



Lecture 20: Select and Aggregation



Logistics

- Exercise sheet 2 released
- Programming assignment 3 (B+Tree) due on **Nov 2**
- Extra-credit project update report due on **Oct 29 (+4 days)**
 - Share GitHub link immediately after project title
 - Focus on completed concrete tasks
 - Benchmarking accuracy, latency etc.

Recap

- Query Execution
- Scan Operator
- Predicate

Lecture Overview

- Select Operator
- Deep vs Shallow Copy
- Using Select Operator
- More Complex Predicate
- Aggregation Operator
- Database and Unix

Select Operator



Select Operator

Fields Being Compared

TUPLE[0]	>	10
INDIRECT FIELD	PREDICATE TYPE	DIRECT FIELD

Select Operator

```
class SelectOperator : public UnaryOperator {  
private:  
    Predicate predicate; // Condition to evaluate on each tuple  
    bool has_next; // Indicator if there's a next tuple satisfying the  
    predicate  
    std::vector<std::unique_ptr<Field>> currentOutput; // Current tuple  
};
```


next

next()

Fetches next tuple that satisfies the predicate

```
bool next() override {  
    while (input->next()) {  
        const auto &output = input->getOutput();  
        if (predicate.checkPredicate(output)) {  
            currentOutput = duplicateFields(output);  
            has_next = true;  
            return true;  
        }  
    }  
    has_next = false;  
    currentOutput.clear();  
    return false;  
}
```


open and close

```
void open() override {  
    input->open(); // Initialize the input operator  
    has_next = false;  
    currentOutput.clear(); // Prepare for new output  
}  
  
void close() override {  
    input->close();  
    currentOutput.clear();  
}
```

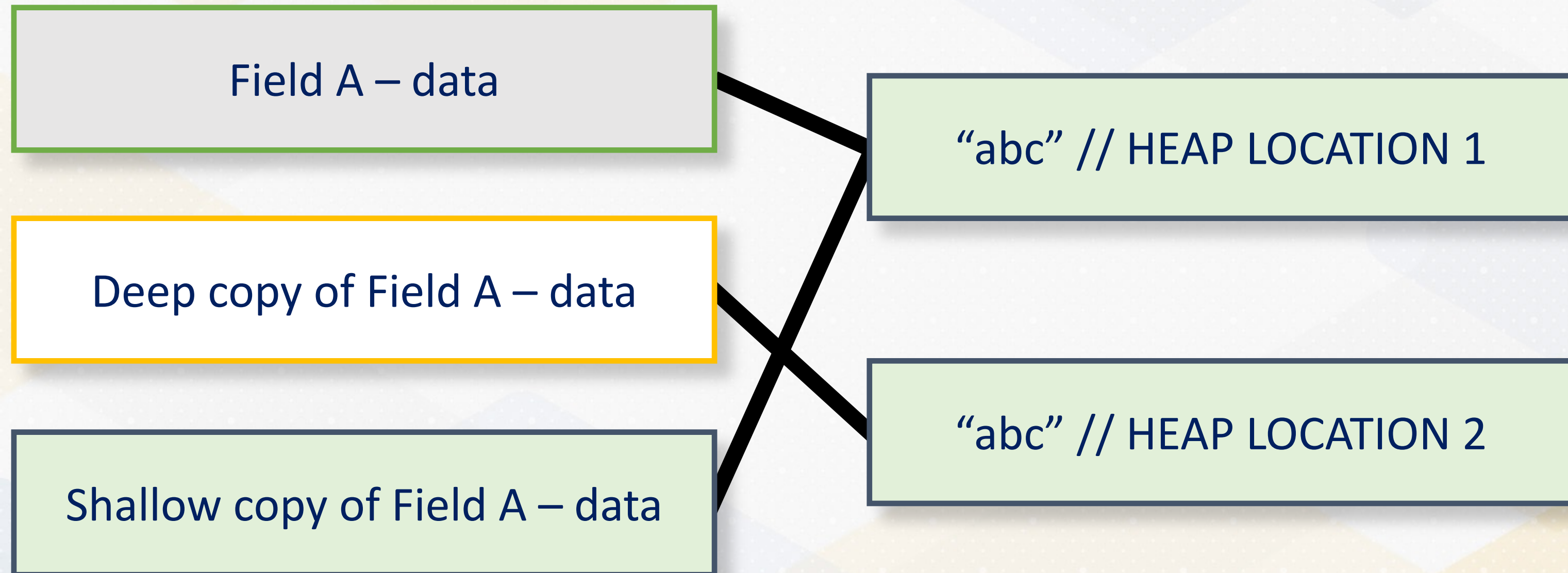

getOutput()

```
std::vector<std::unique_ptr<Field>> getOutput() override {  
    if (has_next) {  
        return duplicateFields(  
            currentOutput); // Return a copy of the current valid output  
    }  
    return {}; // Return empty if no more valid tuples  
}
```


