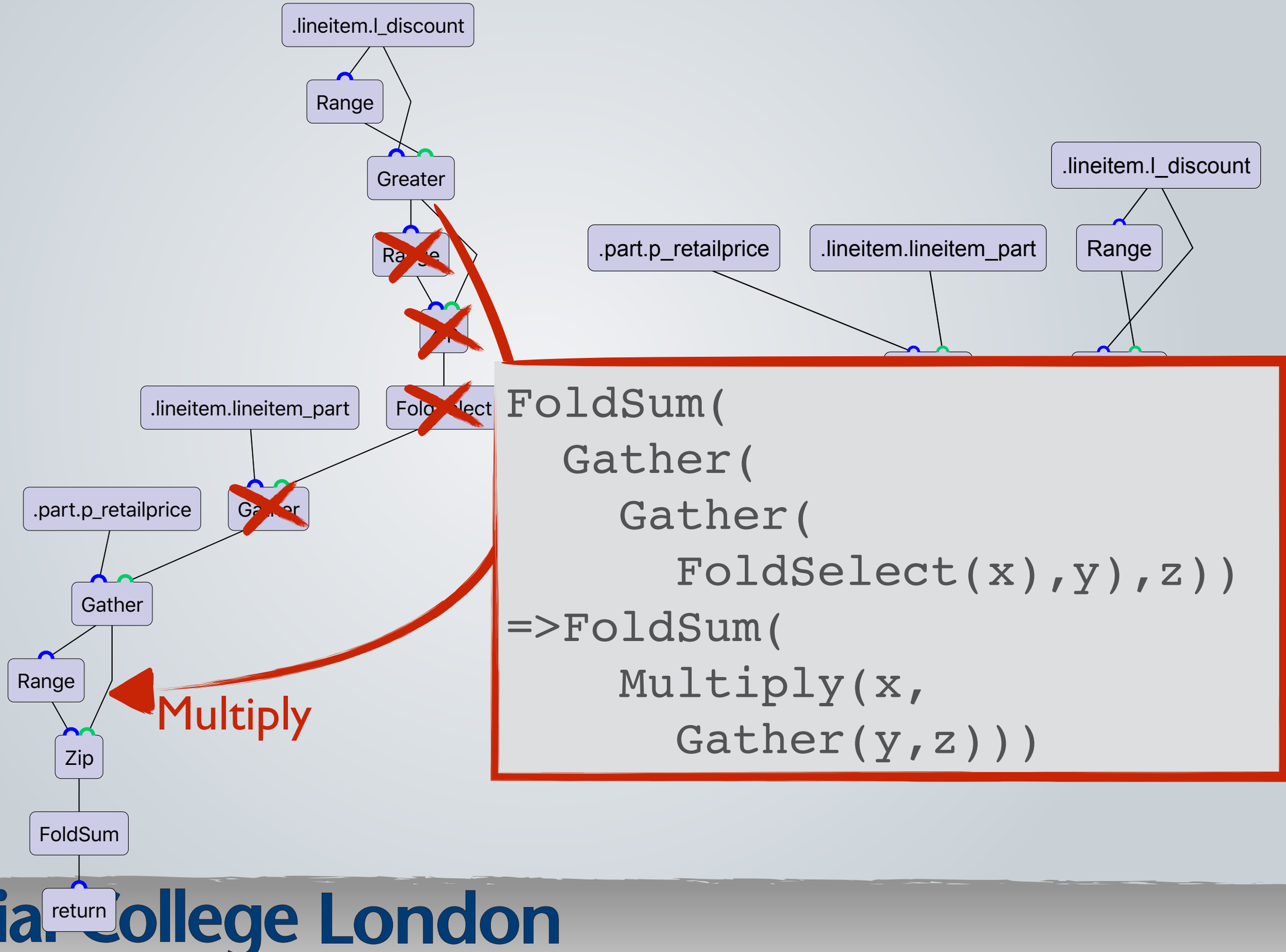
Queries:

merge the sequence ACCCTA into the graph while minimizing the number of new branches

