Lecture 4:
Query Execution

Logistics

- Turing Point app
- Intro sheet (due on Jan 29)
- Programming assignment 1 (due on Jan 29)

Recap

- Indexing
- Hash Table
- Double Hashing
- Range Query
- B+Tree
- Trie
- RTree

Lecture Overview

- Modular Query Execution
- Scan Operator
- Predicate & Select Operator
- Aggregation Operator
- Query Parsing & Compilation
- Columnar Storage
- Compressed Columnar Storage
- Vectorized Execution

Modular Query Execution

Limitations of Hard-Coded Query

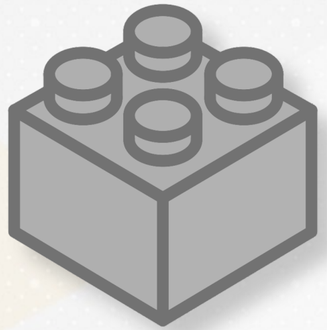
Hard to Change Query

Tight Coupling

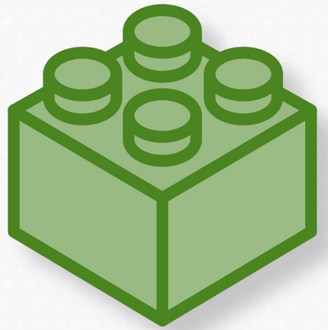
```
void scanTableToBuildIndex() {
  for (size_t page_itr = 0; page_itr < num_pages; page_itr++) {
    for (size_t slot_itr = 0; slot_itr < MAX_SLOTS; slot_itr++) {
      if (slot_array[slot_itr].empty == false) {
        hash_index.insertOrUpdate(key, value); ...
  }
}
void selectGroupBySum(int lowerBound,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

Modular Query Execution

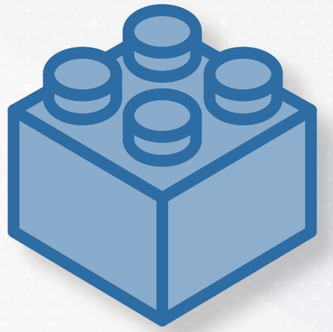
Scan Operator



Select Operator



Group By Operator



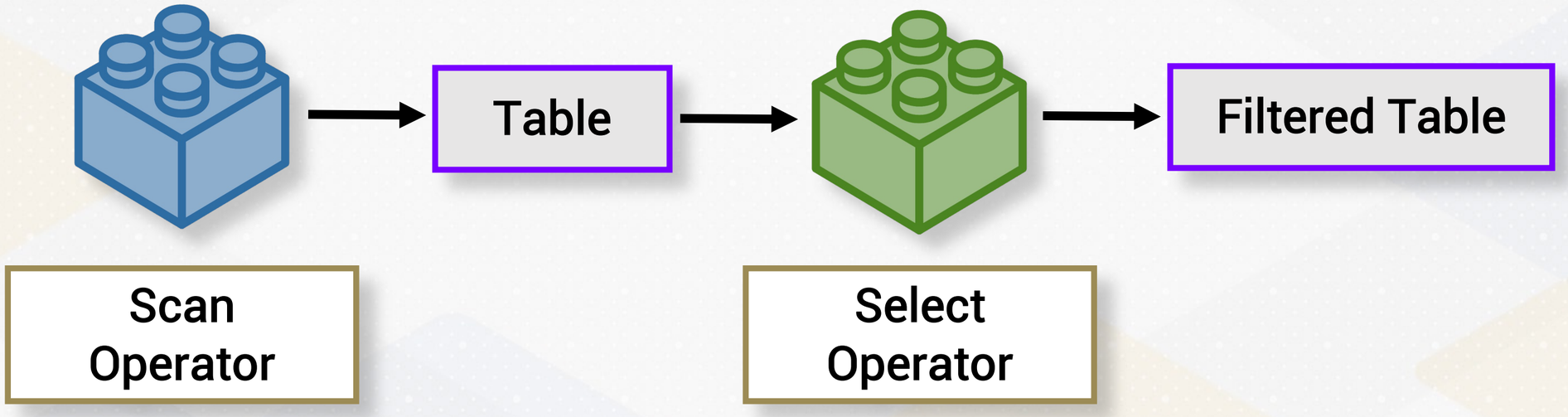
Flexibility

Flexible Query Configuration

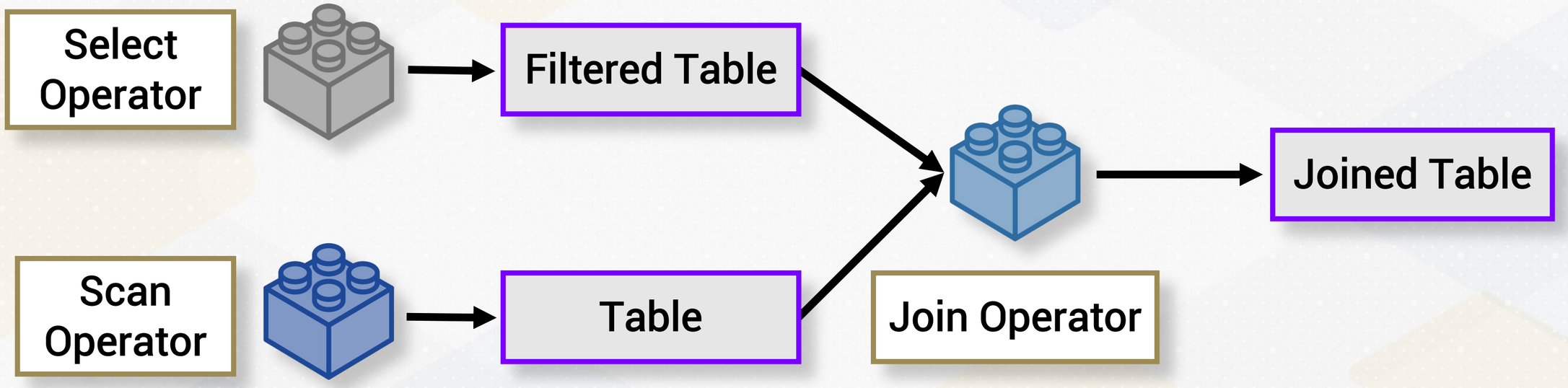
Isolation

Operator Changes Isolated

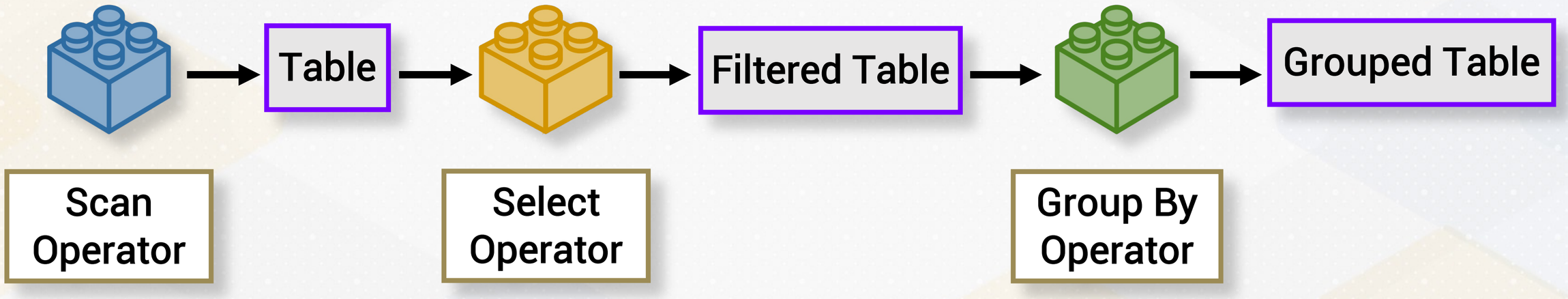
Operators are like Lego Blocks



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;
};
```

