Acknowledgments

This book would not have been possible without the support of my family, who are incredibly patient with me when I get absorbed with yet another side project.

Special thanks go to Matthew Powers, a.k.a. Mr. Powers, for inspiring me to write this book in the first place. Matthew is the author of the book "Writing Beautiful Apache Spark Code" which is also available on Leanpub.

I also need to thank the countless people who have interacted with me over the past couple of years as I worked on the DataFusion project, especially the Apache Arrow PMC, committers, and contributors.

Finally, I want to thank Chris George and Joe Buszkiewic for their support and encouragement while working at RMS, where I further developed my understanding of query engines.

This book is also available for purchase in ePub, MOBI, and PDF format from https://leanpub.com/how-query-engines-work

Introduction

Since starting my first job in software engineering, I have been fascinated with databases and query languages. It seems almost magical to ask a computer a question and get meaningful data back efficiently. After many years of working as a generalist software developer and an end-user of data technologies, I started working for a startup that threw me into the deep end of distributed database development. This is the book that I wish had existed when I started on this journey. Although this is only an introductory-level book, I hope to demystify how query engines work.

My interest in query engines eventually led to me becoming involved in the Apache Arrow project, where I donated the initial Rust implementation in 2018, then donated the DataFusion in-memory query engine in 2019, and finally, donated the Ballista distributed compute project in 2021. I do not plan on building anything else outside the Arrow project and am now continuing to contribute to these projects within Arrow.

The Arrow project now has many active committers and contributors working on the Rust implementation, and it has improved significantly compared to my initial contribution.

Although Rust is a great choice for a high-performance query engine, it is not ideal for teaching the concepts around query engines, so I recently built a new query engine, implemented in Kotlin, as I was writing this book. Kotlin is a very concise language and easy to read, making it possible to include source code examples in this book. I would encourage you to get familiar with the source code as you work your way through this book and consider making some contributions. There is no better way to learn than to get some hands-on experience!

The query engine covered in this book was originally intended to be part of the Ballista project (and was for a while) but as the project evolved, it became apparent that it would make more sense to keep the query engine in Rust and support Java, and other languages, through a UDF mechanism rather than duplicating large amounts of query execution logic in multiple languages.

Now that Ballista has been donated to Apache Arrow, I have updated this book to refer to the query engine in the companion repository simply as "KQuery", short for Kotlin Query Engine, but if anyone has suggestions for a better name, please let me know!

Updates to this book will be made available free of charge as they become available, so please check back occasionally or follow me on Twitter (@andygrove_io) to receive notifications when new content is available.

Feedback

Please send me a DM on Twitter at @andygrove_io or send an email to agrove@apache.org if you have any feedback on this book.

This book is also available for purchase in ePub, MOBI, and PDF format from https://leanpub.com/how-query-engines-work

What Is a Query Engine?

A query engine is a piece of software that can execute queries against data to produce answers to questions, such as:

- What were my average sales by month so far this year?
- What were the five most popular web pages on my site in the past day?
- How does web traffic compare month-by-month with the previous year?

The most widespread query language is Structured Query Language (abbreviated as SQL). Many developers will have encountered relational databases at some point in their careers, such as MySQL, Postgres, Oracle, or SQL Server. All of these databases contain query engines that support SQL.

Here are some example SQL queries.

SQL Example: Average Sales By Month

```
SELECT month, AVG(sales)
FROM product_sales
WHERE year = 2020
GROUP BY month;
```

SQL Example: Top 5 Web Pages Yesterday

```
SELECT page_url, COUNT(*) AS num_visits
FROM apache_log
WHERE event_date = yesterday()
GROUP BY page_url
ORDER BY num_visits DESC
LIMIT 5;
```

SQL is powerful and widely understood but has limitations in the world of so-called "Big Data," where data scientists often need to mix in custom code with their queries. Platforms and tools such as Apache Hadoop, Apache Hive, and Apache Spark are now widely used to query and manipulate vast data volumes.

Here is an example that demonstrates how Apache Spark can be used to perform a simple aggregate query against a Parquet data set. The real power of Spark is that this query can be run on a laptop or on a cluster of hundreds of servers with no code changes required.

Example of Apache Spark Query using DataFrame

```
val spark: SparkSession = SparkSession.builder
  .appName("Example")
  .master("local[*]")
  .getOrCreate()

val df = spark.read.parquet("/mnt/nyctaxi/parquet")
  .groupBy("passenger_count")
  .sum("fare_amount")
  .orderBy("passenger_count")

df.show()
```

Why Are Query Engines Popular?

Data is growing at an ever-increasing pace and often cannot fit on a single computer. Specialist engineering skills are needed to write distributed code for querying data, and it isn't practical to write custom code each time new answers are needed from data.

Query engines provide a set of standard operations and transformations that the end-user can combine in different ways through a simple query language or application programming interface and are tuned for good performance.

What This Book Covers

This book provides an overview of every step involved in building a general-purpose query engine.

The query engine discussed in this book is a simple query engine developed specifically for this book, with the code being developed alongside writing the book content to make sure that I could write about topics while I was facing design decisions.

Source Code

Full source code for the query engine discussed in this book is located in the following GitHub repository.

```
https://github.com/andygrove/how-query-engines-work
```

Refer to the README in the project for up-to-date instructions for building the project using Gradle.

Why Kotlin?

The focus of this book is query engine design, which is generally programming language-agnostic. I chose Kotlin for this book because it is concise and easy to comprehend. It is also 100% compatible with Java, meaning that you can call Kotlin code from Java, and other Java-based languages, such as Scala.

However, the DataFusion query engine in the Apache Arrow project is also primarily based on the design in this book. Readers who are more interested in Rust than JVM can refer to the DataFusion source code in conjunction with this book.

This book is also available for purchase in ePub, MOBI, and PDF format from https://leanpub.com/how-query-engines-work

Apache Arrow

Apache Arrow started as a specification for a memory format for columnar data, with implementations in Java and C++. The memory format is efficient for vectorized processing on modern hardware such as CPUs with SIMD (Single Instruction, Multiple Data) support and GPUs.

There are several benefits to having a standardized memory format for data:

- High-level languages such as Python or Java can make calls into lower-level languages such as Rust or C++ for compute-intensive tasks by passing pointers to the data, rather than making a copy of the data in a different format, which would be very expensive.
- Data can be transferred between processes efficiently without much serialization overhead because the memory format is also the network format (although data can also be compressed).
- It should make it easier to build connectors, drivers, and integrations between various open-source and commercial projects in the data science and data analytics space and allow developers to use their favorite language to leverage these platforms.

Apache Arrow now has implementations in many programming languages, including C, C++, C#, Go, Java, JavaScript, Julia, MATLAB, Python, R, Ruby, and Rust.

Arrow Memory Model

The memory model is described in detail on the Arrow web site, but essentially each column is represented by a single vector holding the raw data, along with separate vectors representing null values and offsets into the raw data for variable-width types.

Inter-Process Communication (IPC)

As I mentioned earlier, data can be passed between processes by passing a pointer to the data. However, the receiving process needs to know how to interpret this data, so an IPC format is defined for exchanging metadata such as schema information. Arrow uses Google Flatbuffers to define the metadata format.

Compute Kernels

The scope of Apache Arrow has expanded to provide computational libraries for evaluating expressions against data. The Java, C++, C, Python, Ruby, Go, Rust, and JavaScript implementations contain computational libraries for performing computations on Arrow memory.

Since this book mostly refers to the Java implementation, it is worth pointing out that Dremio recently donated Gandiva, which is a Java library that compiles expressions down to LLVM and supports SIMD. JVM developers can delegate operations to the Gandiva library and benefit from performance gains that wouldn't be possible natively in Java.

Arrow Flight Protocol

More recently, Arrow has defined a Flight protocol for efficiently streaming Arrow data over the network. Flight is based on gRPC and Google Protocol Buffers.

The Flight protocol defines a FlightService with the following methods:

Handshake

Handshake between client and server. Depending on the server, the handshake may be required to determine the token that should be used for future operations. Both request and response are streams to allow multiple round-trips depending on the auth mechanism.

ListFlights

Get a list of available streams given a particular criteria. Most flight services will expose one or more streams that are readily available for retrieval. This API allows listing the streams available for consumption. A user can also provide a criteria. The criteria can limit the subset of streams that can be listed via this interface. Each flight service allows its own definition of how to consume criteria.

GetFlightInfo

For a given FlightDescriptor, get information about how the flight can be consumed. This is a useful interface if the consumer of the interface can already identify the specific flight to consume. This interface can also allow a consumer to generate a flight stream through a specified descriptor. For example, a flight descriptor might be something that includes a SQL statement or a Pickled Python operation that will be executed. In those cases, the descriptor will not be previously available within the list of available streams provided by ListFlights, but will be available for consumption for the duration defined by the specific flight service.

GetSchema

For a given FlightDescriptor, get the Schema as described in Schema.fbs::Schema. This is used when a consumer needs the Schema of flight stream. Similar to GetFlightInfo, this interface may generate a new flight that was not previously available in ListFlights.

DoGet

Retrieve a single stream associated with a particular descriptor associated with the referenced ticket. A Flight can be composed of one or more streams where each stream can be retrieved using a separate opaque ticket that the flight service uses for managing a collection of streams.

DoPut

Push a stream to the flight service associated with a particular flight stream. This allows a client of a flight service to upload a stream of data. Depending on the particular flight service, a client consumer could be allowed to upload a single stream per descriptor or an unlimited number. In the latter, the service might implement a 'seal' action that can be applied to a descriptor once all streams are uploaded.

DoExchange

Open a bidirectional data channel for a given descriptor. This allows clients to send and receive arbitrary Arrow data and application-specific metadata in a single logical stream. In contrast to DoGet/DoPut, this is more suited for clients offloading computation (rather than storage) to a Flight service.

DoAction

Flight services can support an arbitrary number of simple actions in addition to the possible ListFlights, GetFlightInfo, DoGet, DoPut operations that are potentially available. DoAction allows a flight client to do a specific action against a flight service. An action includes opaque request and response objects that are specific to the type of action being undertaken.

ListActions

A flight service exposes all of the available action types that it has along with descriptions. This allows different flight consumers to understand the capabilities of the flight service.

Arrow Flight SQL

There is a proposal to add SQL capabilities to Arrow Flight. At the time of writing (Jan 2021), there is a PR up for a C++ implementation and the tracking issue is ARROW-14698.

Query Engines

DataFusion

The Rust implementation of Arrow contains an in-memory query engine named DataFusion, which was donated to the project in 2019. This project is maturing rapidly and is gaining traction. For example, InfluxData is building the core of the next generation of InfluxDB by leveraging DataFusion.

Ballista

Ballista is a distributed compute platform primarily implemented in Rust, and powered by Apache Arrow. It is built on an architecture that allows other programming languages (such as Python, C++, and Java) to be supported as first-class citizens without paying a penalty for serialization costs.

The foundational technologies in Ballista are:

- **Apache Arrow** for the memory model and type system.
- **Apache Arrow Flight** protocol for efficient data transfer between processes.
- **Apache Arrow Flight SQL** protocol for use by business intelligence tools and JDBC drivers to connect to a Ballista cluster
- **Google Protocol Buffers** for serializing query plans.
- **Docker** for packaging up executors along with user-defined code.
- **Kubernetes** for deployment and management of the executor docker containers.

Ballista was donated to the Arrow project in 2021 and is not ready for production use although it is capable of running a number of queries from the popular TPC-H benchmark with good performance.

C++ Query Engine

The C++ implementation has work in progress to add a query engine and the current focus is on implementing efficient compute primitives and a Dataset API.

Choosing a Type System

_The source code discussed in this chapter can be found in the `datatypes` module of the KQuery project. _ The first step in building a query engine is to choose a type system to represent the different types of data that the query engine will be processing. One option would be to invent a proprietary type system specific to the query engine. Another option is to use the type system of the data source that the query engine is designed to query from.

If the query engine is going to support querying multiple data sources, which is often the case, then there is likely some conversion required between each supported data source and the query engine's type system, and it will be important to use a type system capable of representing all the data types of all the supported data sources.

Row-Based or Columnar?

An important consideration is whether the query engine will process data row-by-row or whether it will represent data in a columnar format.

Many of today's query engines are based on the Volcano Query Planner where each step in the physical plan is essentially an iterator over rows. This is a simple model to implement but tends to introduce per-row overheads that add up pretty quickly when running a query against billions of rows. This overhead can be reduced by instead iterating over batches of data. Furthermore, if these batches represent columnar data rather than rows, it is possible to use "vectorized processing" and take advantage of SIMD (Single Instruction Multiple Data) to process multiple values within a column with a single CPU instruction. This concept can be taken even further by leveraging GPUs to process much larger quantities of data in parallel.

Interoperability

Another consideration is that we may want to make our query engine accessible from multiple programming languages. It is common for users of query engines to use languages such as Python, R, or Java. We may also want to build ODBC or JDBC drivers to make it easy to build integrations.

Given these requirements, it would be good to find an industry standard for representing columnar data and for exchanging this data efficiently between processes.

It will probably come as little surprise that I believe that Apache Arrow provides an ideal foundation.

Type System

We will use Apache Arrow as the basis for our type system. The following Arrow classes are used to represent schema, fields, and data types.

- *Schema* provides metadata for a data source or the results from a query. A schema consists of one or more fields.
- *Field* provides the name and data type for a field within a schema, and specifies whether it allows null values or not.
- *FieldVector* provides columnar storage for data for a field.
- *ArrowType* represents a data type.

KQuery introduces some additional classes and helpers as an abstraction over the Apache Arrow type system.

