Push vs. Pull-Based Loop Fusion in Query Engines

Amir Shaikhha

23/10/2018

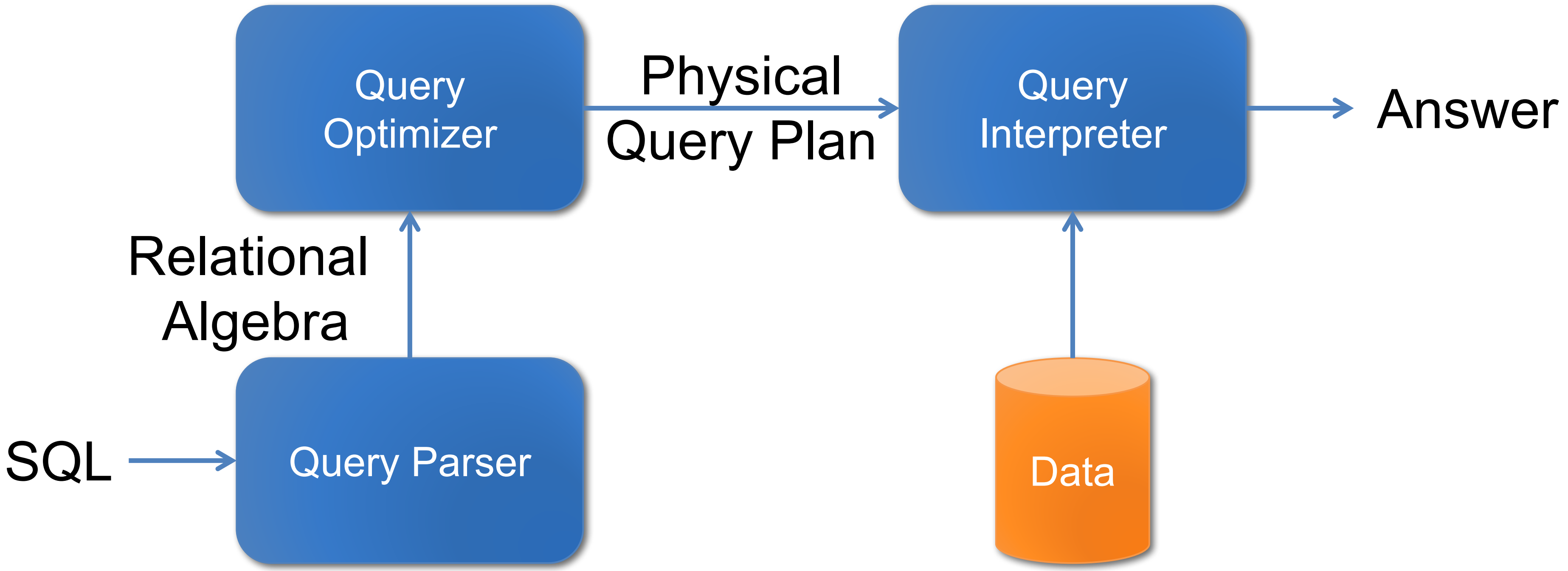
NII Shonan Meeting

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Introduction

- DBMSes are essential components of software systems
- Persistence layer
- Expose a declarative language (SQL) to users

Query Processing



Physical Query Plan

- Join has different implementations
  - Hash Join
  - Nested Loop Join
  - Sort Merge Join
- Each one is appropriate for a particular scenario
- Physical Query Plan
  - Annotated Relational Algebra
  - Concrete implementation choice for each operator
  - Can improve the execution time of a query from 1 year to 0.1 seconds

Query Interpretation

- Runtime library
- Iterator Model (pull based)
- Used in mainstream DBMSes for a long time

