Acorn

Aggressive Caching in Distributed Data Processing Frameworks

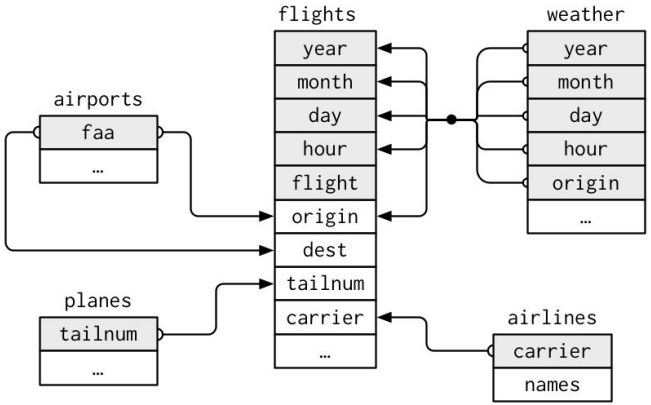
Lana Ramjit
UCLA

Matteo Interlandi
Microsoft

Eugene Wu
Columbia

Ravi Netravali
UCLA

Big Data Processing

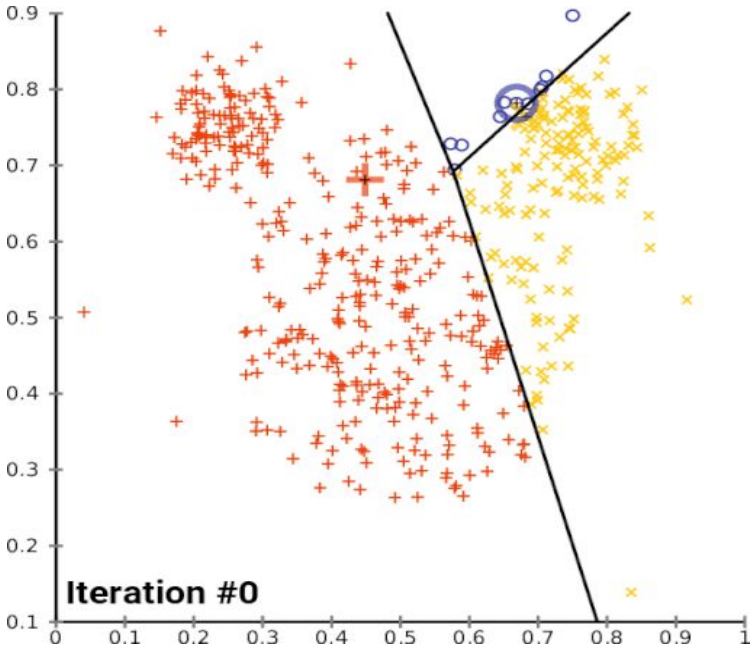


relational data

distributed computing

Iterative Workloads

- Graph Processing
  - Connected Components
  - PageRank
- Machine Learning
  - Belief Propagation
  - k-means clustering
- Interactive Data Exploration
  - single or multiple users



iterations of the k-means algorithm

Overlap between iterations should not be recomputed!

Caching Avoids Recomputing Overlap

Solution: Cache manager should find as many opportunities as possible to reuse old results

- transparently find caching opportunities
  - no user input!
- automatically rewrite incoming queries to use cache

Two Challenges

1. Pipeline introduces obstacles for effective caching

2. User Defined Functions (UDFs)

A Series of Queries

Query 1

UDF!

```
people.filter(p => p.age > 18)
```

Query 2

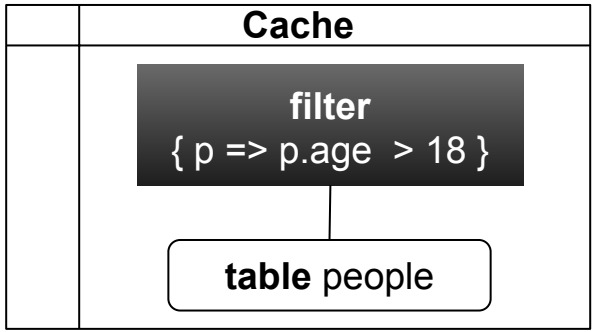
```
people.join(pets, "id === owner")
.filter(people.age > 18)
```

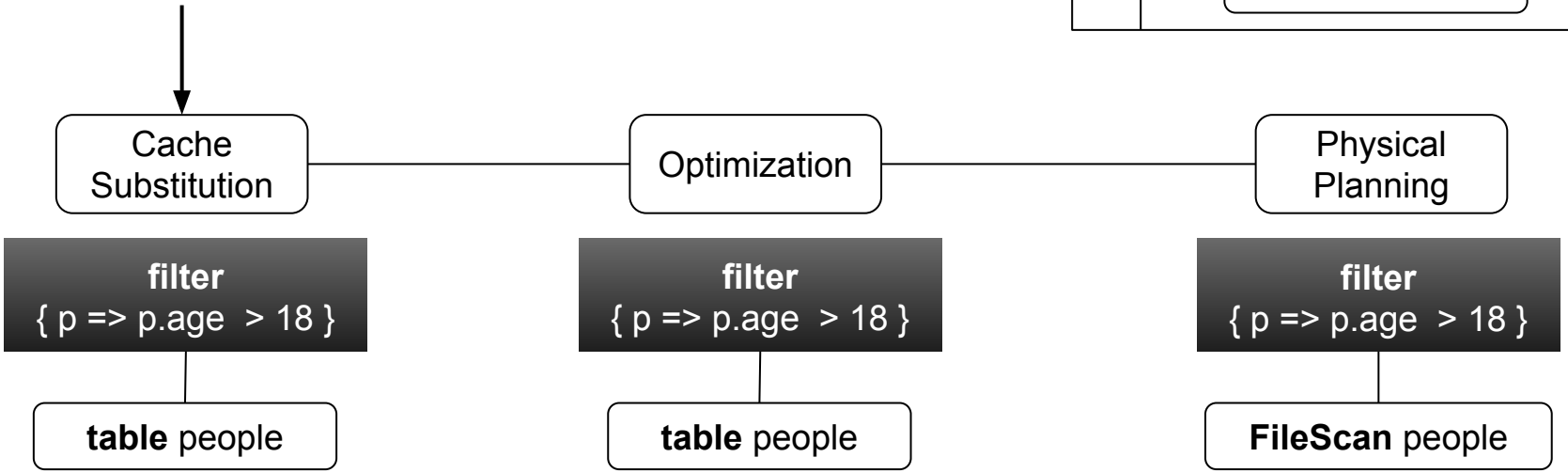
people		
name	**id**	**age**
stephanie	1	19
dylan	2	26
mary kate	3	17