Deep vs Shallow Copy



Deep vs Shallow Copy in C++



Deep Copy

original.data

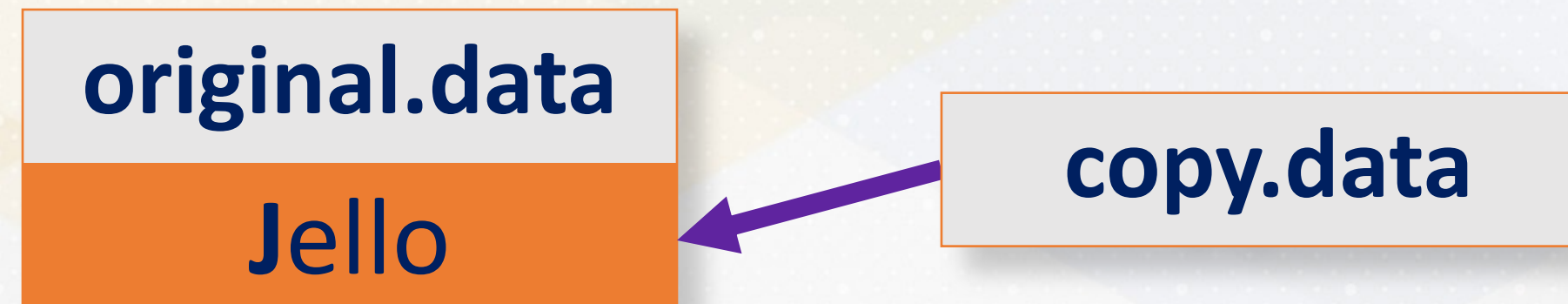
Hello

copy.data

Jello

```
class Deep {  
    char *data;  
    // Deep copy constructor  
    Deep(const Deep &other) {  
        // New memory allocation  
        this->data = new char[strlen(other.data) + 1];  
        strcpy(this->data, other.data);  
    }  
};  
Deep original("Hello");  
Deep copy = original; // Deep copying  
copy.data[0] = 'J';
```


Shallow Copy



```
class Shallow {
    char *data;
    // Shallow copy constructor
    Shallow(const Shallow &other) {
        // Copy the pointer
        this->data = other.data;
    }
};

Shallow original("Hello");
Shallow copy = original; // Shallow copying
copy.data[0] = 'J';
```


duplicateFields

duplicateFields

Creates an
Independent
Object Copy

Each Operator
Isolated from
Other Operators

```
std::vector<std::unique_ptr<Field>>
duplicateFields(const std::vector<std::unique_ptr<Field>> &fields) {
    std::vector<std::unique_ptr<Field>> outputCopy;
    for (const auto &field : fields) {
        outputCopy.push_back(field->clone());
    }
    return outputCopy;
}
```