Dominating cost is I/O

Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

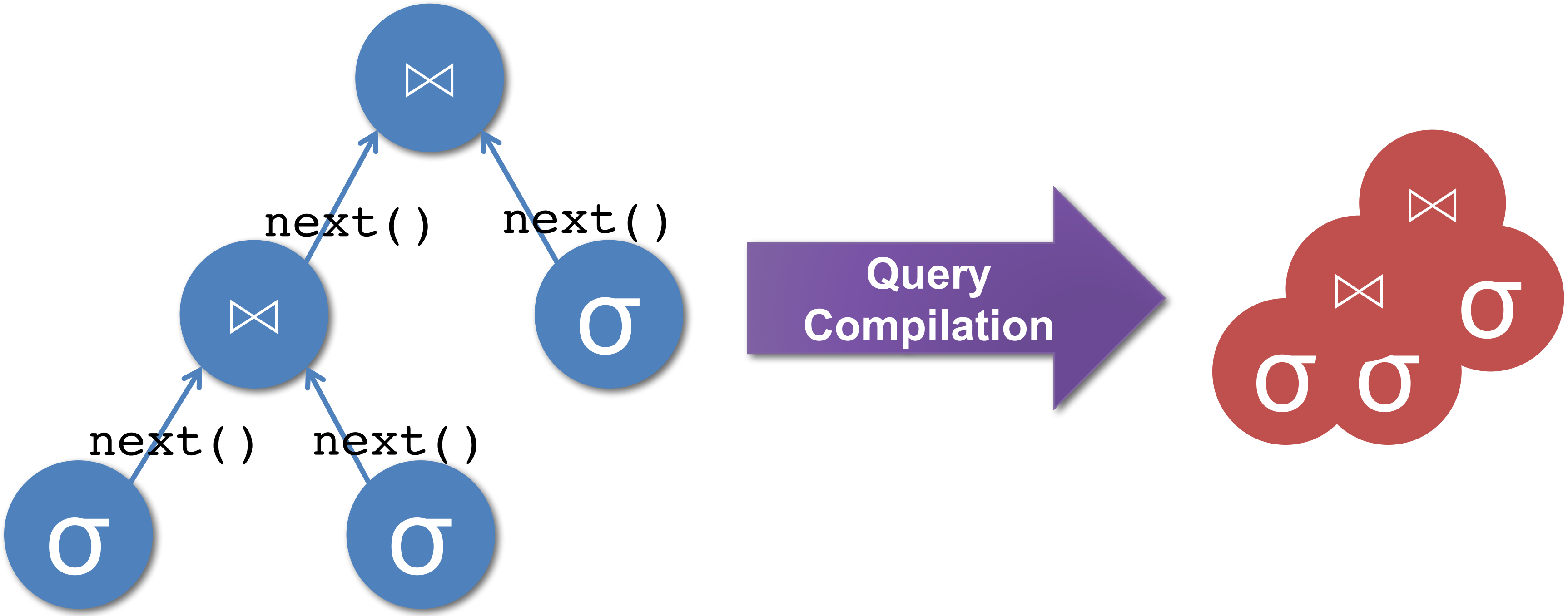
Google

In-Memory Databases

- Modern Hardware Technology
- Servers with 1TB of RAM
- The whole database can fit in the RAM

Code layout is important

Why Query Compilation?



