Lecture 21: Query Parsing

Logistics

- Exercise sheets 2 due on **Nov 21**
- Exercise sheet 3 and assignment 4 due on **Nov 28**
- Project final reports due on **Nov 17**
  - Share GitHub link immediately after project title
  - Focus on completed concrete tasks
  - Benchmarking performance etc.
- In-class presentations on **Nov 21**

Recap

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

Lecture Overview

- Database and Unix
- DDL Operator
- Query Parsing

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

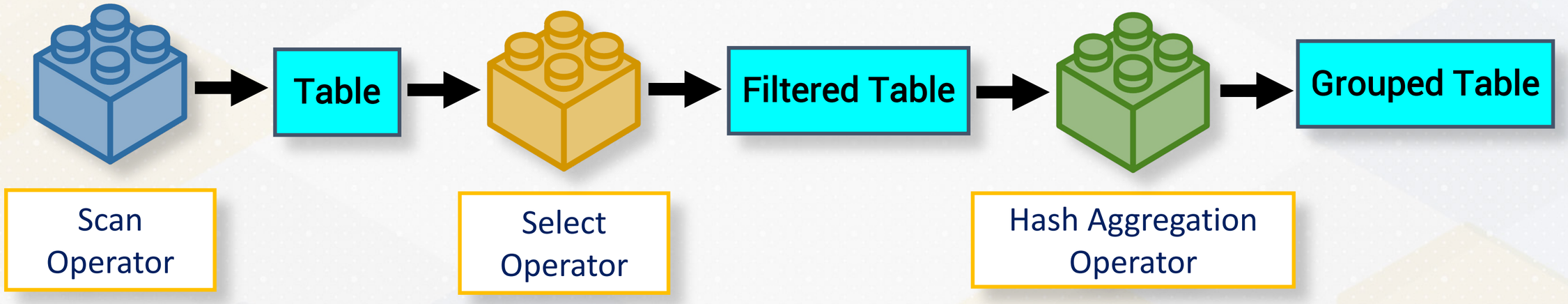
Scans Tuples & Builds Index

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

Unix creators: Ken Thompson and Dennis Ritchie

Simplicity and Modularity

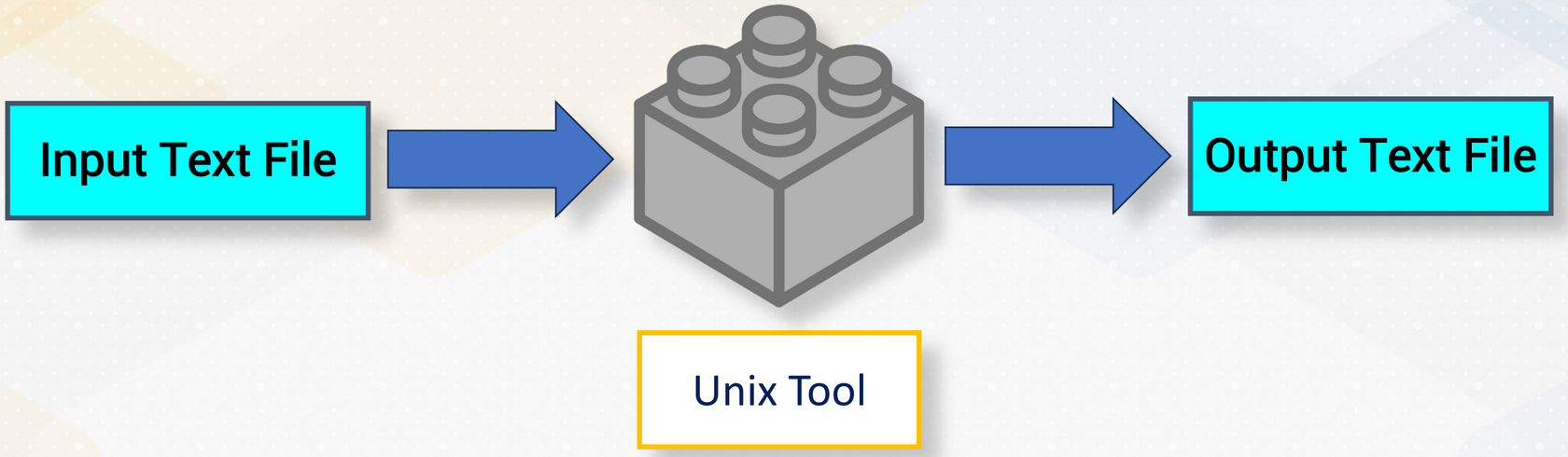
Each tool should do one and only one task well

