Lecture 19:
Query Execution

Logistics

- Exercise sheets 2 and 3 will be released soon
- Programming assignment 4 will also be released soon

Recap

- Learned Index
- Learned Index using Neural Network

Lecture Overview

- Query Execution

Modular Query Execution

Limitations of Hard-Coded Query

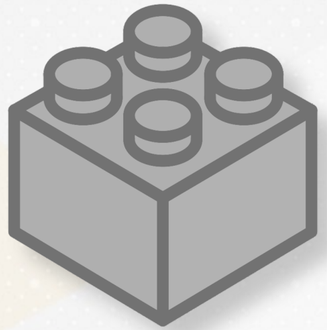
Hard to Change Query

Tight Coupling

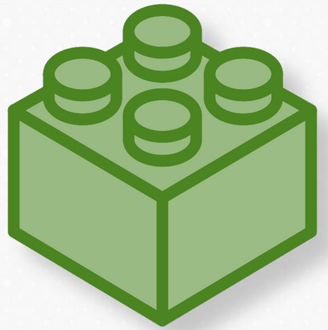
```
void scanTableToBuildIndex() {
  for (size_t page_itr = 0; page_itr < num_pages; page_itr++) {
    for (size_t slot_itr = 0; slot_itr < MAX_SLOTS; slot_itr++) {
      if (slot_array[slot_itr].empty == false) {
        hash_index.insertOrUpdate(key, value); ...
  }
}
void selectGroupBySum(int lowerBound, int upperBound) {
  auto results = index.rangeQuery(lowerBound, upperBound);
  ...
}
```