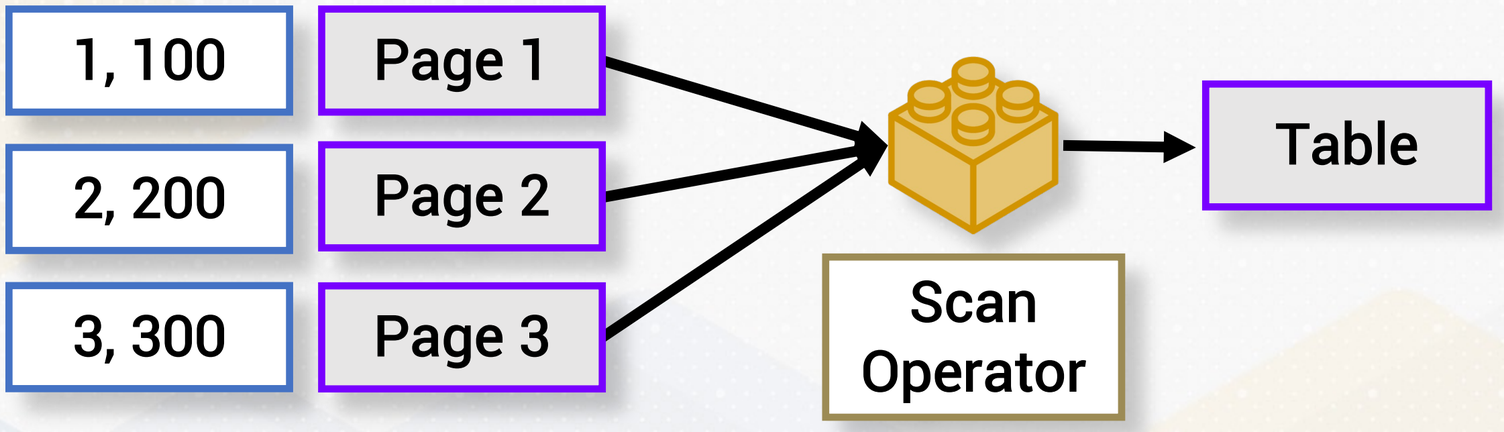
Operator Interface

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();
}
```

Predicate

Select Operator

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

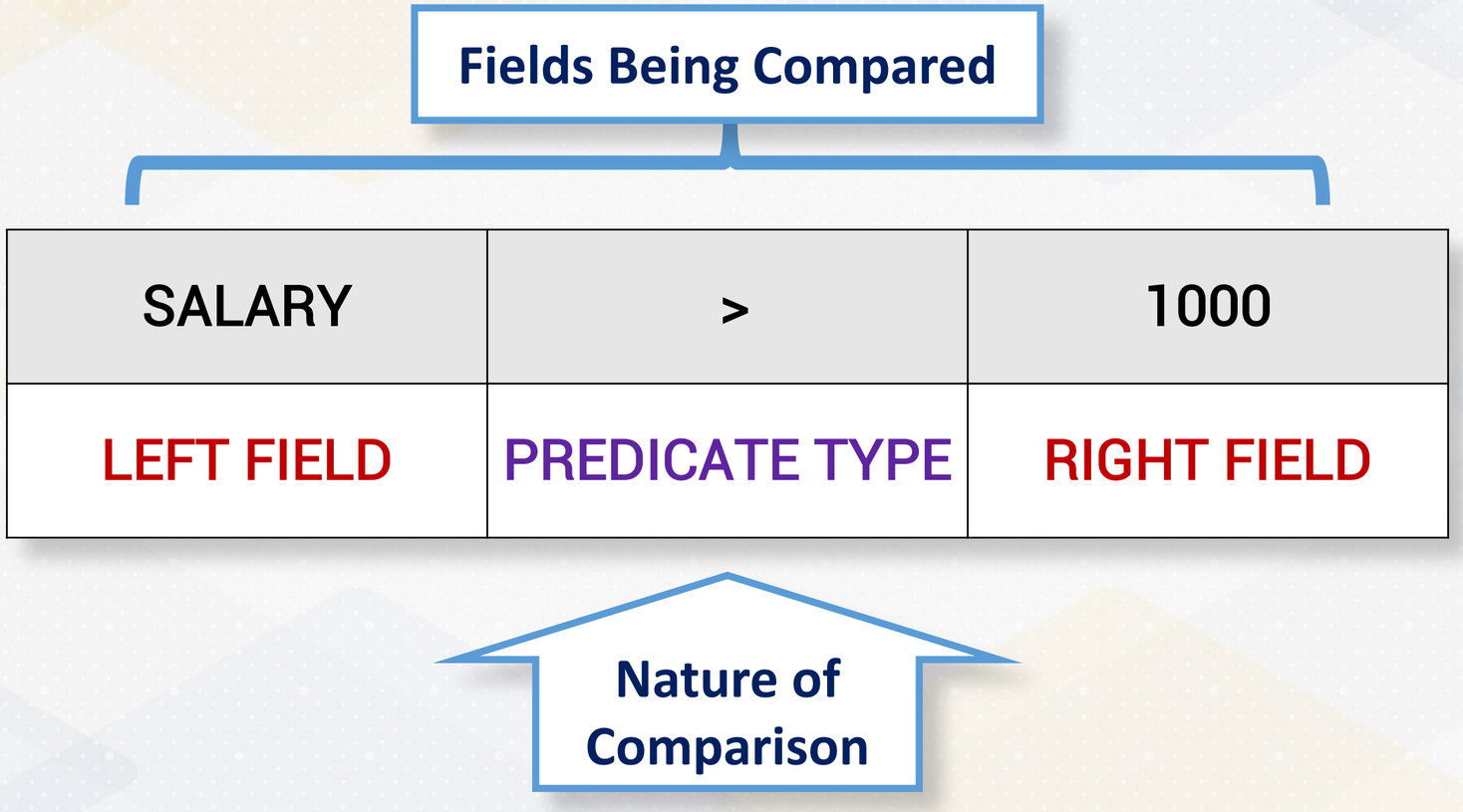
BASE TABLE

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;
    }
}
```