Interpreted Query

✓ Good out-of-core data locality

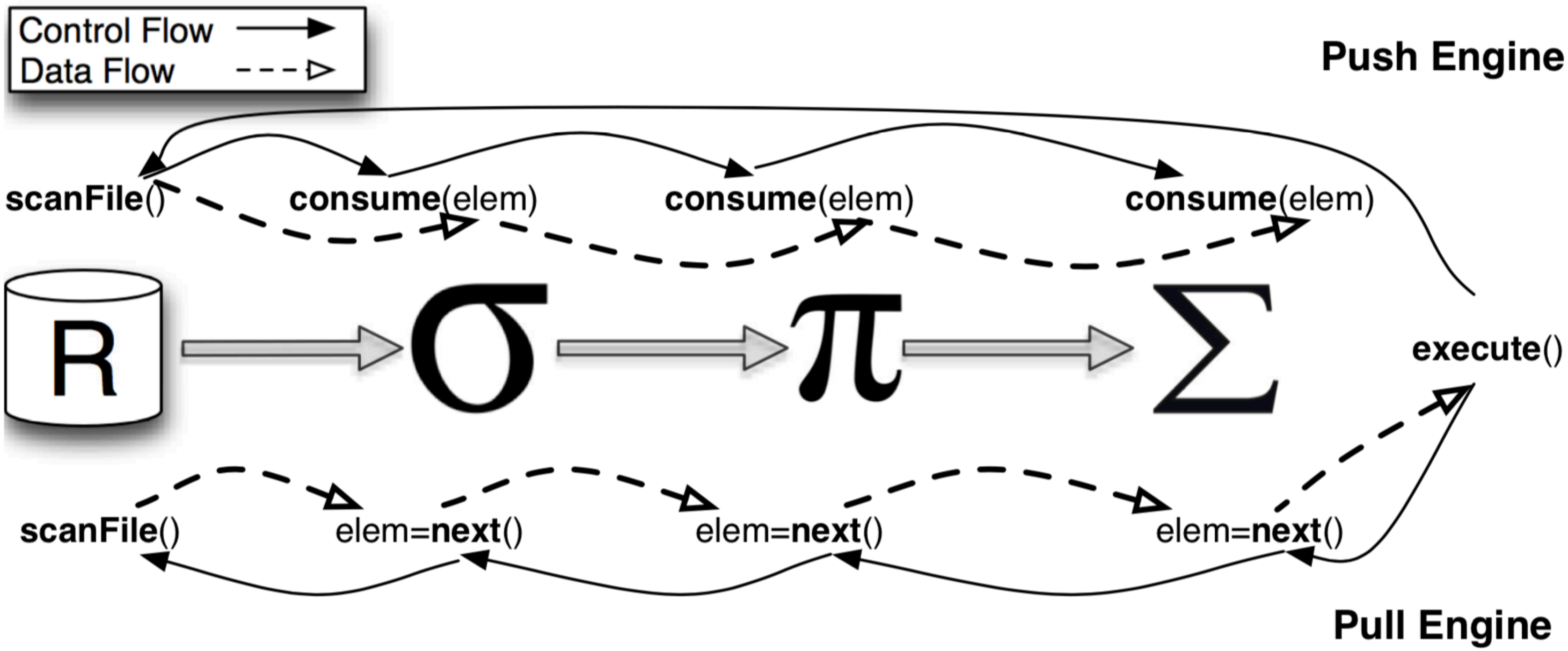
✗ Virtual function calls

  ✗ Bad branch prediction & cache locality

Compiled Query

✓ Good branch prediction & cache locality

Push vs. Pull

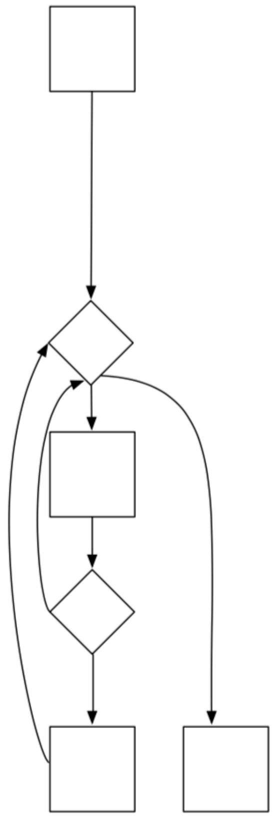

Control Flow
Data Flow
Push Engine
scanFile()
consume(elem)
consume(elem)
consume(elem)
R
σ
π
Σ
execute()
scanFile()
elem=next()
elem=next()
elem=next()
Pull Engine

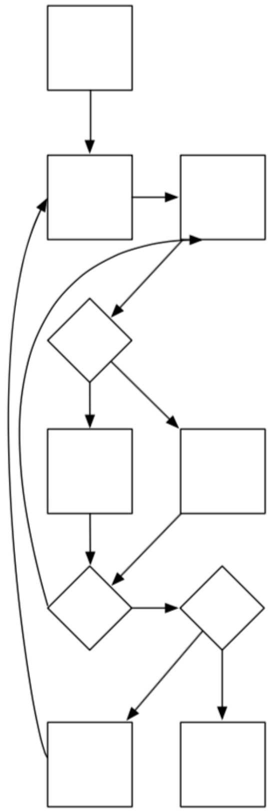
Push vs. Pull

```
var sum = 0.0
var index = 0




while(index < R.length) {
  val rec = R(index)
  index += 1




  if(rec.A < 10)

    sum += rec.B
}
return sum
```

```
var sum = 0.0
var index = 0
while(true) {
  var rec = null
  do {
    if(index < R.length) {
      rec = R(index)
      index += 1
    } else {
      rec = null
    }
  } while(rec != null && !(rec.A < 10))
  if(rec == null) break
  else sum += rec.B
}
return sum
```

