

# Query Execution

## Overview

Relational operators

Execution methods

# Query Execution Overview

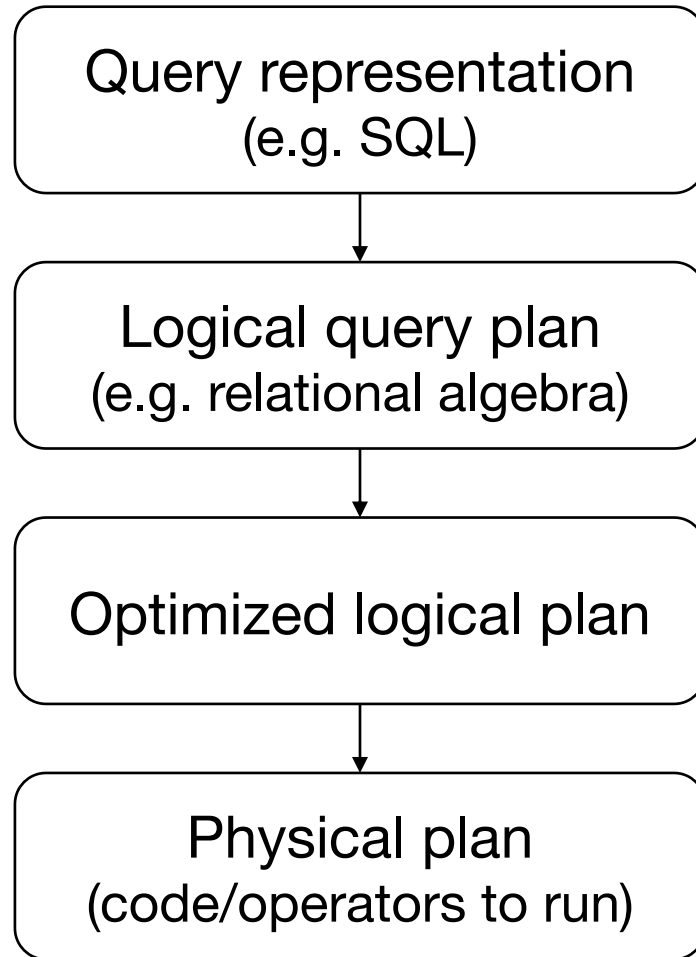
Recall that one of our key principles in data intensive systems was **declarative APIs**

» Specify what you want to compute, not how

We saw how these can translate into many storage strategies

How to execute queries in a declarative API?

# Query Execution Overview



Many execution methods: per-record exec, vectorization, compilation

# Plan Optimization Methods

**Rule-based:** systematically replace some expressions with other expressions

- » Replace  $X \text{ OR } \text{TRUE}$  with  $\text{TRUE}$
- » Replace  $M * A + M * B$  with  $M * (A + B)$  for matrices