Modular Query Execution

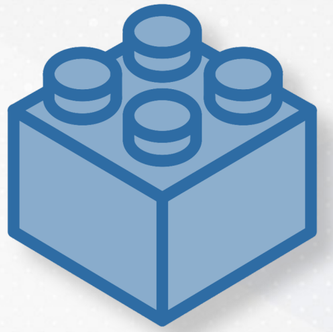
Scan Operator



Select Operator



Group By Operator



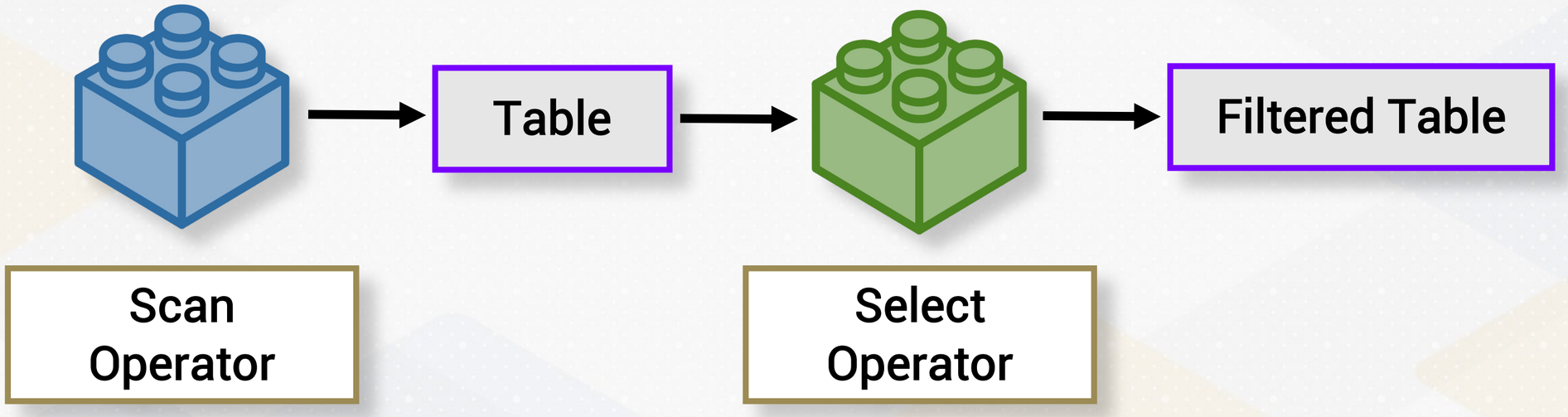
Flexibility

Flexible Query Configuration

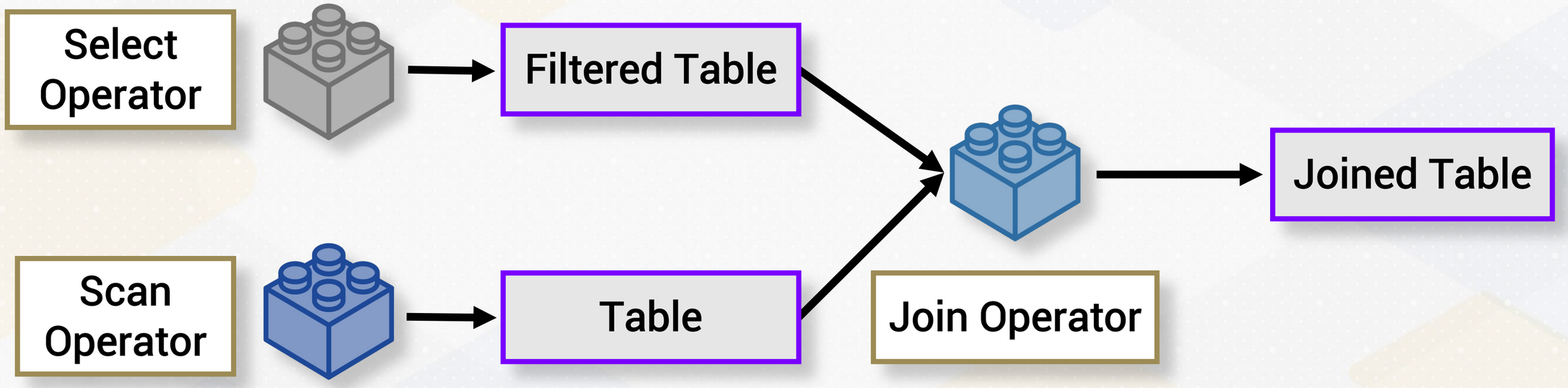
Isolation

Operator Changes Isolated

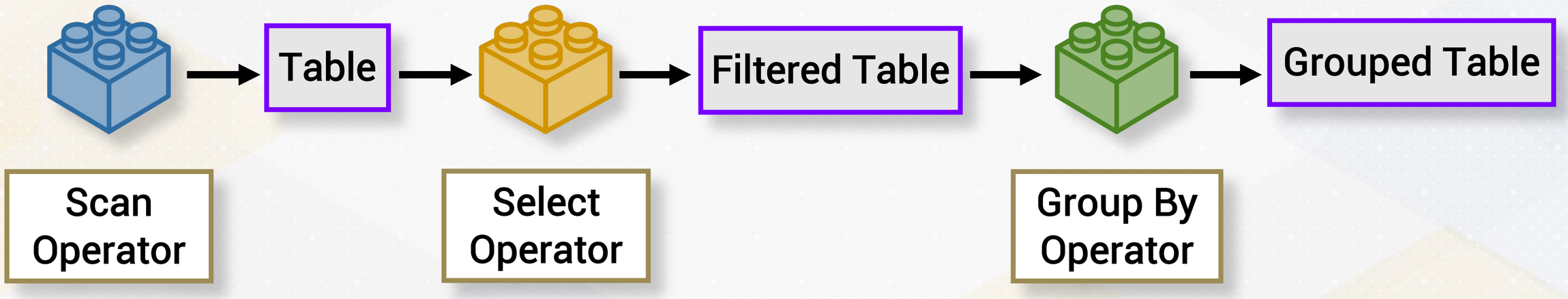
Operators are like Lego Blocks



Composability of Operators



Composability of Operators



Benefits of Operators

Flexibility

Users compose operators to run different queries

Modularity

Each operator encapsulates a specific data manipulation functionality

Reusability

Reduce redundancy and minimize the chance of bugs

Scan Operator

Operator Class

An abstract base class defining the common interface for all operators

```
class Operator {
public:
  virtual ~Operator() = default;
  virtual void open() = 0;
  virtual bool next() = 0;
  virtual std::vector<std::unique_ptr<Field>> getOutput() = 0;
  virtual void close() = 0;
};
```

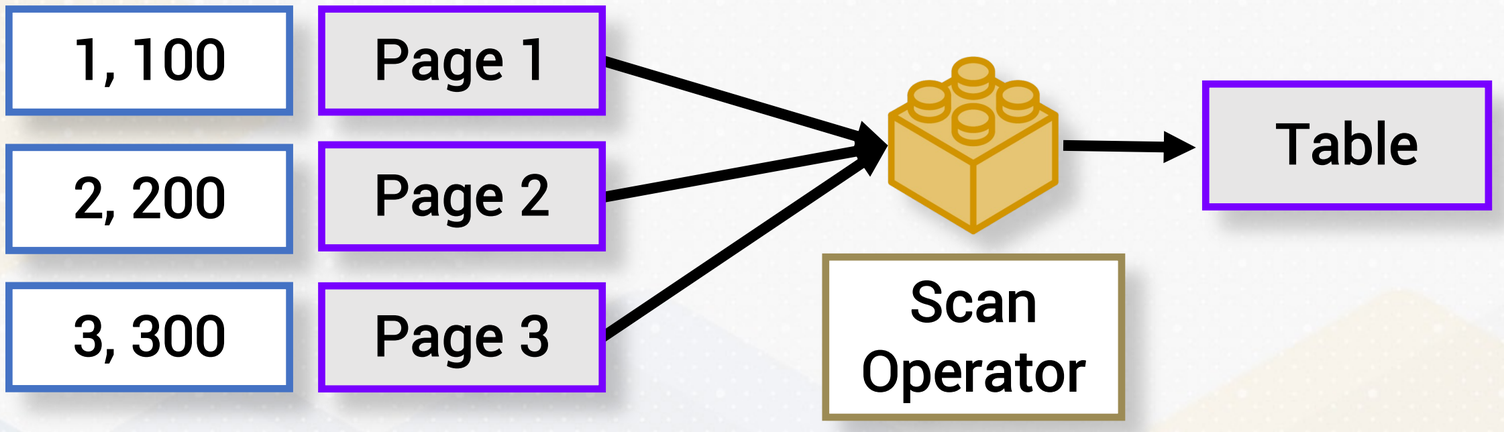
Operator Interface

Interface Methods
void open()
bool next()
std::vector<std::unique_ptr<Field>> getOutput()
void close()

open	next	getOutput	close
Initializes the operator	Progresses to next tuple	Returns the tuple	Cleans up resources