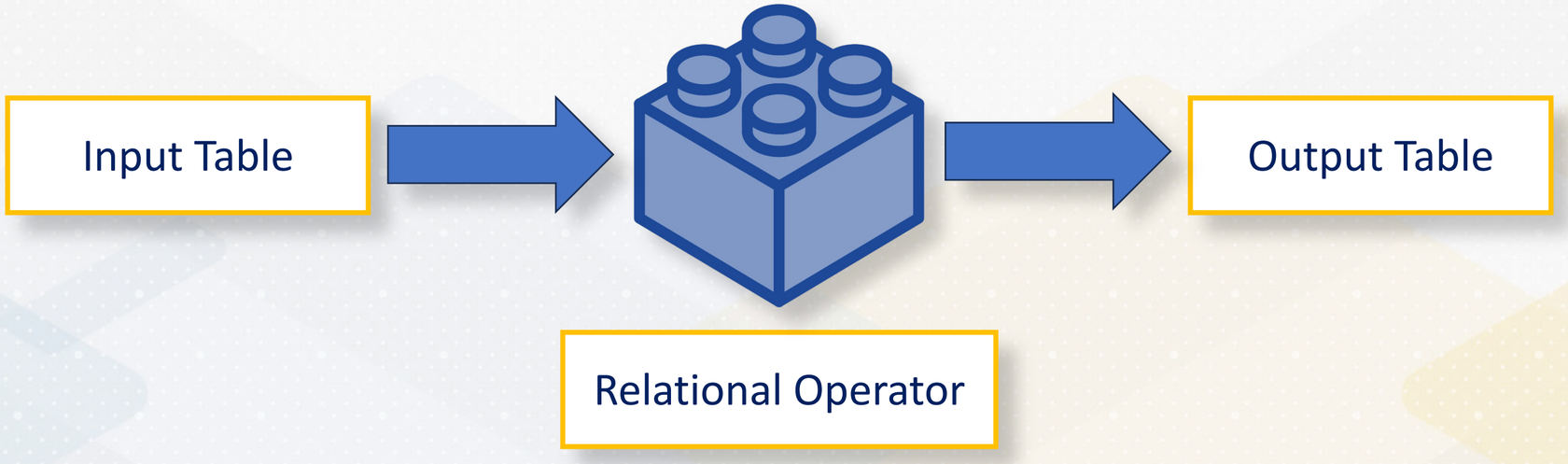
```
// GREP TOOL
grep “error” file.txt

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

Pipelining and Composability

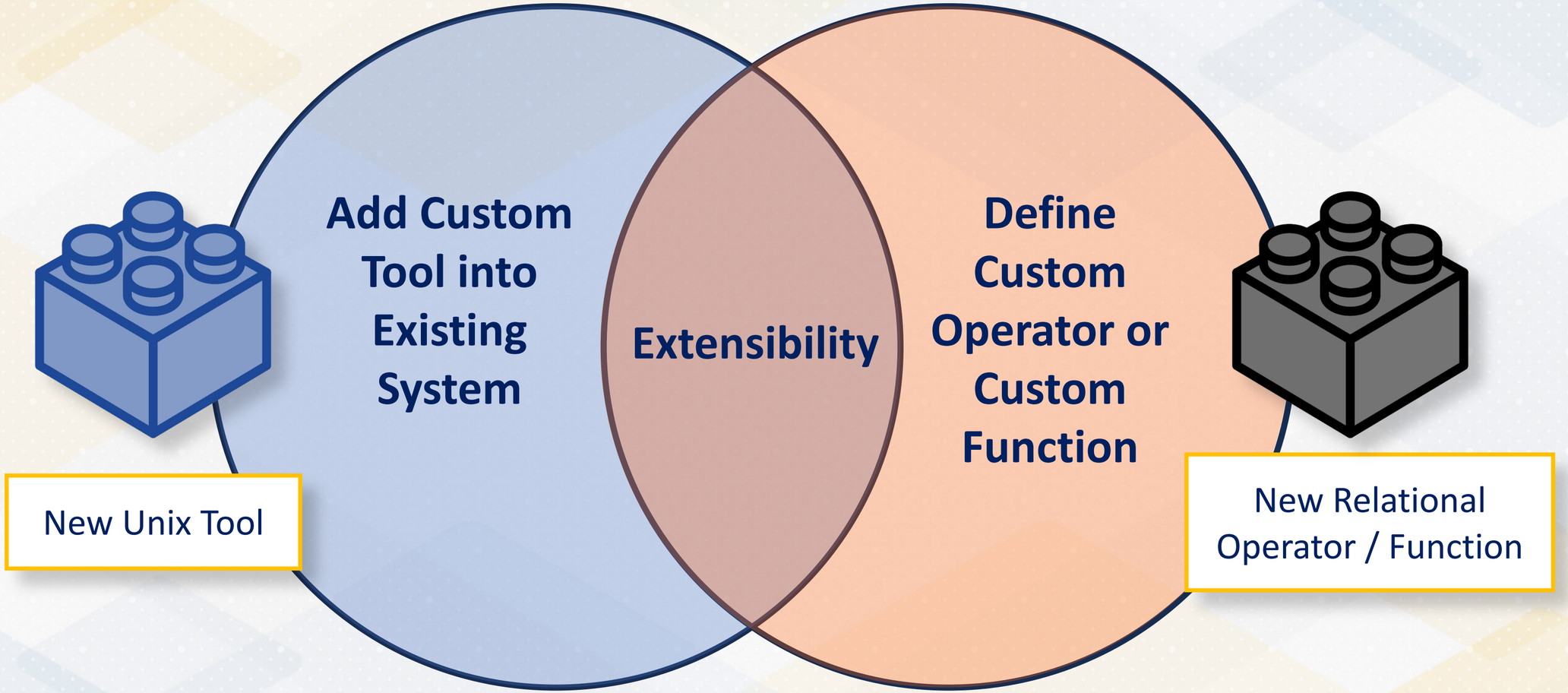
```
// UNIX TOOL PIPELINE
cat file.txt | grep "error" | sort | uniq -c