**Cost-based:** propose several execution plans and pick best based on a **cost model**

**Adaptive:** update execution plan at runtime

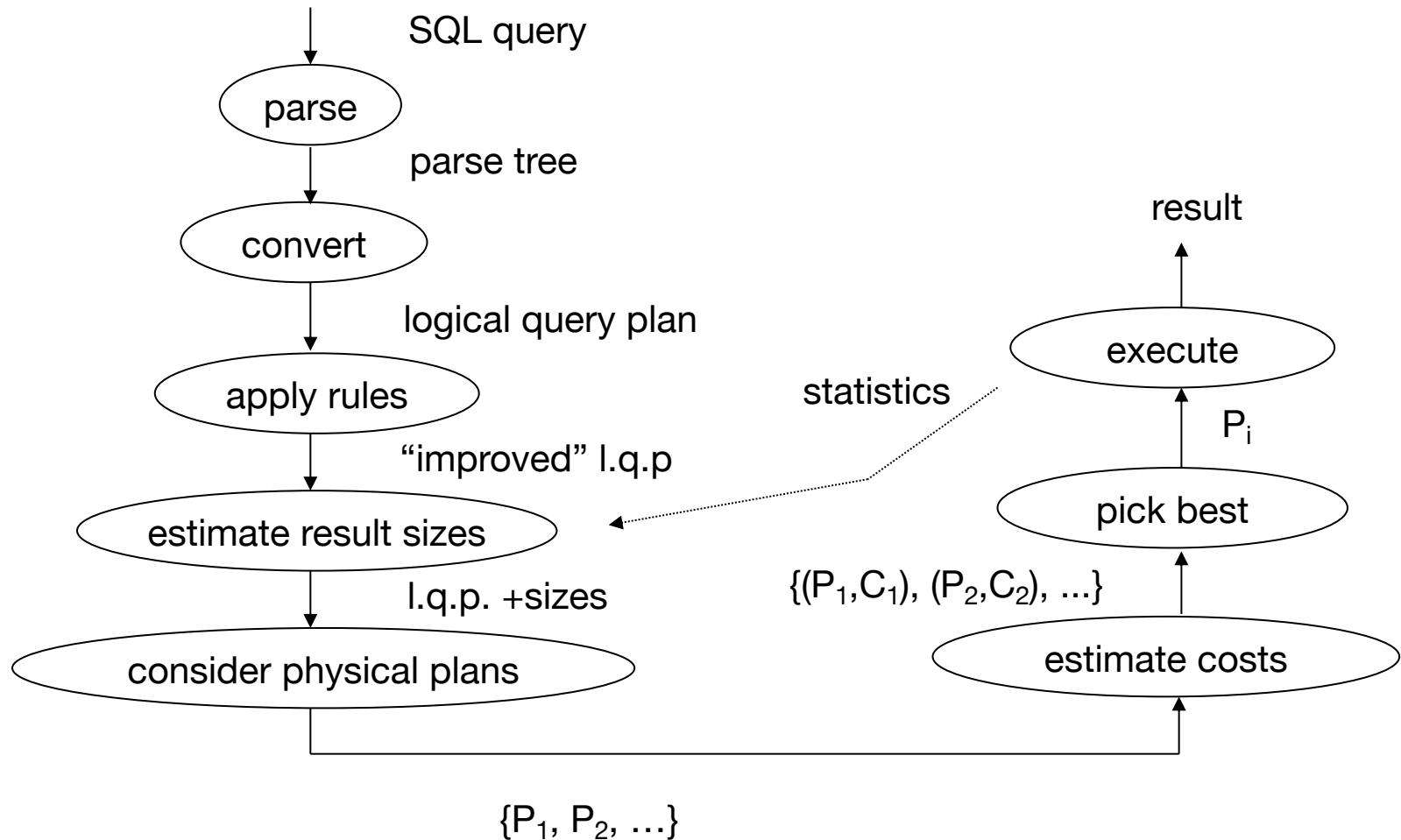
# Execution Methods

**Interpretation:** walk through query plan operators for each record

**Vectorization:** walk through in batches

**Compilation:** generate code (like System R)

# Typical RDBMS Execution



# Query Execution

Overview

Relational operators

Execution methods

# The Relational Algebra

Collection of operators over tables (relations)

» Each table has named attributes (fields)

Codd's original RA: tables are **sets of tuples** (unordered and tuples cannot repeat)

SQL's RA: tables are **bags (multisets) of tuples**; unordered but each tuple may repeat



# Relational Algebra Operators

Basic set operators:

**Intersection:**  $R \cap S$

**Union:**  $R \cup S$

**Difference:**  $R - S$

for tables with same schema

**Cartesian Product:**  $R \times S \quad \{ (r, s) \mid r \in R, s \in S \}$

# Relational Algebra Operators

Basic set operators:

**Intersection:**  $R \cap S$

**Union:**  $R \cup S$   consider both distinct (set union)  
and non-distinct (bag union)

**Difference:**  $R - S$

**Cartesian Product:**  $R \times S$

# Relational Algebra Operators

Special query processing operators:

**Selection:**  $\sigma_{\text{condition}}(R)$      $\{ r \in R \mid \text{condition}(r) \text{ is true} \}$

**Projection:**  $\Pi_{\text{expressions}}(R)$      $\{ \text{expressions}(r) \mid r \in R \}$

**Natural Join:**  $R \bowtie S$      $\{ (r, s) \in R \times S \mid r.\text{key} = s.\text{key} \}$   
where key is the common fields

# Relational Algebra Operators

Special query processing operators:

**Aggregation:**  $\text{keys } G_{\text{agg}(\text{attr})}(R)$       `SELECT agg(attr)  
FROM R  
GROUP BY keys`

**Examples:**       $\text{department } G_{\text{Max}(\text{salary})}(\text{Employees})$   
  
 $G_{\text{Max}(\text{salary})}(\text{Employees})$

# Algebraic Properties

Many properties about which combinations of operators are equivalent

» That's why it's called an algebra!

# Properties: Unions, Products and Joins

$$R \cup S = S \cup R$$

Tuple order in a relation  
doesn't matter (unordered)

$$R \cup (S \cup T) = (R \cup S) \cup T$$

$$R \times S = S \times R$$