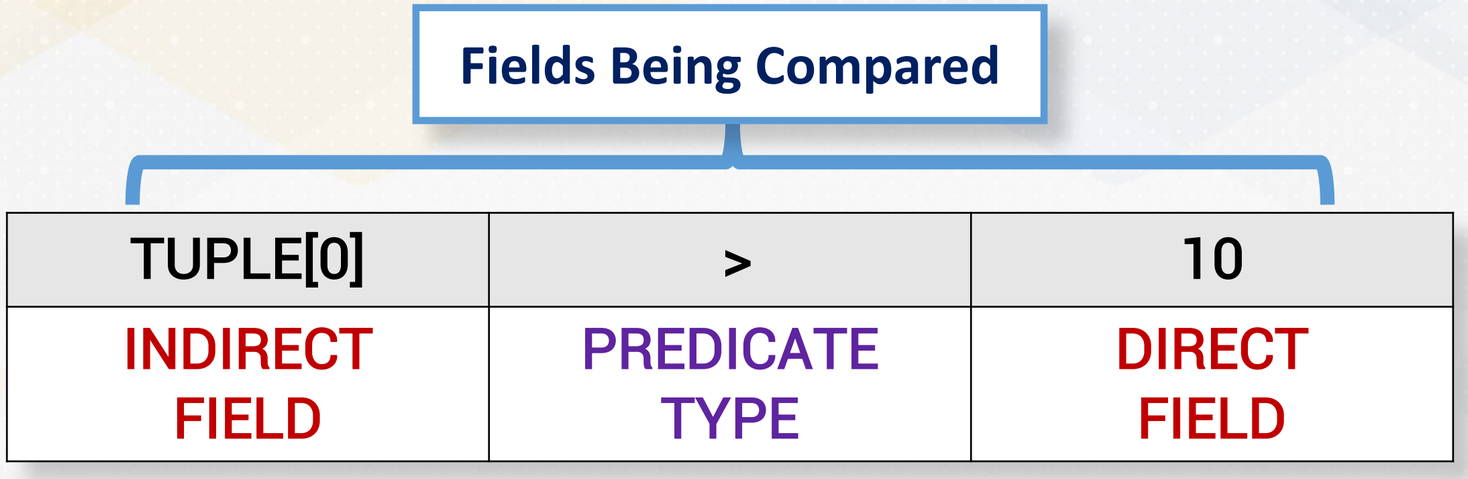
Predicate Example

```
// 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;
```

Select Operator

Select Operator



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
}
```

More Complex Predicate

Need for More Complex Predicate

```sql
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]	<	6
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]	<	6
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>(6)),
      SimplePredicate::ComparisonOperator::LT);
```

Illustrative Complex Predicate

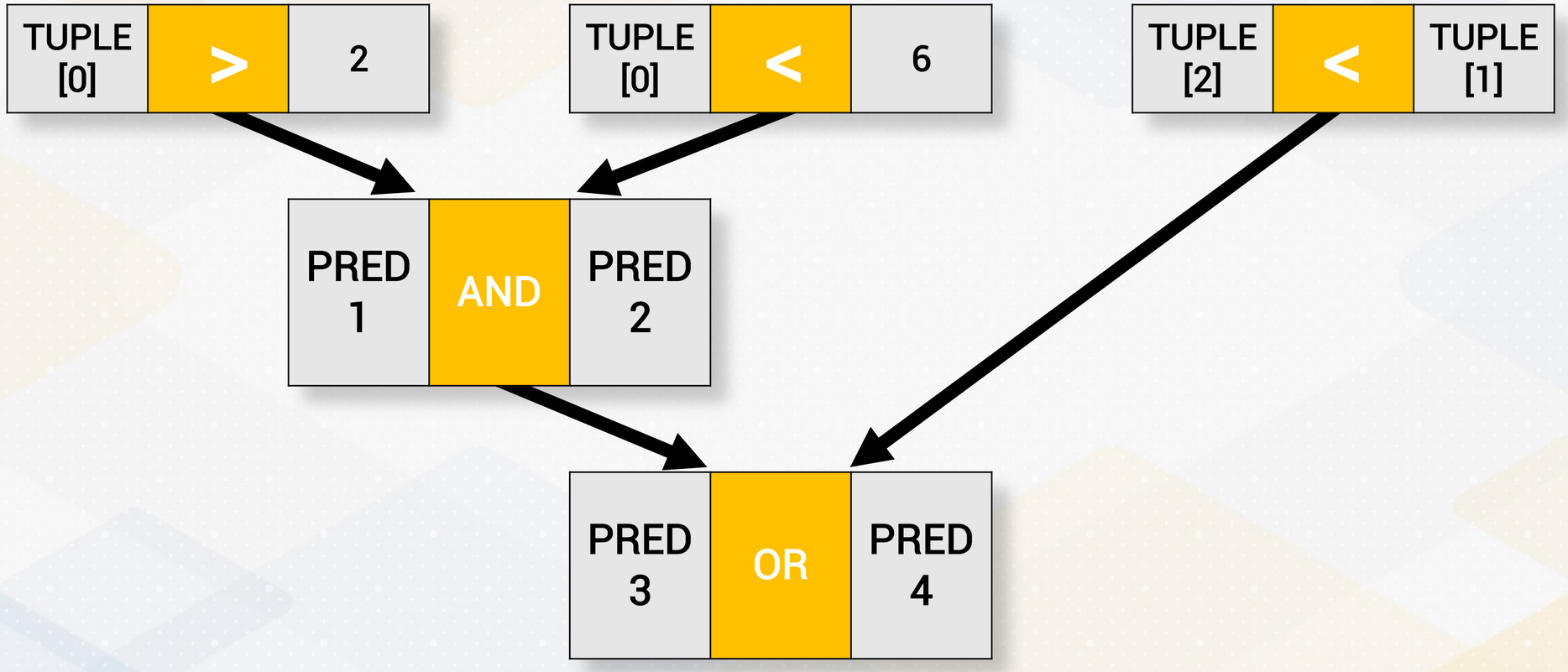
PREDICATE 1	AND	PREDICATE 2
IPREDICATE OBJECT POINTER	LOGICAL OPERATOR	IPREDICATE OBJECT POINTER