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

Standardized Interface





Extensibility



Create Index Operator

SQL

Consists of Sub-Languages

Data Definition Language (DDL)	Data Manipulation Language (DML)

Data Definition Language (DDL)

OPERATION	OBJECT

Tables

Primary Data Structures for Tuples

Indexes

Derived Data Structures

Views

Virtual Tables

Create Index Operator

Builds index on a specific column

HASH INDEX
Column Offset: 1

EMP ID	SALARY	AGE
1	4000	35
2	5000	25
3	5000	55

KEY (SALARY)	VALUE (EMP ID)
4000	1
5000	2, 3

Create Index Executor

```
void open() override {
  input->open();
  while (input->next()) {
    const auto &tuple = input->getOutput();
    int key = tuple[attributeIndex]->asInt();
    int value = tuple[1]->asInt(); // Example uses a fixed
second attribute
    hash_index.insertOrUpdate(key, value);
  }
}
```

Create Index Executor

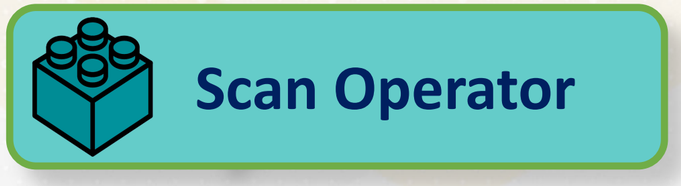
```
bool next() override {
  // Index creation is done in open(), so nothing to do here.
  return false;
}
```

Accessing the Index

Index accessed using getIndex()

```
const HashIndex &getIndex() const {
    return hash_index;
}
```

Create Index Operator Usage




```
// We want to index on the second attribute
CreateIndexOperator createIndexOperator(scanOperator, 1);
createIndexOperator.open(); // Builds the index
createIndexOperator.close();

const auto &index = createIndexOperator.getIndex();
index.print(); // Displays the index contents
```