Attribute order in a relation  
doesn't matter either

$$(R \times S) \times T = R \times (S \times T)$$

$$R \bowtie S = S \bowtie R$$

$$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

# Properties: Selects

$$\sigma_{p \wedge q}(R) =$$

$$\sigma_{p \vee q}(R) =$$

# Properties: Selects

$$\sigma_{p \wedge q}(R) = \sigma_p(\sigma_q(R))$$

$$\sigma_{p \vee q}(R) = \sigma_p(R) \cup \sigma_q(R)$$



careful with repeated elements



# Bags vs. Sets

$$R = \{a, a, b, b, b, c\}$$

$$S = \{b, b, c, c, d\}$$

$$R \cup S = ?$$

# Bags vs. Sets

$$R = \{a, a, b, b, b, c\}$$

$$S = \{b, b, c, c, d\}$$

$$R \cup S = ?$$

- Option 1: SUM of counts

$$R \cup S = \{a, a, b, b, b, b, b, c, c, c, d\}$$

- Option 2: MAX of counts

$$R \cup S = \{a, a, b, b, b, c, c, d\}$$

# Executive Decision

Use “SUM” option for bag unions

Some rules that work for set unions cannot be used for bags

# Properties: Project

Let:  $X$  = set of attributes

$Y$  = set of attributes

$$\Pi_{X \cup Y}(R) =$$

# Properties: Project

Let:  $X$  = set of attributes

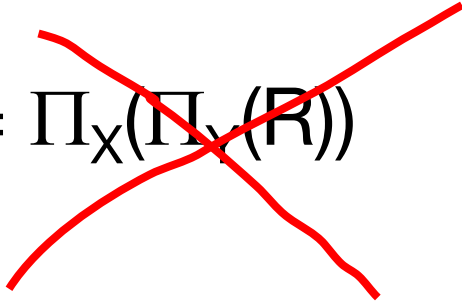
$Y$  = set of attributes

$$\Pi_{X \cup Y}(R) = \Pi_X(\Pi_Y(R))$$

# Properties: Project

Let:  $X$  = set of attributes

$Y$  = set of attributes

$$\Pi_{X \cup Y}(R) = \Pi_X(\Pi_Y(R))$$


# Properties: $\sigma$ + $\bowtie$

Let  $p$  = predicate with only  $R$  attribs

$q$  = predicate with only  $S$  attribs

$m$  = predicate with only  $R, S$  attribs

$$\sigma_p(R \bowtie S) =$$

$$\sigma_q(R \bowtie S) =$$

# Properties: $\sigma$ + $\bowtie$

Let  $p$  = predicate with only  $R$  attribs

$q$  = predicate with only  $S$  attribs

$m$  = predicate with only  $R, S$  attribs

$$\sigma_p(R \bowtie S) = \sigma_p(R) \bowtie S$$

$$\sigma_q(R \bowtie S) = R \bowtie \sigma_q(S)$$



# Properties: $\sigma$ + $\bowtie$

Some rules can be derived:

$$\sigma_{p \wedge q}(R \bowtie S) =$$

$$\sigma_{p \wedge q \wedge m}(R \bowtie S) =$$

$$\sigma_{p \vee q}(R \bowtie S) =$$

# Properties: $\sigma$ + $\bowtie$

Some rules can be derived:

$$\sigma_{p \wedge q}(R \bowtie S) = \sigma_p(R) \bowtie \sigma_q(S)$$

$$\sigma_{p \wedge q \wedge m}(R \bowtie S) = \sigma_m(\sigma_p(R) \bowtie \sigma_q(S))$$

$$\sigma_{p \vee q}(R \bowtie S) = (\sigma_p(R) \bowtie S) \cup (R \bowtie \sigma_q(S))$$