```
// 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

```sql
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

```sql
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));
  }
}
```

Query Parsing

Query Parsing


```
Query 1: "{1} WHERE {1} > 2 AND {1} < 6"
Query 2: "SUM{1} WHERE {1} > 2 AND {1} < 6"
Query 3: "SUM{1} GROUP BY {2} WHERE {1} > 2 AND {1} < 6"
```

Query Parsing

Regex

Query Parsing

```
QueryComponents parseQuery(const std::string &query) {
  QueryComponents components;
  // Example: Parse SELECT attributes
  std::regex selectRegex("\\{(\\d+)\\}(, \\{(\\d+)\\})?");
  std::smatch selectMatches;
  std::string::const_iterator queryStart(query.cbegin());
  while (std::regex_search(queryStart, query.cend(), selectMatches, selectRegex)) {
      ...
  }
  // Further parsing for SUM, GROUP BY, and WHERE conditions
  return components;
}
```

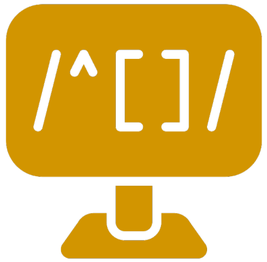
Query Components

```
struct QueryComponents {
  std::vector<int> selectAttributes;
  bool sumOperation = false;
  int sumAttributeIndex = -1;
  bool groupBy = false;
  int groupByAttributeIndex = -1;
  ...
};
```

Query Components

COMPONENT	TYPE	DESCRIPTION
selectAttributes	std::vector<int>	Indices of attributes selected in the query.
sumOperation	bool	Indicates whether a SUM operation is included.
sumAttributeIndex	int	Targets the attribute for the SUM operation.
groupBy	bool	Specifies if there's a GROUP BY clause.
groupByAttributeIndex	int	Determines which attribute to group by.
whereCondition	bool	Checks for the presence of a WHERE clause with bounds.
whereAttributeIndex	int	Determines which attribute to filter by.
lowerBound	int	Specifies the lower bound for the filtered column
upperBound	int	Specifies the upper bound for the filtered column

Query Parsing



Regex Expressions

REGULAR EXPRESSIONS USED

Regular Expression

REGULAR EXPRESSION	QUERY STRING
SUM\{(\d+)\}	SUM{3}

SUM
Detects & Parses SUM Operation

SUM
Matches Text "SUM"

d+
Capturing Group Matches 1 or More Digits

Pretty Printing Parsed Query

```
void prettyPrint(const QueryComponents &components) {
  std::cout << "Query Components:\n";
  std::cout << "  Selected Attributes: ";
  for (auto attr : components.selectAttributes) {
    std::cout << "{" << attr + 1 << "} ";
  }
  ...
}
```

executeQuery

```
void executeQuery(const QueryComponents &components,
                  BufferManager &buffer_manager) {
  ScanOperator scanOp(buffer_manager);
  Operator *rootOp = &scanOp; // Start with the basic scan operation
  std::optional<SelectOperator> selectOpBuffer;
  std::optional<HashAggregationOperator> hashAggOpBuffer;
  // Apply WHERE conditions
  if (components.whereAttributeIndex != -1) {
    // Construct predicates and a complex predicate for AND conditions
    selectOpBuffer.emplace(*rootOp, std::move(complexPredicate));
    rootOp = &*selectOpBuffer; // Chain the select operator
  }
  // Execute the query
}
```

WHERE clause

```
if (components.whereAttributeIndex != -1) {
  auto complexPredicate = makeComplexPredicate(components);
  selectOpBuffer.emplace(*rootOp, std::move(complexPredicate));
  rootOp = &*selectOpBuffer;
}
```

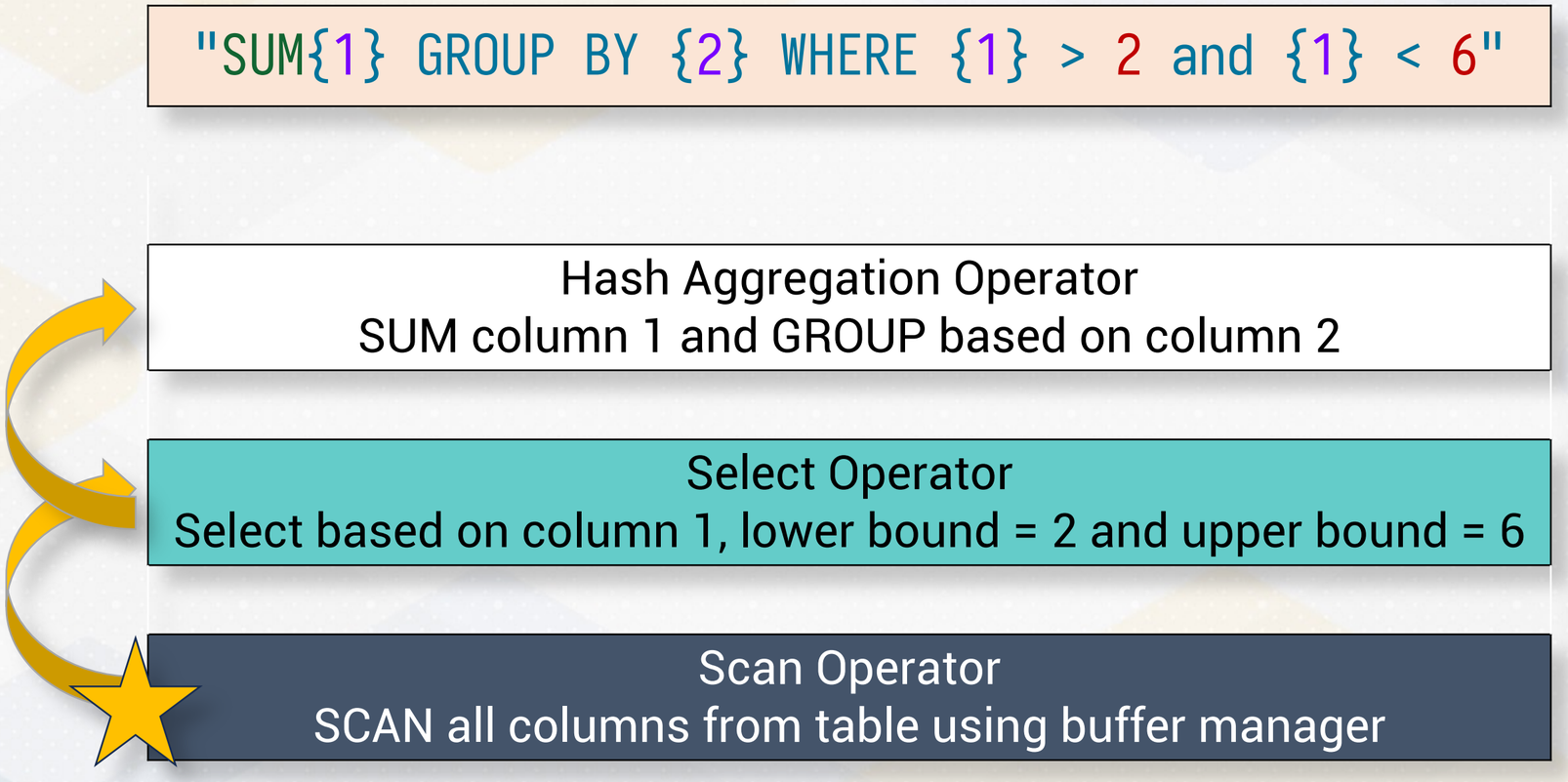
Grouping and Aggregation