KQuery provides constants that can be referenced for the supported Arrow data types

```
object ArrowTypes {
    val BooleanType = ArrowType.Bool()
    val Int8Type = ArrowType.Int(8, true)
    val Int16Type = ArrowType.Int(16, true)
    val Int32Type = ArrowType.Int(32, true)
    val Int64Type = ArrowType.Int(64, true)
    val UInt8Type = ArrowType.Int(8, false)
    val UInt16Type = ArrowType.Int(16, false)
    val UInt32Type = ArrowType.Int(32, false)
    val UInt64Type = ArrowType.Int(64, false)
    val FloatType = ArrowType.FloatingPoint(FloatingPointPrecision.SINGLE)
    val DoubleType = ArrowType.FloatingPoint(FloatingPointPrecision.DOUBLE)
    val StringType = ArrowType.Utf8()
}
```

Rather than working directly with `FieldVector`, KQuery introduces a `ColumnVector` interface as an abstraction to provide more convenient accessor methods, avoiding the need to case to a specific `FieldVector` implementation for each data type.

```
interface ColumnVector {
  fun getType(): ArrowType
  fun getValue(i: Int) : Any?
  fun size(): Int
}
```

This abstraction also makes it possible to have an implementation for scalar values, avoiding the need to create and populate a `FieldVector` with a literal value repeated for every index in the column.

```
class LiteralValueVector(
    val arrowType: ArrowType,
    val value: Any?,
    val size: Int) : ColumnVector {

  override fun getType(): ArrowType {
    return arrowType
  }

  override fun getValue(i: Int): Any? {
    if (i<0 || i>=size) {
      throw IndexOutOfBoundsException()
    }
    return value
  }

  override fun size(): Int {
    return size
  }

}
```

KQuery also provides a *RecordBatch* class to represent a batch of columnar data.

```
class RecordBatch(val schema: Schema, val fields: List<ColumnVector>) {

  fun rowCount() = fields.first().size()

  fun columnCount() = fields.size

  /** Access one column by index */
  fun field(i: Int): ColumnVector {
      return fields[i]
  }

}
```

Data Sources

_The source code discussed in this chapter can be found in the `datasource` module of the KQuery project. _ A query engine is of little use without a data source to read from and we want the ability to support multiple data sources, so it is important to create an interface that the query engine can use to interact with data sources. This also allows users to use our query engine with their custom data sources. Data sources are often files or databases but could also be in-memory objects.

Data Source Interface

During query planning, it is important to understand the schema of the data source so that the query plan can be validated to make sure that referenced columns exist and that data types are compatible with the expressions being used to reference them. In some cases, the schema might not be available, because some data sources do not have a fixed schema and are generally referred to as "schema-less". JSON documents are one example of a schema-less data source.

During query execution, we need the ability to fetch data from the data source and need to be able to specify which columns to load into memory for efficiency. There is no sense loading columns into memory if the query doesn't reference them.

KQuery DataSource Interface

```
interface DataSource {

  /** Return the schema for the underlying data source */
  fun schema(): Schema

  /** Scan the data source, selecting the specified columns */
  fun scan(projection: List<String>): Sequence<RecordBatch>
}
```

Data Source Examples

There are a number of data sources that are often encountered in data science or analytics.

Comma-Separated Values (CSV)

CSV files are text files with one record per line and fields are separated with commas, hence the name "Comma Separated Values". CSV files do not contain schema information (other than optional column names on the first line in the file) although it is possible to derive the schema by reading the file first. This can be an expensive operation.

JSON

The JavaScript Object Notation format (JSON) is another popular text-based file format. Unlike CSV files, JSON files are structured and can store complex nested data types.

Parquet

Parquet was created to provide a compressed, efficient columnar data representation and is a popular file format in the Hadoop ecosystem. Parquet is built from the ground up with complex nested data structures in mind, and uses the record shredding and assembly algorithm described in the Dremel paper.

Parquet files contain schema information and data is stored in batches (referred to as "row groups") where each batch consists of columns. The row groups can contain compressed data and can also contain optional metadata such as

minimum and maximum values for each column. Query engines can be optimised to use this metadata to determine when row groups can be skipped during a scan.

Orc

The Optimized Row Columnar (Orc) format is similar to Parquet. Data is stored in columnar batches called "stripes".

Logical Plans & Expressions

_The source code discussed in this chapter can be found in the `logical-plan` module of the KQuery project. _ A logical plan represents a relation (a set of tuples) with a known schema. Each logical plan can have zero or more logical plans as inputs. It is convenient for a logical plan to expose its child plans so that a visitor pattern can be used to walk through the plan.

```
interface LogicalPlan {
  fun schema(): Schema
  fun children(): List<LogicalPlan>
}
```

Printing Logical Plans

It is important to be able to print logical plans in human-readable form to help with debugging. Logical plans are typically printed as a hierarchical structure with child nodes indented.

We can implement a simple recursive helper function to format a logical plan.

```
fun format(plan: LogicalPlan, indent: Int = 0): String {
  val b = StringBuilder()
  0.rangeTo(indent).forEach { b.append("\t") }
  b.append(plan.toString()).append("\n")
  plan.children().forEach { b.append(format(it, indent+1)) }
  return b.toString()
}
```

Here is an example of a logical plan formatted using this method.

```
Projection: #id, #first_name, #last_name, #state, #salary
  Filter: #state = 'CO'
    Scan: employee.csv; projection=None
```

Serialization

It is sometimes desirable to be able to serialize query plans so that they can easily be transferred to another process. It is good practice to add serialization early on as a precaution against accidentally referencing data structures that cannot be serialized (such as file handles or database connections).

One approach would be to use the implementation languages' default mechanism for serializing data structures to/from a format such as JSON. In Java, the Jackson library could be used. Kotlin has the `kotlinx.serialization` library, and Rust has a serde crate, for example.

Another option would be to define a language-agnostic serialization format using Avro, Thrift, or Protocol Buffers and then write code to translate between this format and the language-specific implementation.

Since publishing the first edition of this book, a new standard named "substrait" has emerged, with the goal of providing cross-language serialization for relational algebra. I am excited about this project and predict that it will become the de-facto standard for representing query plans and open up many integration possibilities. For example, it would be possible to use a mature Java-based query planner such as Apache Calcite, serialize the plan in Substrait format, and then execute the plan in a query engine implemented in a lower-level language, such as C++ or Rust. For more information, visit https://substrait.io/.

Logical Expressions

One of the fundamental building blocks of a query plan is the concept of an expression that can be evaluated against data at runtime.

Here are some examples of expressions that are typically supported in query engines.

Expression	Examples
Literal Value	"hello", 12.34
Column Reference	user_id, first_name, last_name
Math Expression	salary * state_tax
Comparison Expression	x >= y
Boolean Expression	birthday = today() AND age >= 21
Aggregate Expression	MIN(salary), MAX(salary), SUM(salary), AVG(salary), COUNT(*)
Scalar Function	CONCAT(first_name, " ", last_name)
Aliased Expression	salary * 0.02 AS pay_increase

Of course, all of these expressions can be combined to form deeply nested expression trees. Expression evaluation is a textbook case of recursive programming.

When we are planning queries, we will need to know some basic metadata about the output of an expression. Specifically, we need to have a name for the expression so that other expressions can reference it and we need to know the data type of the values that the expression will produce when evaluated so that we can validate that the query plan is valid. For example, if we have an expression `a + b` then it can only be valid if both `a` and `b` are numeric types.

Also note that the data type of an expression can be dependent on the input data. For example, a column reference will have the data type of the column it is referencing, but a comparison expression always returns a Boolean value.

```
interface LogicalExpr {
  fun toField(input: LogicalPlan): Field
}
```

Column Expressions

The `Column` expression simply represents a reference to a named column. The metadata for this expression is derived by finding the named column in the input and returning that column's metadata. Note that the term "column" here refers to a column produced by the input logical plan and could represent a column in a data source, or it could represent the result of an expression being evaluated against other inputs.

```
class Column(val name: String): LogicalExpr {

  override fun toField(input: LogicalPlan): Field {
    return input.schema().fields.find { it.name == name } ?:
      throw SQLException("No column named '$name'")
  }

  override fun toString(): String {
    return "#$name"
  }

}
```

Literal Expressions

We need the ability to represent literal values as expressions so that we can write expressions such as `salary * 0.05`.

Here is an example expression for literal strings.

```
class LiteralString(val str: String): LogicalExpr {

  override fun toField(input: LogicalPlan): Field {
    return Field(str, ArrowTypes.StringType)
  }

  override fun toString(): String {
    return "'$str'"
  }

}
```

Here is an example expression for literal longs.

```
class LiteralLong(val n: Long): LogicalExpr {

  override fun toField(input: LogicalPlan): Field {
      return Field(n.toString(), ArrowTypes.Int64Type)
  }

  override fun toString(): String {
      return n.toString()
  }

}
```

Binary Expressions

Binary expressions are simply expressions that take two inputs. There are three categories of binary expressions that we will implement, and those are comparison expressions, Boolean expressions, and math expressions. Because the string representation is the same for all of these, we can use a common base class that provides the `toString` method. The variables "l" and "r" refer to the left and right inputs.

```
abstract class BinaryExpr(
    val name: String,
    val op: String,
    val l: LogicalExpr,
    val r: LogicalExpr) : LogicalExpr {

  override fun toString(): String {
    return "$l $op $r"
  }
}
```

Comparison expressions such as `=` or `<` compare two values of the same data type and return a Boolean value. We also need to implement Boolean operators `AND` and `OR` which also take two arguments and produce a Boolean result, so we can use a common base class for these as well.

```
abstract class BooleanBinaryExpr(
    name: String,
    op: String,
    l: LogicalExpr,
    r: LogicalExpr) : BinaryExpr(name, op, l, r) {

  override fun toField(input: LogicalPlan): Field {
      return Field(name, ArrowTypes.BooleanType)
  }

}
```

This base class provides a concise way to implement the concrete comparison expressions.

Comparison Expressions

```
/** Equality (`=`) comparison */
class Eq(l: LogicalExpr, r: LogicalExpr)
    : BooleanBinaryExpr("eq", "=", l, r)

/** Inequality (`!=`) comparison */
class Neq(l: LogicalExpr, r: LogicalExpr)
    : BooleanBinaryExpr("neq", "!=", l, r)

/** Greater than (`>`) comparison */
class Gt(l: LogicalExpr, r: LogicalExpr)
    : BooleanBinaryExpr("gt", ">", l, r)

/** Greater than or equals (`>=`) comparison */
class GtEq(l: LogicalExpr, r: LogicalExpr)
    : BooleanBinaryExpr("gteq", ">=", l, r)

/** Less than (`<`) comparison */
class Lt(l: LogicalExpr, r: LogicalExpr)
    : BooleanBinaryExpr("lt", "<", l, r)

/** Less than or equals (`<=`) comparison */
class LtEq(l: LogicalExpr, r: LogicalExpr)
    : BooleanBinaryExpr("lteq", "<=", l, r)
```

Boolean Expressions

The base class also provides a concise way to implement the concrete Boolean logic expressions.

```
/** Logical AND */
class And(l: LogicalExpr, r: LogicalExpr)
    : BooleanBinaryExpr("and", "AND", l, r)

/** Logical OR */
class Or(l: LogicalExpr, r: LogicalExpr)
    : BooleanBinaryExpr("or", "OR", l, r)
```

Math Expressions

Math expressions are another specialization of a binary expression. Math expressions typically operate on values of the same data type and produce a result of the same data type.

```
abstract class MathExpr(
    name: String,
    op: String,
    l: LogicalExpr,
    r: LogicalExpr) : BinaryExpr(name, op, l, r) {

  override fun toField(input: LogicalPlan): Field {
      return Field("mult", l.toField(input).dataType)
  }

}

class Add(l: LogicalExpr, r: LogicalExpr) : MathExpr("add", "+", l, r)
class Subtract(l: LogicalExpr, r: LogicalExpr) : MathExpr("subtract", "-", l, r)
class Multiply(l: LogicalExpr, r: LogicalExpr) : MathExpr("mult", "*", l, r)
class Divide(l: LogicalExpr, r: LogicalExpr) : MathExpr("div", "/", l, r)
class Modulus(l: LogicalExpr, r: LogicalExpr) : MathExpr("mod", "%", l, r)
```

Aggregate Expressions

Aggregate expressions perform an aggregate function such as `MIN`, `MAX`, `COUNT`, `SUM`, or `AVG` on an input expression.

```
abstract class AggregateExpr(
    val name: String,
    val expr: LogicalExpr) : LogicalExpr {

  override fun toField(input: LogicalPlan): Field {
    return Field(name, expr.toField(input).dataType)
  }

  override fun toString(): String {
    return "$name($expr)"
  }
}
```

For aggregate expressions where the aggregated data type is the same as the input type, we can simply extend this base class.

```
class Sum(input: LogicalExpr) : AggregateExpr("SUM", input)
class Min(input: LogicalExpr) : AggregateExpr("MIN", input)
class Max(input: LogicalExpr) : AggregateExpr("MAX", input)
class Avg(input: LogicalExpr) : AggregateExpr("AVG", input)
```

For aggregate expressions where the data type is not dependent on the input type, we need to override the `toField` method. For example, the "COUNT" aggregate expression always produces an integer regardless of the data type of the values being counted.

```
class Count(input: LogicalExpr) : AggregateExpr("COUNT", input) {

  override fun toField(input: LogicalPlan): Field {
    return Field("COUNT", ArrowTypes.Int32Type)
  }

  override fun toString(): String {
    return "COUNT($expr)"
  }
}
```

Logical Plans

With the logical expressions in place, we can now implement the logical plans for the various transformations that the query engine will support.