# Prove One, Others for Practice

$$\begin{aligned}\sigma_{p \wedge q}(R \bowtie S) &= \sigma_p(\sigma_q(R \bowtie S)) \\ &= \sigma_p(R \bowtie \sigma_q(S)) \\ &= \sigma_p(R) \bowtie \sigma_q(S)\end{aligned}$$

# Properties: $\Pi + \sigma$

Let  $x$  = subset of  $R$  attributes

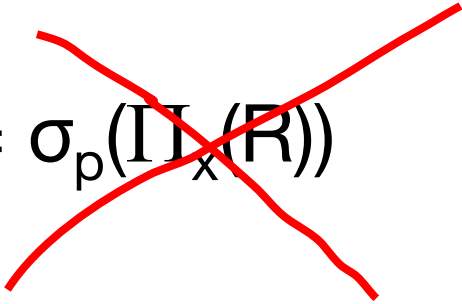
$z$  = attributes in predicate  $p$   
(subset of  $R$  attributes)

$$\Pi_x(\sigma_p(R)) =$$

# Properties: $\Pi + \sigma$

Let  $x$  = subset of  $R$  attributes

$z$  = attributes in predicate  $p$   
(subset of  $R$  attributes)

$$\Pi_x(\sigma_p(R)) = \sigma_p(\Pi_x(R))$$


# Properties: $\Pi + \sigma$

Let  $x$  = subset of  $R$  attributes

$z$  = attributes in predicate  $p$   
(subset of  $R$  attributes)

$$\Pi_x(\sigma_p(R)) = \Pi_x(\sigma_p(\Pi_{x \cup z}(R)))$$

# Properties: $\Pi$ + $\bowtie$

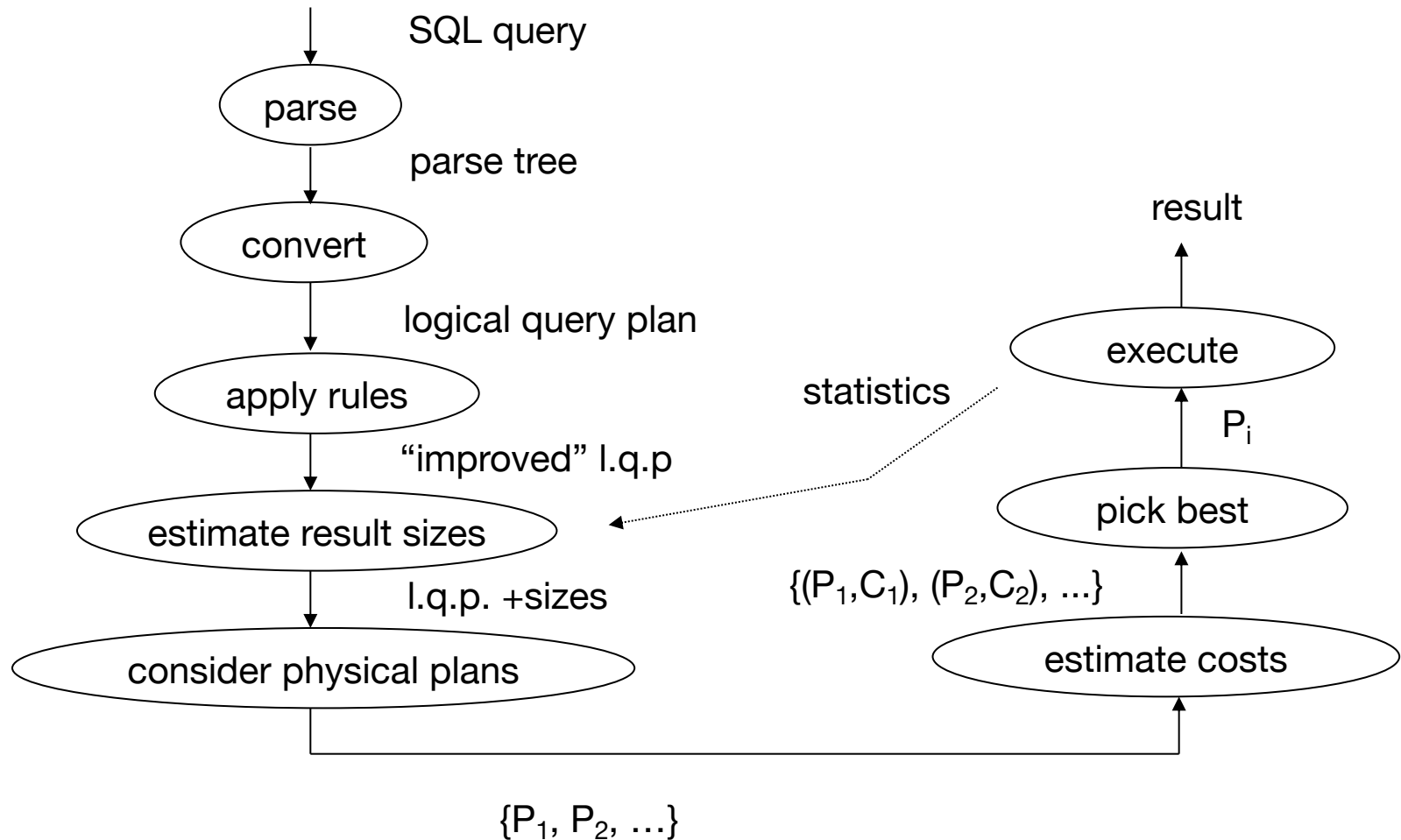
Let  $x$  = subset of  $R$  attributes

$y$  = subset of  $S$  attributes

$z$  = intersection of  $R, S$  attributes

$$\Pi_{x \cup y}(R \bowtie S) = \Pi_{x \cup y}((\Pi_{x \cup z}(R)) \bowtie (\Pi_{y \cup z}(S)))$$