Scan Operator

"Scans" tuples from a table's pages on disk using Buffer Manager



ID	Value
1	100
2	200
3	300

Scan Operator

“Scans” tuples from a table’s pages on disk using Buffer Manager

```
class ScanOperator : public Operator {
private:
  size_t currentPageIndex = 0;
  std::unique_ptr<SlottedPage> currentPage;
  size_t currentSlotIndex = 0;
  std::unique_ptr<Tuple> currentTuple;
  BufferManager &bufferManager;
  // Initialization and tuple management code here...
};
```

open()

open(): Prepares First Page for Tuple Extraction

```
void ScanOperator::open() {
  currentPageIndex = 0; // Reset page index
  currentSlotIndex = 0; // Reset slot index
  loadNextTuple();
}
```

next()

next(): Sequentially moves through slots, loading tuples

```
bool ScanOperator::next() {
  if (!currentPage)
    return false;  // No more pages available
  loadNextTuple(); // load next tuple from page into currentTuple
  return currentTuple != nullptr;
}
```

loadNextTuple()

```
void loadNextTuple() {
  while (currentPageIndex < bufferManager.getNumPages()) {
    Slot *slot_array = reinterpret_cast<Slot *>(currentPage->page_data.get());
    while (currentSlotIndex < MAX_SLOTS) {
      // Extract tuple from current slot
      ...
      currentSlotIndex++;
    }
    currentPageIndex++; currentSlotIndex = 0;
  }
  // If we've reached here, no more tuples are available
  currentTuple.reset();
}
```

getOutput()

getOutput(): Returns fields of current tuple

```
std::vector<std::unique_ptr<Field>> getOutput() override {
  if (currentTuple) {
    return std::move(currentTuple->fields);
  }
  return {}; // Return an empty vector if no tuple is available
}
```

close()

close(): Release resources and perform cleanup operations

```
void ScanOperator::close() override {
  currentPage.reset();
  currentTuple.reset();
}
```

Abstract Base Class (ABC)

Abstract Base Class

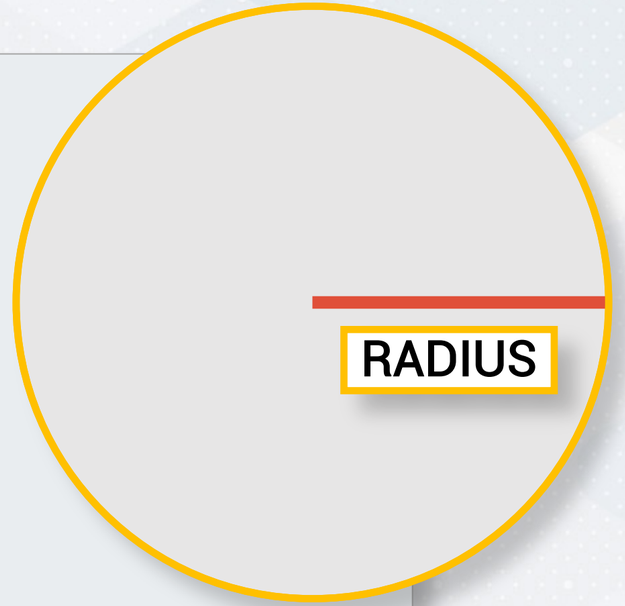
```
class Shape {
protected:
  float x, y; // Location of the shape
public:
  Shape(float x, float y, string color) : x(x), y(y), color(color) {}
  // Pure virtual function for drawing the shape
  virtual void draw() const = 0;
  // Virtual function for moving the shape
  virtual void move(float newX, float newY) {
    x = newX;
    y = newY;
    cout << "Moved to (" << x << ", " << y << ")." << endl;
  }
};
```

COMMON VARIABLES

PURE VIRTUAL FUNCTION

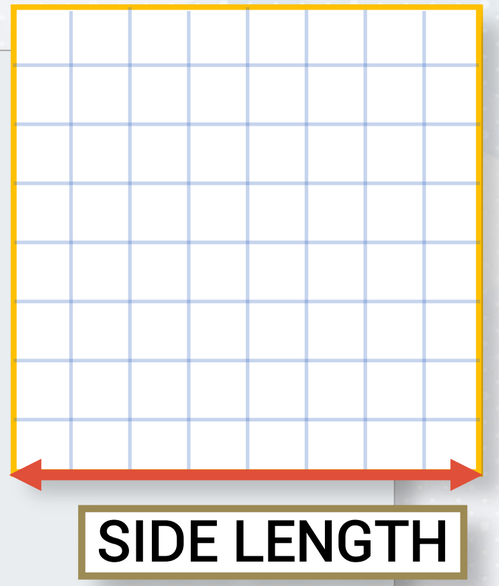
Derived Class: Circle

```
class Circle : public Shape {
private:
  float radius;
public:
  // Constructor for Circle initializes Shape and radius
  Circle(float x, float y, string color, float radius)
      : Shape(x, y, color), radius(radius) {}
  // Implementation of the pure virtual draw function
  void draw() const override {
    cout << "Drawing a circle at (" << x << ", " << y << ") with radius "
         << radius << " and color " << color << "." << endl;
  }
};
```