Using Select Operator



Illustrative Query with Scan & Select Operators

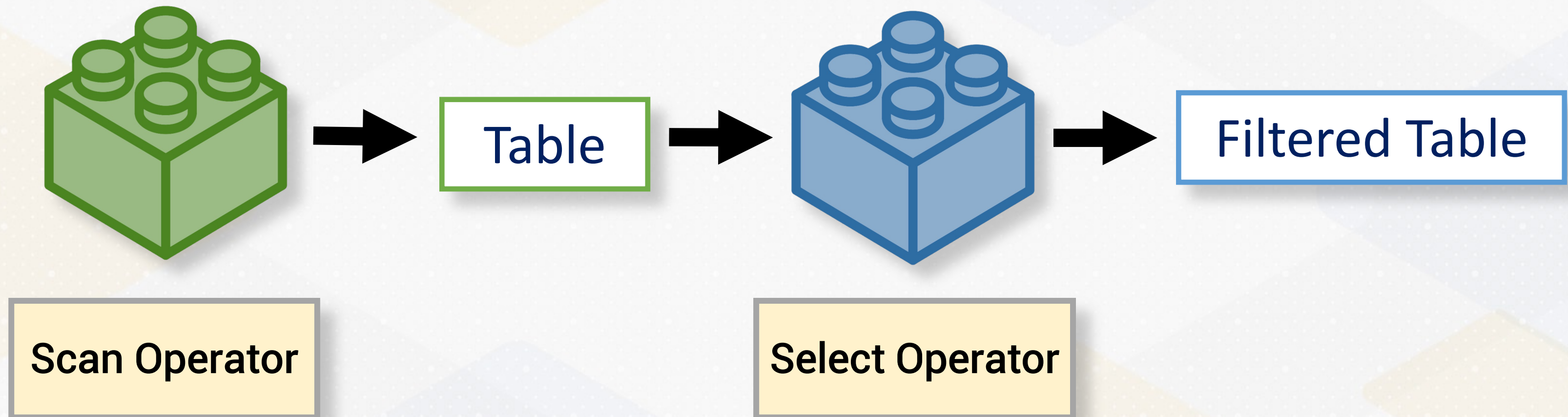
TUPLE [0] > 2

```
void executeQuery() {  
    ScanOperator scanOperator(buffer_manager);  
    Predicate predicate(Predicate::Operand(0),  
                        Predicate::Operand(std::make_unique<Field>(2)),  
                        PredicateType::GT);  
  
    SelectOperator selectOperator(scanOperator, std::move(predicate));  
    selectOperator.open();  
}
```


Illustrative Query with Scan & Select Operators

```
void executeQuery() {  
    ...  
    while (selectOperator.next()) {  
        const auto &fields = selectOperator.getOutput();  
        for (const auto &field_ptr : fields) {  
            field_ptr->print();  
            std::cout << " ";  
        }  
        std::cout << "\n";  
    }  
    selectOperator.close();  
}
```


Composability of Operators like Lego Blocks



More Complex Predicate

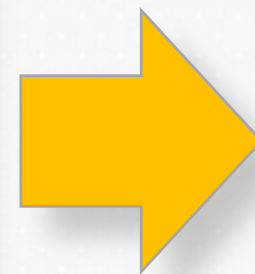


Need for More Complex Predicate

```
SELECT *  
FROM employees  
WHERE salary > 1000 AND age < 30;
```

BASE TABLE

ID	SALARY	AGE
1	500	20
2	1500	20
3	800	40
4	2000	40



FILTERED TABLE

ID	SALARY	AGE
2	1500	20

IPredicate Interface



Common interface for all predicate types

```
class IPredicate {  
public:  
    virtual ~IPredicate() = default;  
    virtual bool  
    check(const std::vector<std::unique_ptr<Field>> &tupleFields) const = 0;  
};
```


SimplePredicate: Comparison Operator



Implements the **IPredicate** interface for basic comparison operations

SALARY	>	1000
INDIRECT OPERAND	COMPARISON OPERATOR	DIRECT OPERAND

SimplePredicate: Comparison Operator



Implements the **IPredicate** interface for basic comparison operations

```
class SimplePredicate : public IPredicate {
    Operand left_operand, right_operand;
    ComparisonOperator comparison_operator;

    bool
    check(const std::vector<std::unique_ptr<Field>> &tupleFields) const override {
        // Fetch fields based on operand type (DIRECT or INDIRECT)
        // Compare fields based on the specified ComparisonOperator
    }
};
```


ComplexPredicate: Logical Operator



Implements the **IPredicate** interface for logical operations (AND/OR)

PREDICATE 1	AND/OR	PREDICATE 2
	LOGICAL OPERATOR	

ComplexPredicate: Logical Operator

```
class ComplexPredicate : public IPredicate {  
    std::vector<std::unique_ptr<IPredicate>> predicates;  
    LogicOperator logic_operator;  
  
    bool  
    check(const std::vector<std::unique_ptr<Field>> &tupleFields) const override {  
        // Evaluate all contained predicates based on the logic operator (AND, OR)  
    }  
};
```


Evaluating Complex Predicate

```
if (logic_operator == AND) {  
    for (const auto &pred : predicates) {  
        if (!pred->check(tupleFields)) {  
            return false; // All must pass  
        }  
    }  
    return true;  
}  
else if (logic_operator == OR) {  
    for (const auto &pred : predicates) {  
        if (pred->check(tupleFields)) {  
            return true; // Any one can pass  
        }  
    }  
    return false;  
}
```