# Typical RDBMS Execution



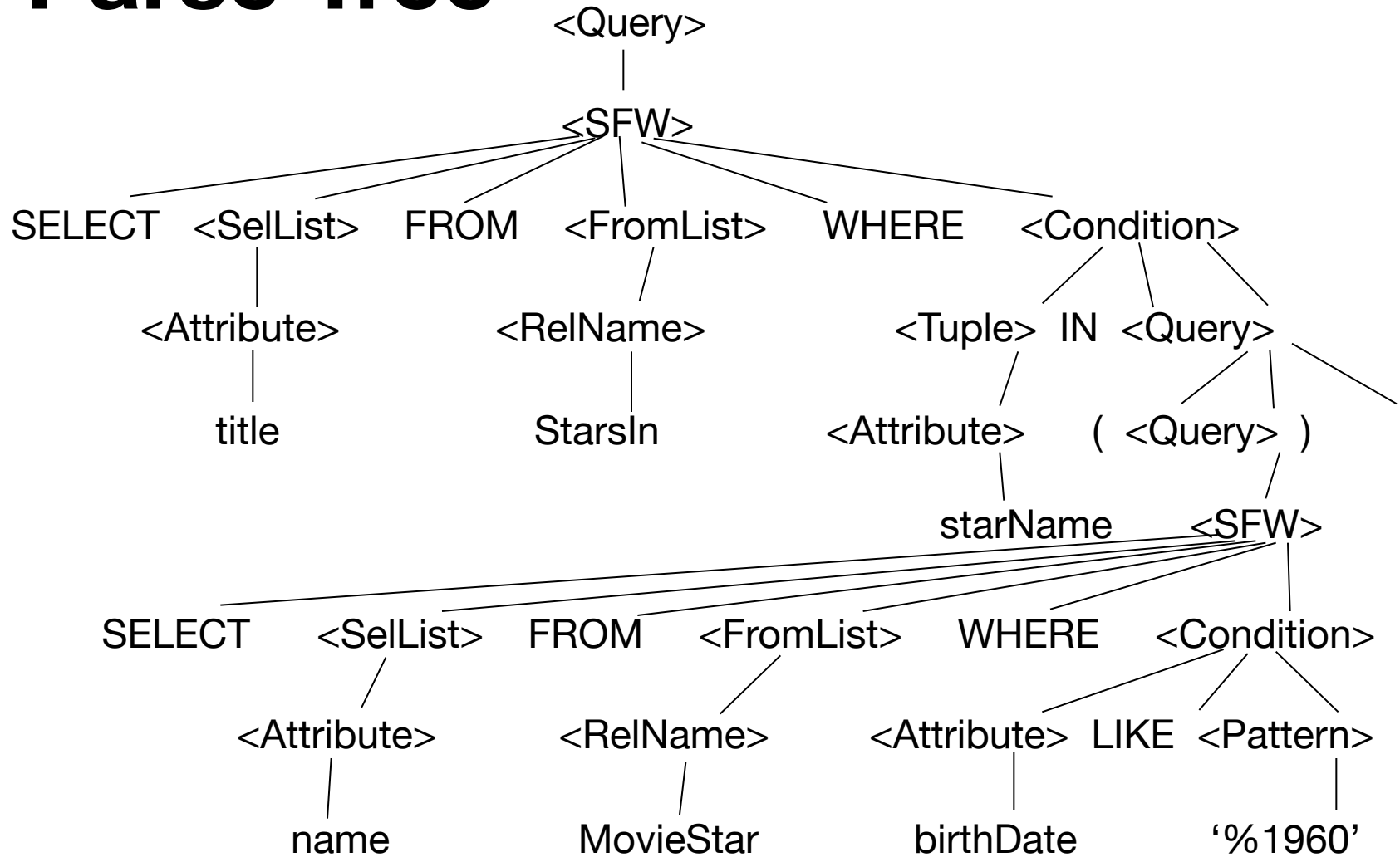


# Example SQL Query

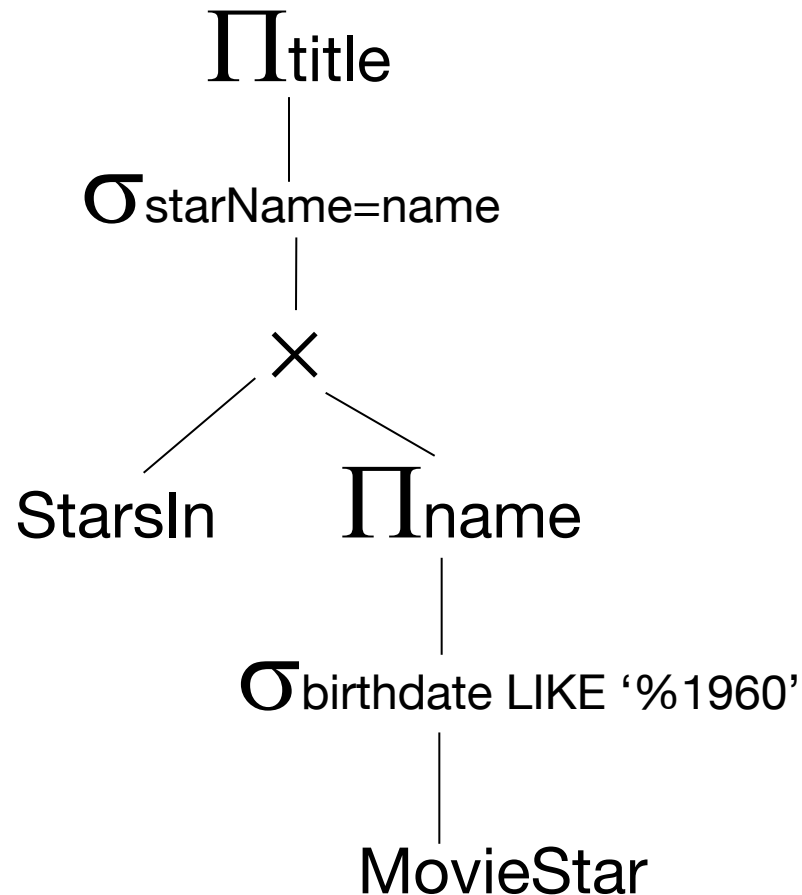
```
SELECT title
FROM StarsIn
WHERE starName IN (
    SELECT name
    FROM MovieStar
    WHERE birthdate LIKE '%1960'
);
```