pets	
name	**owner**
catsidy	2
gigi	3

Example: Query 1

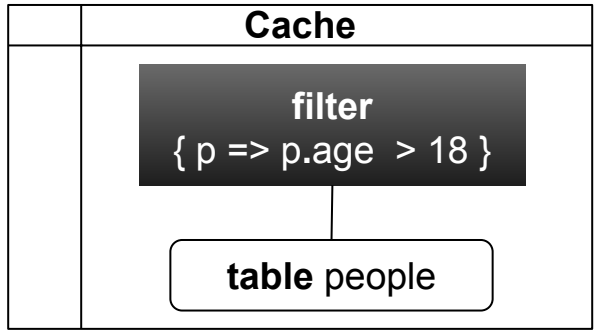
```
people.filter(age > 18)
```

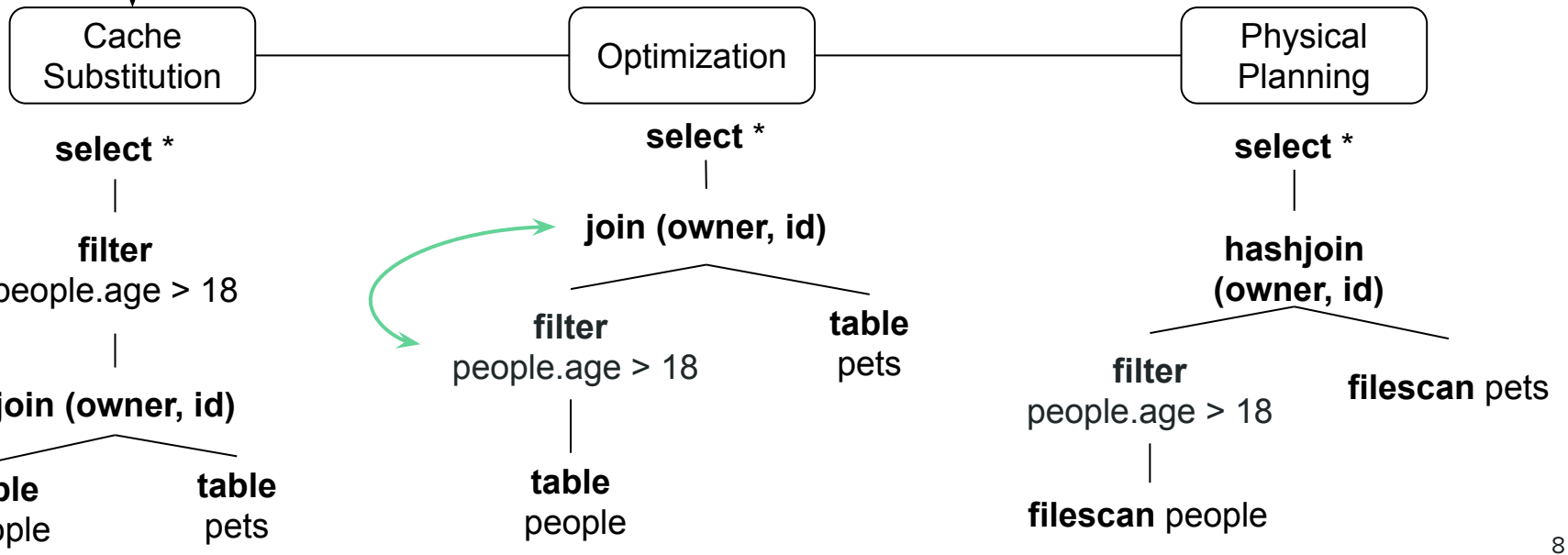




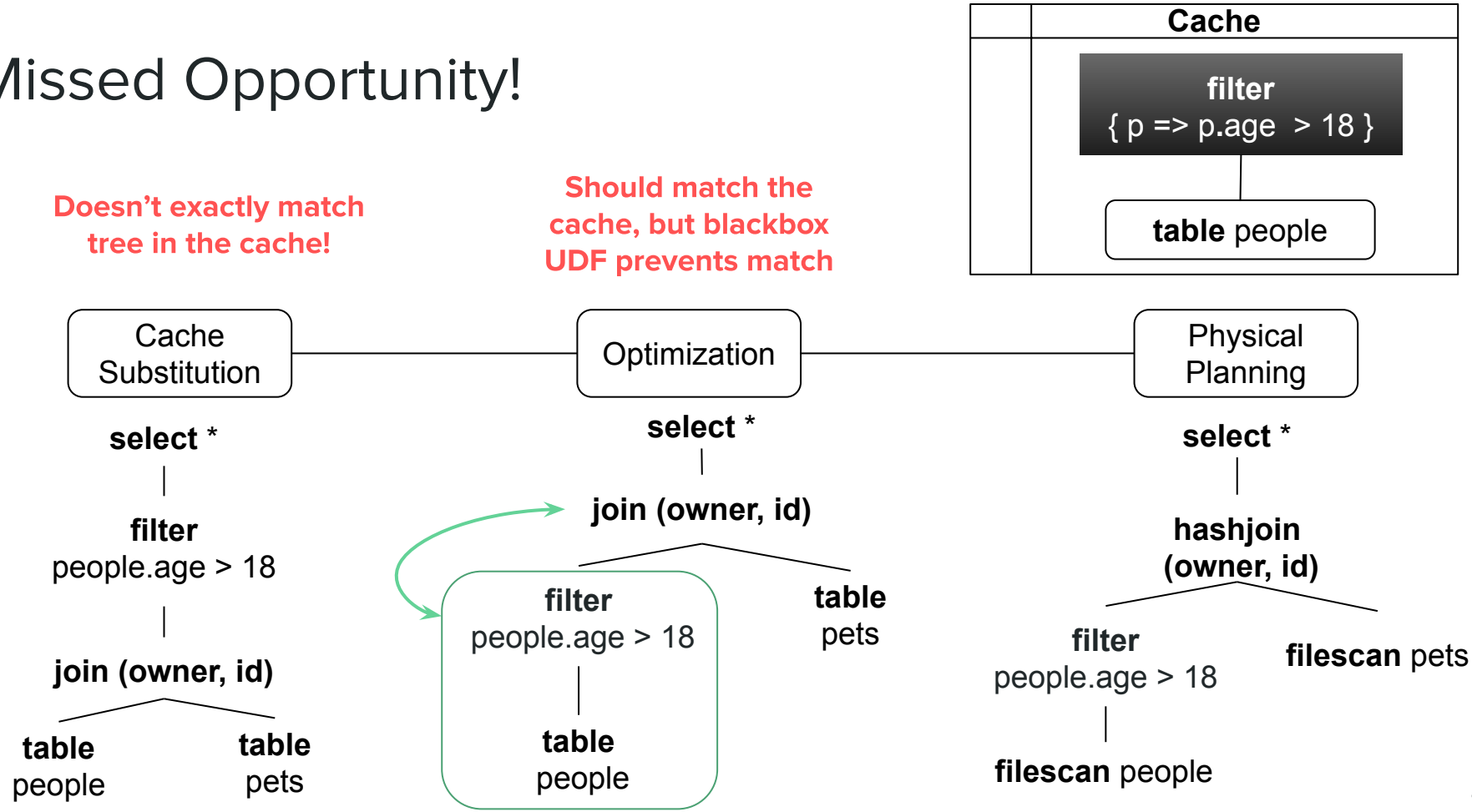
Example: Query 2

```
people.join(pets, "id === owner")
    .filter(people.age > 18)
```





Missed Opportunity!



