Goals

- 75% - Integrate with Postgres and execute simple queries
  - Implement base table sampling

- 100% - Dynamic programming approach
  - Tuple-based cost model

- 125% - Use sampling and static/dynamic replanning to select query plans
Goals

● 75% - Integrate with Postgres and execute simple queries ✓

   Implement base table sampling ✓

● 100% - Cost-guided Dynamic Programming Framework ✓

● 125% - Use sampling and static/dynamic replanning to select query plans
Cost-guided Dynamic Programming Framework

- Cascades-style incremental plan space exploration
  - Space of plans is explored on demand and only as needed
  - Both Logical and Physical transformations are applied at the same time
  - Alternative is a two-phase optimization
    - Generate entire plan space
    - Cost and choose most optimal physical plan

- Optimizer components
  - Plan Representation
  - Memo Table
    - Equivalence classes
  - Rule Interface
  - Pattern Matching
    - Binding traversal
  - Plan Exploration
Plan Representation

- Logical, Physical, and Expression operators
- Composed to create an operator tree
  - OpExpression represents a concrete plan
- Easily extensible with new types
  - Required input & output physical properties
  - Hash function
  - Equality

Credit: Orca Paper
Memo Table

- Recursive plan space exploration has redundant sub computations
  - Memo Table enables sub problem reuse throughout optimization
- Insertion of query into Memo creates an initial set of Groups
  - Equivalence classes for intermediate results
Memo Table

- As exploration of plan space proceeds, equivalent expressions are grouped together.

Credit: Orca Paper
Rule Interface

- Extensible rule interface
  - Rule implementer only provides
    - Pattern to match against
    - Validation function
    - Transformation function
- Decoupled from optimizer and exploration process

```
class Rule {
public:
    virtual ~Rule() {};
    std::shared_ptr<Pattern> GetMatchPattern() const { return match_pattern; }
    bool isPhysical() const { return physical; }
    bool isLogical() const { return logical; }
    virtual bool Check(std::shared_ptr<OpExpression> expr) const = 0;
    virtual void Transform(
        std::shared_ptr<OpExpression> input,
        std::vector<std::shared_ptr<OpExpression>> &transformed) const = 0;
protected:
    std::shared_ptr<Pattern> match_pattern;
    bool physical = false;
    bool logical = false;
};
```

```
void InnerJoinCommutativity::Transform(
    std::shared_ptr<OpExpression> input,
    std::vector<std::shared_ptr<OpExpression>> &transformed) const
{
    auto result_plan = std::make_shared<OpExpression>(LogicalInnerJoin::make());
    std::vector<std::shared_ptr<OpExpression>> children = input->Children();
    assert(children.size() == 3);
    result_plan->PushChild(children[0]);
    result_plan->PushChild(children[1]);
    result_plan->PushChild(children[2]);
    transformed.push_back(result_plan);
}
```
Pattern Matching

- InnerJoin
  - Leaf
  - Leaf

- Binding

Group 0
- Sort [Group 0]
- InnerJoin [Group 1, Group 2]

Group 1
- Scan [Table 1]
- Get [Table 1]

Group 2
- Scan [Table 2]
- Get [Table 2]
Pattern Matching

InnerJoin

Leaf
Leaf

Group 0

Sort
[Group 0]
InnerJoin
[Group 1, Group 2]

Group 1

Scan
[Table 1]
Get
[Table 1]

Group 2

Scan
[Table 2]
Get
[Table 2]
Pattern Matching

Group 0
- Sort [Group 0]
- InnerJoin [Group 1, Group 2]

Group 1
- Scan [Table 1]
- Get [Table 1]

Group 2
- Scan [Table 2]
- Get [Table 2]
Pattern Matching

Group 0
- Sort [Group 0]
- InnerJoin [Group 1, Group 2]

Group 1
- Scan [Table 1]
- Get [Table 1]

Group 2
- Scan [Table 2]
- Get [Table 2]
Pattern Matching

Group 0
- Sort [Group 0]
- InnerJoin [Group 1, Group 2]

Group 1
- Scan [Table 1]
- Get [Table 1]

Group 2
- Scan [Table 2]
- Get [Table 2]
Pattern Matching

Group 0
- Sort [Group 0]
- InnerJoin [Group 1, Group 2]

Group 1
- Scan [Table 1]
- Get [Table 1]

Group 2
- Scan [Table 2]
- Get [Table 2]

InnerJoin

Scan [Table 1]

Leaf

Leaf

Binding

InnerJoin
Pattern Matching

Group 0
- Sort [Group 0]
- InnerJoin [Group 1, Group 2]

Group 1
- Scan [Table 1]
- Get [Table 1]

Group 2
- Scan [Table 2]
- Get [Table 2]
Pattern Matching

Group 0
- Sort [Group 0]
- InnerJoin [Group 1, Group 2]

Group 1
- Scan [Table 1]
- Get [Table 1]

Group 2
- Scan [Table 2]
- Get [Table 2]

InnerJoin
- Scan [Table 2]

Leaf
- Leaf

Binding
- InnerJoin
- Leaf (Group 1)
Pattern Matching

![Diagram of pattern matching with nodes and edges representing InnerJoin, Sort, Get, Scan, and binding with Group 0, Group 1, and Group 2.]
Plan Exploration

- Series of individual tasks
  - Optimization
  - Exploration
  - Rule Application
  - Costing
- Kicks off by optimizing root group
  - Recursively optimize input groups for each operator variant
Demo
Retrospective

- Implementing the basic infrastructure was a significant undertaking
  - Synthesizing a concrete implementation from several decades of research
  - Designing extensible representations
  - Generic search process that is invariant of specific rules or operators
- Shuttling between Postgres, Peloton, and the optimizer representation
  - Converting from Postgres query
  - Converting back into Peloton plan
Still to be done

- **Optimizer core**
  - Implement statistics for cost calculation
    - Table sampling
    - Join intermediate sampling
  - Additional memoization
    - Some rules are still being explored and applied redundantly

- **Extensions to base functionality - Logical, Physical operators and rules**
  - Operators
    - Merge & nested loop join
    - Index scan
    - Insert, update, delete
    - Aggregate
    - Subqueries
  - Rules
    - Predicate pushdown & pullup
    - Subquery fusion
    - Aggregate pushdown
    - etc...
Future Work

• End-to-end planning, analysis, and compilation
  ○ Most compilers work directly in terms of the code to be executed
  ○ RDBMs abstract away from low-level operator representation
    ■ Use a high-level and simple cost model

• Semi-static and dynamic replanning
  ○ Semi-static
    ■ Generate a tree of potential static plans at points of high variance
  ○ Dynamic
    ■ Perform initial coarse & guided optimization pass
    ■ Refine after executing predicates and joins