```
if (components.sumOperation || components.groupBy) {
  std::vector<AggrFunc> aggrFuncs = prepareAggregationFunctions(components);
  hashAggOpBuffer.emplace(*rootOp, groupByAttrs, aggrFuncs);
  rootOp = &*hashAggOpBuffer;
}
```

Final Query Execution

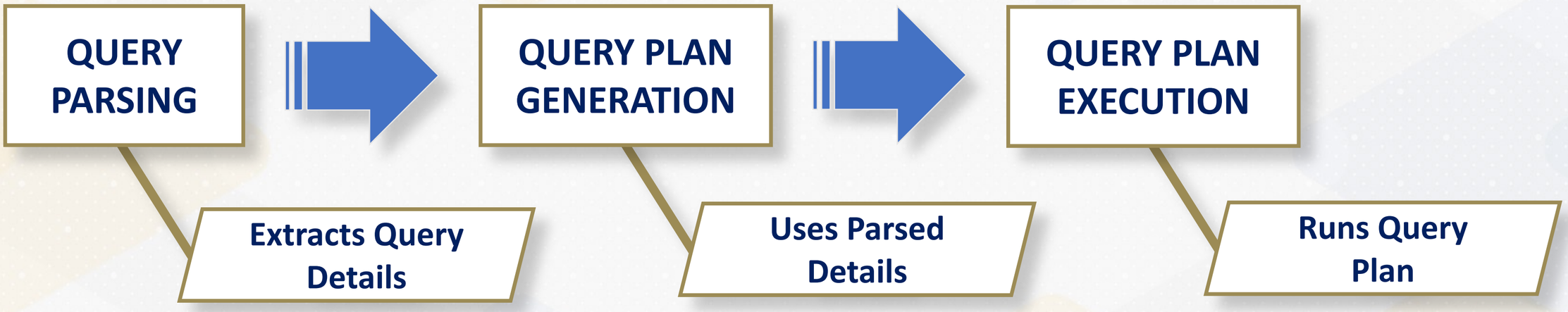
```
rootOp->open();
while (rootOp->next()) {
  const auto &output = rootOp->getOutput();
  printTupleFields(output);
}
rootOp->close();
```

Example


"SUM{1} GROUP BY {2} WHERE {1} > 2 and {1} < 6"
Hash Aggregation Operator
SUM column 1 and GROUP based on column 2
Select Operator
Select based on column 1, lower bound = 2 and upper bound = 6
Scan Operator
SCAN all columns from table using buffer manager

Query Compilation

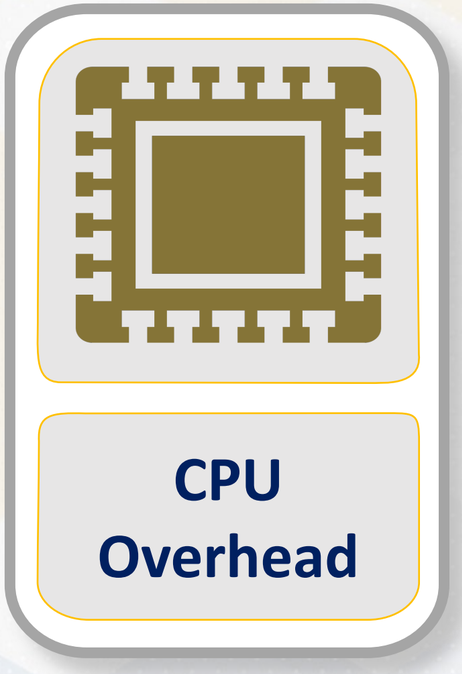
Query Interpretation

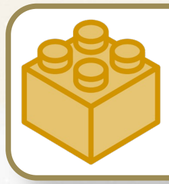


Query Interpretation