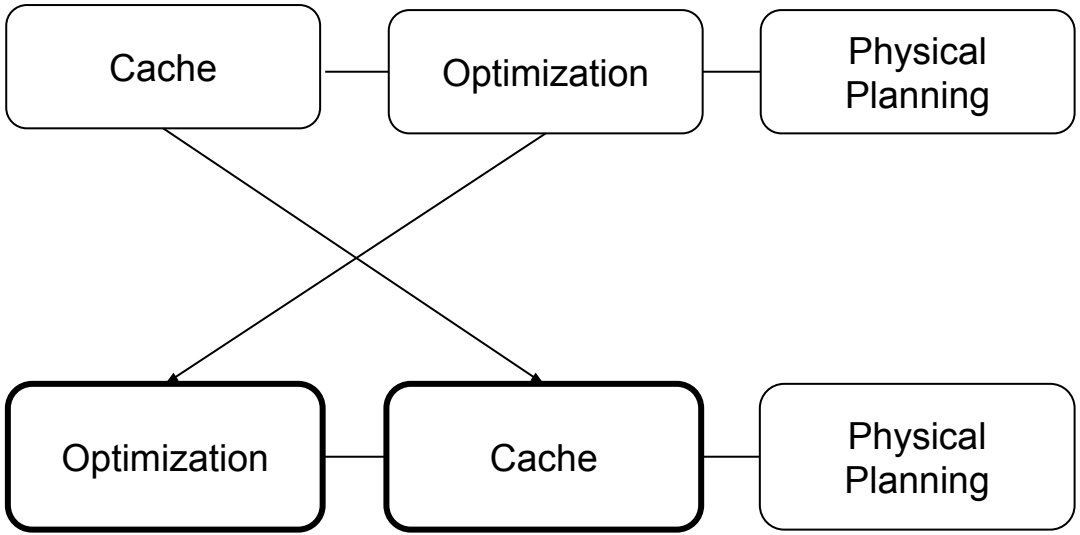
Two Challenges (recap)

1. **Pipeline introduces obstacles for effective caching**
   - **Cache compares unoptimized instead of optimized plans**
   - **unoptimized == uncanonicalized**

2. User Defined Functions (UDFs)
   - prevent high coverage
   - blackboxes to optimizers

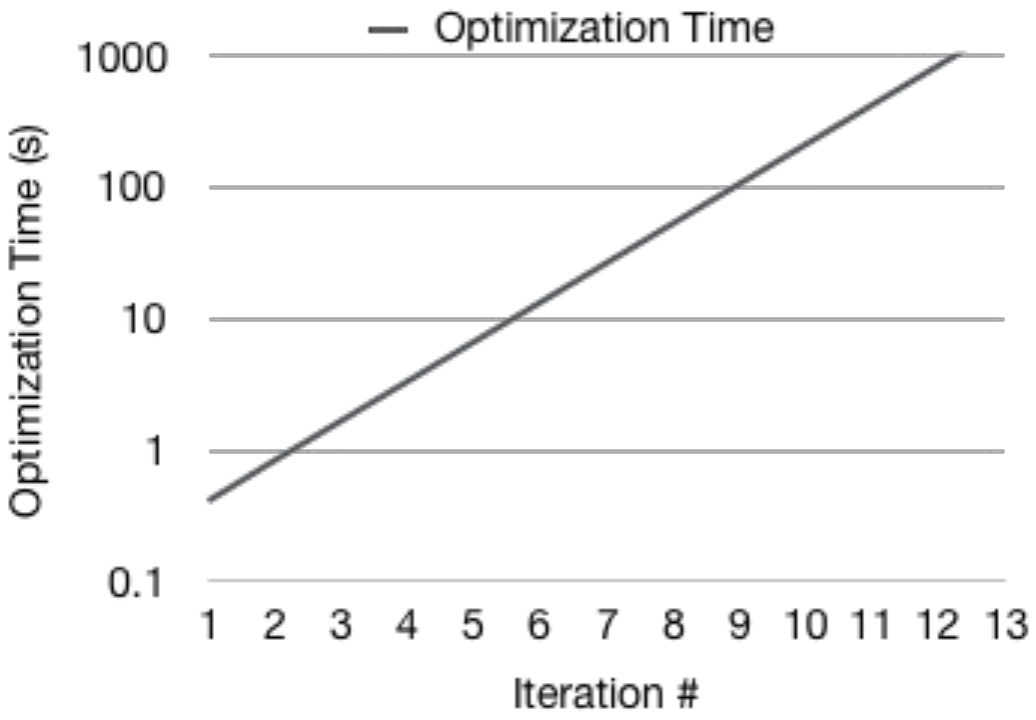
So, fix the pipeline?

Current Pipeline



Optimization-first pipeline

Optimization Is Slow



log scale!

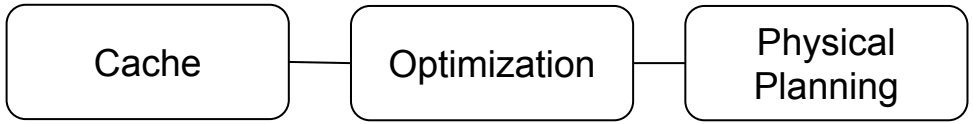
Optimization time per iteration of connected components algorithm

Why Is Optimization Slow?

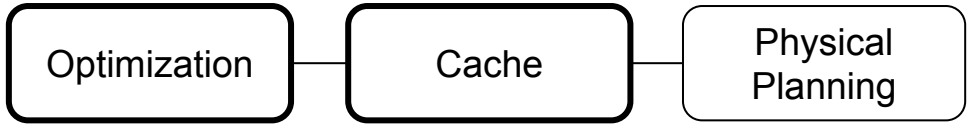
- General optimizer
  - targeting diverse workloads
  - custom rules
- Immutable data
  - can't "update" underlying data
  - all updates are logged in query plan
  - ➜ very large query plans

So, fix the pipeline?

Current Pipeline

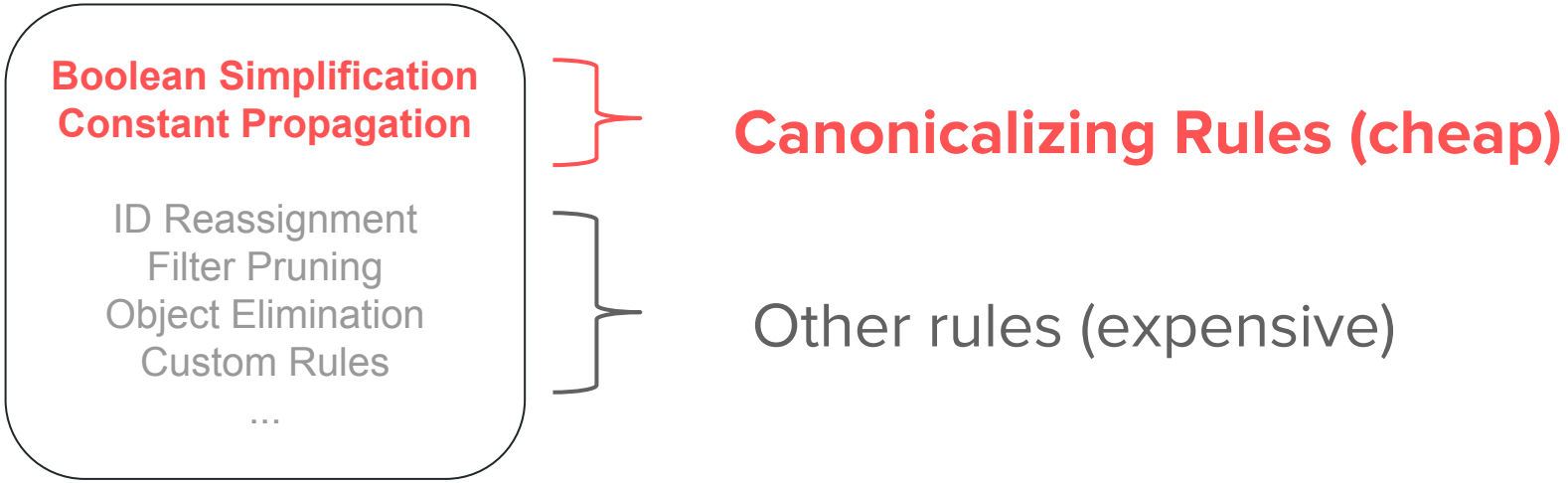


Optimization-first pipeline (slow!)