Derived Class: Square

```
class Square : public Shape {
private:
  float sideLength;
public:
  // Constructor for Square initializes Shape and side length
  Square(float x, float y, string color, float sideLength)
      : Shape(x, y, color), sideLength(sideLength) {

  // Implementation of the pure virtual draw function
  void draw() const override {
    cout << "Drawing a square at (" << x << ", " << y << ") with side length "
         << sideLength << " and color " << color << "." << endl;
  }
};
```

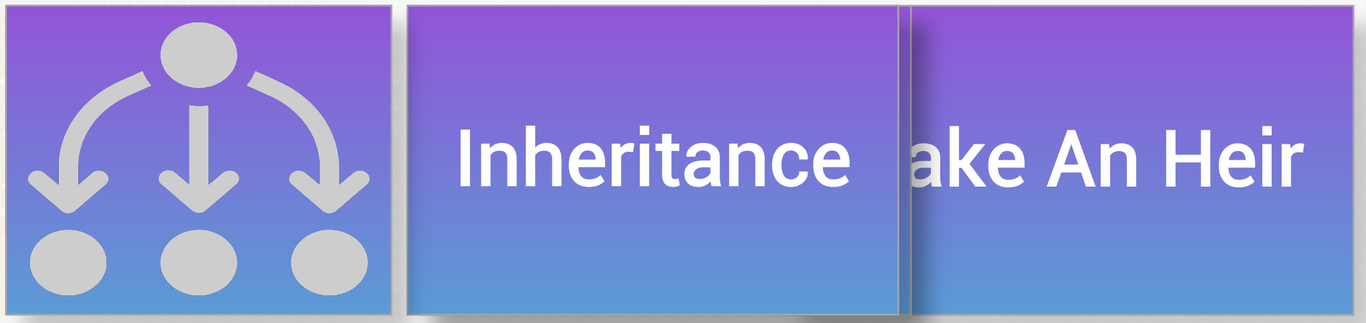


Inheritance

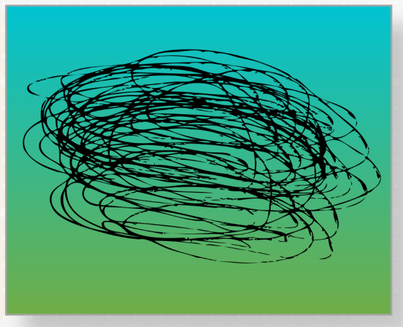
Inheritance
Derived classes can override Base class methods

Polymorphism
Derived classes can override Base class methods with different implementations