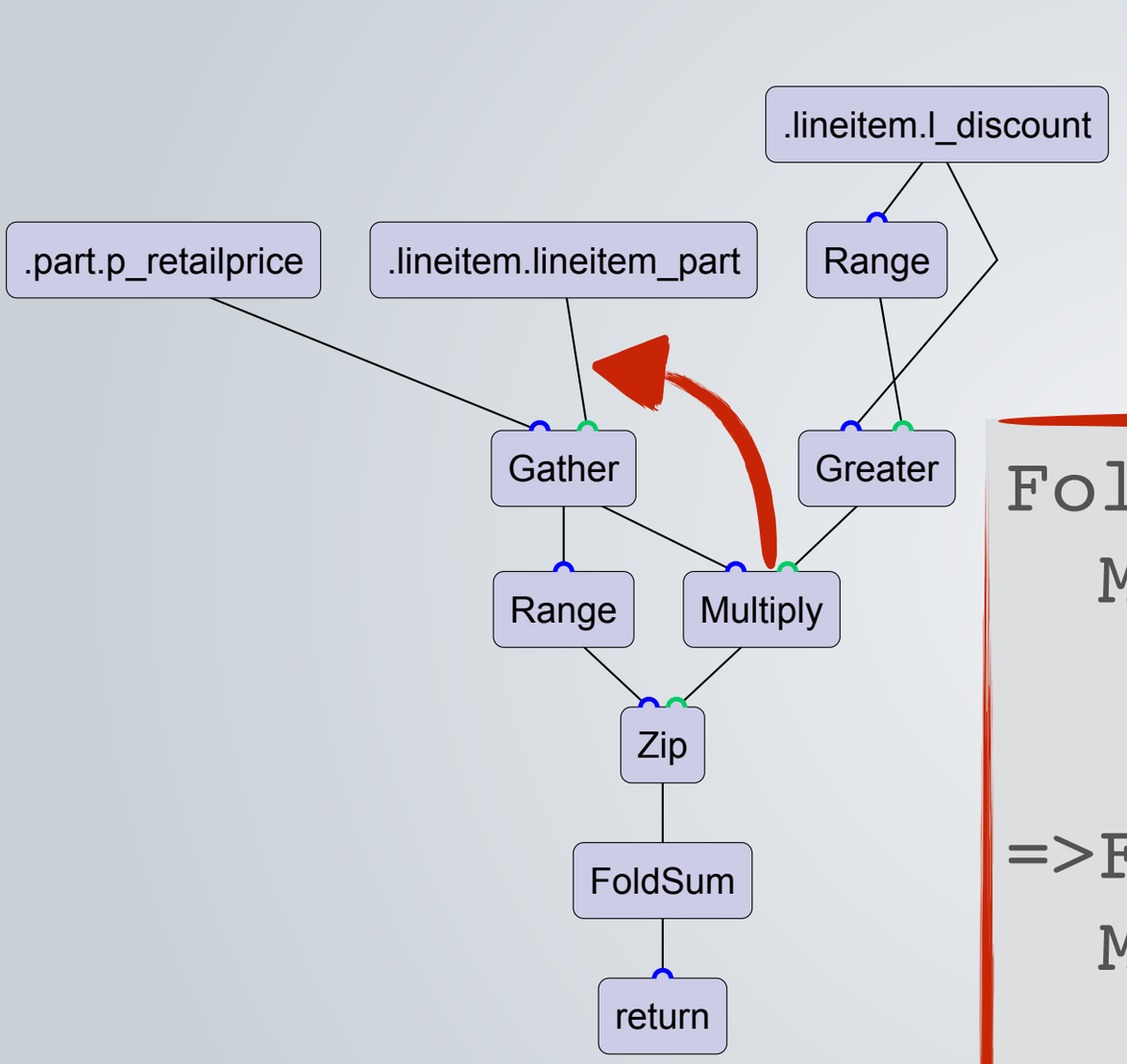
count the number of mutations of a gene

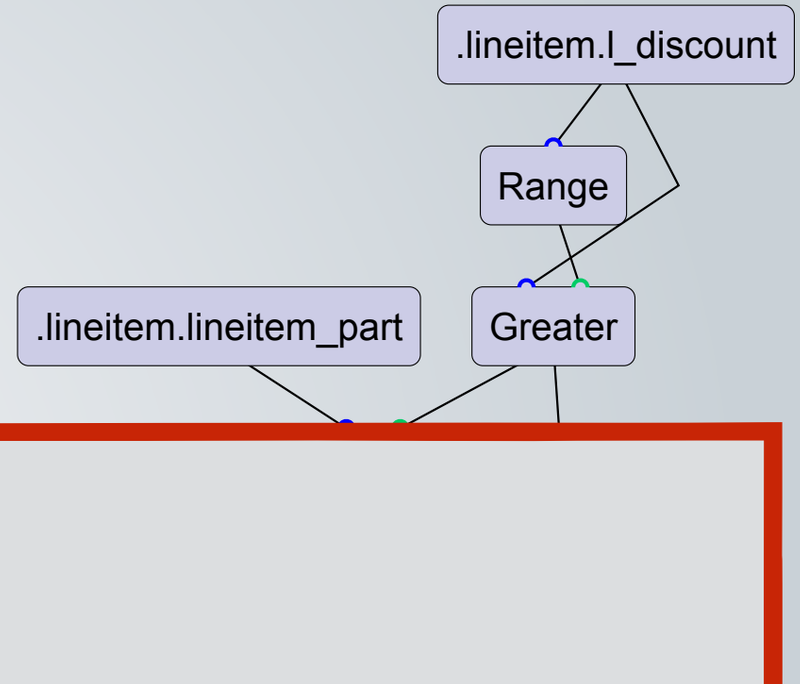
find the most common allele of a gene

Predicated Foreign-Key Joins



Double-predicated Foreign-Key Joins



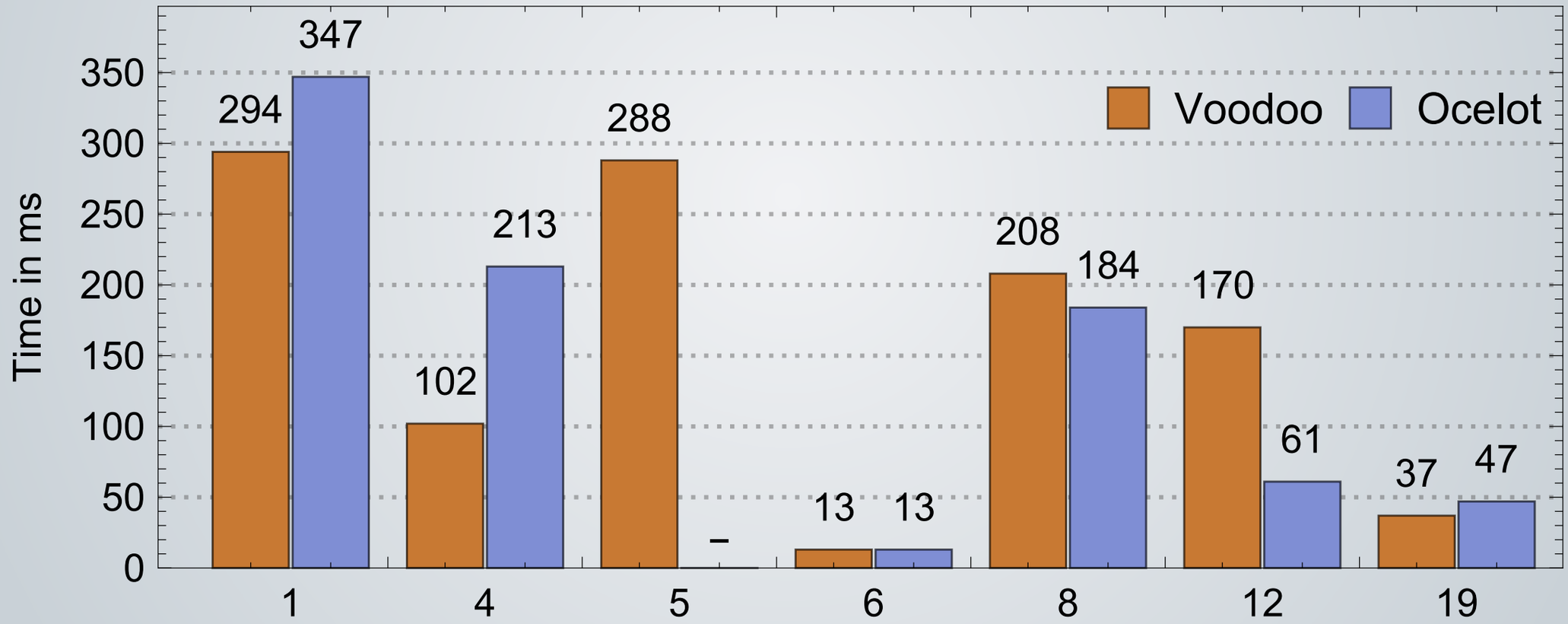



```
FoldSum(
  Multiply(
    Gather(
      x,y),z))
=>FoldSum(
  Multiply(
    Gather(
      Multiply(x,z),y),z))
```

TPC-H Query 9 in Hyper

- Select (on dimension/target table) & Join Query
- HyPeR strategy: select scan + hash-join
  - Ignores foreign key index
- Voodoo strategy: select scan, build bitmap, join using FK-Index
  - Takes advantage of foreign key index

TPC-H ON GPUS


Time in ms
0
50
100
150
200
250
300
350
Voodoo
Ocelot
294
347
102
213
288
–
13
13
208
184
170
61
37
47
1
4
5
6
8
12
19

Why Exploration?
Why not machine Learning?

- Because it takes ML-experts
- There often isn't enough data (e.g., for catastrophic events)
- It is slow (in particular the training)

…when generating code

- LLVM
  - No Multithreading support
  - SIMD support is best-effort
  - No transactional memory support
  - No *real* GPU support
  - No NUMA support