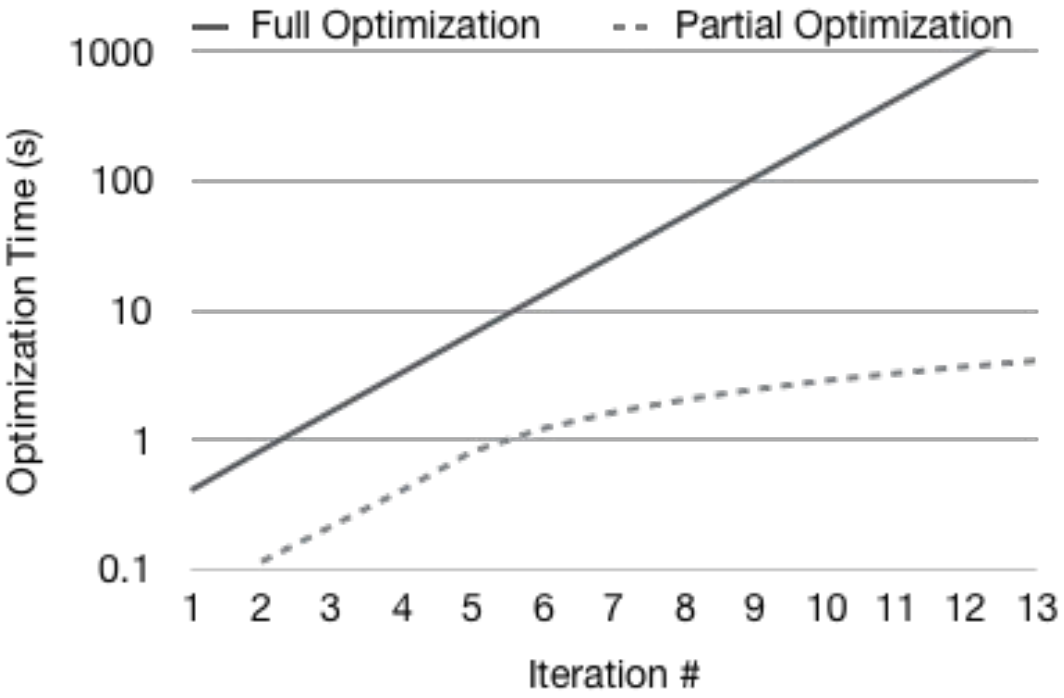


Insight: not all optimizations help caching!

Partial Optimization



Partial Optimization Scales

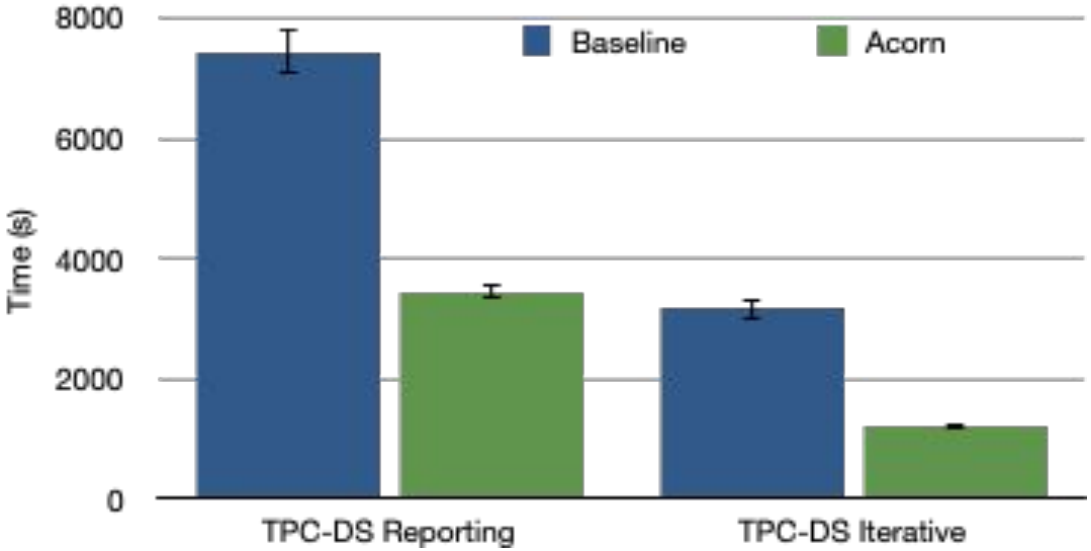


log scale!

Keep the canonicalizing benefits of optimization without the price

Partial Optimization Uses Cache More

- TPC-DS benchmark
- iterative style queries
- scaled to 100GB on a 16 machine cluster
- **Performance improved by 2.2X**



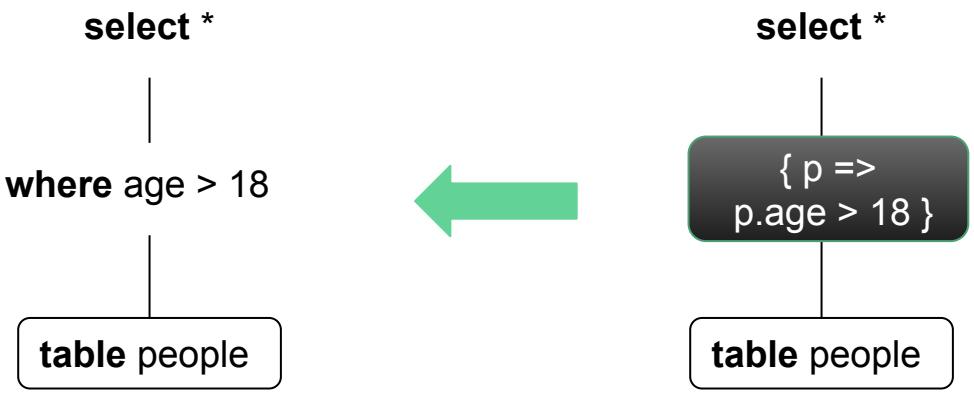
Two Challenges

1. Pipeline introduces obstacles for effective caching
   - Cache compares unoptimized instead of optimized plans
   - unoptimized == uncanonicalized

2. **UDFs**
   - **prevent high coverage**
   - **blackboxes to optimizers**

UDFs

UDFs are blackboxes that hide caching opportunities



UDF Translation

Program Synthesis	User Annotation	Froid	Acorn

UDF Translation

	Program Synthesis	User Annotation	Froid	Acorn
Correct	✓	✓	✓	✓

UDF Translation

	Program Synthesis	User Annotation	Froid	Acorn
Correct	✓	✓	✓	✓
Transparent	✓	X	✓	✓

UDF Translation

	Program Synthesis	User Annotation	Froid	Acorn
Correct	✓	✓	✓	✓
Transparent	✓	X	✓	✓
General (Java, Scala)	✓	X	X	✓