Scan

The `Scan` logical plan represents fetching data from a `DataSource` with an optional projection. `Scan` is the only logical plan in our query engine that does not have another logical plan as an input. It is a leaf node in the query tree.

```
class Scan(
    val path: String,
    val dataSource: DataSource,
    val projection: List<String>): LogicalPlan {

  val schema = deriveSchema()

  override fun schema(): Schema {
    return schema
  }

  private fun deriveSchema() : Schema {
    val schema = dataSource.schema()
    if (projection.isEmpty()) {
      return schema
    } else {
      return schema.select(projection)
    }
  }

  override fun children(): List<LogicalPlan> {
    return listOf()
  }

  override fun toString(): String {
    return if (projection.isEmpty()) {
      "Scan: $path; projection=None"
    } else {
      "Scan: $path; projection=$projection"
    }
  }

}
```

Projection

The `Projection` logical plan applies a projection to its input. A projection is a list of expressions to be evaluated against the input data. Sometimes this is as simple as a list of columns, such as `SELECT a, b, c FROM foo`, but it could also include any other type of expression that is supported. A more complex example would be `SELECT (CAST(a AS float) * 3.141592)) AS my_float FROM foo`.

```
class Projection(
    val input: LogicalPlan,
    val expr: List<LogicalExpr>): LogicalPlan {

  override fun schema(): Schema {
    return Schema(expr.map { it.toField(input) })
  }

  override fun children(): List<LogicalPlan> {
    return listOf(input)
  }

  override fun toString(): String {
    return "Projection: ${ expr.map {
        it.toString() }.joinToString(", ")
    }"
  }
}
```

Selection (also known as Filter)

The `selection` logical plan applies a filter expression to determine which rows should be selected (included) in its output. This is represented by the `WHERE` clause in SQL. A simple example would be `SELECT * FROM foo WHERE a > 5`. The filter expression needs to evaluate to a Boolean result.

```
class Selection(
    val input: LogicalPlan,
    val expr: Expr): LogicalPlan {

  override fun schema(): Schema {
    // selection does not change the schema of the input
    return input.schema()
  }

  override fun children(): List<LogicalPlan> {
    return listOf(input)
  }

  override fun toString(): String {
    return "Filter: $expr"
  }
}
```

Aggregate

The `Aggregate` logical plan is more complex than `Projection`, `Selection`, or `Scan` and calculates aggregates of underlying data such as calculating minimum, maximum, averages, and sums of data. Aggregates are often grouped by other columns (or expressions). A simple example would be `SELECT job_title, AVG(salary) FROM employee GROUP BY job_title`.

```
class Aggregate(
    val input: LogicalPlan,
    val groupExpr: List<LogicalExpr>,
    val aggregateExpr: List<AggregateExpr>) : LogicalPlan {

  override fun schema(): Schema {
    return Schema(groupExpr.map { it.toField(input) } +
                  aggregateExpr.map { it.toField(input) })
  }

  override fun children(): List<LogicalPlan> {
    return listOf(input)
  }

  override fun toString(): String {
    return "Aggregate: groupExpr=$groupExpr, aggregateExpr=$aggregateExpr"
  }
}
```

Note that in this implementation, the output of the aggregate plan is organized with grouping columns followed by aggregate expressions. It will often be necessary to wrap the aggregate logical plan in a projection so that columns are returned in the order requested in the original query.

Building Logical Plans

_The source code discussed in this chapter can be found in the `dataframe` module of the KQuery project. _

Building Logical Plans The Hard Way

Now that we have defined classes for a subset of logical plans, we can combine them programmatically.

Here is some verbose code for building a plan for the query `SELECT * FROM employee WHERE state = 'CO'` against a CSV file containing the columns `id, first_name, last_name, state, job_title, salary`.

```
// create a plan to represent the data source
val csv = CsvDataSource("employee.csv")

// create a plan to represent the scan of the data source (FROM)
val scan = Scan("employee", csv, listOf())

// create a plan to represent the selection (WHERE)
val filterExpr = Eq(Column("state"), LiteralString("CO"))
val selection = Selection(scan, filterExpr)

// create a plan to represent the projection (SELECT)
val projectionList = listOf(Column("id"),
                            Column("first_name"),
                            Column("last_name"),
                            Column("state"),
                            Column("salary"))
val plan = Projection(selection, projectionList)

// print the plan
println(format(plan))
```

This prints the following plan:

```
Projection: #id, #first_name, #last_name, #state, #salary
    Filter: #state = 'CO'
        Scan: employee; projection=None
```

The same code can also be written more concisely like this:

```
val plan = Projection(
  Selection(
    Scan("employee", CsvDataSource("employee.csv"), listOf()),
    Eq(Column(3), LiteralString("CO"))
  ),
  listOf(Column("id"),
         Column("first_name"),
         Column("last_name"),
         Column("state"),
         Column("salary"))
)
println(format(plan))
```

Although this is more concise, it is also harder to interpret, so it would be nice to have a more elegant way to create logical plans. This is where a DataFrame interface can help.

Building Logical Plans using DataFrames

Implementing a DataFrame style API allows us to build logical query plans in a much more user-friendly way. A DataFrame is just an abstraction around a logical query plan and has methods to perform transformations and actions. It is similar to a fluent-style builder API.

Here is a minimal starting point for a DataFrame interface that allows us to apply projections and selections to an existing DataFrame.

```
interface DataFrame {

  /** Apply a projection */
  fun project(expr: List<LogicalExpr>): DataFrame

  /** Apply a filter */
  fun filter(expr: LogicalExpr): DataFrame

  /** Aggregate */
  fun aggregate(groupBy: List<LogicalExpr>,
                aggregateExpr: List<AggregateExpr>): DataFrame

  /** Returns the schema of the data that will be produced by this DataFrame. */
  fun schema(): Schema

  /** Get the logical plan */
  fun logicalPlan() : LogicalPlan

}
```

Here is the implementation of this interface.

```
class DataFrameImpl(private val plan: LogicalPlan) : DataFrame {

  override fun project(expr: List<LogicalExpr>): DataFrame {
    return DataFrameImpl(Projection(plan, expr))
  }

  override fun filter(expr: LogicalExpr): DataFrame {
    return DataFrameImpl(Selection(plan, expr))
  }

  override fun aggregate(groupBy: List<LogicalExpr>,
                         aggregateExpr: List<AggregateExpr>): DataFrame {
    return DataFrameImpl(Aggregate(plan, groupBy, aggregateExpr))
  }

  override fun schema(): Schema {
    return plan.schema()
  }

  override fun logicalPlan(): LogicalPlan {
    return plan
  }

}
```

Before we can apply a projection or selection, we need a way to create an initial DataFrame that represents an underlying data source. This is usually obtained through an execution context.

Here is a simple starting point for an execution context that we will enhance later.

```
class ExecutionContext {

  fun csv(filename: String): DataFrame {
    return DataFrameImpl(Scan(filename, CsvDataSource(filename), listOf()))
  }

  fun parquet(filename: String): DataFrame {
    return DataFrameImpl(Scan(filename, ParquetDataSource(filename), listOf()))
  }
}
```

With this groundwork in place, we can now create a logical query plan using the context and the DataFrame API.

```
val ctx = ExecutionContext()

val plan = ctx.csv("employee.csv")
              .filter(Eq(Column("state"), LiteralString("CO")))
              .select(listOf(Column("id"),
                             Column("first_name"),
                             Column("last_name"),
                             Column("state"),
                             Column("salary")))
```

This is much cleaner and more intuitive, but we can go a step further and add some convenience methods to make this a little more comprehensible. This is specific to Kotlin, but other languages have similar concepts.

We can create some convenience methods for creating the supported expression objects.

```
fun col(name: String) = Column(name)
fun lit(value: String) = LiteralString(value)
fun lit(value: Long) = LiteralLong(value)
fun lit(value: Double) = LiteralDouble(value)
```

We can also define infix operators on the `LogicalExpr` interface for building binary expressions.

```
infix fun LogicalExpr.eq(rhs: LogicalExpr): LogicalExpr { return Eq(this, rhs) }
infix fun LogicalExpr.neq(rhs: LogicalExpr): LogicalExpr { return Neq(this, rhs) }
infix fun LogicalExpr.gt(rhs: LogicalExpr): LogicalExpr { return Gt(this, rhs) }
infix fun LogicalExpr.gteq(rhs: LogicalExpr): LogicalExpr { return GtEq(this, rhs) }
infix fun LogicalExpr.lt(rhs: LogicalExpr): LogicalExpr { return Lt(this, rhs) }
infix fun LogicalExpr.lteq(rhs: LogicalExpr): LogicalExpr { return LtEq(this, rhs) }
```

With these convenience methods in place, we can now write expressive code to build our logical query plan.

```
val df = ctx.csv(employeeCsv)
   .filter(col("state") eq lit("CO"))
   .select(listOf(
       col("id"),
       col("first_name"),
       col("last_name"),
       col("salary"),
       (col("salary") mult lit(0.1)) alias "bonus"))
   .filter(col("bonus") gt lit(1000))
```

Physical Plans & Expressions

_The source code discussed in this chapter can be found in the `physical-plan` module of the KQuery project. _ The logical plans defined in chapter five specify *what* to do but not *how* to do it, and it is good practice to have separate logical and physical plans, although it is possible to combine them to reduce complexity.

One reason to keep logical and physical plans separate is that sometimes there can be multiple ways to execute a particular operation, meaning that there is a one-to-many relationship between logical plans and physical plans.

For example, there could be separate physical plans for single-process versus distributed execution, or CPU versus GPU execution.

Also, operations such as `Aggregate` and `Join` can be implemented with a variety of algorithms with different performance trade-offs. When aggregating data that is already sorted by the grouping keys, it is efficient to use a Group Aggregate (also known as a Sort Aggregate) which only needs to hold state for one set of grouping keys at a time and can emit a result as soon as one set of grouping keys ends. If the data is not sorted, then a Hash Aggregate is typically used. A Hash Aggregate maintains a HashMap of accumulators by grouping keys.

Joins have an even wider variety of algorithms, including Nested Loop Join, Sort-Merge Join, and Hash Join.

Physical plans return iterators over record batches.

```
interface PhysicalPlan {
  fun schema(): Schema
  fun execute(): Sequence<RecordBatch>
  fun children(): List<PhysicalPlan>
}
```

Physical Expressions

We have defined logical expressions that are referenced in the logical plans, but we now need to implement physical expression classes containing the code to evaluate the expressions at runtime.

There could be multiple physical expression implementations for each logical expression. For example, for the logical expression `AddExpr` that adds two numbers, we could have one implementation that uses the CPU and one that uses the GPU. The query planner could choose which one to use based on the hardware capabilities of the server that the code is running on.

Physical expressions are evaluated against record batches and the results are columns.

Here is the interface that we will use to represent physical expressions.

```
interface Expression {
  fun evaluate(input: RecordBatch): ColumnVector
}
```

Column Expressions

The `Column` expression simply evaluates to a reference to the `ColumnVector` in the `RecordBatch` being processed. The logical expression for `Column` references inputs by name, which is user-friendly for writing queries, but for the physical expression we want to avoid the cost of name lookups every time the expression is evaluated, so it references columns by index instead.

```
class ColumnExpression(val i: Int) : Expression {

  override fun evaluate(input: RecordBatch): ColumnVector {
    return input.field(i)
  }

  override fun toString(): String {
    return "#$i"
  }
}
```

Literal Expressions

The physical implementation of a literal expression is simply a literal value wrapped in a class that implements the appropriate trait and provides the same value for every index in a column.

```
class LiteralValueVector(
    val arrowType: ArrowType,
    val value: Any?,
    val size: Int) : ColumnVector {

  override fun getType(): ArrowType {
    return arrowType
  }

  override fun getValue(i: Int): Any? {
    if (i<0 || i>=size) {
      throw IndexOutOfBoundsException()
    }
    return value
  }

  override fun size(): Int {
    return size
  }

}
```

With this class in place, we can create our physical expressions for literal expressions of each data type.

```
class LiteralLongExpression(val value: Long) : Expression {
  override fun evaluate(input: RecordBatch): ColumnVector {
    return LiteralValueVector(ArrowTypes.Int64Type,
                              value,
                              input.rowCount())
  }
}

class LiteralDoubleExpression(val value: Double) : Expression {
  override fun evaluate(input: RecordBatch): ColumnVector {
    return LiteralValueVector(ArrowTypes.DoubleType,
                              value,
                              input.rowCount())
  }
}

class LiteralStringExpression(val value: String) : Expression {
  override fun evaluate(input: RecordBatch): ColumnVector {
    return LiteralValueVector(ArrowTypes.StringType,
                              value.toByteArray(),
                              input.rowCount())
  }
}
```

Binary Expressions

For binary expressions, we need to evaluate the left and right input expressions and then evaluate the specific binary operator against those input values, so we can provide a base class to simplify the implementation for each operator.

```
abstract class BinaryExpression(val l: Expression, val r: Expression) : Expression {
  override fun evaluate(input: RecordBatch): ColumnVector {
    val ll = l.evaluate(input)
    val rr = r.evaluate(input)
    assert(ll.size() == rr.size())
    if (ll.getType() != rr.getType()) {
      throw IllegalStateException(
          "Binary expression operands do not have the same type: " +
          "${ll.getType()} != ${rr.getType()}")
    }
    return evaluate(ll, rr)
  }

  abstract fun evaluate(l: ColumnVector, r: ColumnVector) : ColumnVector
}
```

Comparison Expressions

The comparison expressions simply compare all values in the two input columns and produce a new column (a bit vector) containing the results.