Illustrative Complex Predicate

TUPLE [0]	>	2
INDIRECT OPERAND	COMPARISON OPERATOR	DIRECT OPERAND



TUPLE [0]	>	7
INDIRECT OPERAND	COMPARISON OPERATOR	DIRECT OPERAND



PREDICATE 1	AND	PREDICATE 2
IPREDICATE OBJECT POINTER	LOGICAL OPERATOR	IPREDICATE OBJECT POINTER

Illustrative Complex Predicate

TUPLE [0]	>	2
INDIRECT OPERAND	COMPARISON OPERATOR	DIRECT OPERAND

TUPLE [0]	>	7
INDIRECT OPERAND	COMPARISON OPERATOR	DIRECT OPERAND

```
// Creating simple comparison predicates
auto greaterThanTwo = std::make_unique<SimplePredicate>(
    SimplePredicate::Operand(0),
    SimplePredicate::Operand(std::make_unique<Field>(2)),
    SimplePredicate::ComparisonOperator::GT);

auto lessThanSix = std::make_unique<SimplePredicate>(
    SimplePredicate::Operand(0),
    SimplePredicate::Operand(std::make_unique<Field>(7)),
    SimplePredicate::ComparisonOperator::LT);
```

Illustrative Complex Predicate

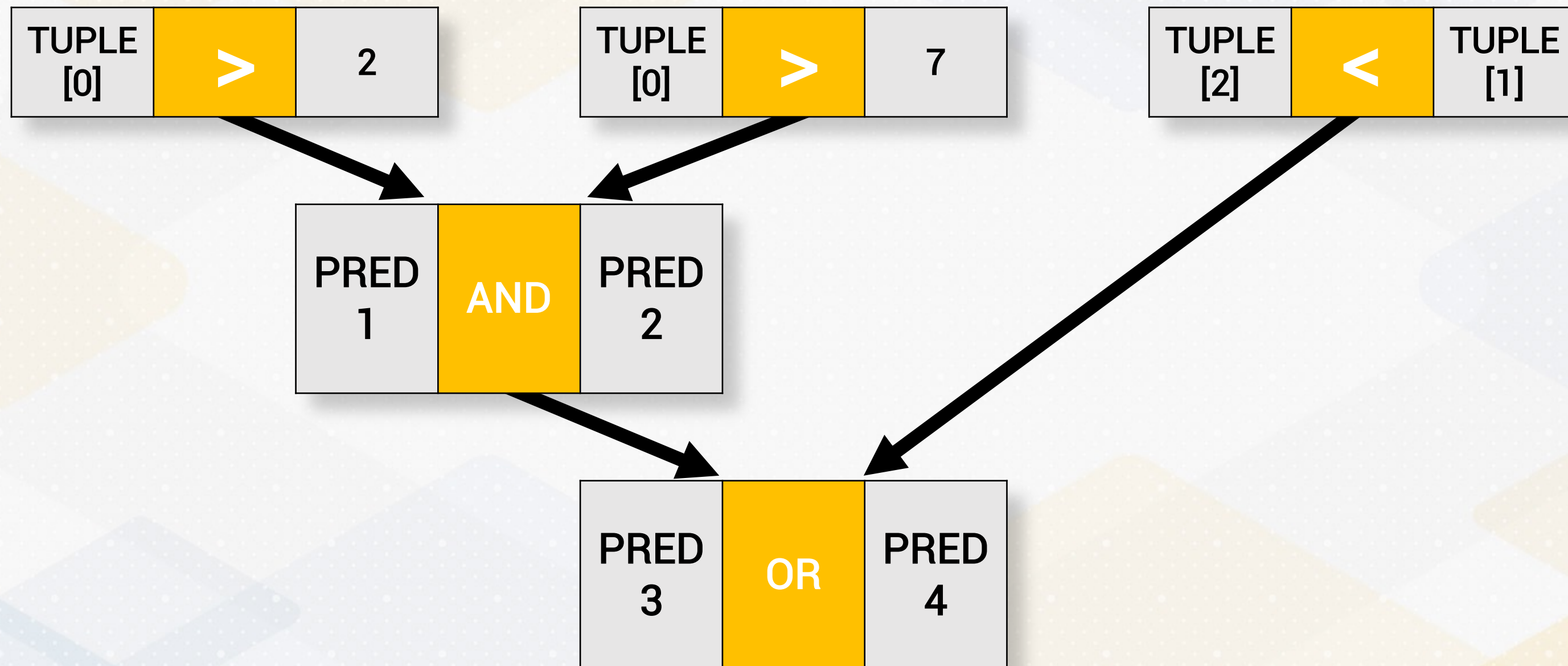
PREDICATE 1	AND	PREDICATE 2
IPREDICATE OBJECT POINTER	LOGICAL OPERATOR	IPREDICATE OBJECT POINTER

```
// Combining predicates into a complex predicate with AND logic
auto complexPredicate =
    std::make_unique<ComplexPredicate>(ComplexPredicate::LogicOperator::AND);

complexPredicate->addPredicate(std::move(greaterThanTwo));
complexPredicate->addPredicate(std::move(lessThanSix));

// Integrating ComplexPredicate with a SelectOperator for query processing
SelectOperator selectOperator(scanOperator, std::move(complexPredicate));
```