UDF Translation

	Program Synthesis	User Annotation	Froid	Acorn
Correct	✓	✓	✓	✓
Transparent	✓	X	✓	✓
General (Java, Scala)	✓	X	X	✓
Fast	X	✓	✓	✓

UDF Translation

- arbitrarily long or complex
- user-defined types (classes)
- anonymous functions

translate via symbolic execution

Scala | **Native Spark**

TPC-H

```
def q6cond(shipDate: Long,
    disc: Double, qty: Int) =
{
  val d1 = 757468799
  vale d2 = 31536000
  if ( shipDate < d1
    && shipdate >= d1 + d2
    && qty >= 24)
        return false
  val epsilon = .01
  val dec = .06
  var lower = dec - epsilon
  var upper= dec + epsilon
  return discount >= lower
    && discount <= upper
}
```

```
If(LessThan(shipDate,  757468799),
If(GreaterThanOrEqual(
shipdate, Add(757468799,
31536000)),
If(GreaterThanOrEqual(qty,
Literal(24)), If
(GreaterThanOrEqual(discount,
Subtract(Literal(.01),
Literal(.06))),
If(LessThanOrEqual(discount,
Add(Literal(.01), Literal(.06))),
true, false), false), false),
false, false)
```

Open Source

```
_.births > 100
```

```
GreaterThan(StructField("numBirth
s", IntegerType), Literal(100,
IntegerType)
```

Step 1: Translate to an IR

`person.filter(p => p.age > 18)`

```
 1  aload_1
 2  invokeinterface
 3  dload_1
 4  ldc2_w
 5  dcmpg
 6  ifge 18
 7  iconst_1
 8  goto 10
 9  iconst_0
10  aload_0
11  aload_1
```

```
1  Person r1 := @param0
2  double $d0 = r1.age()
3  int $d1 = 18
4  if $d0 < $d1
5  goto 8
6  boolean $zo = 1
7  goto 9
8  $zo = 0
9  return $zo
```

Step 2: Symbolic Execution

```
1  Person r1 := @param0
2  double $d0 = r1.age()
3  int $d1 = 18
4  if $d0 > $d1
5  goto 8
6  boolean $zo = 1
7  goto 9
8  $zo = 0
9  return $zo
```

Name	Type	Expression
r1	class[Person]	this

Step 2: Symbolic Execution

```
1  Person r1 := @param0
2  double $d0 = r1.age()
3  int $d1 = 18
4  if $d0 > $d1
5  goto 8
6  boolean $zo = 1
7  goto 9
8  $zo = 0
9  return $zo
```

Name	Type	Expression
r1	class[Person]	this
d0	double	Attribute("age")

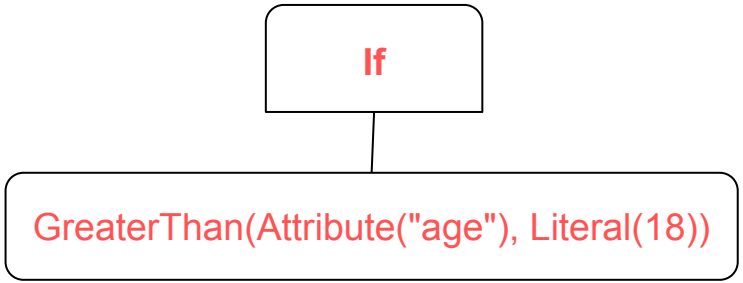
Step 2: Symbolic Execution

```
1  Person r1 := @param0
2  double $d0 = r1.age()
3  int $d1 = 18
4  if $d0 > $d1
5  goto 8
6  boolean $zo = 1
7  goto 9
8  $zo = 0
9  return $zo
```

Name	Type	Expression
r1	class[Person]	this
d0	double	Attribute("age")
d1	**int**	**Literal(18)**

Step 2: Symbolic Execution

```
1  Person r1 := @param0
2  double $d0 = r1.age()
3  int $d1 = 18
4  if $d0 > $d1
5  goto 8
6  boolean $zo = 1
7  goto 9
8  $zo = 0
9  return $zo
```

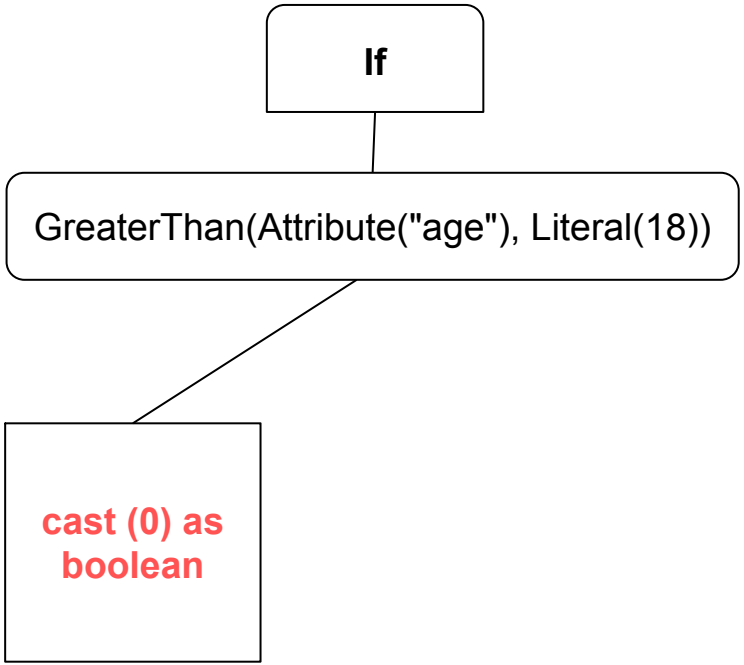


Name	Type	Expression
r1	class[Person]	this
d0	double	Attribute("age")
d1	int	Literal(18)

Step 2: Symbolic Execution

```
1  Person r1 := @param0
2  double $d0 = r1.age()
3  int $d1 = 18
4  if $d0 > $d1
5  goto 8
6  boolean $zo = 1
7  goto 9
8  $zo = 0
9  return $zo
```

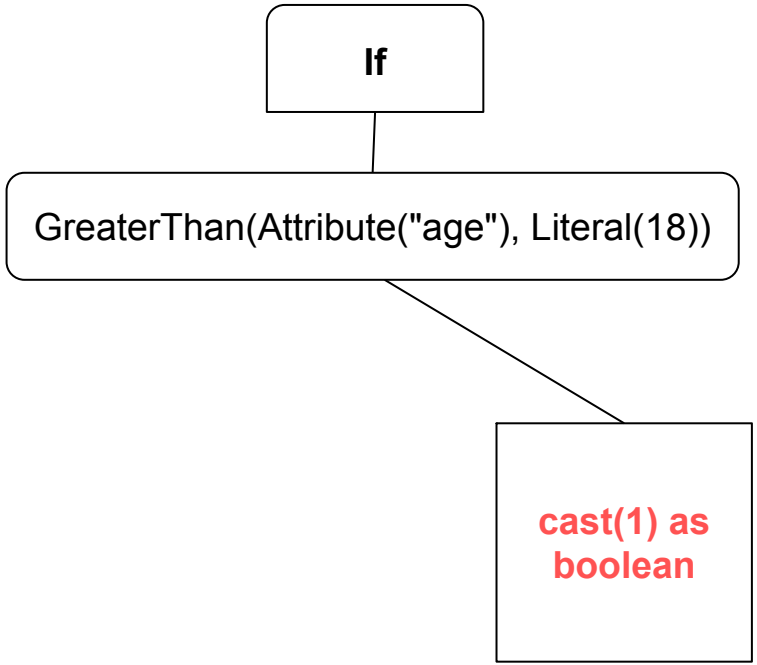
Name	Type	Expression
r1	class[Person]	this
d0	double	Attribute("age")
d1	int	Literal(18)