Etymology of Inheritance



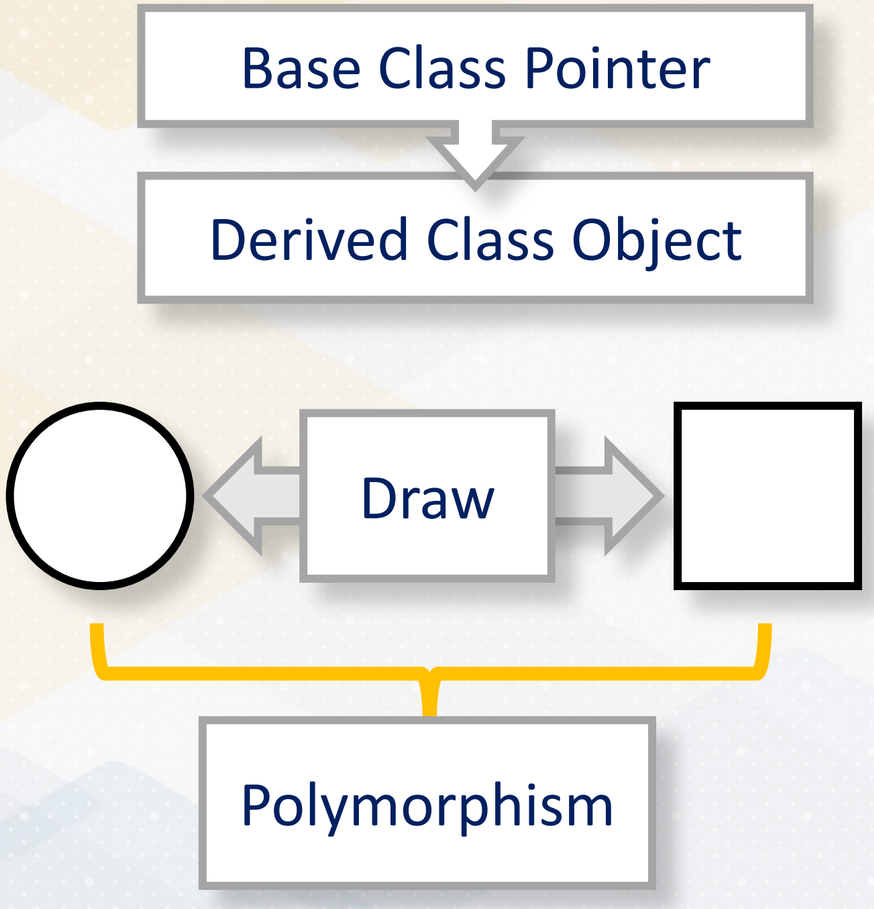
Etymology of Abstract



Abstract

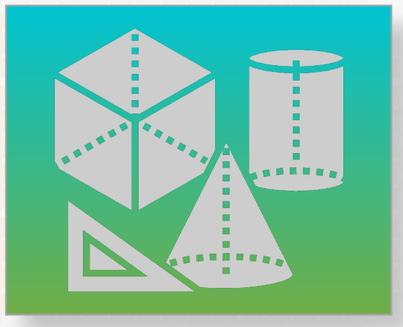
awn away
Detached

Shape Example and Polymorphism




```
int main() {
  Shape *shapes[2];
  shapes[0] = new Circle();
  shapes[1] = new Square();


  for (int i = 0; i < 2; ++i) {
    shapes[i]->draw(); // Polymorphic call}
    return 0;
  }
}
```