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
}
```

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;
}
```

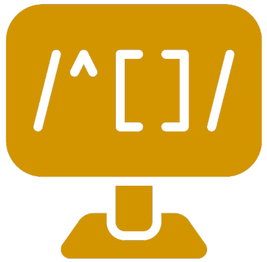
Query Components

```
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

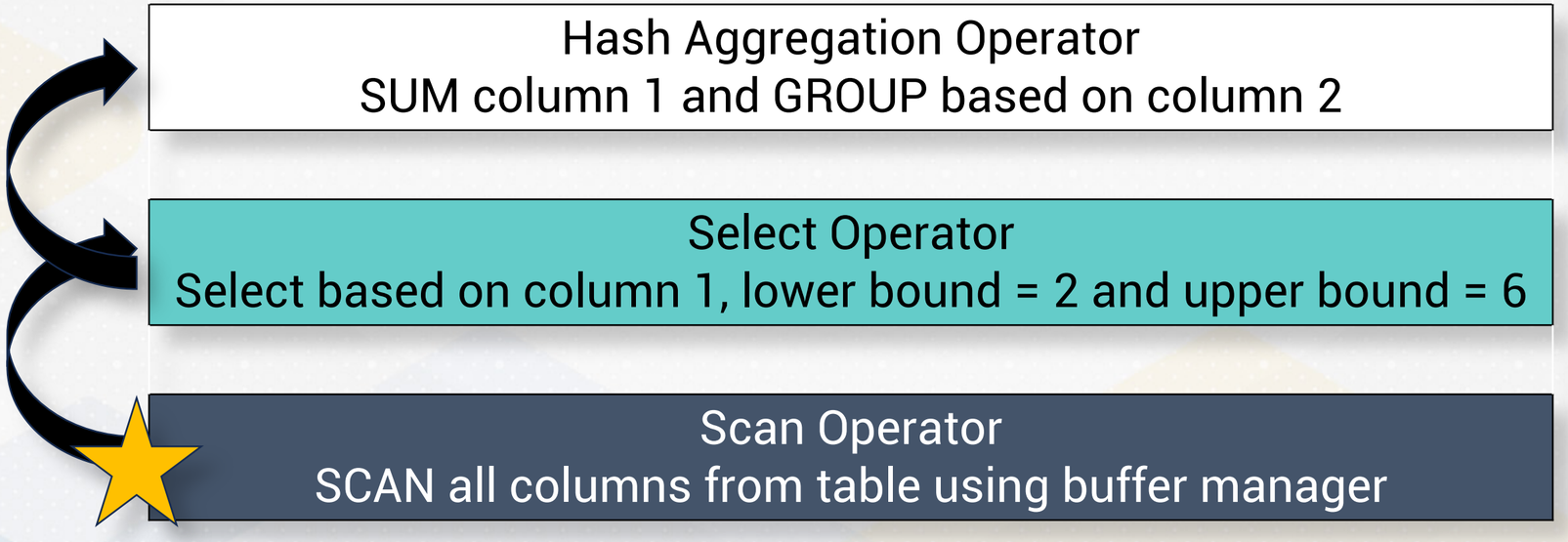
```
if (components.sumOperation || components.groupBy) {
  std::vector<AggrFunc> aggrFuncs = prepareAggregationFunctions(components);
  hashAggOpBuffer.emplace(*rootOp, groupByAttrs, aggrFuncs);
  rootOp = &*hashAggOpBuffer;
}
```

Final Query Execution

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

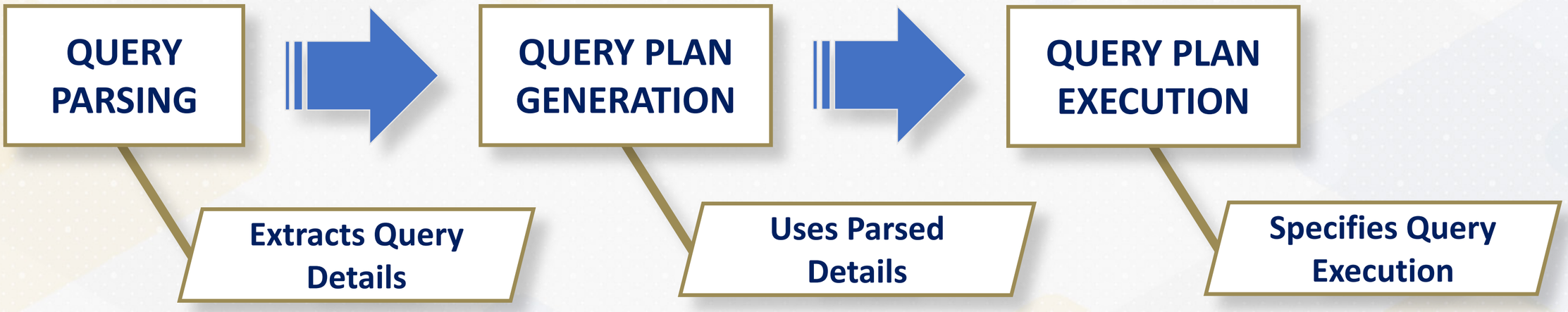
Example