Here is an example for the equality operator.

```
class EqExpression(l: Expression,
                   r: Expression): BooleanExpression(l,r) {

  override fun evaluate(l: Any?, r: Any?, arrowType: ArrowType) : Boolean {
    return when (arrowType) {
      ArrowTypes.Int8Type -> (l as Byte) == (r as Byte)
      ArrowTypes.Int16Type -> (l as Short) == (r as Short)
      ArrowTypes.Int32Type -> (l as Int) == (r as Int)
      ArrowTypes.Int64Type -> (l as Long) == (r as Long)
      ArrowTypes.FloatType -> (l as Float) == (r as Float)
      ArrowTypes.DoubleType -> (l as Double) == (r as Double)
      ArrowTypes.StringType -> toString(l) == toString(r)
      else -> throw IllegalStateException(
          "Unsupported data type in comparison expression: $arrowType")
    }
  }
}
```

Math Expressions

The implementation for math expressions is very similar to the code for comparison expressions. A base class is used for all math expressions.

```
abstract class MathExpression(l: Expression,
                              r: Expression): BinaryExpression(l,r) {

  override fun evaluate(l: ColumnVector, r: ColumnVector): ColumnVector {
    val fieldVector = FieldVectorFactory.create(l.getType(), l.size())
    val builder = ArrowVectorBuilder(fieldVector)
    (0 until l.size()).forEach {
      val value = evaluate(l.getValue(it), r.getValue(it), l.getType())
      builder.set(it, value)
    }
    builder.setValueCount(l.size())
    return builder.build()
  }

  abstract fun evaluate(l: Any?, r: Any?, arrowType: ArrowType) : Any?
}
```

Here is an example of a specific math expression extending this base class.

```
class AddExpression(l: Expression,
                    r: Expression): MathExpression(l,r) {

  override fun evaluate(l: Any?, r: Any?, arrowType: ArrowType) : Any? {
      return when (arrowType) {
        ArrowTypes.Int8Type -> (l as Byte) + (r as Byte)
        ArrowTypes.Int16Type -> (l as Short) + (r as Short)
        ArrowTypes.Int32Type -> (l as Int) + (r as Int)
        ArrowTypes.Int64Type -> (l as Long) + (r as Long)
        ArrowTypes.FloatType -> (l as Float) + (r as Float)
        ArrowTypes.DoubleType -> (l as Double) + (r as Double)
        else -> throw IllegalStateException(
            "Unsupported data type in math expression: $arrowType")
      }
  }

  override fun toString(): String {
    return "$l+$r"
  }
}
```

Aggregate Expressions

The expressions we have looked at so far produce one output column from one or more input columns in each batch. Aggregate expressions are more complex because they aggregate values across multiple batches of data and then produce one final value, so we need to introduce the concept of accumulators, and the physical representation of each aggregate expression needs to know how to produce an appropriate accumulator for the query engine to pass input data to.

Here are the main interfaces for representing aggregate expressions and accumulators.

```
interface AggregateExpression {
  fun inputExpression(): Expression
  fun createAccumulator(): Accumulator
}

interface Accumulator {
  fun accumulate(value: Any?)
  fun finalValue(): Any?
}
```

The implementation for the `Max` aggregate expression would produce a specific MaxAccumulator.

```
class MaxExpression(private val expr: Expression) : AggregateExpression {

  override fun inputExpression(): Expression {
    return expr
  }

  override fun createAccumulator(): Accumulator {
    return MaxAccumulator()
  }

  override fun toString(): String {
    return "MAX($expr)"
  }
}
```

Here is an example implementation of the MaxAccumulator.

```
class MaxAccumulator : Accumulator {

  var value: Any? = null

  override fun accumulate(value: Any?) {
    if (value != null) {
      if (this.value == null) {
        this.value = value
      } else {
        val isMax = when (value) {
          is Byte -> value > this.value as Byte
          is Short -> value > this.value as Short
          is Int -> value > this.value as Int
          is Long -> value > this.value as Long
          is Float -> value > this.value as Float
          is Double -> value > this.value as Double
          is String -> value > this.value as String
          else -> throw UnsupportedOperationException(
            "MAX is not implemented for data type: ${value.javaClass.name}")
        }

        if (isMax) {
          this.value = value
        }
      }
    }
  }

  override fun finalValue(): Any? {
    return value
  }
}
```

Physical Plans

With the physical expressions in place, we can now implement the physical plans for the various transformations that the query engine will support.

Scan

The Scan execution plan simply delegates to a data source, passing in a projection to limit the number of columns to load into memory. No additional logic is performed.

```
class ScanExec(val ds: DataSource, val projection: List<String>) : PhysicalPlan {

  override fun schema(): Schema {
    return ds.schema().select(projection)
  }

  override fun children(): List<PhysicalPlan> {
    // Scan is a leaf node and has no child plans
    return listOf()
  }

  override fun execute(): Sequence<RecordBatch> {
    return ds.scan(projection);
  }

  override fun toString(): String {
    return "ScanExec: schema=${schema()}, projection=$projection"
  }
}
```

Projection

The projection execution plan simply evaluates the projection expressions against the input columns and then produces a record batch containing the derived columns. Note that for the case of projection expressions that reference existing columns by name, the derived column is simply a pointer or reference to the input column, so the underlying data values are not being copied.

```
class ProjectionExec(
    val input: PhysicalPlan,
    val schema: Schema,
    val expr: List<Expression>) : PhysicalPlan {

  override fun schema(): Schema {
    return schema
  }

  override fun children(): List<PhysicalPlan> {
    return listOf(input)
  }

  override fun execute(): Sequence<RecordBatch> {
    return input.execute().map { batch ->
      val columns = expr.map { it.evaluate(batch) }
        RecordBatch(schema, columns)
      }
  }

  override fun toString(): String {
    return "ProjectionExec: $expr"
  }
}
```

Selection (also known as Filter)

The selection execution plan is the first non-trivial plan, since it has conditional logic to determine which rows from the input record batch should be included in the output batches.

For each input batch, the filter expression is evaluated to return a bit vector containing bits representing the Boolean result of the expression, with one bit for each row. This bit vector is then used to filter the input columns to produce new output columns. This is a simple implementation that could be optimized for cases where the bit vector contains all ones or all zeros to avoid overhead of copying data to new vectors.

```
class SelectionExec(
    val input: PhysicalPlan,
    val expr: Expression) : PhysicalPlan {

  override fun schema(): Schema {
    return input.schema()
  }

  override fun children(): List<PhysicalPlan> {
    return listOf(input)
  }

  override fun execute(): Sequence<RecordBatch> {
    val input = input.execute()
    return input.map { batch ->
      val result = (expr.evaluate(batch) as ArrowFieldVector).field as BitVector
      val schema = batch.schema
      val columnCount = batch.schema.fields.size
      val filteredFields = (0 until columnCount).map {
          filter(batch.field(it), result)
      }
      val fields = filteredFields.map { ArrowFieldVector(it) }
      RecordBatch(schema, fields)
    }

  private fun filter(v: ColumnVector, selection: BitVector) : FieldVector {
    val filteredVector = VarCharVector("v",
                                       RootAllocator(Long.MAX_VALUE))
    filteredVector.allocateNew()

    val builder = ArrowVectorBuilder(filteredVector)

    var count = 0
    (0 until selection.valueCount).forEach {
      if (selection.get(it) == 1) {
        builder.set(count, v.getValue(it))
        count++
      }
    }
    filteredVector.valueCount = count
    return filteredVector
  }
}
```

Hash Aggregate

The HashAggregate plan is more complex than the previous plans because it must process all incoming batches and maintain a HashMap of accumulators and update the accumulators for each row being processed. Finally, the results from the accumulators are used to create one record batch at the end containing the results of the aggregate query.

```
class HashAggregateExec(
    val input: PhysicalPlan,
    val groupExpr: List<PhysicalExpr>,
    val aggregateExpr: List<PhysicalAggregateExpr>,
    val schema: Schema) : PhysicalPlan {

  override fun schema(): Schema {
    return schema
  }

  override fun children(): List<PhysicalPlan> {
    return listOf(input)
  }

  override fun toString(): String {
    return "HashAggregateExec: groupExpr=$groupExpr, aggrExpr=$aggregateExpr"
  }

  override fun execute(): Sequence<RecordBatch> {
    val map = HashMap<List<Any?>, List<Accumulator>>()

    // for each batch from the input executor
    input.execute().iterator().forEach { batch ->

    // evaluate the grouping expressions
    val groupKeys = groupExpr.map { it.evaluate(batch) }

    // evaluate the expressions that are inputs to the aggregate functions
    val aggrInputValues = aggregateExpr.map {
        it.inputExpression().evaluate(batch)
    }

    // for each row in the batch
    (0 until batch.rowCount()).forEach { rowIndex ->
      // create the key for the hash map
      val rowKey = groupKeys.map {
      val value = it.getValue(rowIndex)
      when (value) {
        is ByteArray -> String(value)
        else -> value
      }
    }

    // get or create accumulators for this grouping key
    val accumulators = map.getOrPut(rowKey) {
        aggregateExpr.map { it.createAccumulator() }
    }

    // perform accumulation
    accumulators.withIndex().forEach { accum ->
      val value = aggrInputValues[accum.index].getValue(rowIndex)
      accum.value.accumulate(value)
    }

    // create result batch containing final aggregate values
    val allocator = RootAllocator(Long.MAX_VALUE)
    val root = VectorSchemaRoot.create(schema.toArrow(), allocator)
    root.allocateNew()
    root.rowCount = map.size

    val builders = root.fieldVectors.map { ArrowVectorBuilder(it) }

    map.entries.withIndex().forEach { entry ->
      val rowIndex = entry.index
      val groupingKey = entry.value.key
      val accumulators = entry.value.value
      groupExpr.indices.forEach {
        builders[it].set(rowIndex, groupingKey[it])
      }
      aggregateExpr.indices.forEach {
        builders[groupExpr.size+it].set(rowIndex, accumulators[it].finalValue())
      }
    }

    val outputBatch = RecordBatch(schema, root.fieldVectors.map {
       ArrowFieldVector(it)
    })

    return listOf(outputBatch).asSequence()
  }
```

```
}
```

Joins

As the name suggests, the Join operator joins rows from two relations. There are a number of different types of joins with different semantics:

- `[INNER] JOIN`: This is the most commonly used join type and creates a new relation containing rows from both the left and right inputs. When the join expression consists only of equality comparisons between columns from the left and right inputs then the join is known as an "equi-join". An example of an equi-join would be `SELECT * FROM customer JOIN orders ON customer.id = order.customer_id`.
- `LEFT [OUTER] JOIN`: A left outer join produces rows that contain all values from the left input, and optionally rows from the right input. Where this is no match on the right-hand side then null values are produced for the right columns.
- `RIGHT [OUTER] JOIN`: This is the opposite of the left join. All rows from the right are returned along with rows from the left where available.
- `SEMI JOIN`: A semi join is similar to a left join but it only returns rows from the left input where there is match to the right input. No data is returned from the right input. Not all SQL implementations support semi joins explicitly and they are often written as subqueries instead. An example of a semi join would be `SELECT id FROM foo WHERE EXISTS (SELECT * FROM bar WHERE foo.id = bar.id)`.
- `ANTI JOIN`: An into join is the opposite of a semi join. It only returns rows from the left input where this is match on the right input. An example of an anti join would be `SELECT id FROM foo WHERE NOT EXISTS (SELECT * FROM bar WHERE foo.id = bar.id)`.
- `CROSS JOIN`: A cross join returns every possible combination of rows from the left input combined with rows from the right input. If the left input contains 100 rows and the right input contains 200 rows then 20,000 rows will be returned. This is known as a cartesian product.

KQuery does not yet implement the join operator.

Subqueries

Subqueries are queries within queries. They can be correlated or uncorrelated (involving a join to other relations or not). When a subquery returns a single value then it is known as a scalar subquery.

Scalar subqueries

A scalar subquery returns a single value and can be used in many SQL expressions where a literal value could be used.

Here is an example of a correlated scalar subquery:

```
SELECT id, name, (SELECT count(*) FROM orders WHERE customer_id = customer.id) AS num_orders FROM customers
```

Here is an example of an uncorrelated scalar subquery:

```
SELECT * FROM orders WHERE total > (SELECT avg(total) FROM sales WHERE customer_state = 'CA')
```

Correlated subqueries are translated into joins before execution (this is explained in chapter 9).

Uncorrelated queries can be executed individually and the resulting value can be substituted into the top-level query.

EXISTS and IN subqueries

The `EXISTS` and `IN` expressions (and their negated forms, `NOT EXISTS` and `NOT IN`) can be used to create semi-joins and anti-joins.

Here is an example of a semi-join that selects all rows from the left relation (`foo`) where there is a matching row returned by the subquery.

```
SELECT id FROM foo WHERE EXISTS (SELECT * FROM bar WHERE foo.id = bar.id)
```

Correlated subqueries are typically converted into joins during logical plan optimization (this is explained in chapter 9)

KQuery does not yet implement subqueries.

Creating Physical Plans

With our physical plans in place, the next step is to build a query planner to create physical plans from logical plans, which we cover in the next chapter.

This book is also available for purchase in ePub, MOBI, and PDF format from https://leanpub.com/how-query-engines-work

Query Planner

_The source code discussed in this chapter can be found in the `query-planner` module of the KQuery project. _ We have defined logical and physical query plans, and now we need a query planner that can translate the logical plan into the physical plan.

The query planner may choose different physical plans based on configuration options or based on the target platform's hardware capabilities. For example, queries could be executed on CPU or GPU, on a single node, or distributed in a cluster.