Step 2: Symbolic Execution

```
1  Person r1 := @param0
2  double $d0 = r1.age()
3  int $d1 = 18
4  if $d0 > $d1
5  goto 8
6  boolean $zo = 1
7  goto 9
8  $zo = 0
9  return $zo
```

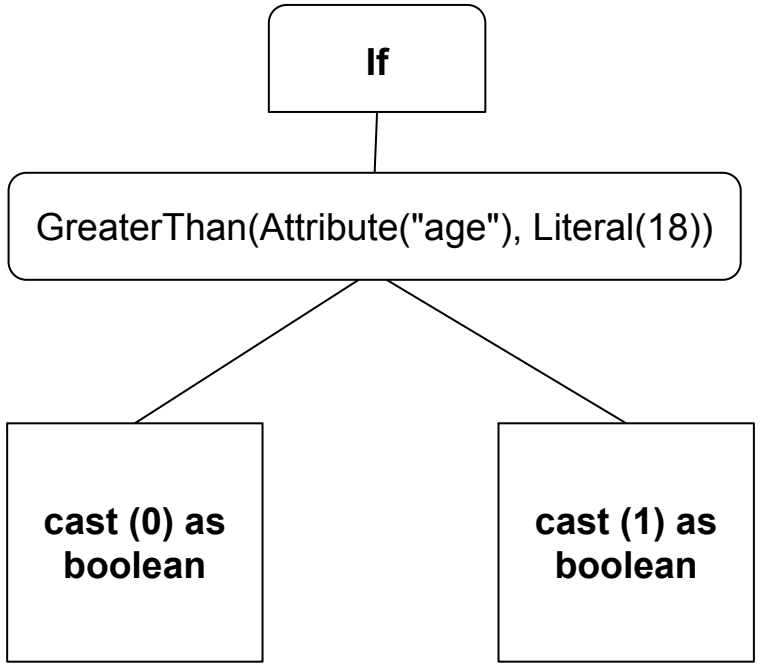
Name	Type	Expression
r1	class[Person]	this
d0	double	Attribute("age")
d1	int	Literal(18)



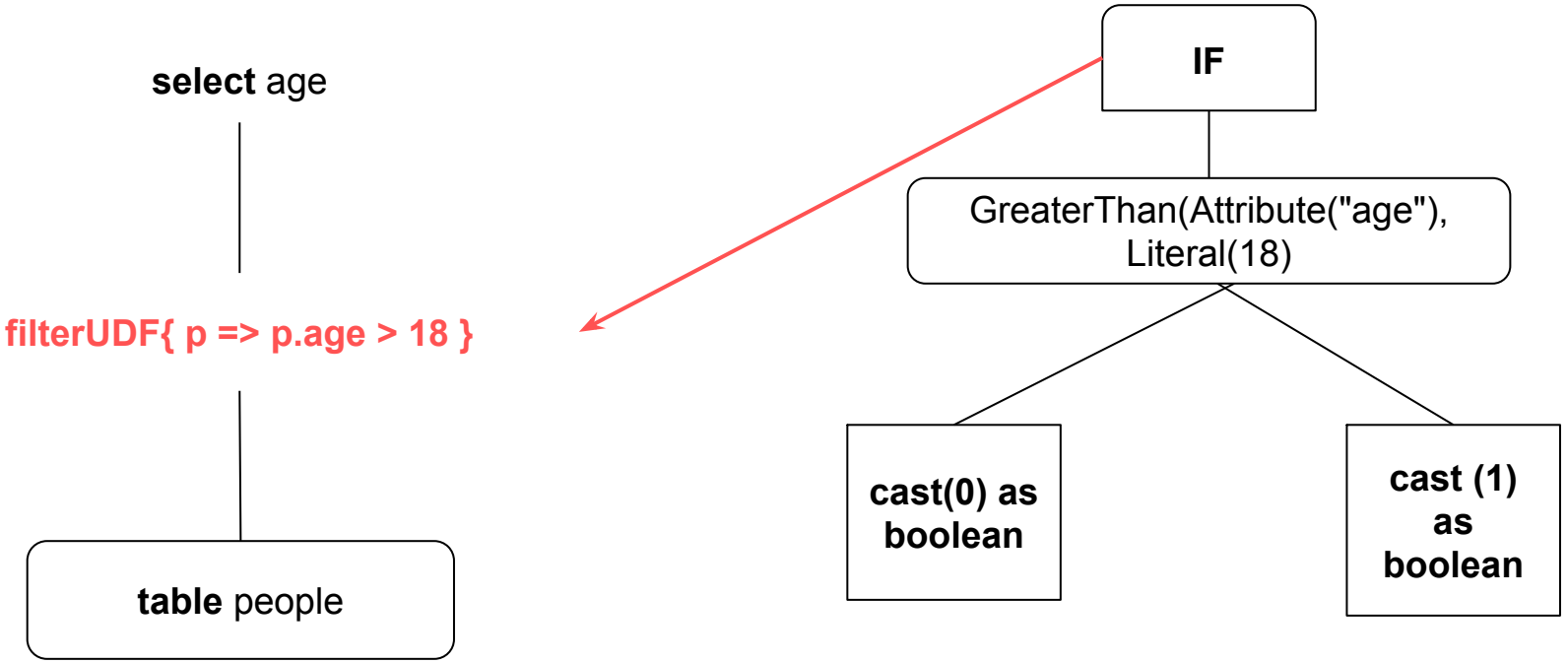
Step 2: Symbolic Execution

```
1  Person r1 := @param0
2  double $d0 = r1.age()
3  int $d1 = 18
4  if $d0 > $d1
5  goto 8
6  boolean $zo = 1
7  goto 9
8  $zo = 0
9  return $zo
```

Name	Type	Expression
r1	class[Person]	this
d0	double	Attribute("age")
d1	int	Literal(18)