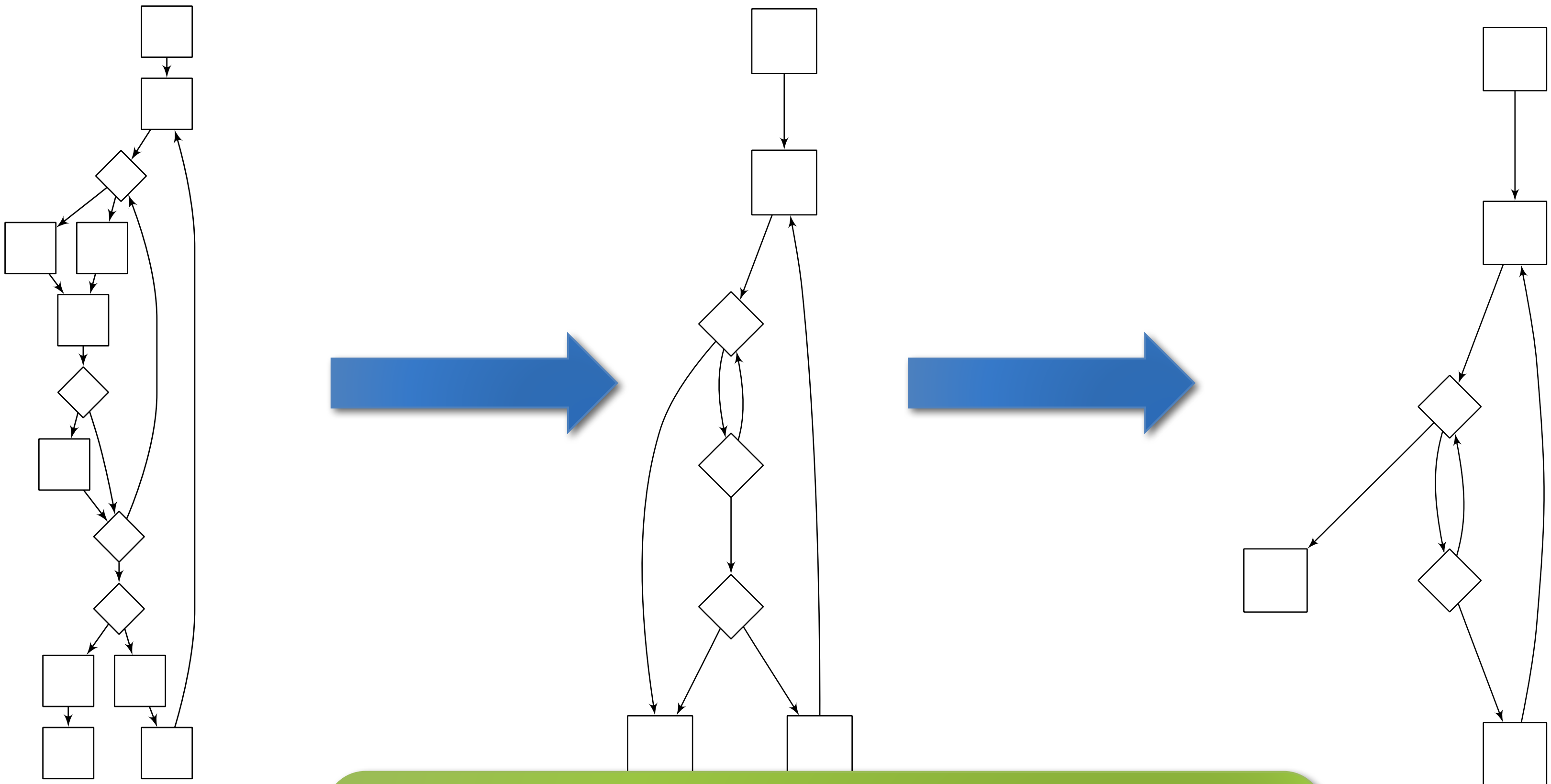
CFG of Push vs. Pull





Pull Engine produces a more complicated CFG

Simplifying CFG



Optimizing Compilers successfully simplify CFG

Push Engine Issues

- Hard to handle
  - Merge & Zip-like operator
  - Limit operator
- Solutions
  - Give up & materialize
  - Ad-hoc fused version of operators
  - Rely on hacky mechanisms
    - Makes the interface more complicated

Pipelining in SQL = Fusion in collections

```
SELECT SUM(R.B)
FROM R
WHERE R.A < 10
```