Translating Logical Expressions

The first step is to define a method to translate logical expressions to physical expressions recursively. The following code sample demonstrates an implementation based on a switch statement and shows how translating a binary expression, which has two input expressions, causes the code to recurse back into the same method to translate those inputs. This approach walks the entire logical expression tree and creates a corresponding physical expression tree.

```
fun createPhysicalExpr(expr: LogicalExpr,
                       input: LogicalPlan): PhysicalExpr = when (expr) {
  is ColumnIndex -> ColumnExpression(expr.i)
  is LiteralString -> LiteralStringExpression(expr.str)
  is BinaryExpr -> {
    val l = createPhysicalExpr(expr.l, input)
    val r = createPhysicalExpr(expr.r, input)
    ...
  }
  ...
}
```

The following sections will explain the implementation for each type of expression.

Column Expressions

The logical Column expression references columns by name, but the physical expression uses column indices for improved performance, so the query planner needs to perform the translation from column name to column index and throw an exception if the column name is not valid.

This simplified example looks for the first matching column name and does not check if there are multiple matching columns, which should be an error condition.

```
is Column -> {
  val i = input.schema().fields.indexOfFirst { it.name == expr.name }
  if (i == -1) {
    throw SQLException("No column named '${expr.name}'")
  }
  ColumnExpression(i)
```

Literal Expressions

The physical expressions for literal values are straightforward, and the mapping from logical to physical expression is trivial because we need to copy the literal value over.

```
is LiteralLong -> LiteralLongExpression(expr.n)
is LiteralDouble -> LiteralDoubleExpression(expr.n)
is LiteralString -> LiteralStringExpression(expr.str)
```

Binary Expressions

To create a physical expression for a binary expression we first need to create the physical expression for the left and right inputs and then we need to create the specific physical expression.

```
is BinaryExpr -> {
  val l = createPhysicalExpr(expr.l, input)
  val r = createPhysicalExpr(expr.r, input)
  when (expr) {
    // comparision
    is Eq -> EqExpression(l, r)
    is Neq -> NeqExpression(l, r)
    is Gt -> GtExpression(l, r)
    is GtEq -> GtEqExpression(l, r)
    is Lt -> LtExpression(l, r)
    is LtEq -> LtEqExpression(l, r)

    // boolean
    is And -> AndExpression(l, r)
    is Or -> OrExpression(l, r)

    // math
    is Add -> AddExpression(l, r)
    is Subtract -> SubtractExpression(l, r)
    is Multiply -> MultiplyExpression(l, r)
    is Divide -> DivideExpression(l, r)

    else -> throw IllegalStateException(
        "Unsupported binary expression: $expr")
    }
}
```

Translating Logical Plans

We need to implement a recursive function to walk the logical plan tree and translate it into a physical plan, using the same pattern described earlier for translating expressions.

```
fun createPhysicalPlan(plan: LogicalPlan) : PhysicalPlan {
  return when (plan) {
    is Scan -> ...
    is Selection -> ...
    ...
}
```

Scan

Translating the Scan plan simply requires copying the data source reference and the logical plan's projection.

```
is Scan -> ScanExec(plan.dataSource, plan.projection)
```

Projection

There are two steps to translating a projection. First, we need to create a physical plan for the projection's input, and then we need to convert the projection's logical expressions to physical expressions.

```
is Projection -> {
  val input = createPhysicalPlan(plan.input)
  val projectionExpr = plan.expr.map { createPhysicalExpr(it, plan.input) }
  val projectionSchema = Schema(plan.expr.map { it.toField(plan.input) })
  ProjectionExec(input, projectionSchema, projectionExpr)
}
```

Selection (also known as Filter)

The query planning step for `Selection` is very similar to `Projection`.

```
is Selection -> {
  val input = createPhysicalPlan(plan.input)
  val filterExpr = createPhysicalExpr(plan.expr, plan.input)
  SelectionExec(input, filterExpr)
}
```

Aggregate

The query planning step for aggregate queries involves evaluating the expressions that define the optional grouping keys and evaluating the expressions that are the inputs to the aggregate functions, and then creating the physical aggregate expressions.

```
is Aggregate -> {
  val input = createPhysicalPlan(plan.input)
  val groupExpr = plan.groupExpr.map { createPhysicalExpr(it, plan.input) }
  val aggregateExpr = plan.aggregateExpr.map {
    when (it) {
      is Max -> MaxExpression(createPhysicalExpr(it.expr, plan.input))
      is Min -> MinExpression(createPhysicalExpr(it.expr, plan.input))
      is Sum -> SumExpression(createPhysicalExpr(it.expr, plan.input))
      else -> throw java.lang.IllegalStateException(
          "Unsupported aggregate function: $it")
    }
  }
  HashAggregateExec(input, groupExpr, aggregateExpr, plan.schema())
}
```

Query Optimizations

_The source code discussed in this chapter can be found in the `optimizer` module of the KQuery project. _ We now have functional query plans, but we rely on the end-user to construct the plans in an efficient way. For example, we expect the user to construct the plan so that filters happen as early as possible, especially before joins, since this limits the amount of data that needs to be processed.

This is a good time to implement a simple rules-based query optimizer that can re-arrange the query plan to make it more efficient.

This is going to become even more important once we start supporting SQL in chapter eleven, because the SQL language only defines how the query should work and does not always allow the user to specify the order that operators and expressions are evaluated in.

Rule-Based Optimizations

Rule based optimizations are a simple and pragmatic approach to apply common sense optimizations to a query plan. These optimizations are typically executed against the logical plan before the physical plan is created, although rule-based optimizations can also be applied to physical plans.

The optimizations work by walking through the logical plan using the visitor pattern and creating a copy of each step in the plan with any necessary modifications applied. This is a much simpler design than attempting to mutate state while walking the plan and is well aligned with a functional programming style that prefers immutable state.

We will use the following interface to represent optimizer rules.

```
interface OptimizerRule {
  fun optimize(plan: LogicalPlan) : LogicalPlan
}
```

We will now look at some common optimization rules that most query engines implement.

Projection Push-Down

The goal of the projection push-down rule is to filter out columns as soon as possible after reading data from disk and before other phases of query execution, to reduce the amount of data that is kept in memory (and potentially transfered over the network in the case of distributed queries) between operators.

In order to know which columns are referenced in a query, we must write recursive code to examine expressions and build up a list of columns.

```
fun extractColumns(expr: List<LogicalExpr>,
                   input: LogicalPlan,
                   accum: MutableSet<String>) {

  expr.forEach { extractColumns(it, input, accum) }
}

fun extractColumns(expr: LogicalExpr,
                   input: LogicalPlan,
                   accum: MutableSet<String>) {

  when (expr) {
    is ColumnIndex -> accum.add(input.schema().fields[expr.i].name)
    is Column -> accum.add(expr.name)
    is BinaryExpr -> {
       extractColumns(expr.l, input, accum)
       extractColumns(expr.r, input, accum)
    }
    is Alias -> extractColumns(expr.expr, input, accum)
    is CastExpr -> extractColumns(expr.expr, input, accum)
    is LiteralString -> {}
    is LiteralLong -> {}
    is LiteralDouble -> {}
    else -> throw IllegalStateException(
        "extractColumns does not support expression: $expr")
  }
}
```

With this utility code in place, we can go ahead and implement the optimizer rule. Note that for the `Projection`, `Selection`, and `Aggregate` plans we are building up the list of column names, but when we reach the `Scan` (which is a leaf node) we replace it with a version of the scan that has the list of column names used elsewhere in the query.

```
class ProjectionPushDownRule : OptimizerRule {

  override fun optimize(plan: LogicalPlan): LogicalPlan {
    return pushDown(plan, mutableSetOf())
  }

  private fun pushDown(plan: LogicalPlan,
                       columnNames: MutableSet<String>): LogicalPlan {
    return when (plan) {
      is Projection -> {
        extractColumns(plan.expr, columnNames)
        val input = pushDown(plan.input, columnNames)
        Projection(input, plan.expr)
      }
      is Selection -> {
        extractColumns(plan.expr, columnNames)
        val input = pushDown(plan.input, columnNames)
        Selection(input, plan.expr)
      }
      is Aggregate -> {
        extractColumns(plan.groupExpr, columnNames)
        extractColumns(plan.aggregateExpr.map { it.inputExpr() }, columnNames)
        val input = pushDown(plan.input, columnNames)
        Aggregate(input, plan.groupExpr, plan.aggregateExpr)
      }
      is Scan -> Scan(plan.name, plan.dataSource, columnNames.toList().sorted())
      else -> throw new UnsupportedOperationException()
    }
  }

}
```

Given this input logical plan:

```
Projection: #id, #first_name, #last_name
  Filter: #state = 'CO'
    Scan: employee; projection=None
```

This optimizer rule will transform it to the following plan.

```
Projection: #id, #first_name, #last_name
  Filter: #state = 'CO'
    Scan: employee; projection=[first_name, id, last_name, state]
```

Predicate Push-Down

The Predicate Push-Down optimization aims to filter out rows as early as possible within a query, to avoid redundant processing. Consider the following which joins an `employee` table and `dept` table and then filters on employees based in Colorado.

```
Projection: #dept_name, #first_name, #last_name
  Filter: #state = 'CO'
    Join: #employee.dept_id = #dept.id
      Scan: employee; projection=[first_name, id, last_name, state]
      Scan: dept; projection=[id, dept_name]
```

The query will produce the correct results but will have the overhead of performing the join for all employees and not just those employees that are based in Colorado. The predicate push-down rule would push the filter down into the join as shown in the following query plan.

```
Projection: #dept_name, #first_name, #last_name
  Join: #employee.dept_id = #dept.id
    Filter: #state = 'CO'
      Scan: employee; projection=[first_name, id, last_name, state]
    Scan: dept; projection=[id, dept_name]
```

The join will now only process a subset of employees, resulting in better performance.

Eliminate Common Subexpressions

Given a query such as `SELECT sum(price * qty) as total_price, sum(price * qty * tax_rate) as total_tax FROM ...`, we can see that the expression `price * qty` appears twice. Rather than perform this computation twice, we could choose to re-write the plan to compute it once.

Original plan:

```
Projection: sum(#price * #qty), sum(#price * #qty * #tax)
  Scan: sales
```

Optimized plan:

```
Projection: sum(#_price_mult_qty), sum(#_price_mult_qty * #tax)
  Projection: #price * #qty as _price_mult_qty
    Scan: sales
```

Converting Correlated Subqueries to Joins

Given a query such as `SELECT id FROM foo WHERE EXISTS (SELECT * FROM bar WHERE foo.id = bar.id)`, a simple implementation would be to scan all rows in `foo` and then perform a lookup in `bar` for each row in `foo`. This would be extremely inefficient, so query engines typically translate correlated subqueries into joins. This is also known as subquery decorrelation.

This query can be rewritten as `SELECT foo.id FROM foo JOIN bar ON foo.id = bar.id`.

```
Projection: foo.id
  LeftSemi Join: foo.id = bar.id
    TableScan: foo projection=[id]
    TableScan: bar projection=[id]
```

If the query is modified to use `NOT EXISTS` rather than `EXISTS` then the query plan would use a `LeftAnti` rather than `LeftSemi` join.

```
Projection: foo.id
  LeftAnti Join: foo.id = bar.id
    TableScan: foo projection=[id]
    TableScan: bar projection=[id]
```

Cost-Based Optimizations

Cost-based optimization refers to optimization rules that use statistics about the underlying data to determine a cost of executing a particular query and then choose an optimal execution plan by looking for one with a low cost. Good examples would be choosing which join algorithm to use, or choosing which order tables should be joined in, based on the sizes of the underlying tables.

One major drawback to cost-based optimizations is that they depend on the availability of accurate and detailed statistics about the underlying data. Such statistics would typically include per-column statistics such as the number of null values, number of distinct values, min and max values, and histograms showing the distribution of values within the column. The histogram is essential to be able to detect that a predicate such as `state = 'CA'` is likely to produce more rows than `state = 'WY'` for example (California is the most populated US state, with 39 million residents, and Wyoming is the least populated state, with fewer than 1 million residents).

When working with file formats such as Orc or Parquet, some of these statistics are available, but generally it is necessary to run a process to build these statistics, and when working with multiple terabytes of data, this can be prohibitive, and outweigh the benefit, especially for ad-hoc queries.

Query Execution

We are now able to write code to execute optimized queries against CSV files.

Before we execute the query with KQuery, it might be useful to use a trusted alternative so that we know what the correct results should be and to get some baseline performance metrics for comparison.