More Complex Predicate



Aggregation Operator



Aggregation Query

```
SELECT Category_ID,  
       SUM(Profit) AS Total_Profit  
FROM Sales_Data  
WHERE MONTH(DATE) = 'JUNE'  
GROUP BY Category_ID;
```

BASE TABLE

PRODUCT ID	PROFIT	CATEGORY ID	DATE
1	500	1	May 15
2	1000	1	June 10
3	1000	2	June 20
4	2000	2	June 30

HASH TABLE

KEY	VALUE
1	1000
2	3000

Hash Aggregation Operator

BASE TABLE

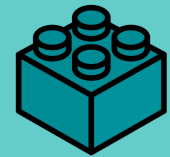
PRODUCT ID	PROFIT	CATEGORY ID	DATE
1	500	1	May 15
2	1000	1	June 10
3	1000	2	June 20
4	2000	2	June 30



HASH TABLE

KEY	VALUE
1	1000
2	3000

Hash Aggregation Operator



Unary Operator Class



Hash Aggregation



group_by_attrs



aggr_funcs vector



output_tuples

```
class HashAggregationOperator : public UnaryOperator {  
private:  
    std::vector<size_t> group_by_attrs;  
    std::vector<AggrFunc> aggr_funcs;  
    std::vector<Tuple> output_tuples;  
    ...  
};
```

Hash Aggregation Operator

BASE TABLE

PRODUCT ID	PROFIT	CATEGORY ID	DATE
1	500	1	May 15
2	1000	1	June 10
3	1000	2	June 20
4	2000	2	June 30



HASH AGGREGATION OPERATOR

GROUP BY ATTRS	{3}
AGGREGATE FUNCTIONS	{SUM, 2}
OUTPUT TUPLES	{1, 1000}, {2, 3000}

Aggregation Logic



```
void HashAggregationOperator::open() {  
    std::unordered_map<std::vector<Field>, std::vector<Field>, FieldVectorHasher>  
        hash_table;  
    // Build the hash table with the aggregate tuples  
}  
  
bool HashAggregationOperator::next() {  
    // Iterate over the aggregate tuples in the hash table  
}
```

Multi-Column Grouping

```
SELECT Category_ID,  
       MONTH(DATE),  
       SUM(Profit) AS Total_Profit  
FROM Sales_Data  
GROUP BY Category_ID, MONTH(DATE);
```

BASE TABLE

PRODUCT ID	PROFIT	CATEGORY ID	DATE
1	500	1	May 15
2	1000	1	June 10
3	1000	2	June 20
4	2000	2	June 30

AGGREGATED TABLE

CATEGORY ID	MONTH (DATE)	TOTAL PROFIT
1	May	500
1	June	1000
2	June	3000

Field Vector Hasher



```
struct FieldVectorHasher {  
    std::size_t operator()(const std::vector<Field> &fields) const {  
        std::size_t hash = 0;  
        for (const auto &field : fields) {  
            std::hash<std::string> hasher;  
            std::size_t fieldHash = 0;  
            fieldHash = hasher(...);  
            hash ^= fieldHash + 0x9e3779b9 + (hash << 6) + (hash >> 2);  
        }  
        return hash;  
    }  
};
```

Hash Aggregation Operator

FIELD	OVERALL HASH VALUE	FIELD HASH VALUE

Update Aggregate

```
SELECT Category_ID,  
       MONTH(DATE),  
       SUM(Profit) AS Total_Profit  
FROM Sales_Data  
GROUP BY Category_ID, MONTH(DATE);
```