(Find the movies with stars born in 1960)

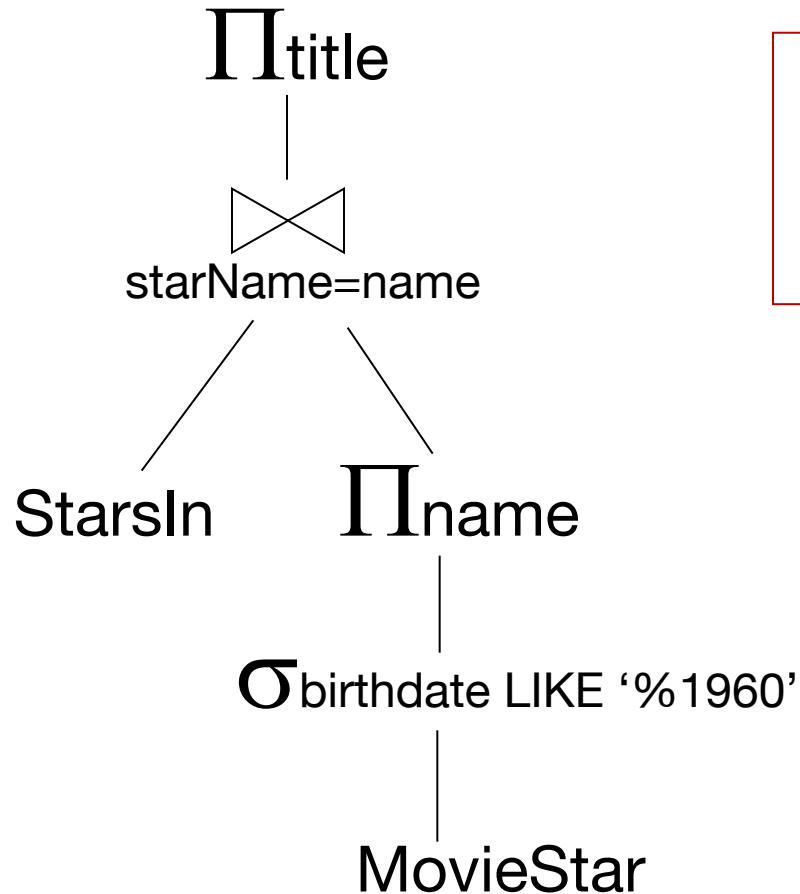
# Parse Tree



# Logical Query Plan

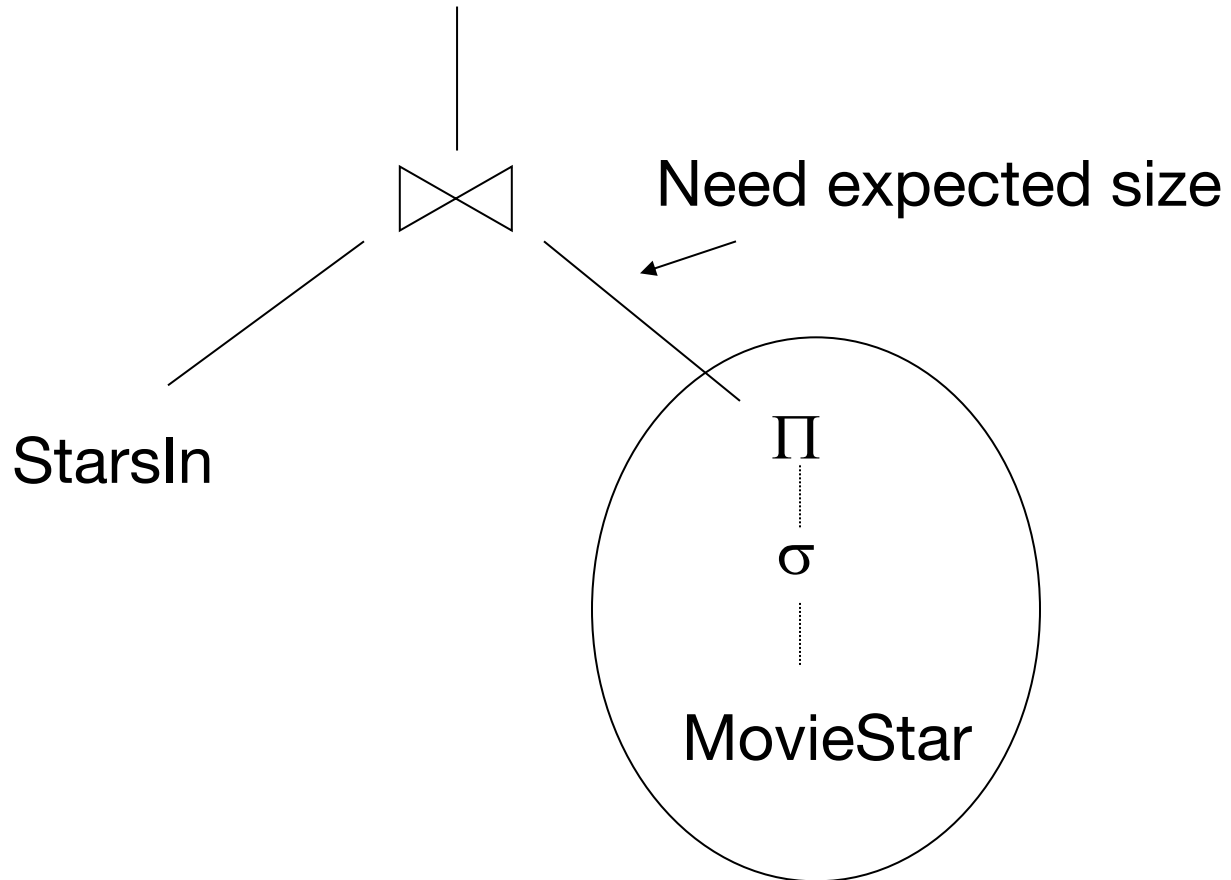


# Improved Logical Query Plan

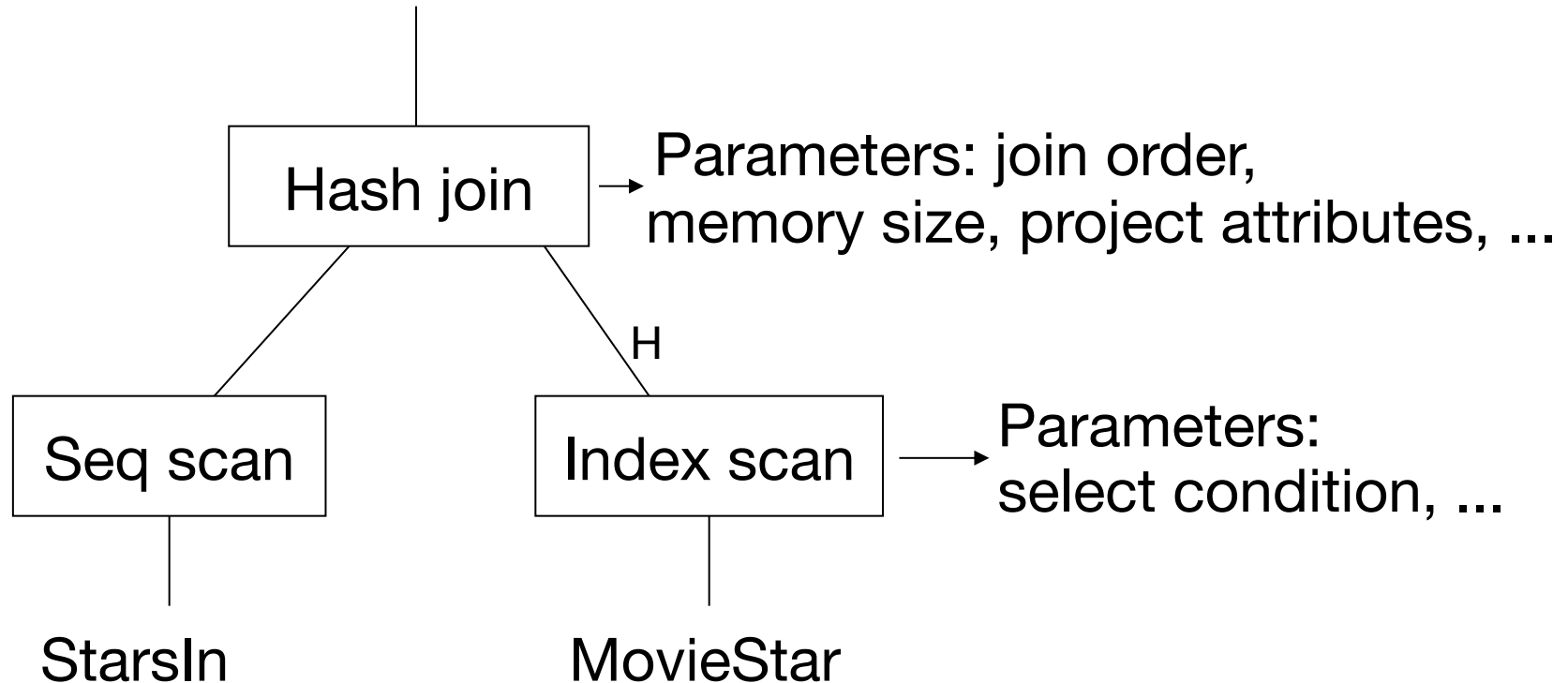


Question:  
Push  $\Pi_{\text{title}}$   
to StarsIn?