```
"SUM{1} GROUP BY {2} WHERE {1} > 2 and {1} < 6"
```



Query Compilation

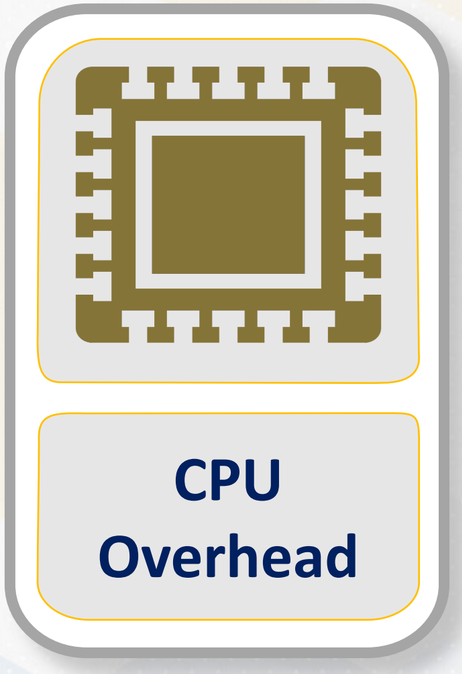
Query Interpretation

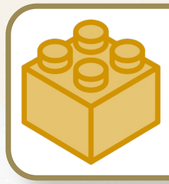


Query Interpretation