BASE TABLE

PRODUCT ID	PROFIT	CATEGORY ID	DATE
1	500	1	May 15
2	1000	1	June 10
3	1000	2	June 20
4	2000	2	June 30

AGGREGATED TABLE

CATEGORY ID	MONTH (DATE)	TOTAL PROFIT
1	May	500
1	June	1000
2	June	3000

Update Aggregate

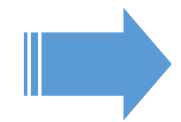


```
Field updateAggregate(const AggrFunc &aggrFunc, const Field &currentAggr,
                      const Field &newValue) {
    // Check for type consistency
    if (currentAggr.getType() != newValue.getType()) {
        throw std::runtime_error("Mismatched Field types in aggregation.");
    }
    // Perform the specified aggregation operation
    switch (aggrFunc.func) {
    case AggrFuncType::SUM:
        return Field(currentAggr.asInt() + newValue.asInt());
        // other cases...
    }
}
```


getOutput

Appends Group Keys

and Aggregate Value



Aggregate Tuples

```
for (const auto &entry : hash_table) {  
    const auto &group_keys = entry.first;  
    const auto &aggr_values = entry.second;  
    for (const auto &key : group_keys) {  
        // Add group keys to the tuple  
        output_tuple.addField(std::make_unique<Field>(key));  
    }  
    for (const auto &value : aggr_values) {  
        // Add aggregated values to the tuple  
        output_tuple.addField(std::make_unique<Field>(value));  
    }  
}
```


Database and Unix



Hard-Coded Query Function



Selects
Tuples
from
Table

Filters
Tuples
from
Column 1

Groups
Filters
Tuples

Sums
Values of
Column 2

```
SELECT column1, SUM(column2)
FROM tableName
WHERE column1 > 2 AND column1 < 7
GROUP BY column1;
```

Hard-Coded Query Function

selectGroupBySum

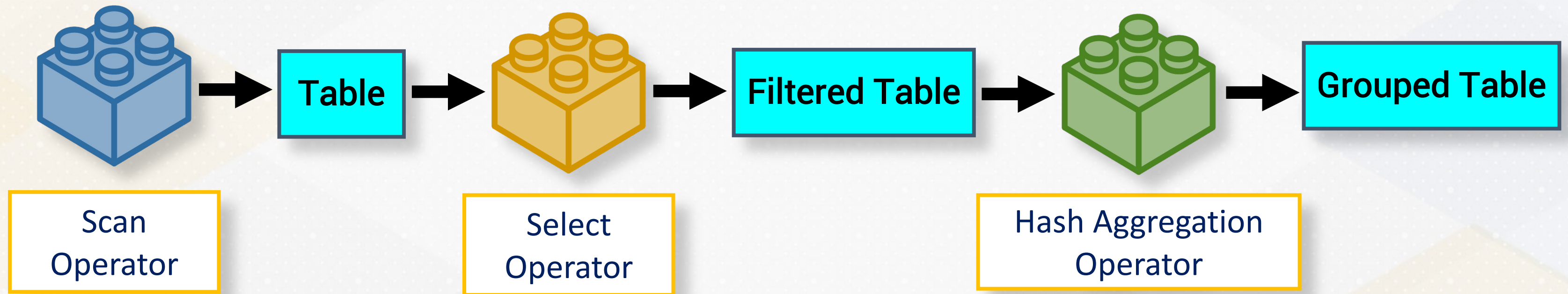
Scans Tuples &
Builds Index

Groups Filtered
Tuples

Sums Value of
Column 2

```
int main() {  
    db.scanTableToBuildIndex();  
    ...  
    int lowerBound = 2;  
    int upperBound = 7;  
    db.selectGroupBySum(lowerBound, upperBound);  
}
```


Relational Operator-Based Query Pipeline



Relational Operator-Based Query Pipeline



```
void executeQuery() {  
    ...  
    SelectOperator selectOperator(scanOperator, std::move(complexPredicate));  
    std::vector<AggrFunc> aggrFuncs{{AggrFuncType::SUM, 1}};  
    std::vector<size_t> groupByAttrs{0};  
    HashAggregationOperator hashAggregationOperator(selectOperator, groupByAttrs,  
                                                    aggrFuncs);  
  
    hashAggregationOperator.open();  
    while (hashAggregationOperator.next()) {  
        const auto &aggregatedFields = hashAggregationOperator.getOutput();  
        // Output key and aggregated value  
    }  
    hashAggregationOperator.close();  
}
```