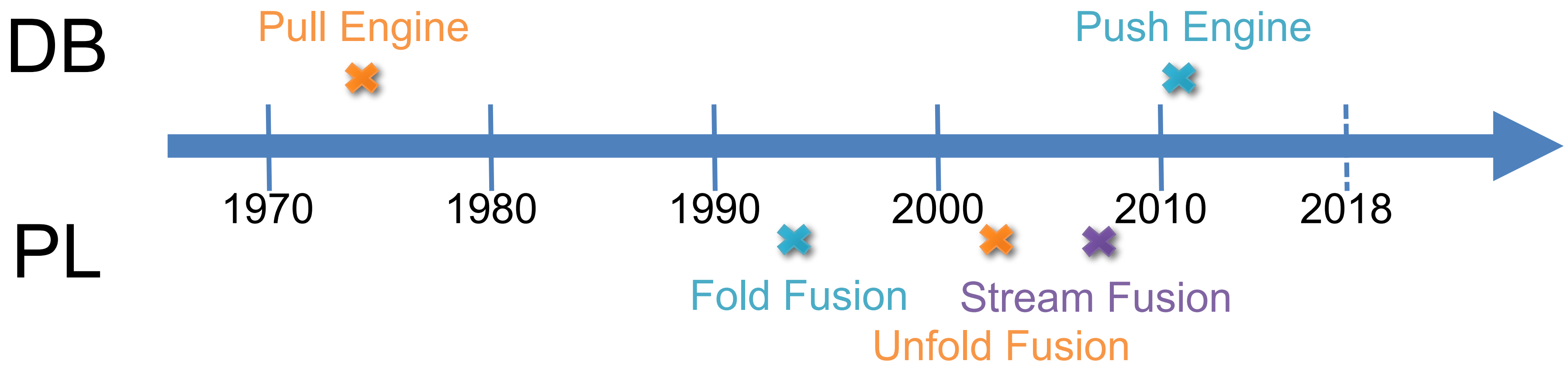
=

```
R.filter(r => r.A < 10)
 .map(r => r.B)
 .fold(0)((s, r) => s + r)
```

→

```
var sum = 0.0
var index = 0
while(index < R.length) {
  val rec = R(index)
  index += 1
  if(rec.A < 10) sum += rec.B
}
return sum
```

Pipelining/Fusion History



Push

```
class ProjectOp[R, P](f: R => P) {
  def consume(e: R): Unit =
     dest.consume(f(e))
}
class SelectOp[R](p: R => Boolean) {
  def consume(e: R): Unit =
    if(p(e))
       dest.consume(e)
}
```

```
def map[S](f: R => S) = build { k =>
  for(e <- this)
     k(f(e))
}
def filter(p: R => Boolean) = build { k =>
  for(e <- this)
    if(p(e))
     k(e)
}
```

Pull

```
class SelectOp[R](p: R => Boolean) {
  def next(): R = {
    var elem: R = null
    do {
      elem = source.next()
    } while (elem != null && !p(elem))
    elem
} }
class ProjectOp[R, P](f: R => P) {
  def next(): P = {
    val elem = source.next()
    if(elem == null) null
    else f(elem)
} }
```

```
def filter(p: R=>Boolean) = destroy { n =>
  generate { () =>
    var elem: R = null
    do {
      elem = n()
    } while(elem != null && !p(elem))
    elem
} }
def map[P](p: R => P) = destroy { n =>
  generate { () =>
    val elem = n()
    if(elem == null) null
    else f(elem)
} }
```

Fusion/Pipelining Correspondence

- Fold Fusion = Push Engine
- Unfold Fusion = Pull Engine

New

- Stream Fusion = Stream-Fusion Engine

Stream-Fusion Engine

- A pipelined query engine
- Inspired by the Stream Fusion approach developed for functional collections
- Combines the benefits of pull and push-based engines

Stream Fusion

```
class SelectOp[R](p: R => Boolean) {
  def stream(): Step[R] = {
    source.stream().filter(p)
} }
class ProjectOp[R, P](f: R => P) {
  def stream(): Step[P] = {
    source.stream().map(f)
} }
```