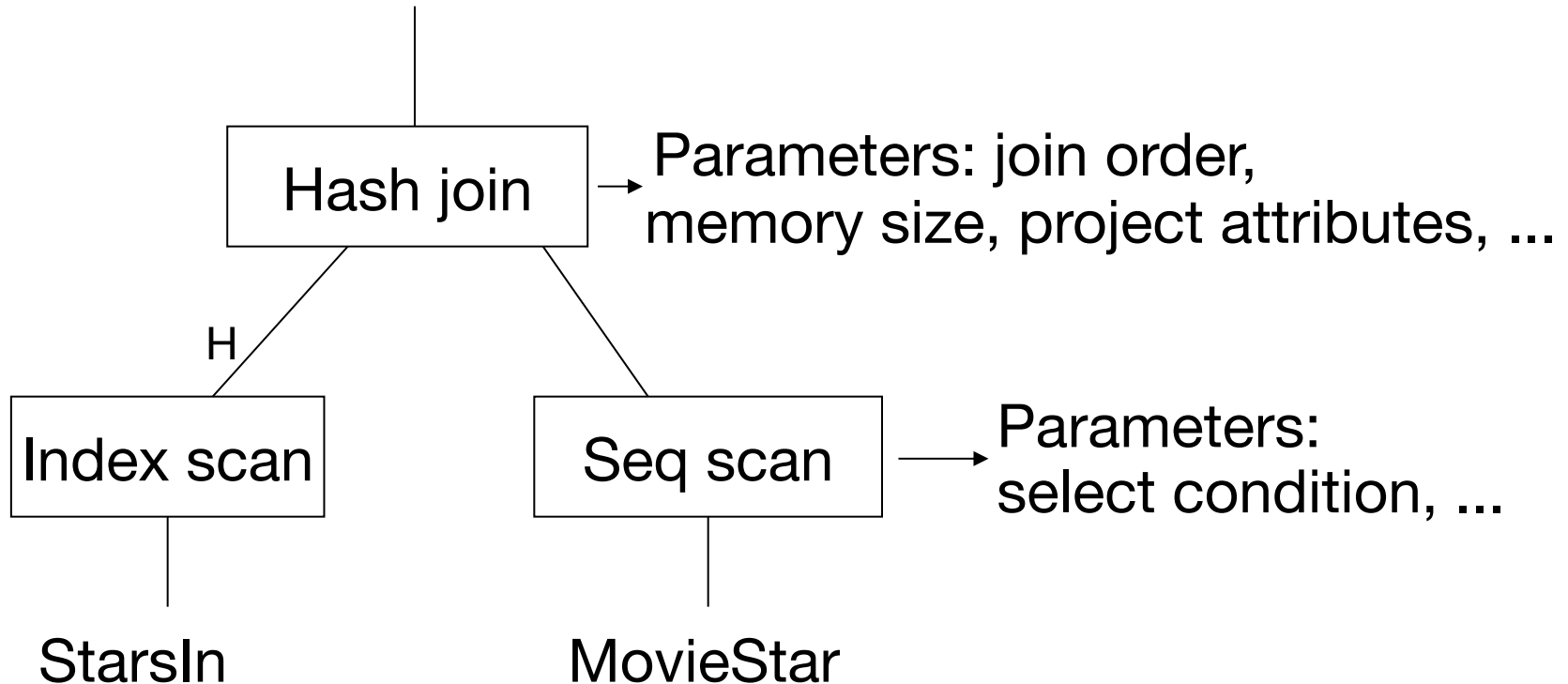
# Estimate Result Sizes



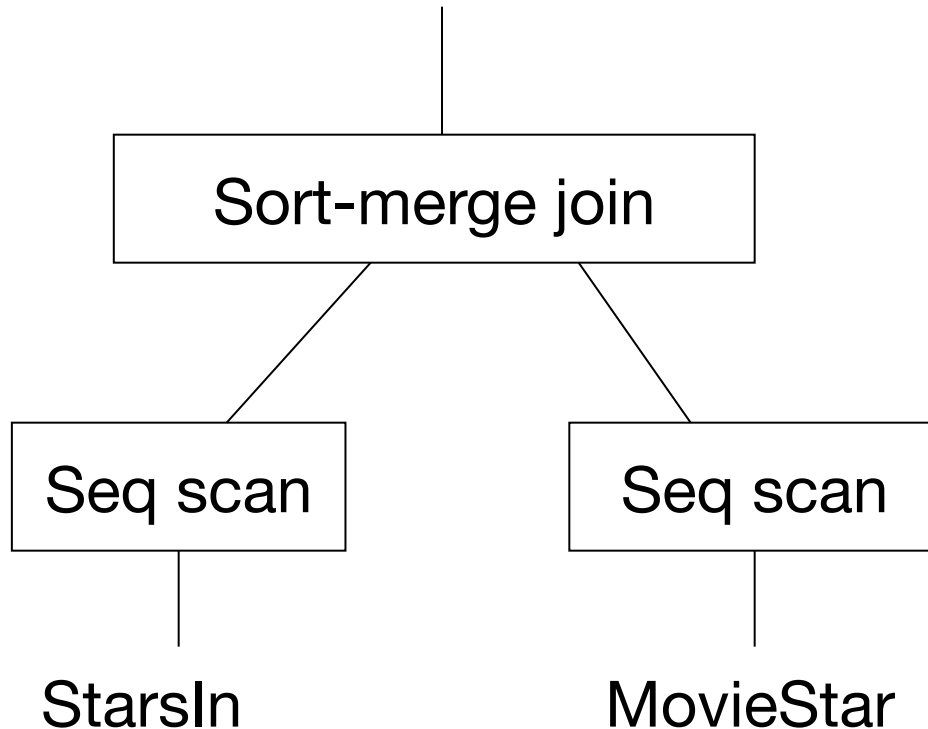
# One Physical Plan



# Another Physical Plan



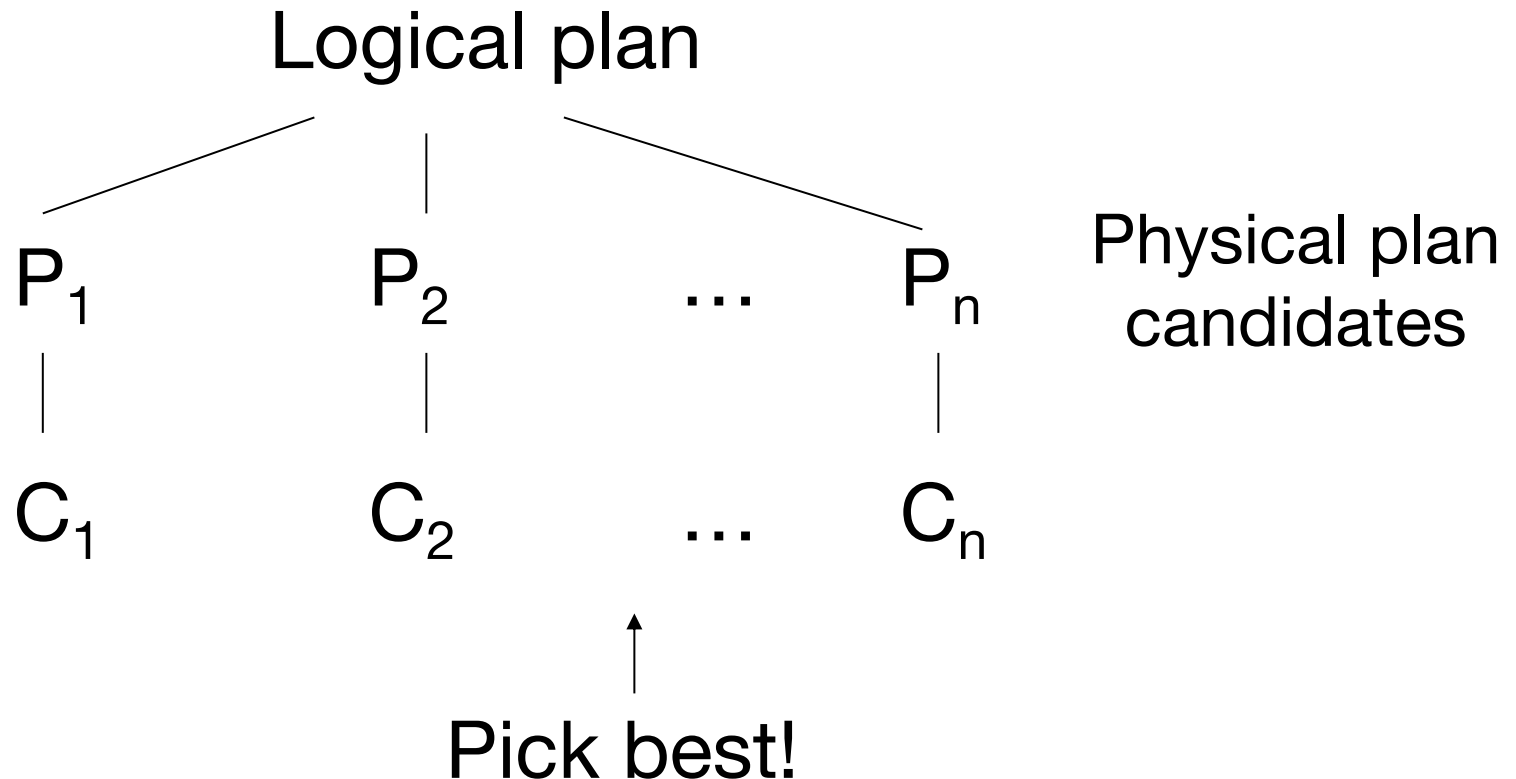
# Another Physical Plan



Which plan is likely to be better?



# Estimating Plan Costs



Covered in next few lectures!

# Query Execution

Overview

Relational operators

Execution methods

# **Now That We Have a Plan, How Do We Run it?**

Several different options that trade between complexity, setup time & performance

# Example: Simple Query

```
SELECT quantity * price  
  FROM orders  
 WHERE productId = 75
```

$$\Pi_{\text{quantity} * \text{price}} (\sigma_{\text{productId}=75} (\text{orders}))$$

# Method 1: Interpretation

```
interface Operator {  
    Tuple next();  
}
```

```
class TableScan: Operator {  
    String tableName;  
}
```

```
class Select: Operator {  
    Operator parent;  
    Expression condition;  
}
```

```
class Project: Operator {  
    Operator parent;  
    Expression[] exprs;  
}
```

```
interface Expression {  
    Value compute(Tuple in);  
}
```

```
class Attribute: Expression {  
    String name;  
}
```

```
class Times: Expression {  
    Expression left, right;  
}
```

```
class Equals: Expression {  
    Expression left, right;  
}
```

# Example Expression Classes

```
class Attribute: Expression {  
    String name;  
  
    Value compute(Tuple in) {  
        return in.getField(name);  
    }  
}
```

← probably better to use a  
numeric field ID instead

```
class Times: Expression {  
    Expression left, right;  
  
    Value compute(Tuple in) {  
        return left.compute(in) * right.compute(in);  
    }  
}
```

# Example Operator Classes

```
class TableScan: Operator {
    String tableName;

    Tuple next() {
        // read & return next record from file
    }
}
```

```
class Project: Operator {
    Operator parent;
    Expression[] exprs;

    Tuple next() {
        tuple = parent.next();
        fields = [expr.compute(tuple) for expr in exprs];
        return new Tuple(fields);
    }
}
```

# Running Our Query with Interpretation

```
ops = Project(  
    expr = Times(Attr("quantity"), Attr("price")),  
    parent = Select(  
        expr = Equals(Attr("productId"), Literal(75)),  
        parent = TableScan("orders")  
    )  
);
```

```
while(true) {  
    Tuple t = ops.next();  
    if (t != null) {  
        out.write(t);  
    } else {  
        break;  