Etymology of Polymorphism

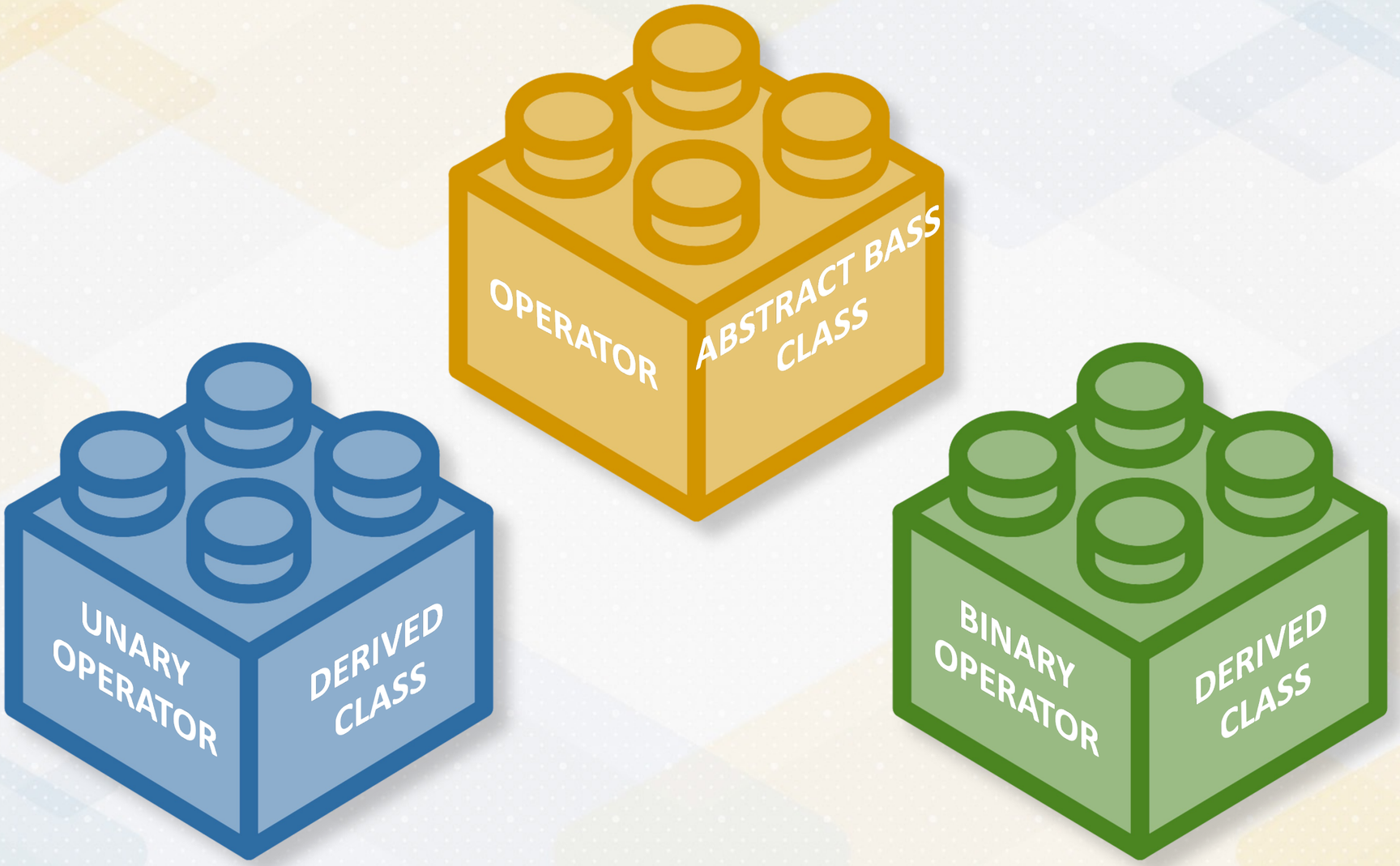


Polymorphism

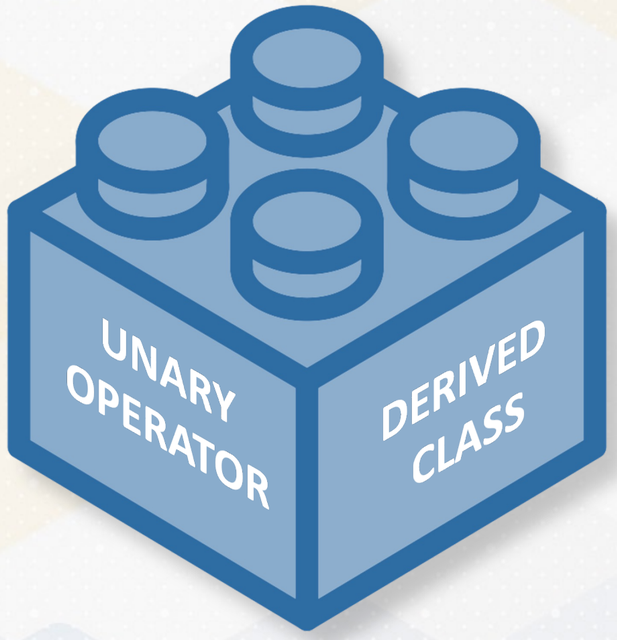
ny Shapes

Operator Inheritance

Derived Classes of Operator

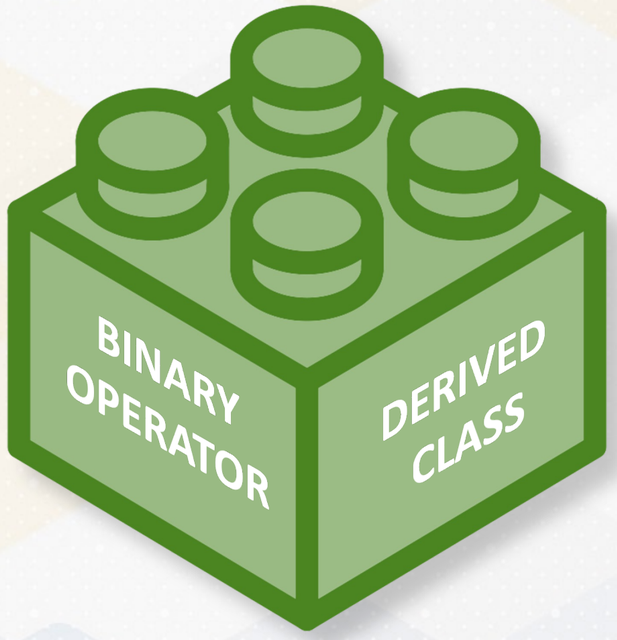


UnaryOperator




```
class UnaryOperator : public Operator {
  protected:
    Operator *input;

  public:
    explicit UnaryOperator(Operator &input) :
input(&input) {}

    ~UnaryOperator() override = default;
  };
```

BinaryOperator




```
class BinaryOperator : public Operator {
  protected:
    Operator *input_left;
    Operator *input_right;

  public:
    explicit BinaryOperator(Operator &input_left,
                            Operator &input_right)
        : input_left(&input_left),
input_right(&input_right) {}

    ~BinaryOperator() override = default;
  };
```