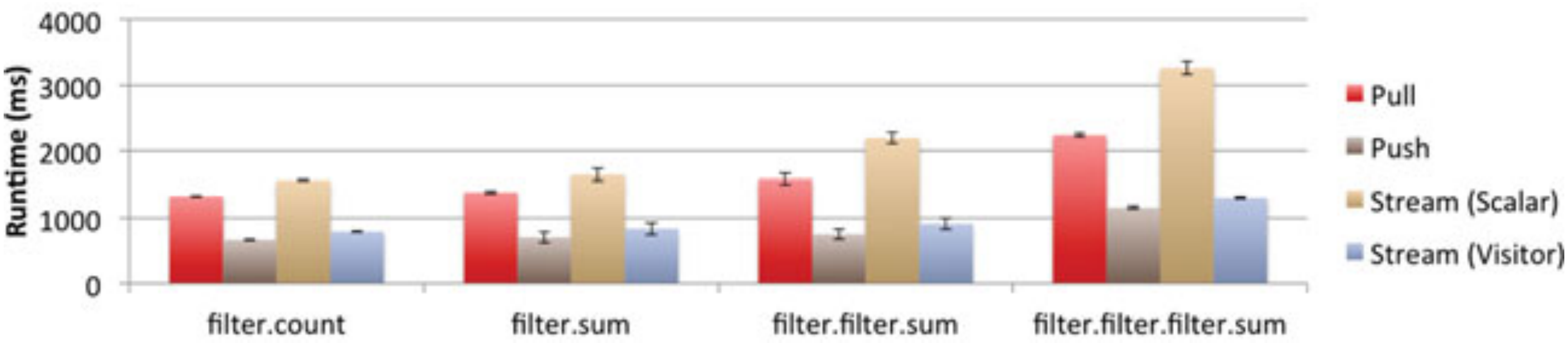
```
def filter(p: R => Boolean) = {
  unstream { () =>
    stream().filter(p)
} }
def map[P](f: R => P) = {
  unstream { () =>
    stream().map(f)
} }
```

Intermediate Step Objects

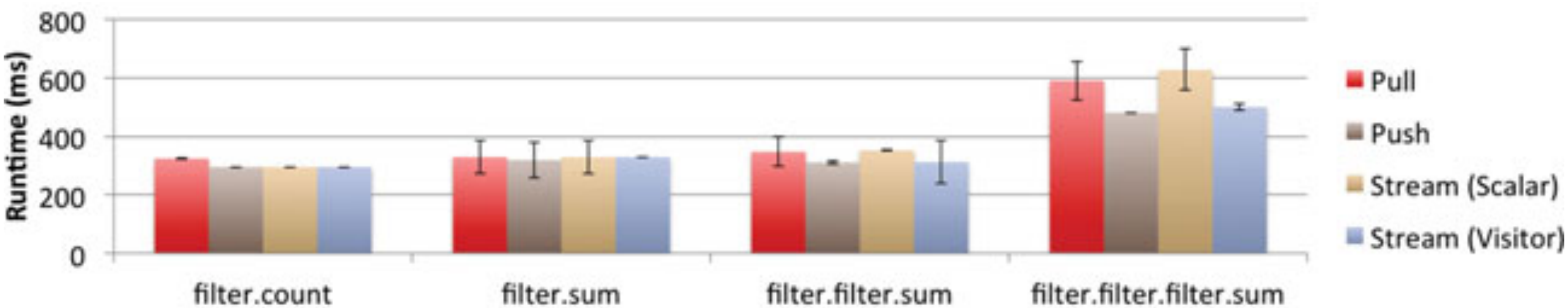
- Stream-fusion creates intermediate step objects:
  - Yield
  - Skip
  - Done
- Two solutions to remove their allocation
  - Scalar replacement
  - Church encoding of the sum type

Type of engine	Run time (ms)
Pull (Interpreted)	3,486
Pull (Naïve)	2,405
Pull (Inline-Friendly)	2,165
Stream (Scalar replacement for `Step` objects)	2,447
Stream (Visitor model for `Step` objects)	2,217
Stream (No removal of `Step` objects)	6,886