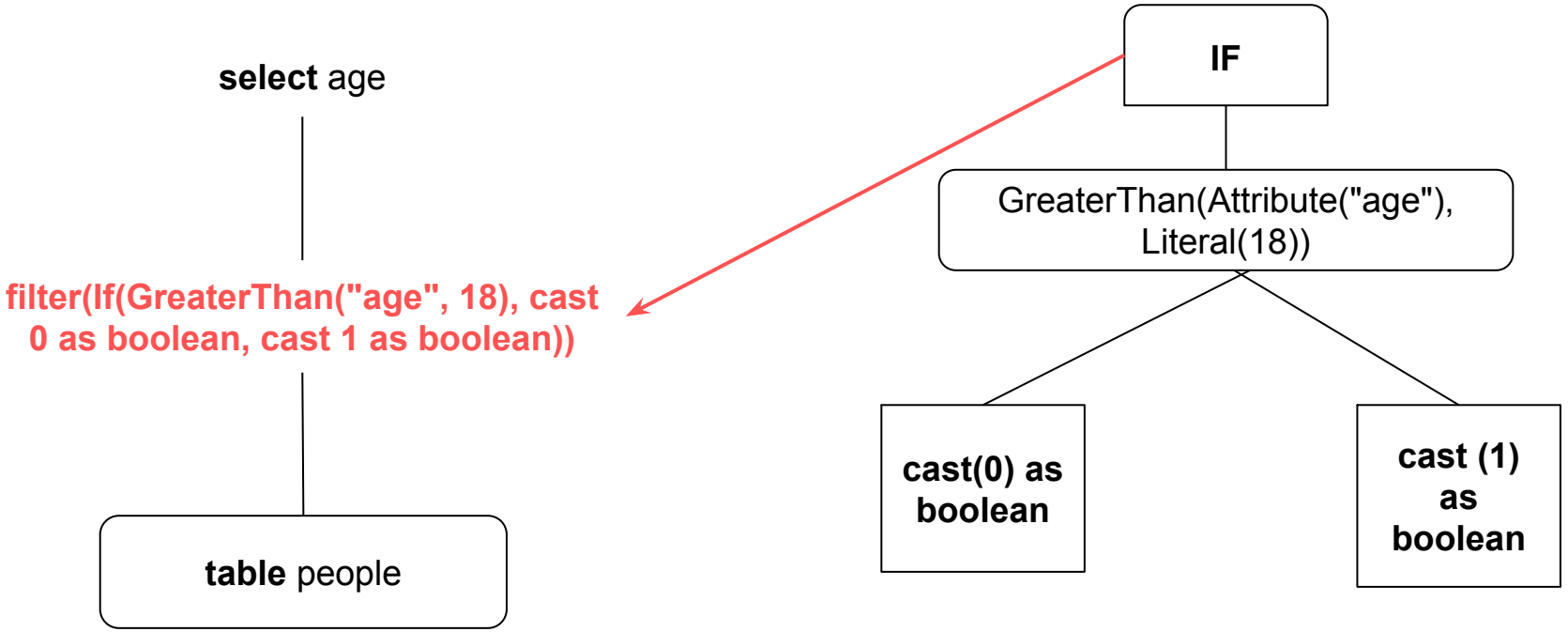


Step 3: Rewriting



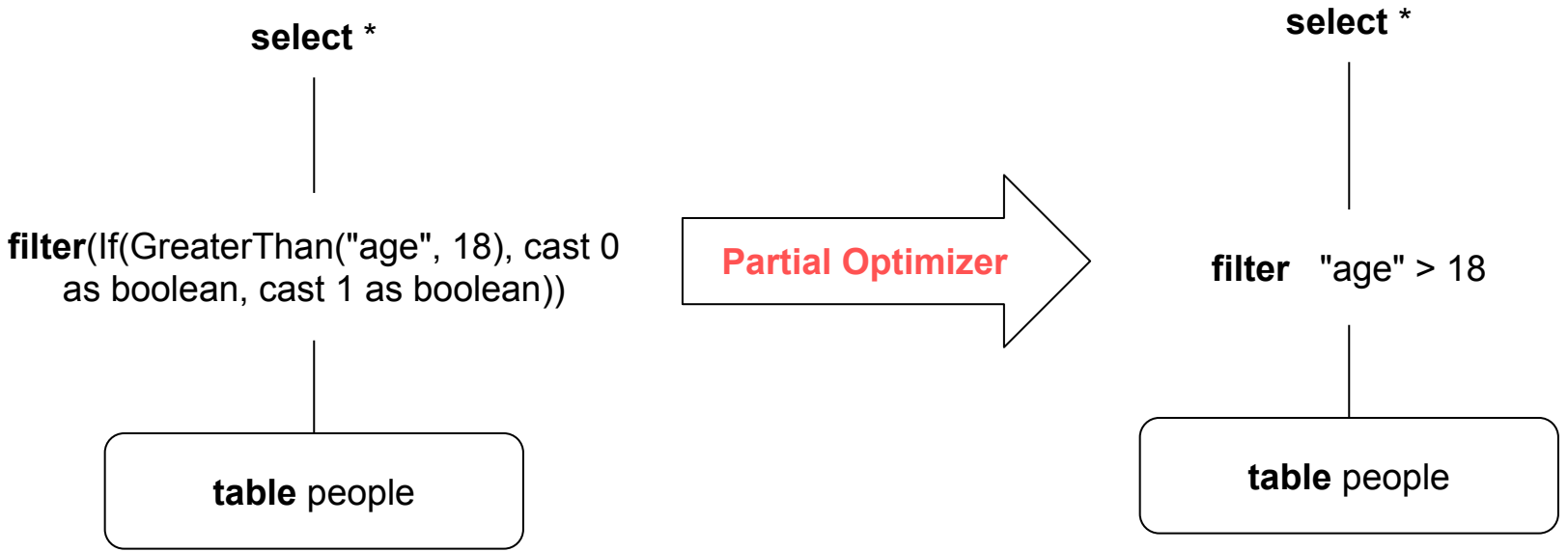

```
person.filter(p => p.age > 18)
```

Step 3: Rewriting




```
person.filter(p => p.age > 18)
```

Step 3: Rewriting




```
person.filter(p => p.age > 18)
```

```
person.filter(age > 18)
```