Apache Spark Example

_The source code discussed in this chapter can be found in the `spark` module of KQuery. _ First, we need to create a Spark context. Note that we are using a single thread for execution so that we can make a relatively fair comparison to the performance of the single threaded implementation in KQuery.

```
val spark = SparkSession.builder()
  .master("local[1]")
  .getOrCreate()
```

Next, we need to register the CSV file as a DataFrame against the context.

```
val schema = StructType(Seq(
  StructField("VendorID", DataTypes.IntegerType),
  StructField("tpep_pickup_datetime", DataTypes.TimestampType),
  StructField("tpep_dropoff_datetime", DataTypes.TimestampType),
  StructField("passenger_count", DataTypes.IntegerType),
  StructField("trip_distance", DataTypes.DoubleType),
  StructField("RatecodeID", DataTypes.IntegerType),
  StructField("store_and_fwd_flag", DataTypes.StringType),
  StructField("PULocationID", DataTypes.IntegerType),
  StructField("DOLocationID", DataTypes.IntegerType),
  StructField("payment_type", DataTypes.IntegerType),
  StructField("fare_amount", DataTypes.DoubleType),
  StructField("extra", DataTypes.DoubleType),
  StructField("mta_tax", DataTypes.DoubleType),
  StructField("tip_amount", DataTypes.DoubleType),
  StructField("tolls_amount", DataTypes.DoubleType),
  StructField("improvement_surcharge", DataTypes.DoubleType),
  StructField("total_amount", DataTypes.DoubleType)
))

val tripdata = spark.read.format("csv")
  .option("header", "true")
  .schema(schema)
  .load("/mnt/nyctaxi/csv/yellow_tripdata_2019-01.csv")

tripdata.createOrReplaceTempView("tripdata")
```

Finally, we can go ahead and execute SQL against the DataFrame.

```
val start = System.currentTimeMillis()

val df = spark.sql(
  """SELECT passenger_count, MAX(fare_amount)
    |FROM tripdata
    |GROUP BY passenger_count""".stripMargin)

df.foreach(row => println(row))

val duration = System.currentTimeMillis() - start

println(s"Query took $duration ms")
```

Executing this code on my desktop produces the following output.

```
[1,623259.86]
[6,262.5]
[3,350.0]
[5,760.0]
[9,92.0]
[4,500.0]
[8,87.0]
[7,78.0]
[2,492.5]
[0,36090.3]
Query took 14418 ms
```

KQuery Examples

_The source code discussed in this chapter can be found in the `examples` module of the KQuery project. _ Here is the equivalent query implemented with KQuery. Note that this code differs from the Spark example because KQuery doesn't have the option of specifying the schema of the CSV file yet, so all data types are strings, and this means that we need to add an explicit cast to the query plan to convert the `fare_amount` column to a numeric type.

```
val time = measureTimeMillis {

val ctx = ExecutionContext()

val df = ctx.csv("/mnt/nyctaxi/csv/yellow_tripdata_2019-01.csv", 1*1024)
            .aggregate(
               listOf(col("passenger_count")),
               listOf(max(cast(col("fare_amount"), ArrowTypes.FloatType))))

val optimizedPlan = Optimizer().optimize(df.logicalPlan())
val results = ctx.execute(optimizedPlan)

results.forEach { println(it.toCSV()) }

println("Query took $time ms")
```

This produces the following output on my desktop.

```
Schema<passenger_count: Utf8, MAX: FloatingPoint(DOUBLE)>
1,623259.86
2,492.5
3,350.0
4,500.0
5,760.0
6,262.5
7,78.0
8,87.0
9,92.0
0,36090.3

Query took 6740 ms
```

We can see that the results match those produced by Apache Spark. We also see that the performance is respectable for this size of input. It is very likely that Apache Spark will outperform KQuery with larger data sets since it is optimized for "Big Data".

Removing The Query Optimizer

Let's remove the optimizations and see how much they helped with performance.

```
val time = measureTimeMillis {

val ctx = ExecutionContext()

val df = ctx.csv("/mnt/nyctaxi/csv/yellow_tripdata_2019-01.csv", 1*1024)
            .aggregate(
               listOf(col("passenger_count")),
               listOf(max(cast(col("fare_amount"), ArrowTypes.FloatType))))

val results = ctx.execute(df.logicalPlan())

results.forEach { println(it.toCSV()) }

println("Query took $time ms")
```

This produces the following output on my desktop.

```
1,623259.86
2,492.5
3,350.0
4,500.0
5,760.0
6,262.5
7,78.0
8,87.0
9,92.0
0,36090.3

Query took 36090 ms
```

The results are the same, but the query took about five times as long to execute. This clearly shows the benefit of the projection push-down optimization that was discussed in the previous chapter.

This book is also available for purchase in ePub, MOBI, and PDF format from https://leanpub.com/how-query-engines-work

SQL Support

_The source code discussed in this chapter can be found in the `sql` module of the KQuery project. _ In addition to having the ability to hand-code logical plans, it would be more convenient in some cases to just write SQL. In this chapter, we will build a SQL parser and query planner that can translate SQL queries into logical plans.

Tokenizer

The first step is to convert the SQL query string into a list of tokens representing keywords, literals, identifiers, and operators.

This is a subset of all possible tokens, but it is sufficient for now.

```
interface Token
data class IdentifierToken(val s: String) : Token
data class LiteralStringToken(val s: String) : Token
data class LiteralLongToken(val s: String) : Token
data class KeywordToken(val s: String) : Token
data class OperatorToken(val s: String) : Token
```

We will then need a tokenizer class. This is not particularly interesting to walk through here, and full source code can be found in the companion GitHub repository.

```
class Tokenizer {
  fun tokenize(sql: String): List<Token> {
    // see github repo for code
  }
}
```

Given the input `"SELECT a + b FROM c"` we expect the output to be as follows:

```
listOf(
  KeywordToken("SELECT"),
  IdentifierToken("a"),
  OperatorToken("+"),
  IdentifierToken("b"),
  KeywordToken("FROM"),
  IdentifierToken("c")
)
```

Pratt Parser

We are going to hand-code a SQL parser based on the Top Down Operator Precedence paper published by Vaughan R. Pratt in 1973. Although there are other approaches to building SQL parsers such as using Parser Generators and Parser Combinators, I have found Pratt's approach to work well and it results in code that is efficient, easy to comprehend, and easy to debug.

Here is a bare-bones implementation of a Pratt parser. In my opinion, it is beautiful in its simplicity. Expression parsing is performed by a simple loop that parses a "prefix" expression followed by an optional "infix" expression and keeps doing this until the precedence changes in such a way that the parser recognizes that it has finished parsing the expression. Of course, the implementation of `parsePrefix` and `parseInfix` can recursively call back into the `parse` method and this is where it becomes very powerful.

```
interface PrattParser {

  /** Parse an expression */
  fun parse(precedence: Int = 0): SqlExpr? {
    var expr = parsePrefix() ?: return null
    while (precedence < nextPrecedence()) {
      expr = parseInfix(expr, nextPrecedence())
    }
    return expr
  }

  /** Get the precedence of the next token */
  fun nextPrecedence(): Int

  /** Parse the next prefix expression */
  fun parsePrefix(): SqlExpr?

  /** Parse the next infix expression */
  fun parseInfix(left: SqlExpr, precedence: Int): SqlExpr

}
```

This interface refers to a new `SqlExpr` class which will be our representation of a parsed expression and will largely be a one to one mapping to the expressions defined in the logical plan but for binary expressions we can use a more generic structure where the operator is a string rather than create separate data structures for all the different binary expressions that we will support.

Here are some examples of `SqlExpr` implementations.

```
/** SQL Expression */
interface SqlExpr

/** Simple SQL identifier such as a table or column name */
data class SqlIdentifier(val id: String) : SqlExpr {
  override fun toString() = id
}

/** Binary expression */
data class SqlBinaryExpr(val l: SqlExpr, val op: String, val r: SqlExpr) : SqlExpr {
  override fun toString(): String = "$l $op $r"
}

/** SQL literal string */
data class SqlString(val value: String) : SqlExpr {
  override fun toString() = "'$value'"
}
```

With these classes in place it is possible to represent the expression `foo = 'bar'` with the following code.

```
val sqlExpr = SqlBinaryExpr(SqlIdentifier("foo"), "=", SqlString("bar"))
```

Parsing SQL Expressions

Let's walk through this approach for parsing a simple math expression such as `1 + 2 * 3`. This expression consists of the following tokens.

```
listOf(
  LiteralLongToken("1"),
  OperatorToken("+"),
  LiteralLongToken("2"),
  OperatorToken("*"),
  LiteralLongToken("3")
)
```

We need to create an implementation of the `PrattParser` trait and pass the tokens into the constructor. The tokens are wrapped in a `TokenStream` class that provides some convenience methods such as `next` for consuming the next token, and `peek` for when we want to look ahead without consuming a token.

```
class SqlParser(val tokens: TokenStream) : PrattParser {
}
```

Implementing the `nextPrecedence` method is simple because we only have a small number of tokens that have any precedence here and we need to have the multiplication and division operators have higher precedence than the addition and subtraction operator. Note that the specific numbers returned by this method are not important since they are just used for comparisons. A good reference for operator precedence can be found in the PostgreSQL documentation.

```
override fun nextPrecedence(): Int {
  val token = tokens.peek()
  return when (token) {
    is OperatorToken -> {
      when (token.s) {
        "+", "-" -> 50
        "*", "/" -> 60
        else -> 0
      }
    }
    else -> 0
  }
}
```

The prefix parser just needs to know how to parse literal numeric values.

```
override fun parsePrefix(): SqlExpr? {
  val token = tokens.next() ?: return null
  return when (token) {
    is LiteralLongToken -> SqlLong(token.s.toLong())
    else -> throw IllegalStateException("Unexpected token $token")
  }
}
```

The infix parser just needs to know how to parse operators. Note that after parsing an operator, this method recursively calls back into the top level `parse` method to parse the expression following the operator (the right-hand side of the binary expression).

```
override fun parseInfix(left: SqlExpr, precedence: Int): SqlExpr {
  val token = tokens.peek()
  return when (token) {
    is OperatorToken -> {
      tokens.next()
      SqlBinaryExpr(left, token.s, parse(precedence) ?:
                    throw SQLException("Error parsing infix"))
    }
    else -> throw IllegalStateException("Unexpected infix token $token")
  }
}
```

The precedence logic can be demonstrated by parsing the math expressions `1 + 2 * 3` and `1 * 2 + 3` which should be parsed as `1 + (2 * 3)` and `(1 * 2) + 3` respectively.

_Example: Parsing `1 + 2 _ 3`*

These are the tokens along with their precedence values.

```
Tokens:      [1]  [+]  [2]  [*]  [3]
Precedence:  [0] [50]  [0] [60]  [0]
```

The final result correctly represents the expression as `1 + (2 * 3)`.

```
SqlBinaryExpr(
    SqlLong(1),
    "+",
    SqlBinaryExpr(SqlLong(2), "*", SqlLong(3))
)
```

_Example: Parsing `1 _ 2 + 3`*

```
Tokens:      [1]  [*]  [2]  [+]  [3]
Precedence:  [0] [60]  [0] [50]  [0]
```

The final result correctly represents the expression as `(1 * 2) + 3`.

```
SqlBinaryExpr(
    SqlBinaryExpr(SqlLong(1), "*", SqlLong(2)),
    "+",
    SqlLong(3)
)
```

Parsing a SELECT statement

Now that we have the ability to parse some simple expressions, the next step is to extend the parser to support parsing a SELECT statement into a concrete syntax tree (CST). Note that with other approaches to parsing such as using a parser generator like ANTLR there is an intermediate stage known as an Abstract Syntax Tree (AST) which then needs to be translated to a Concrete Syntax Tree but with the Pratt Parser approach we go directly from tokens to the CST.

Here is an example CST that can represent a simple single-table query with a projection and selection. This will be extended to support more complex queries in later chapters.

```
data class SqlSelect(
    val projection: List<SqlExpr>,
    val selection: SqlExpr,
    val tableName: String) : SqlRelation
```

SQL Query Planner

The SQL Query Planner translates the SQL Query Tree into a Logical Plan. This turns out to be a much harder problem than translating a logical plan to a physical plan due to the flexibility of the SQL language. For example, consider the following simple query.

```
SELECT id, first_name, last_name, salary/12 AS monthly_salary
FROM employee
WHERE state = 'CO' AND monthly_salary > 1000
```

Although this is intuitive to a human reading the query, the selection part of the query (the `WHERE` clause) refers to one expression (`state`) that is not included in the output of the projection so clearly needs to be applied before the projection but also refers to another expression (`salary/12 AS monthly_salary`) which is only available after the projection is applied. We will face similar issues with the `GROUP BY`, `HAVING`, and `ORDER BY` clauses.

There are multiple solutions to this problem. One approach would be to translate this query to the following logical plan, splitting the selection expression into two steps, one before and one after the projection. However, this is only possible because the selection expression is a conjunctive predicate (the expression is true only if all parts are true) and this approach might not be possible for more complex expressions. If the expression had been `state = 'CO' OR monthly_salary > 1000` then we could not do this.

```
Filter: #monthly_salary > 1000
  Projection: #id, #first_name, #last_name, #salary/12 AS monthly_salary
    Filter: #state = 'CO'
      Scan: table=employee
```

A simpler and more generic approach would be to add all the required expressions to the projection so that the selection can be applied after the projection, and then remove any columns that were added by wrapping the output in another projection.

```
Projection: #id, #first_name, #last_name, #monthly_salary
  Filter: #state = 'CO' AND #monthly_salary > 1000
    Projection: #id, #first_name, #last_name, #salary/12 AS monthly_salary, #state
      Scan: table=employee
```

It is worth noting that we will build a "Predicate Push Down" query optimizer rule in a later chapter that will be able to optimize this plan and push the `state = 'CO'` part of the predicate further down in the plan so that it is before the projection.