Unix Operating System and Relational Database

	Unix Operating System	Relational Database
Timeline	1970s	1970s
Team	Dennis Ritchie, Ken Thompson, and others	Ted Codd, System R Team, and others
Location	AT&T Bell Labs	IBM

Simplicity and Modularity

Each tool should do one and only one task well

```
// GREP TOOL
grep "error" file.txt

// SELECT OPERATOR
SELECT *
FROM TableName
WHERE Message LIKE '%error%';
```


Pipelining and Composability



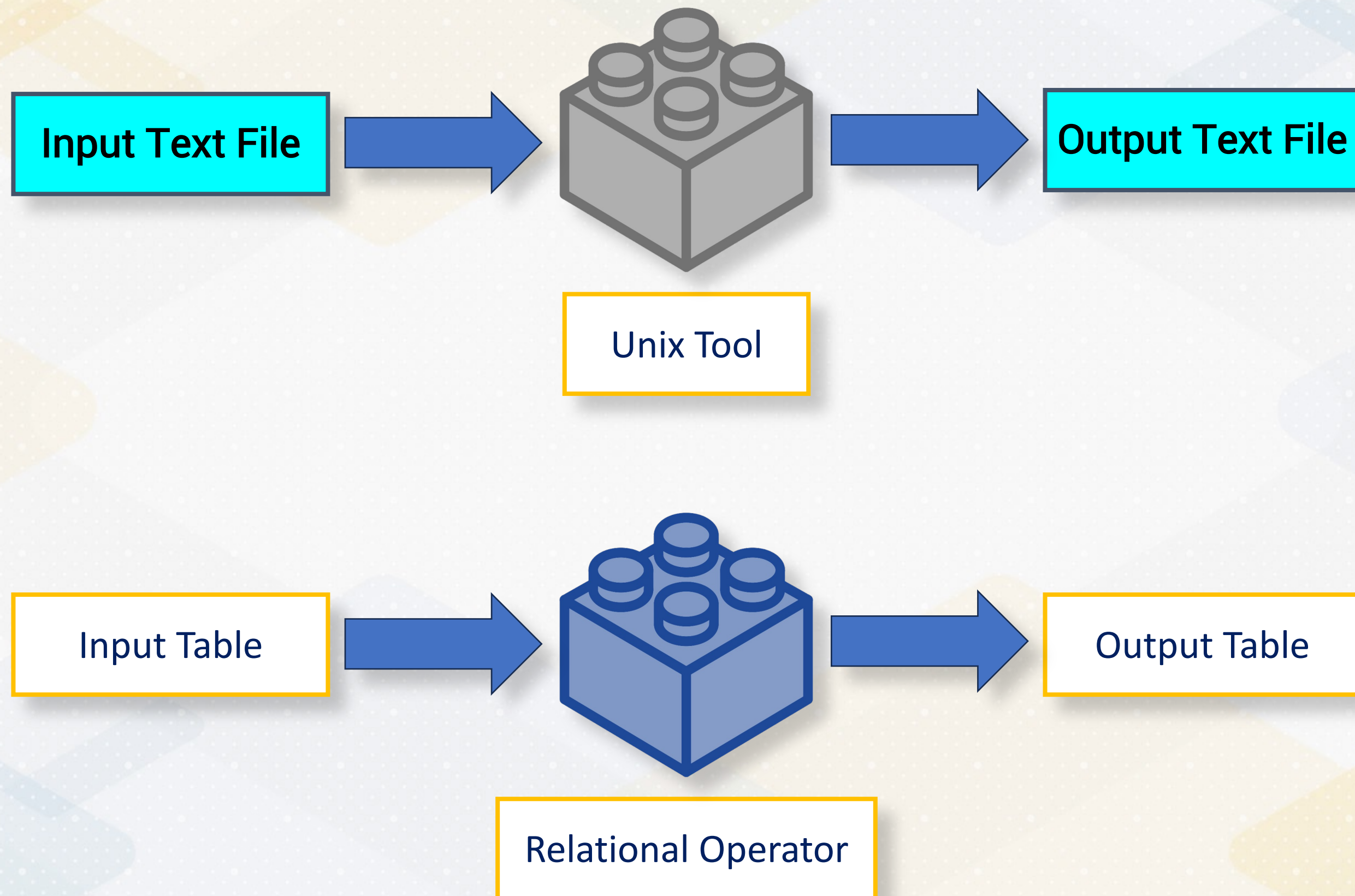
// UNIX TOOL PIPELINE

```
cat file.txt | grep "error" | sort | uniq -c
```