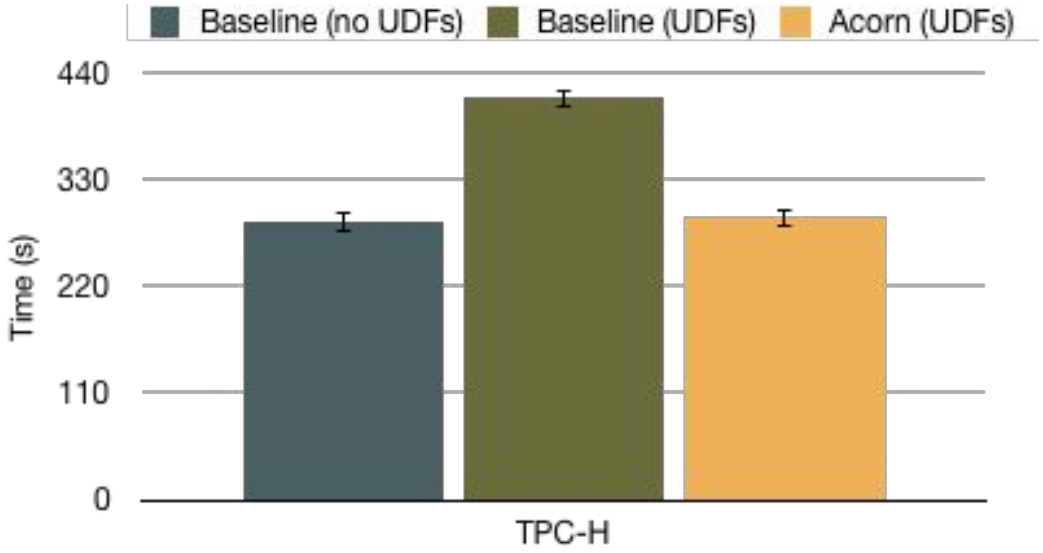
UDF Translation

Inserted UDFs into TPC-H

No caching opportunities

Same benchmark used by Microsoft's Froid

UDF translation is faithful to native SQL



partial optimization + udf translation

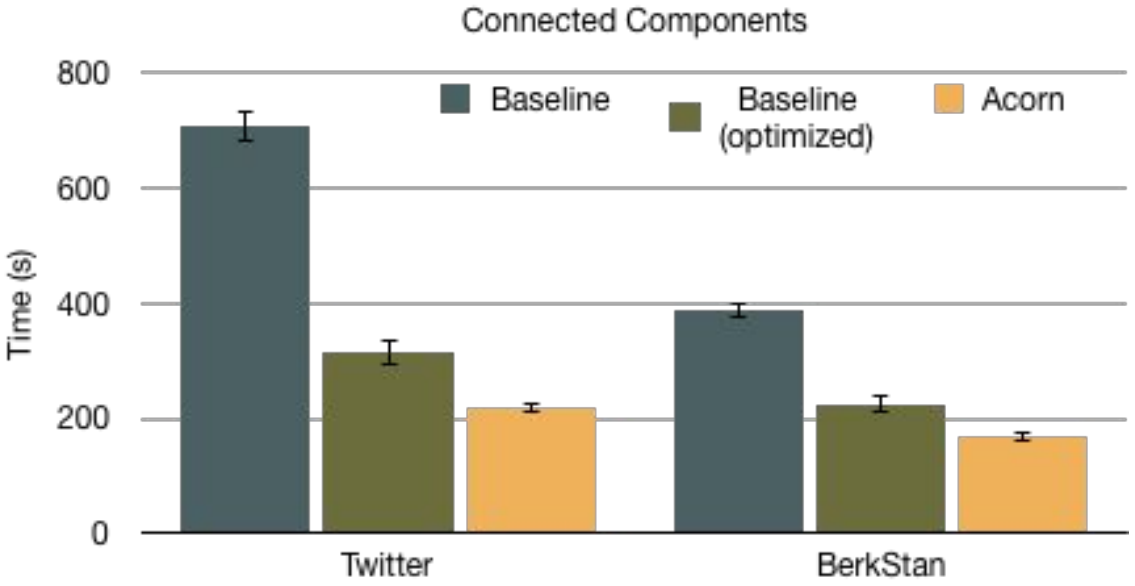
Open Source Applications: Caching+UDFs

connected components on BerkStan and Twitter networks

contains UDFs and iteration

3X improvement over the baseline

hand-optimized workload, 1.4X improvement over baseline



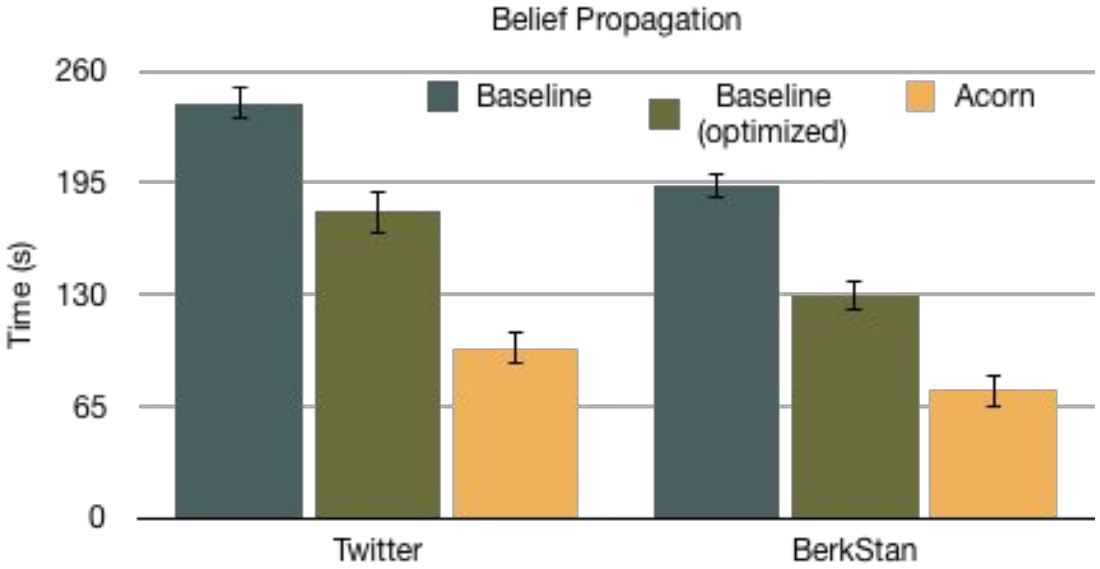
Open Source Applications: Caching+UDFs

belief propagation on BerkStan and Twitter networks

contains UDFs and iteration

3X improvement over the baseline

hand-optimized workload, 1.4X improvement over baseline



Acorn

Aggressive caching in big data systems

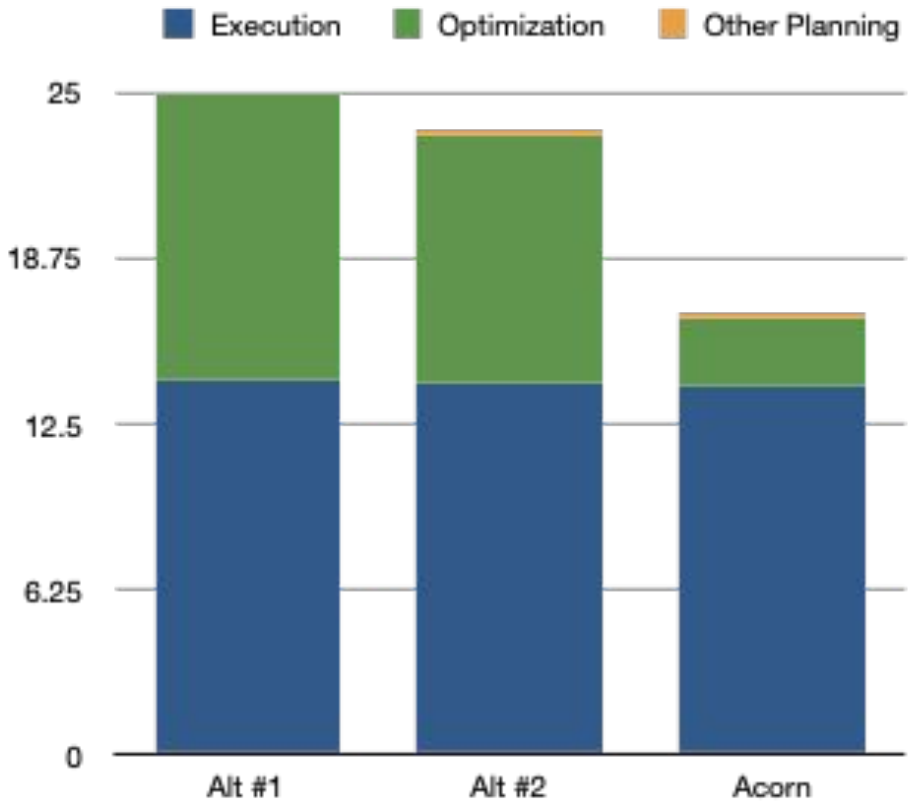
First Java/Scala UDF ➔ SQL Translator

Integrated into Spark 2.3.2

lana@cs.ucla.edu

Comparing Pipelines

- comparing proposed pipelines on TPC-DS benchmark
- time each stage in the pipeline
- **Acorn pipeline minimizes time spent in optimizer without impacting execution time**



Translation Limits

- Inherent limitation
  - unbounded loops (including recursion)
  - bounded loops (ex: foreach) are ok
- Limited by target language
- Non-determinism