Translating SQL Expressions

Translating SQL expressions to logical expressions is fairly simple, as demonstrated in this example code.

```
private fun createLogicalExpr(expr: SqlExpr, input: DataFrame) : LogicalExpr {
  return when (expr) {
    is SqlIdentifier -> Column(expr.id)
    is SqlAlias -> Alias(createLogicalExpr(expr.expr, input), expr.alias.id)
    is SqlString -> LiteralString(expr.value)
    is SqlLong -> LiteralLong(expr.value)
    is SqlDouble -> LiteralDouble(expr.value)
    is SqlBinaryExpr -> {
      val l = createLogicalExpr(expr.l, input)
      val r = createLogicalExpr(expr.r, input)
      when(expr.op) {
        // comparison operators
        "=" -> Eq(l, r)
        "!=" -> Neq(l, r)
        ">" -> Gt(l, r)
        ">=" -> GtEq(l, r)
        "<" -> Lt(l, r)
        "<=" -> LtEq(l, r)
        // boolean operators
        "AND" -> And(l, r)
        "OR" -> Or(l, r)
        // math operators
        "+" -> Add(l, r)
        "-" -> Subtract(l, r)
        "*" -> Multiply(l, r)
        "/" -> Divide(l, r)
        "%" -> Modulus(l, r)
        else -> throw SQLException("Invalid operator ${expr.op}")
      }
    }

    else -> throw new UnsupportedOperationException()
  }
}
```

Planning SELECT

If we only wanted to support the use case where all columns referenced in the selection also exist in the projection we could get away with some very simple logic to build the query plan.

```
fun createDataFrame(select: SqlSelect, tables: Map<String, DataFrame>) : DataFrame {

  // get a reference to the data source
  var df = tables[select.tableName] ?:
      throw SQLException("No table named '${select.tableName}'")

  val projectionExpr = select.projection.map { createLogicalExpr(it, df) }

  if (select.selection == null) {
    // apply projection
    return df.select(projectionExpr)
  }

  // apply projection then wrap in a selection (filter)
  return df.select(projectionExpr)
           .filter(createLogicalExpr(select.selection, df))
}
```

However, because the selection could reference both inputs to the projections and outputs from the projection we need to create a more complex plan with an intermediate projection. The first step is to determine which columns are references by the selection filter expression. To do this we will use the visitor pattern to walk the expression tree and build a mutable set of column names.

Here is the utility method we will use to walk the expression tree.

```
private fun visit(expr: LogicalExpr, accumulator: MutableSet<String>) {
  when (expr) {
    is Column -> accumulator.add(expr.name)
    is Alias -> visit(expr.expr, accumulator)
    is BinaryExpr -> {
      visit(expr.l, accumulator)
      visit(expr.r, accumulator)
     }
  }
}
```

With this in place we can now write the following code to convert a SELECT statement into a valid logical plan. This code sample is not perfect and probably contains some bugs for edge cases where there are name clashes between columns in the data source and aliased expressions but we will ignore this for the moment to keep the code simple.

```
fun createDataFrame(select: SqlSelect, tables: Map<String, DataFrame>) : DataFrame {

  // get a reference to the data source
  var df = tables[select.tableName] ?:
    throw SQLException("No table named '${select.tableName}'")

  // create the logical expressions for the projection
  val projectionExpr = select.projection.map { createLogicalExpr(it, df) }

  if (select.selection == null) {
    // if there is no selection then we can just return the projection
    return df.select(projectionExpr)
  }

  // create the logical expression to represent the selection
  val filterExpr = createLogicalExpr(select.selection, df)

  // get a list of columns references in the projection expression
  val columnsInProjection = projectionExpr
    .map { it.toField(df.logicalPlan()).name}
    .toSet()

  // get a list of columns referenced in the selection expression
  val columnNames = mutableSetOf<String>()
  visit(filterExpr, columnNames)

  // determine if the selection references any columns not in the projection
  val missing = columnNames - columnsInProjection

  // if the selection only references outputs from the projection we can
  // simply apply the filter expression to the DataFrame representing
  // the projection
  if (missing.size == 0) {
    return df.select(projectionExpr)
             .filter(filterExpr)
  }

  // because the selection references some columns that are not in the
  // projection output we need to create an interim projection that has
  // the additional columns and then we need to remove them after the
  // selection has been applied
  return df.select(projectionExpr + missing.map { Column(it) })
           .filter(filterExpr)
           .select(projectionExpr.map {
              Column(it.toField(df.logicalPlan()).name)
            })
}
```

Planning for Aggregate Queries

As you can see, the SQL query planner is relatively complex and the code for parsing aggregate queries is quite involved. If you are interested in learning more, please refer to the source code.

Parallel Query Execution

So far, we have been using a single thread to execute queries against individual files. This approach is not very scalable, because queries will take longer to run with larger files or with multiple files. The next step is to implement distributed query execution so that query execution can utilize multiple CPU cores and multiple servers.

The simplest form of distributed query execution is parallel query execution utilizing multiple CPU cores on a single node using threads.

The NYC taxi data set is already conveniently partitioned because there is one CSV file for each month of each year, meaning that there are twelve partitions for the 2019 data set, for example. The most straightforward approach to parallel query execution would be to use one thread per partition to execute the same query in parallel and then combine the results. Suppose this code is running on a computer with six CPU cores with hyper-threading support. In that case, these twelve queries should execute in the same elapsed time as running one of the queries on a single thread, assuming that each month has a similar amount of data.

Here is an example of running an aggregate SQL query in parallel across twelve partitions. This example is implemented using Kotlin coroutines, rather than using threads directly.

The source code for this example can be found at `jvm/examples/src/main/kotlin/ParallelQuery.kt` in the KQuery GitHub repository.

Let us start with the single-threaded code for running one query against one partition.

```
fun executeQuery(path: String, month: Int, sql: String): List<RecordBatch> {
  val monthStr = String.format("%02d", month);
  val filename = "$path/yellow_tripdata_2019-$monthStr.csv"
  val ctx = ExecutionContext()
  ctx.registerCsv("tripdata", filename)
  val df = ctx.sql(sql)
  return ctx.execute(df).toList()
}
```

With this in place, we can now write the following code to run this query in parallel across each of the twelve partitions of data.

```
val start = System.currentTimeMillis()
val deferred = (1..12).map {month ->
  GlobalScope.async {

    val sql = "SELECT passenger_count, " +
        "MAX(CAST(fare_amount AS double)) AS max_fare " +
        "FROM tripdata " +
        "GROUP BY passenger_count"

    val start = System.currentTimeMillis()
    val result = executeQuery(path, month, sql)
    val duration = System.currentTimeMillis() - start
    println("Query against month $month took $duration ms")
    result
  }
}
val results: List<RecordBatch> = runBlocking {
  deferred.flatMap { it.await() }
}
val duration = System.currentTimeMillis() - start
println("Collected ${results.size} batches in $duration ms")
```

Here is the output from this example, running on a desktop computer with 24 cores.

```
Query against month 8 took 17074 ms
Query against month 9 took 18976 ms
Query against month 7 took 20010 ms
Query against month 2 took 21417 ms
Query against month 11 took 21521 ms
Query against month 12 took 22082 ms
Query against month 6 took 23669 ms
Query against month 1 took 23735 ms
Query against month 10 took 23739 ms
Query against month 3 took 24048 ms
Query against month 5 took 24103 ms
Query against month 4 took 25439 ms
Collected 12 batches in 25505 ms
```

As you can see, the total duration was around the same time as the slowest query.

Although we have successfully executed the aggregate query against the partitions, our result is a list of batches of data with duplicate values. For example, there will most likely be a result for `passenger_count=1` from each of the partitions.

Combining Results

For simple queries consisting of projection and selection operators, the results of the parallel queries can be combined (similar to a SQL `UNION ALL` operation), and no further processing is required. More complex queries involving aggregates, sorts, or joins will require a secondary query to be run on the results of the parallel queries to combine the results. The terms "map" and "reduce" are often used to explain this two-step process. The "map" step refers to running one query in parallel across the partitions, and the "reduce" step refers to combining the results into a single result.

For this particular example, it is now necessary to run a secondary aggregation query almost identical to the aggregate query executed against the partitions. One difference is that the second query may need to apply different aggregate functions. For the aggregate functions `min`, `max`, and `sum`, the same operation is used in the map and reduce steps, to get the min of the min or the sum of the sums. For the count expression, we do not want the count of the counts. We want to see the sum of the counts instead.

```
val sql = "SELECT passenger_count, " +
        "MAX(max_fare) " +
        "FROM tripdata " +
        "GROUP BY passenger_count"

val ctx = ExecutionContext()
ctx.registerDataSource("tripdata", InMemoryDataSource(results.first().schema, results))
val df = ctx.sql(sql)
ctx.execute(df).forEach { println(it) }
```

This produces the final result set:

```
1,671123.14
2,1196.35
3,350.0
4,500.0
5,760.0
6,262.5
7,80.52
8,89.0
9,97.5
0,90000.0
```

Smarter Partitioning

Although the strategy of using one thread per file worked well in this example, it does not work as a general-purpose approach to partitioning. If a data source has thousands of small partitions, starting one thread per partition would be inefficient. A better approach would be for the query planner to decide how to share the available data between a specified number of worker threads (or executors).

Some file formats already have a natural partitioning scheme within them. For example, Apache Parquet files consist of multiple "row groups" containing batches of columnar data. A query planner could inspect the available Parquet files, build a list of row groups and then schedule reading these row groups across a fixed number of threads or executors.

It is even possible to apply this technique to unstructured files such as CSV files, but this is not trivial. It is easy to inspect the file size and break the file into equal-sized chunks, but a record could likely span two chunks, so it is necessary to read backward or forwards from a boundary to find the start or end of the record. It is insufficient to look for a newline character because these often appear within records and are also used to delimit records. It is common practice to convert CSV files into a structured format such as Parquet early on in a processing pipeline to improve the efficiency of subsequent processing.

Partition Keys

One solution to this problem is to place files in directories and use directory names consisting of key-value pairs to specify the contents.

For example, we could organize the files as follows:

```
/mnt/nyxtaxi/csv/year=2019/month=1/tripdata.csv
/mnt/nyxtaxi/csv/year=2019/month=2/tripdata.csv
...
/mnt/nyxtaxi/csv/year=2019/month=12/tripdata.csv
```

Given this structure, the query planner could now implement a form of "predicate push down" to limit the number of partitions included in the physical query plan. This approach is often referred to as "partition pruning".

Parallel Joins

When performing an inner join with a single thread, a simple approach is to load one side of the join into memory and then scan the other side, performing lookups against the data stored in memory. This classic Hash Join algorithm is efficient if one side of the join can fit into memory.

The parallel version of this is known as a Partitioned Hash Join or Parallel Hash Join. It involves partitioning both inputs based on the join keys and performing a classic Hash Join on each pair of input partitions.

Distributed Query Execution

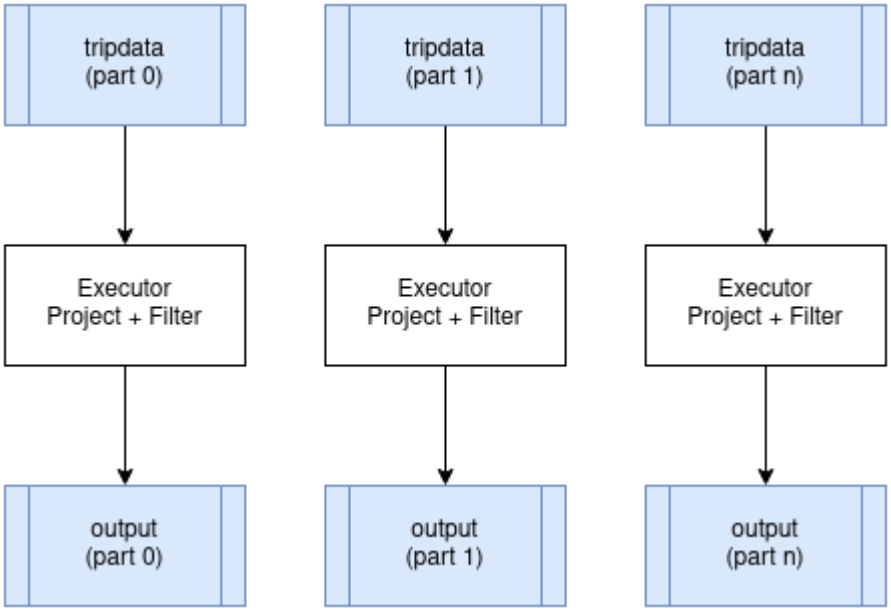
The previous section on Parallel Query Execution covered some fundamental concepts such as partitioning, which we will build on in this section.

To somewhat over-simplify the concept of distributed query execution, the goal is to be able to create a physical query plan which defines how work is distributed to a number of "executors" in a cluster. Distributed query plans will typically contain new operators that describe how data is exchanged between executors at various points during query execution.

In the following sections we will explore how different types of plans are executed in a distributed environment and then discuss building a distributed query scheduler.

Embarrassingly Parallel Operators

Certain operators can run in parallel on partitions of data without any significant overhead when running in a distributed environment. The best examples of these are Projection and Filter. These operators can be applied in parallel to each input partition of the data being operated on and produce a corresponding output partition for each one. These operators do not change the partitioning scheme of the data.