```
void queryInterpretation() {
  std::vector<std::string> test_queries = {
      "SUM{1} GROUP BY {1} WHERE {1} > 2 and {1} < 6"};
  for (const auto &query : test_queries) {
    auto components = parseQuery(query);
    executeQuery(components, buffer_manager);
  }
}
```

Query Interpretation





Operator Framework

Compute Overhead

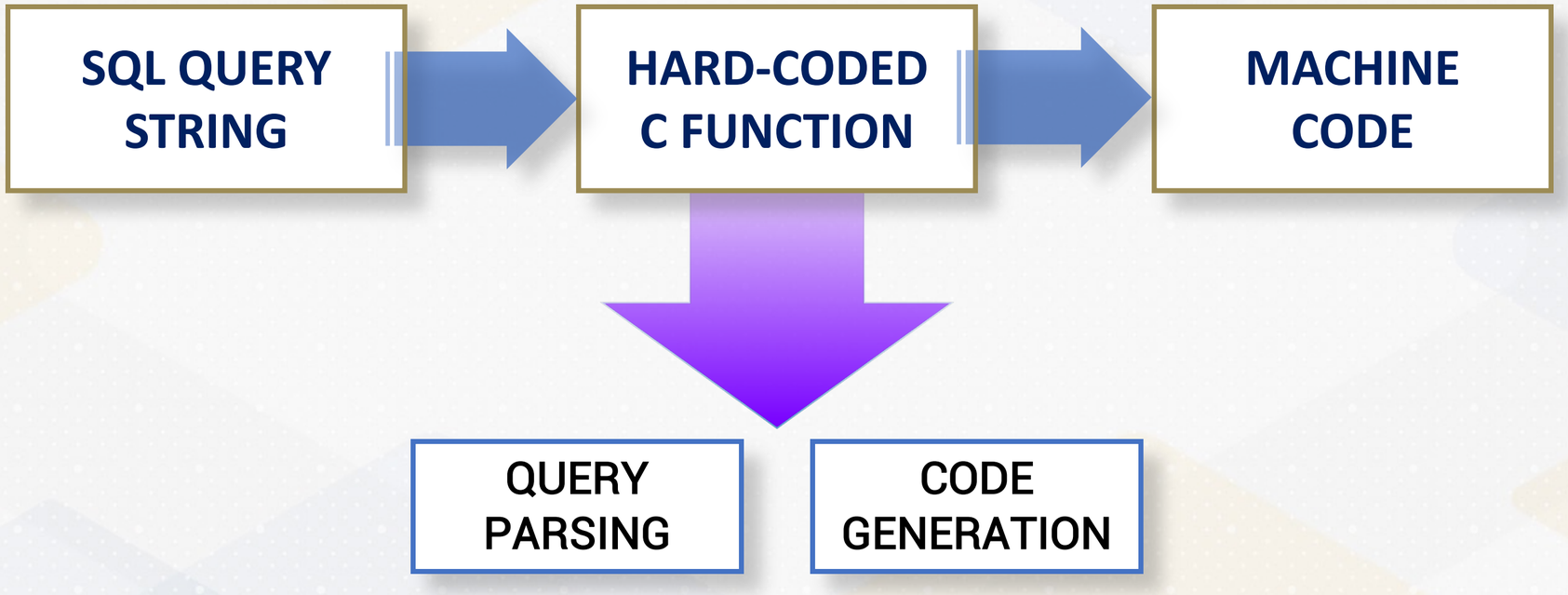
Tuple Deserialization

Memory Overhead

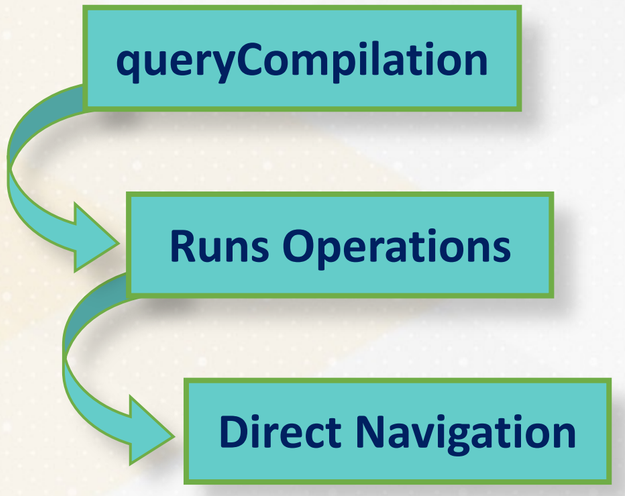
Polymorphism
(Operator* rootOp)

Temporary Objects
std::unique_ptr<Field>

Query Compilation



Query Compilation: Scanning




```
void queryCompilation() {
  while (currentPageIndex < buffer_manager.getNumPages()) {
    auto &currentPage = buffer_manager.getPage(currentPageIndex);
    Slot *slot_array = reinterpret_cast<Slot *>(page_buffer);
    while (currentSlotIndex < MAX_SLOTS) {
      if (!slot_array[currentSlotIndex].empty) {
        ...                    // Process tuple in slot
        currentSlotIndex++; // Move to the next slot
        continue;              // Continue scanning
      }
      currentSlotIndex++;
    }
  }
}
```

Query Compilation: Field Extraction

```
const char *tuple_data = page_buffer + slot_array[currentSlotIndex].offset;
// Navigate to the first field -- text based hard-coded deserialization
// "4 0 4 2 0 4 231 1 4 132.04 2 7 buzzdb"
std::string integerField;
int character_count = 0; int current_count = 0;
int required_count = 4; // to retrieve 2 (first field)
while (current_count < required_count) {
  integerField = "";
  for (const char *p = tuple_data + character_count; *p != ' ' && *p != '\0'; ++p) {
    character_count++; integerField += *p;
  }
  character_count++; current_count++;
}
int fieldValue = std::stoi(integerField);
```

Query Compilation: Aggregation

```
// Maps to store the sum and count for each group
std::unordered_map<int, int> groupSums;

// Apply WHERE condition
if (fieldValue > 2 && fieldValue < 6) {
  groupSums[fieldValue] += fieldValue;
}