```
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

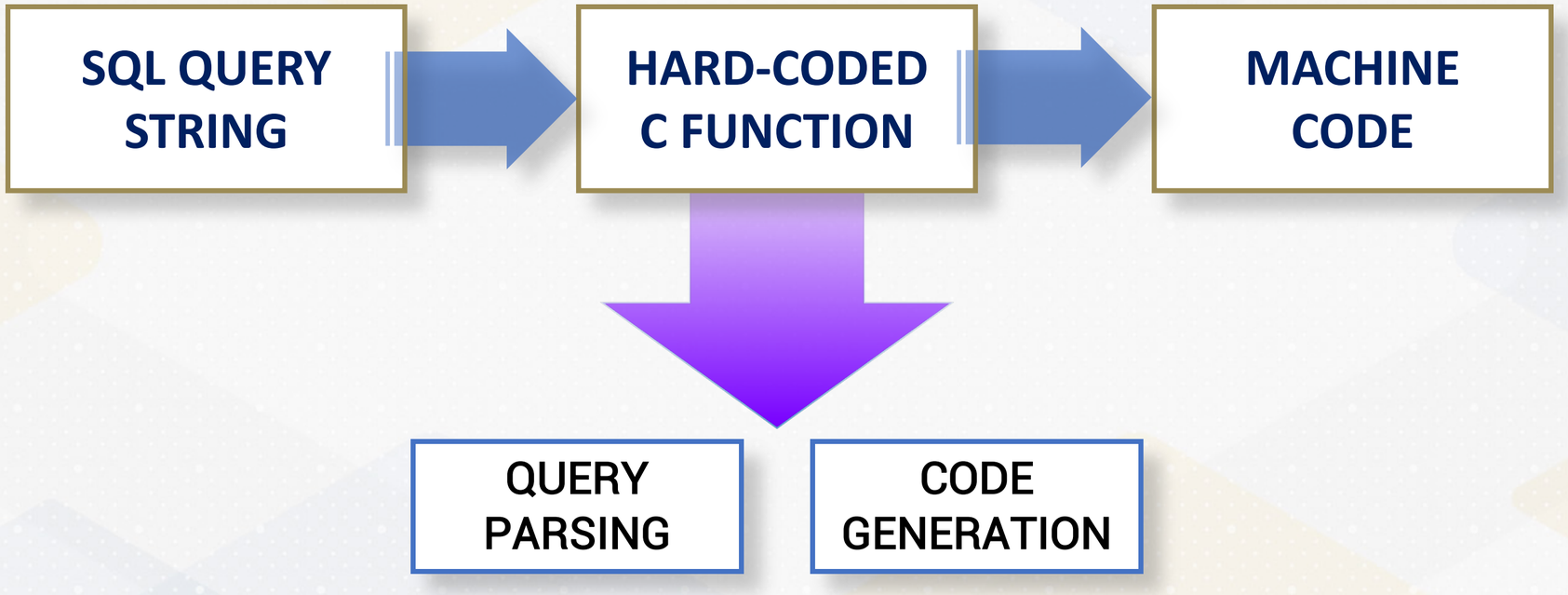
Tuple Deserialization

Memory Overhead

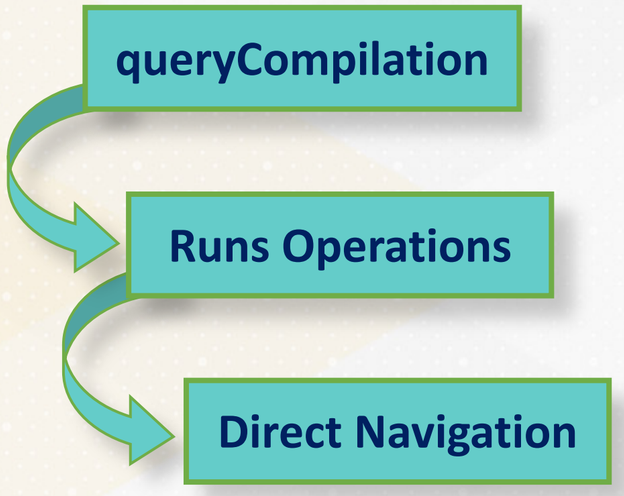
Polymorphism
(Operator* rootOp)

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

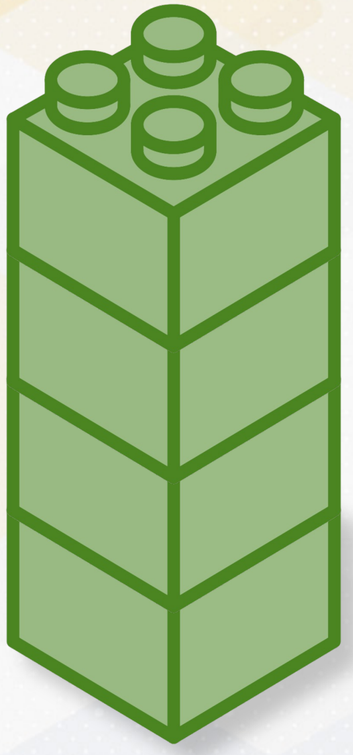
REDUCED INTERPRETATION OVERHEAD

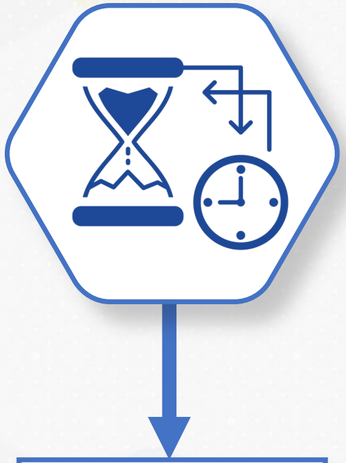
COMPILED QUERY 30 times faster than INTERPRETED QUERY

2

INCREASED EXECUTION SPEED

Drawbacks of Query Compilation





Compilation Process

Long Compile Time

Reduced Flexibility

Additional Compile Time

Conclusion

- Database and Unix
- DDL Operator
- Query Parsing
- Query Compilation