    }  
}
```

recursively calls `Operator.next()`  
and `Expression.compute()`



Pros & cons of this  
approach?



# Method 2: Vectorization

Interpreting query plans one record at a time is simple, but it's too slow

- » Lots of virtual function calls and branches for each record (recall Jeff Dean's numbers)

Keep recursive interpretation, but make Operators and Expressions run on **batches**

# Implementing Vectorization

```
class TupleBatch {  
    // Efficient storage, e.g.  
    // schema + column arrays  
}
```

```
interface Operator {  
    TupleBatch next();  
}
```

```
class Select: Operator {  
    Operator parent;  
    Expression condition;  
}
```

...

```
class ValueBatch {  
    // Efficient storage  
}
```

```
interface Expression {  
    ValueBatch compute(  
        TupleBatch in);  
}
```

```
class Times: Expression {  
    Expression left, right;  
}
```

...

# Typical Implementation

Values stored in columnar arrays (e.g. `int[]`)  
with a separate bit array to mark nulls

Tuple batches fit in L1 or L2 cache

Operators use SIMD instructions to update  
both values and null fields without branching

# Pros & Cons of Vectorization

- + Faster than record-at-a-time if the query processes many records
- + Relatively simple to implement
- Lots of nulls in batches if query is selective
- Data travels between CPU & cache a lot

# Method 3: Compilation

Turn the query into executable code

# Compilation Example

$\Pi_{\text{quantity} * \text{price}} (\sigma_{\text{productId}=75} (\text{orders}))$



```
class MyQuery {  
    void run() {  
        Iterator<OrdersTuple> in = openTable("orders");  
        for(OrdersTuple t: in) {  
            if (t.productId == 75) {  
                out.write(Tuple(t.quantity * t.price));  
            }  
        }  
    }  
}
```

generated class with the right  
field types for orders table

Can also theoretically generate  
vectorized code

# Pros & Cons of Compilation

- + Potential to get fastest possible execution
- + Leverage existing work in compilers
- Complex to implement
- Compilation takes time
- Generated code may not match hand-written

# What's Used Today?

Depends on context & other bottlenecks

**Transactional databases (e.g. MySQL):**  
mostly record-at-a-time interpretation

**Analytical systems (Vertica, Spark SQL):**  
vectorization, sometimes compilation

**ML libs (TensorFlow):** mostly vectorization  
(the records *are* vectors!), some compilation