// Output the results
for (const auto &group : groupSums) {
  std::cout << "KEY " << group.first;
  std::cout << ", SUM: " << groupSums[group.first] << std::endl;
}
```

Advantages of Query Compilation

1

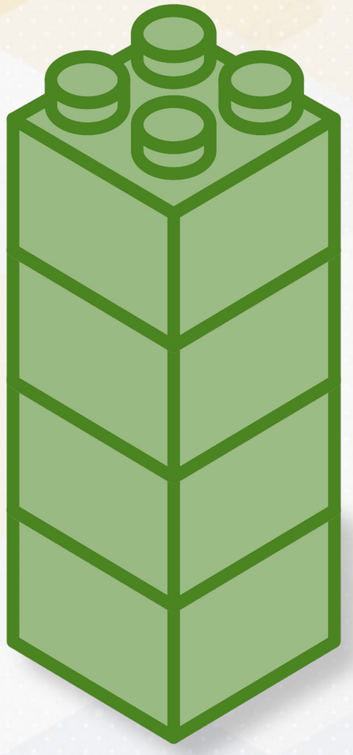
REDUCED INTERPRETATION OVERHEAD

COMPILED QUERY 30 times faster than INTERPRETED QUERY

2

INCREASED EXECUTION SPEED

Drawbacks of Query Compilation



Compilation Process

Long Compile Time

Reduced Flexibility

Additional Compile Time

Columnar Storage

Weather Analysis

- A weather dataset with columns for timestamp, temperature, humidity, and wind speed.
- **Query:** Find average temperate between timestamp 100 and timestamp 150

Row Storage

- Find average temperate between timestamp 5 and timestamp 15

PAGE 1			
Timestamp 1	Temperature 1	Humidity 1	Windspeed 1
Timestamp 2	Temperature 2	Humidity 2	Windspeed 2
…	…	…	…
Timestamp 10	Temperature 10	Humidity 10	Windspeed 10

PAGE 2			
Timestamp 11	Temperature 11	Humidity 11	Windspeed 11
…	…	…	…
Timestamp 20	Temperature 20	Humidity 20	Windspeed 20

Why Columnar Storage?

- **Row Storage vs. Columnar Storage**: Row storage stores all fields of a record together, while columnar storage organizes data by columns.
- **Benefits**: Columnar storage is ideal for analytical queries, which often access only a subset of columns.
- **Example Use Case**: A weather dataset with columns for timestamp, temperature, humidity, and wind speed.

Columnar Storage

- Find average temperate between timestamp 5 and timestamp 15

PAGE 1
Timestamp 1
Timestamp 2
…
Timestamp 80

PAGE 2
Temperature 1
Temperature 2
…
Temperature 80

PAGE 3
Humidity 1
Humidity 2
…
Humidity 80

PAGE 4
Windspeed 1
Windspeed 2
…
Windspeed 80

Row vs Columnar Storage

FEATURE	ROW STORAGE	COLUMNAR STORAGE
Data Organization	All attributes of a record stored together	Each attribute stored in separate files
Ideal Use Case	Transactional workloads (OLTP), where entire records are accessed frequently.	Analytical workloads (OLAP), which often require only a subset of columns
Access Pattern	Efficient for accessing full rows	Efficient for accessing specific columns
Compression	Limited compression potential	High compression potential due to similar data types within columns

Generated Weather Data

Timestamp	Temperature	Humidity	Wind Speed
1609459200	40	74	12
1609459202	37	84	14
1609459207	35	6	31
1609459210	35	90	59

Row Storage

- Single file storing rows (tuples) sequentially.

```
std::ofstream rowFile("row_storage.dat", std::ios::binary);
for (const auto& row : data) {
    rowFile.write(reinterpret_cast<const char*>(row.data()),
                  NUM_COLS * sizeof(int));
}
```

Columnar Storage

- Separate files for each column in the table in binary format.

```
std::vector<std::ofstream> colFiles(NUM_COLS);
for (int i = 0; i < NUM_COLS; ++i) {
    colFiles[i].open("column_storage_" + std::to_string(i) + ".dat",
                     std::ios::binary);
}
```

Querying Data with Row Storage

- Scans all rows and filters by timestamp, accumulating temperature values within the range.

```
double queryAverageTemperatureRowStorage(int startTimestamp, int endTimestamp) {
    std::ifstream rowFile("row_storage.dat", std::ios::binary);
    int sumTemperatures = 0, count = 0;
    while (rowFile.read(reinterpret_cast<char*>(row.data()), NUM_COLS * sizeof(int))) {
        if (row[0] >= startTimestamp && row[0] <= endTimestamp) {
            sumTemperatures += row[1];
            count++;
        }
    }
    return count > 0 ? static_cast<double>(sumTemperatures) / count : 0.0;
}
```

Querying Data with Row Storage

- **Limitations**: Processes entire rows, even though only one column (temperature) is needed.

```
double queryAverageTemperatureRowStorage(int startTimestamp, int endTimestamp) {
    std::ifstream rowFile("row_storage.dat", std::ios::binary);
    int sumTemperatures = 0, count = 0;
    while (rowFile.read(reinterpret_cast<char*>(row.data()), NUM_COLS * sizeof(int))) {
        if (row[0] >= startTimestamp && row[0] <= endTimestamp) {
            sumTemperatures += row[1];
            count++;
        }
    }
    return count > 0 ? static_cast<double>(sumTemperatures) / count : 0.0;
}
```

Querying Data with Columnar Storage

- Only the timestamp and temperature files are accessed, making the query faster than row storage.

```
long long queryColumnarStorage(int& filterPagesRead, int& aggregatePagesRead) {
    std::ifstream filterFile("column_storage_" + std::to_string(FILTER_COLUMN) + ".dat",
                             std::ios::binary);
    std::ifstream aggregateFile("column_storage_" + std::to_string(AGGREGATE_COLUMN) + ".dat",
                             std::ios::binary);
    long long sum = 0;
    std::unordered_map<int, std::vector<int>> pageOffsetMap;
    ..
}
```

Querying Data with Columnar Storage

- Figure out the qualifying tuples by going over the temperature file.

```
// Read the filter column and collect row offsets for qualifying rows
    for (int startRow = 0; startRow < NUM_ROWS; startRow += INTS_PER_PAGE) {
        auto filterPage = readColumnPage(filterFile, startRow, filterPagesRead);
        for (size_t i = 0; i < filterPage.size(); ++i) {
            //std::cout << filterPage[i] << "\n";
            if (filterPage[i] > FILTER_THRESHOLD) {
                pageOffsetMap[startRow / INTS_PER_PAGE].push_back(i);
            }
        }
    }
```

Querying Data with Columnar Storage

- Aggregate the temperature column using the collected tuple offset

```
// Read the aggregate column using the collected row offsets
    for (const auto& [pageIndex, offsets] : pageOffsetMap) {
        auto aggregatePage = readColumnPage(aggregateFile, pageIndex * INTS_PER_PAGE,
aggregatePagesRead);
        for (const auto& rowIndex : offsets) {
            sum += aggregatePage[rowIndex];
        }
    }
```

Illustrative Results

- Fewer pages are read from disk with columnar storage

```
Filter Selectivity: 0.003416
Row Storage Query Result: 7358
Row Storage Query Time: 0.112099 seconds
Columnar Storage Query Result: 7358
Columnar Storage Query Time: 0.00361763 seconds