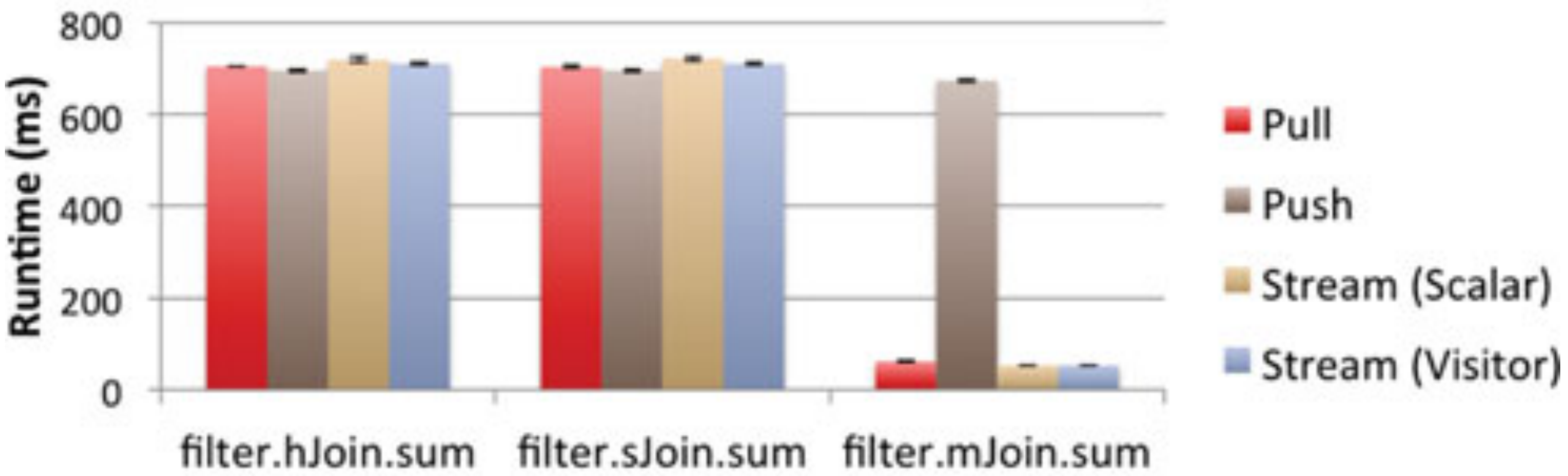
Micro-Benchmark Results



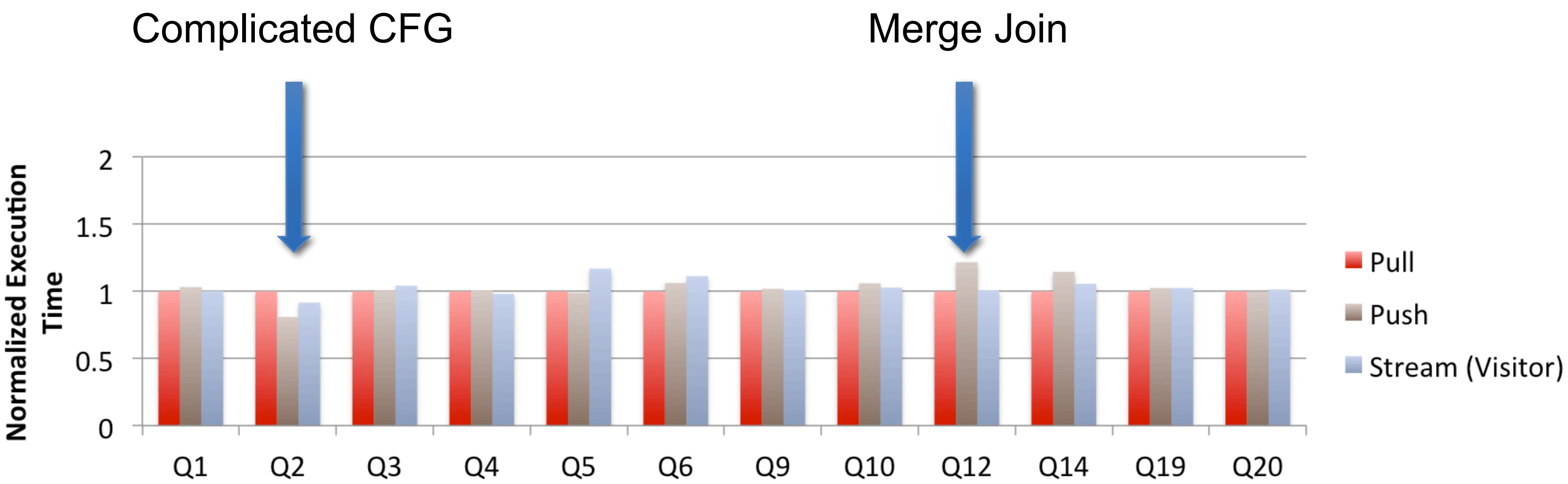
-O3



Micro-Benchmark Results (Cont.)



Experimental Results



In most cases all show similar performance

Conclusions

- Pipelining == Fusion
- DB ⇔ PL
- Many techniques can be exchanged
  - Vectorization == Generalized Stream-Fusion
  - Query optimizers
  - Column stores

Thank You!