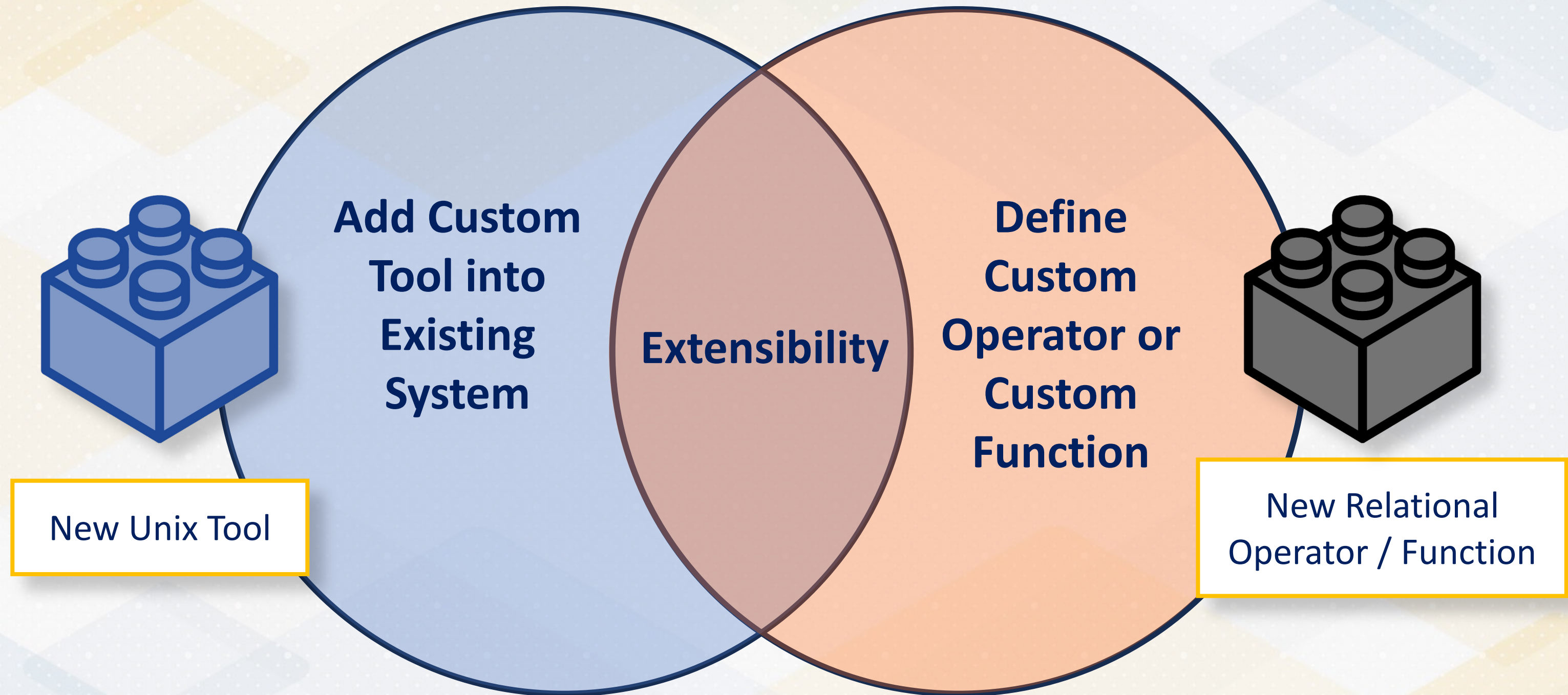
// QUERY PIPELINE

```
SELECT ErrorMessage, COUNT(*) AS ErrorCount  
FROM LogTable  
WHERE ErrorMessage LIKE '%error%'  
GROUP BY ErrorMessage  
ORDER BY ErrorMessage;
```

Standardized Interface



Extensibility



Conclusion

- Select Operator
- Deep vs Shallow Copy
- Using Select Operator
- More Complex Predicate
- Aggregation Operator
- Database and Unix