Pages Read:
Total Row Storage Pages Read: 3907
Filter Pages Read: 489
Aggregate Pages Read: 473
Total Columnar Storage Pages Read: 962
```

Compressed Columnar Storage

Compression in Databases

- **Storage Reduction**: Compression reduces the space required for large tables.
- **Performance Gains**: Reduces I/O time by storing smaller, compressed data on disk.

Compression and Columnar Storage

- **Row Storage**: Mixed data types make it challenging to compress entire rows effectively, as each row contains diverse data (e.g., timestamps, names, quantities).
- **Columnar Storage**: Compression can target the specific patterns within each column, such as delta encoding for timestamps or bit-packing for small integer ranges.

Compression and Columnar Storage

- **Uniform Data Types per Column**: Each column contains only one data type (e.g., integers for temperature, strings for product IDs), which often exhibit similar values or ranges.
- **Increased Data Redundancy**: Since similar values are often grouped together in columns (e.g., temperatures in small ranges, or product IDs with repeated entries), compression algorithms are highly effective.
- **Reduced I/O for Query Performance**: Columnar compression enables more data to fit in memory, reducing I/O operations and speeding up query performance, especially for analytical workloads that access only a subset of columns.

Compression and Columnar Storage

Timestamp
1609459200
1609459202
1609459207
1609459210

Timestamp
1609459200
+2
+5
+3
+2
+5
+3
+5

Compression Algorithms

ALGORITHM	IDEAL USE CASE
Bit-Packing	Small-range numeric values
Huffman Coding	Frequent categorical strings
Byte Dictionary	Repeated values, fewer than 256

Illustrative Results

```
Average Temperature (Row Storage)     : 36.9554°C
Query Time (Row Storage)              : 17.57 milliseconds
Query Time (Columnar Storage)         :  5.13 milliseconds
Query time on compressed data         :  4.08 milliseconds
```

Vectorized Execution

Limitations of Tuple-at-a-time Processing

- Each tuple incurs the cost of:
  - Function calls between operators.
  - Deserializing, interpreting, and processing the tuples.
  - Doesn't leverage the CPU's ability to process batches of data efficiently.
  - Results in frequent pipeline stalls and cache misses.

Vector-at-a-time Processing

- Process a vector of tuples at a time to reduce overhead of function calls etc.
- Use SIMD (Single Instruction, Multiple Data) instructions
- A single instruction operates on multiple data points simultaneously.

18	40	25	15	
				> 20
0	1	1	1	

SIMD in Query Execution

- **SIMD Use Cases**:
  - Filtering (e.g., select rows within a range).
  - Aggregations (e.g., sum, average).
  - Compression (e.g., decoding bit-packed data).
- **Key SIMD Operations**
  - **Vector Loads**: Load multiple data points.
  - **Masks**: Apply conditions to filter data.
  - **Horizontal Reduction**: Sum vector elements.

Filter timestamps using SIMD

- Load data in **batches** using vld1q_s32.
- Perform **vectorized comparisons** using vcgeq_s32 and vcleq_s32.
- Combine results with a **bitwise AND** using vandq_u32.

```
int32x4_t ts_vec = vld1q_s32(&data.timestamps[i]);  // Load timestamps
uint32x4_t mask = vandq_u32(vcgeq_s32(ts_vec, 1),
                            vcleq_s32(ts_vec, end)); // Mask for range
```

Filter timestamps using SIMD

vld1q_s32	18	40	25	15	
vcgeq_s32					> 20
	0	1	1	1	
vcleq_s32					< 30
	1	0	1	1	
vandq_u32					> 20 AND < 30
	0	0	1	0	

Scalar vs SIMD Execution

- **Scalar Execution**:
  - Processes **one data element per cycle**.
  - Repeated instruction fetch, decode, and execute for each element.
- **SIMD Execution**:
  - Processes **multiple data elements per cycle** by leveraging wide registers.
  - Executes the same operation on an entire vector (batch) with a single instruction.

Benefits of SIMD Execution

- **Instruction-Level Efficiency**
  - Fewer instructions due to vectorized operations.
  - Scalar processing requires **N instructions** for **N data points**.
  - SIMD requires **N / W instructions**, where **W** is the width of the SIMD vector.
- **Cache and Memory Efficiency**
  - Contiguous memory access aligns with cache lines.
  - SIMD operates on **contiguous memory** (columnar layouts align well with SIMD).
  - Cache lines are fully utilized, reducing memory latency.

Benefits of SIMD Execution

- **Minimized Control Overhead**:
  - SIMD minimizes branching by applying the same operation to all elements in a vector.
  - With scalar execution, pipeline stalls if the branch prediction is incorrect.
  - With SIMD execution, masks handle conditional operations, avoiding pipeline stalls.
- **Hardware Support**:
  - Modern CPUs have dedicated SIMD execution units optimized for throughput.

SIMD Query

- Task: Calculate the average temperature within a timestamp range

Timestamps	1	2	5	7
Temperature	25.5	26.0	27.2	28.3
> 2	0	1	1	0
< 6	1	1	1	0
> 2 AND < 6	0	1	1	0
Masked Temps	0	26.0	27.2	0
Sum	53.2			

SIMD Query: Loading Data

- Load 4 consecutive timestamps and temperatures into SIMD registers.

```
int32x4_t ts_vec = vld1q_s32(&data.timestamps[i]);  // Load 4 timestamps
float32x4_t temp_vec = vld1q_f32(&data.temperatures[i]);  // Load 4 temperatures
```

SIMD Query: Filtering Timestamps

- Filter timestamps within the query range.
- **Compare for Lower Bound**: vcgeq_s32: Compares timestamps with startTimestamp.
- **Compare for Upper Bound**: vcleq_s32: Compares timestamps with endTimestamp.
- **Combine Results**: vandq_u32: Combines the two masks with a bitwise AND.

```
uint32x4_t in_range_mask = vandq_u32(vcgeq_s32(ts_vec, vdupq_n_s32(startTimestamp)),
                                     vcleq_s32(ts_vec, vdupq_n_s32(endTimestamp)));
```

SIMD Query: Mask Application

- Apply the mask to the temperatures and sum up the valid values.
- vmulq_f32: Multiplies the mask with the temperature vector.
- Keeps valid temperatures, zeros out invalid ones.
- vaddq_f32: Adds the valid temperatures to the running sum.

```
float32x4_t masked_temps = vmulq_f32(temp_vec, vcvtq_f32_u32(in_range_mask));
sum_vec = vaddq_f32(sum_vec, masked_temps);
```

SIMD Query: Final Steps

- **Horizontal Reduction**: Sum up all elements in the SIMD vector.
- **Handle Remaining Scalar Elements**: Process leftover elements not divisible by the SIMD width.

```
float total_sum = vaddvq_f32(sum_vec);
for (int i = count - (count % 4); i < count; ++i) {
    if (data.timestamps[i] >= startTimestamp && data.timestamps[i] <= endTimestamp) {
        total_sum += data.temperatures[i];
    }
}
```

Conclusion

- Modular Query Execution
- Scan Operator
- Predicate & Select Operator
- Aggregation Operator
- Query Parsing & Compilation
- Columnar Storage
- Compressed Columnar Storage
- Vectorized Execution