Predicate

Select Operator

```sql
SELECT *
FROM employees
WHERE salary > 1000;
```

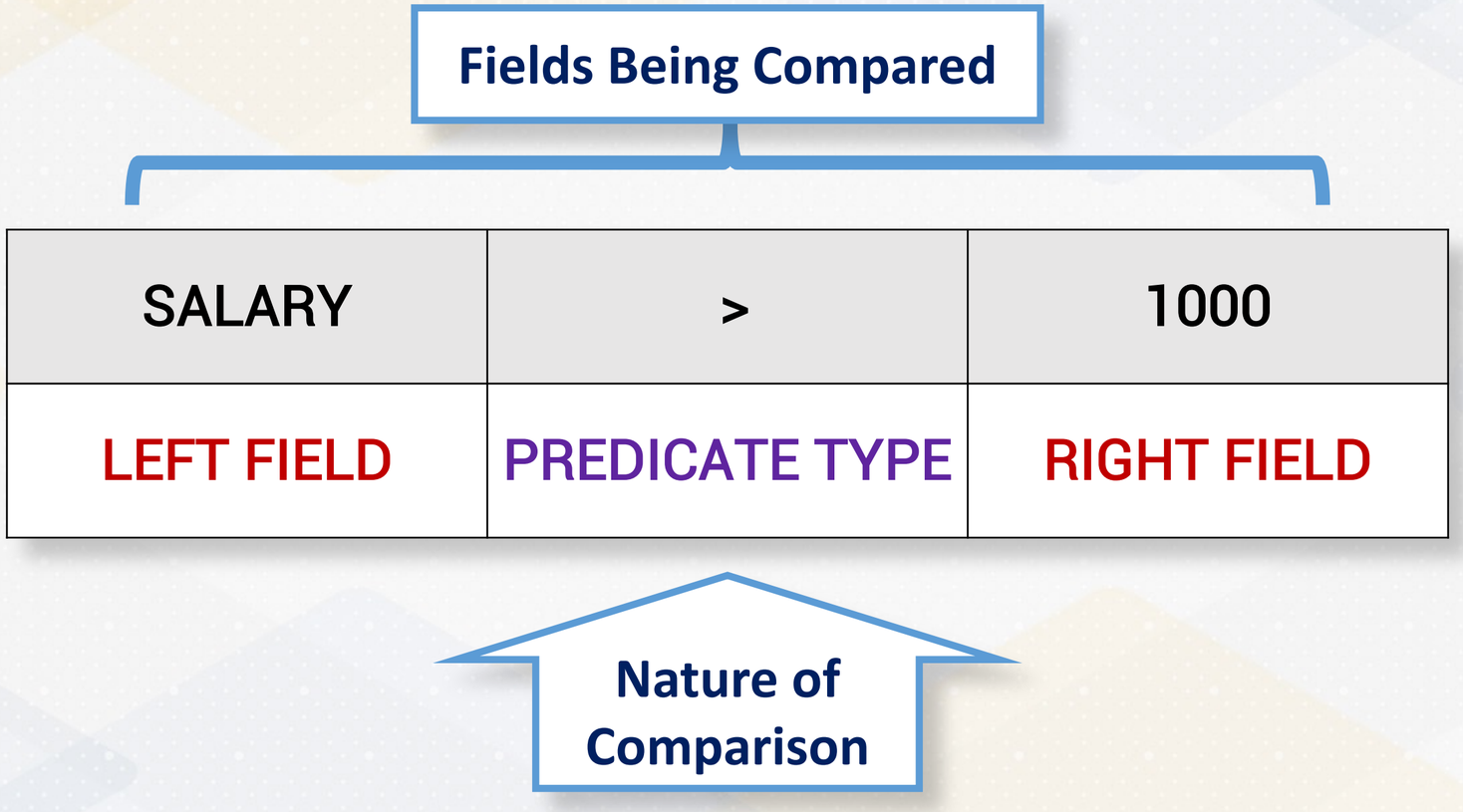
BASE TABLE

ID	SALARY
1	500
2	1500
3	800
4	2000

FILTERED TABLE

ID	SALARY
2	1500
4	2000

Predicate


Fields Being Compared
SALARY
>
1000
LEFT FIELD
PREDICATE TYPE
RIGHT FIELD
Nature of Comparison

Predicate

```
class Predicate {
public:
  std::unique_ptr<Field> left_field;
  std::unique_ptr<Field> right_field;
  PredicateType predicate_type;
};
```

Predicate Type

```
enum class PredicateType {
  EQ, // Equal
  NE, // Not Equal
  GT, // Greater Than
  GE, // Greater Than or Equal
  LT, // Less Than
  LE  // Less Than or Equal
};
```

Predicate Evaluation

```
bool checkPredicate() const {
  switch (left_field->getType()) {
  case INT: {
    int left_val = left_field->asInt();
    int right_val = right_field->asInt();
    return compare(left_val, right_val);
  }
  ...
}
```

Type-Safe Comparison Template

```
template <typename T>
bool compare(const T &left_val, const T &right_val) const {
    switch (predicate_type) {
    case PredicateType::EQ: return left_val == right_val;
    case PredicateType::NE: return left_val != right_val;
    case PredicateType::GT: return left_val > right_val;
    case PredicateType::GE: return left_val >= right_val;
    case PredicateType::LT: return left_val < right_val;
    case PredicateType::LE: return left_val <= right_val;
    default: std::cerr << "Invalid predicate type\n"; return false;
    }
}
```

Predicate Example

```
// Create integer fields for comparison
std::unique_ptr<Field> left = std::make_unique<Field>(10);
std::unique_ptr<Field> right = std::make_unique<Field>(20);
// check if left field value is greater than the right field value
Predicate predicate(std::move(left), std::move(right), PredicateType::GT);

// Evaluate the predicate and print the result
bool result = predicate.checkPredicate();
std::cout << "Predicate result: (10 > 20) : " << std::boolalpha << result
          << std::endl;
```

Complex Predicate

Limitations of Predicate

10	>	20
LEFT CONSTANT FIELD	PREDICATE TYPE	RIGHT CONSTANT FIELD

Complex Predicate

SALARY	>	1000
TUPLE COLUMN	PREDICATE TYPE	RIGHT CONSTANT FIELD

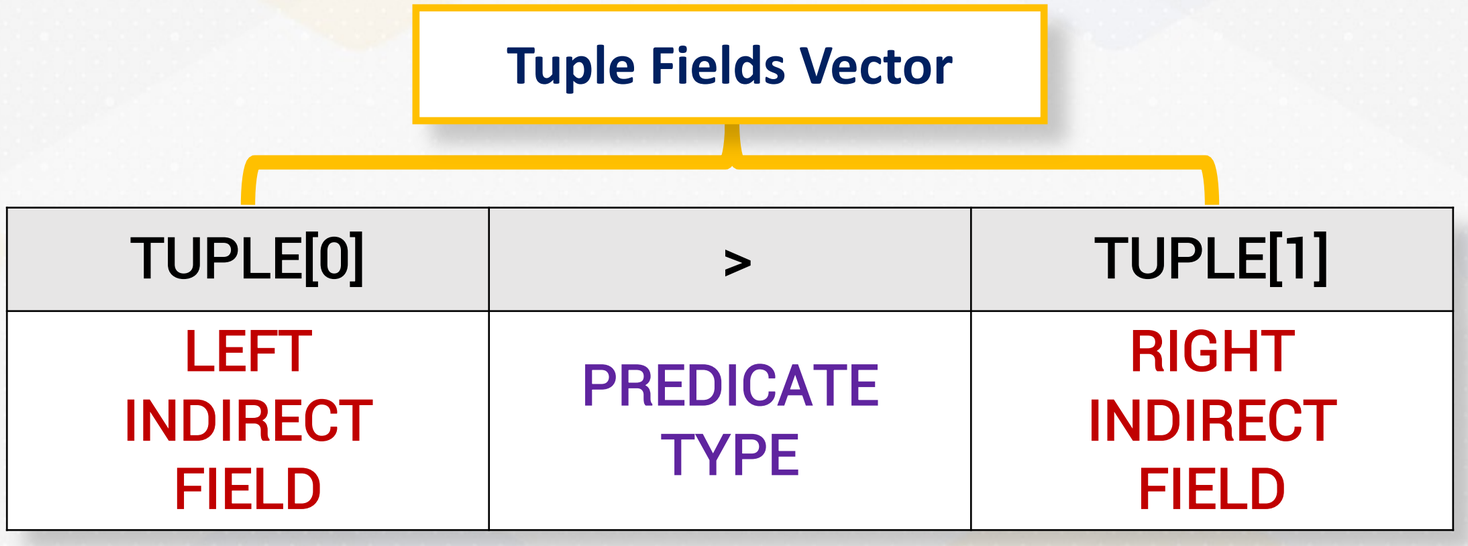
Complex Predicate

```
class Predicate {
  public:
    enum OperandType { DIRECT, INDIRECT };
    struct Operand {
      std::unique_ptr<Field> directValue;
      size_t index;
      OperandType type;
      Operand(std::unique_ptr<Field> value)
          : directValue(std::move(value)), type(DIRECT) {}
      Operand(size_t idx) : index(idx), type(INDIRECT) {}
    };
};
```

Complex Predicate Evaluation

```
bool checkPredicate(const std::vector<std::unique_ptr<Field>> &tupleFields) const {
    const Field *leftField = (left_operand.type == DIRECT)
                                 ? left_operand.directValue.get()
                                 : tupleFields[left_operand.index].get();
    const Field *rightField = (right_operand.type == DIRECT)
                                  ? right_operand.directValue.get()
                                  : tupleFields[right_operand.index].get();
    return compareBasedOnType(leftField, rightField);
}
```

Example with Indirect Field Reference



Example with Indirect Field Reference

```
// Construct tuple
std::vector<std::unique_ptr<Field>> tupleFields;
tupleFields.push_back(std::make_unique<Field>(10));
tupleFields.push_back(std::make_unique<Field>(20));

// Create predicate with indirect references to fields within the tuple
Predicate predicate(Predicate::Operand(0), Predicate::Operand(1),
                    PredicateType::GT);

// Check predicate using the constructed tuple
bool result = predicate.checkPredicate(tupleFields);
std::cout << "Predicate result: (10 > 20) : " << std::boolalpha << result
          << std::endl;
```

Example with Direct Field Reference

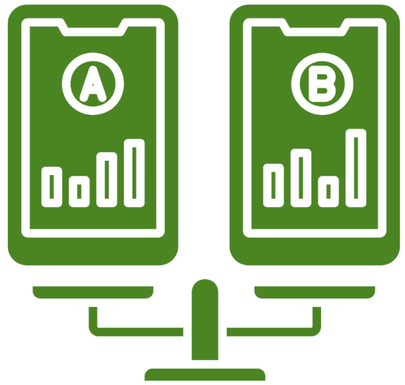
10	>	20
LEFT DIRECT FIELD	PREDICATE TYPE	RIGHT DIRECT FIELD

Example with Direct Field Reference

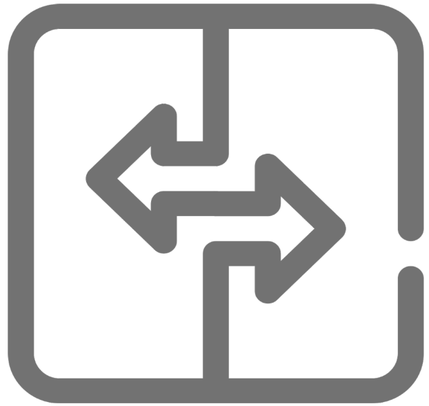
```
// Create direct field values
std::unique_ptr<Field> directLeftField = std::make_unique<Field>(10);
std::unique_ptr<Field> directRightField = std::make_unique<Field>(20);


// Create predicate with direct field references
Predicate predicate(Predicate::Operand(std::move(directLeftField)),
                    Predicate::Operand(std::move(directRightField)),
                    PredicateType::GT);
bool result = predicate.checkPredicate();
std::cout << "Predicate result using direct fields: (10 > 20) : "
          << std::boolalpha << result << std::endl;
```

Benefits of Complex Predicates

DIRECT AND INDIRECT REFERENCING



EASY TO CHANGE PREDICATE TYPE



TYPE-SPECIFIC COMPARISON

Conclusion

- Modular Query Execution
- Scan Operator
- Abstract Base Class
- Operator Inheritance
- Predicate