Distributed Aggregates

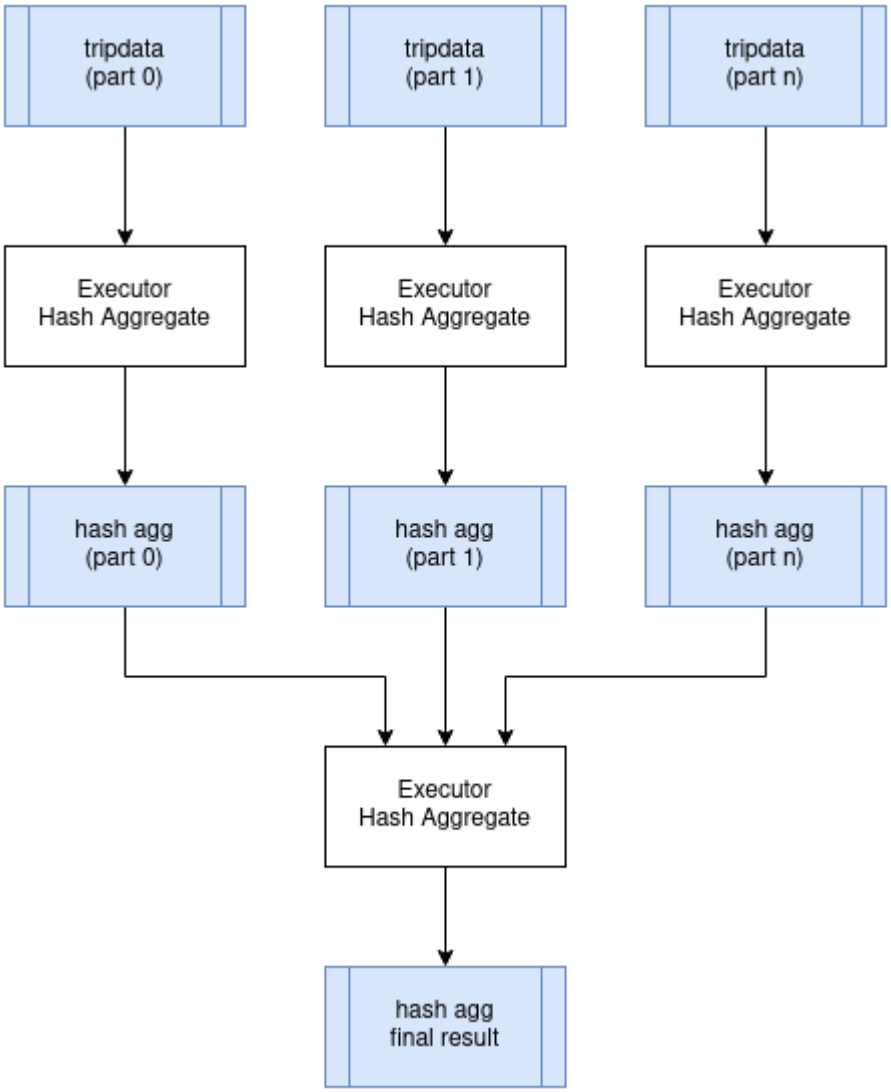
Let's use the example SQL query that we used in the previous chapter on Parallel Query Execution and look at the distributed planning implications of an aggregate query.

```
SELECT passenger_count, MAX(max_fare)
FROM tripdata
GROUP BY passenger_count
```

We can execute this query in parallel on all partitions of the `tripdata` table, with each executor in the cluster processing a subset of these partitions. However, we need to then combine all the resulting aggregated data onto a single node and then apply the final aggregate query so that we get a single result set without duplicate grouping keys (`passenger_count` in this case). Here is one possible logical query plan for representing this. Note the new `Exchange` operator which represents the exchange of data between executors. The physical plan for the exchange could be implemented by writing intermediate results to shared storage, or perhaps by streaming data directly to other executors.

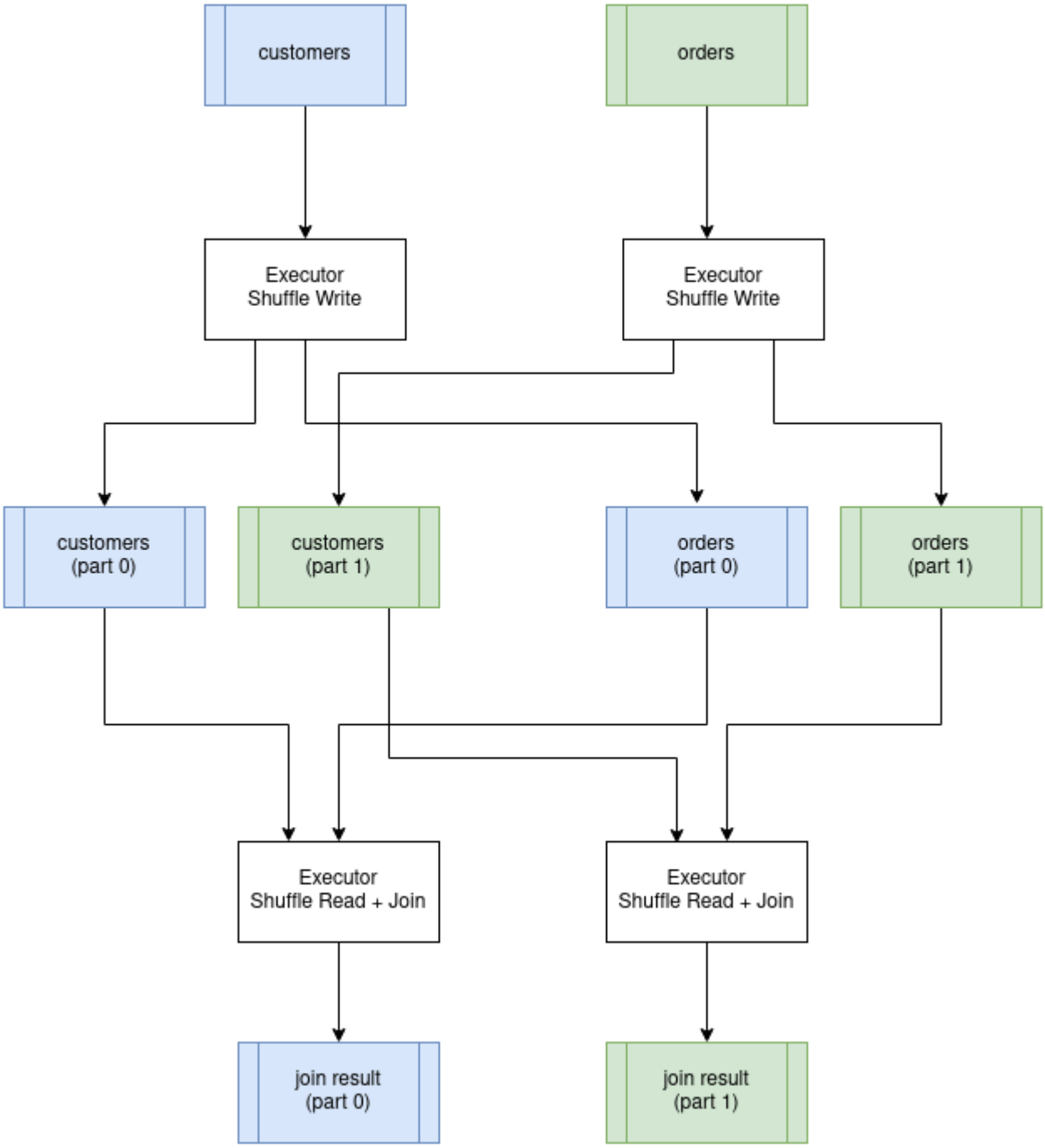
```
HashAggregate: groupBy=[passenger_count], aggr=[MAX(max_fare)]
  Exchange:
    HashAggregate: groupBy=[passenger_count], aggr=[MAX(max_fare)]
      Scan: tripdata.parquet
```

Here is a diagram showing how this query could be executed in a distributed environment:



Distributed Joins

Joins are often the most expensive operation to perform in a distributed environment. The reason for this is that we need to make sure that we organize the data in such a way that both input relations are partitioned on the join keys. For example, if we joining a `customer` table to an `order` table where the join condition is `customer.id = order.customer_id`, then all the rows in both tables for a specific customer must be processed by the same executor. To achieve this, we have to first repartition both tables on the join keys and write the partitions to disk. Once this has completed then we can perform the join in parallel with one join for each partition. The resulting data will remain partitioned by the join keys. This particular join algorithm is called a partitioned hash join. The process of repartitioning the data is known as performing a "shuffle".



Distributed Query Scheduling

Distributed query plans are fundamentally different to in-process query plans because we can't just build a tree of operators and start executing them. The query now requires co-ordination across executors which means that we now need to build a scheduler.

At a high level, the concept of a distributed query scheduler is not complex. The scheduler needs to examine the whole query and break it down into stages that can be executed in isolation (usually in parallel across the executors) and then schedule these stages for execution based on the available resources in the cluster. Once each query stage completes then any subsequent dependent query stages can be scheduled. This process repeats until all query stages have been executed.

The scheduler could also be responsible for managing the compute resources in the cluster so that extra executors can be started on demand to handle the query load.

In the remainder of this chapter, we will discuss the following topics, referring to Ballista and the design that is being implemented in that project.

- Producing a distributed query plan

- Serializing query plans and exchanging them with executors
- Exchange intermediate results between executors
- Optimizing distributed queries

Producing a Distributed Query Plan

As we have seen in the previous examples, some operators can run in parallel on input partitions and some operators require data to be repartitioned. These changes in partitioning are key to planning a distributed query. Changes in partitioning within a plan are sometimes called pipeline breakers and these changes in partitioning define the boundaries between query stages.

We will now use the following SQL query to see how this process works.

```
SELECT customer.id, sum(order.amount) as total_amount
FROM customer JOIN order ON customer.id = order.customer_id
GROUP BY customer.id
```

The physical (non-distributed) plan for this query would look something like this:

```
Projection: #customer.id, #total_amount
  HashAggregate: groupBy=[customer.id], aggr=[MAX(max_fare) AS total_amount]
    Join: condition=[customer.id = order.customer_id]
      Scan: customer
      Scan: order
```

Assuming that the customer and order tables are not already partitioned on customer id, we will need to schedule execution of the first two query stages to repartition this data. These two query stages can run in parallel.

```
Query Stage #1: repartition=[customer.id]
  Scan: customer
Query Stage #2: repartition=[order.customer_id]
  Scan: order
```

Next, we can schedule the join, which will run in parallel for each partition of the two inputs. The next operator after the join is the aggregate, which is split into two parts; the aggregate that runs in parallel and then the final aggregate that requires a single input partition. We can perform the parallel part of this aggregate in the same query stage as the join because this first aggregate does not care how the data is partitioned. This gives us our third query stage, which can now be scheduled for execution. The output of this query stage remains partitioned by customer id.

```
Query Stage #3: repartition=[]
  HashAggregate: groupBy=[customer.id], aggr=[MAX(max_fare) AS total_amount]
    Join: condition=[customer.id = order.customer_id]
      Query Stage #1
      Query Stage #2
```

The final query stage performs the aggregate of the aggregates, reading from all of the partitions from the previous stage.

```
Query Stage #4:
  Projection: #customer.id, #total_amount
    HashAggregate: groupBy=[customer.id], aggr=[MAX(max_fare) AS total_amount]
      QueryStage #3
```

To recap, here is the full distributed query plan showing the query stages that are introduced when data needs to be repartitioned or exchanged between pipelined operations.

```
Query Stage #4:
  Projection: #customer.id, #total_amount
    HashAggregate: groupBy=[customer.id], aggr=[MAX(max_fare) AS total_amount]
      Query Stage #3: repartition=[]
        HashAggregate: groupBy=[customer.id], aggr=[MAX(max_fare) AS total_amount]
          Join: condition=[customer.id = order.customer_id]
            Query Stage #1: repartition=[customer.id]
              Scan: customer
            Query Stage #2: repartition=[order.customer_id]
              Scan: order
```

Serializing a Query Plan

The query scheduler needs to send fragments of the overall query plan to executors for execution.

There are a number of options for serializing a query plan so that it can be passed between processes. Many query engines choose the strategy of using the programming languages native serialization support, which is a suitable choice if there is no requirement to be able to exchange query plans between different programming languages and this is usually the simplest mechanism to implement.

However, there are advantages in using a serialization format that is programming language-agnostic. Ballista uses Google's Protocol Buffers format to define query plans. The project is typically abbreviated as "protobuf".

Here is a subset of the Ballista protocol buffer definition of a query plan.

Full source code can be found at `proto/ballista.proto` in the Ballista github repository.

```
message LogicalPlanNode {
  LogicalPlanNode input = 1;
  FileNode file = 10;
  ProjectionNode projection = 20;
  SelectionNode selection = 21;
  LimitNode limit = 22;
  AggregateNode aggregate = 23;
}

message FileNode {
  string filename = 1;
  Schema schema = 2;
  repeated string projection = 3;
}

message ProjectionNode {
  repeated LogicalExprNode expr = 1;
}

message SelectionNode {
  LogicalExprNode expr = 2;
}

message AggregateNode {
  repeated LogicalExprNode group_expr = 1;
  repeated LogicalExprNode aggr_expr = 2;
}

message LimitNode {
  uint32 limit = 1;
}
```

The protobuf project provides tools for generating language-specific source code for serializing and de-serializing data.

Serializing Data

Data must also be serialized as it is streamed between clients and executors and between executors.

Apache Arrow provides an IPC (Inter-process Communication) format for exchanging data between processes. Because of the standardized memory layout provided by Arrow, the raw bytes can be transferred directly between memory and an

input/output device (disk, network, etc) without the overhead typically associated with serialization. This is effectively a zero copy operation because the data does not have to be transformed from its in-memory format to a separate serialization format.

However, the metadata about the data, such as the schema (column names and data types) does need to be encoded using Google Flatbuffers. This metadata is small and is typically serialized once per result set or per batch so the overhead is small.

Another advantage of using Arrow is that it provides very efficient exchange of data between different programming languages.

Apache Arrow IPC defines the data encoding format but not the mechanism for exchanging it. Arrow IPC could be used to transfer data from a JVM language to C or Rust via JNI for example.

Choosing a Protocol

Now that we have chosen serialization formats for query plans and data, the next question is how do we exchange this data between distributed processes.

Apache Arrow provides a Flight protocol which is intended for this exact purpose. Flight is a new general-purpose client-server framework to simplify high performance transport of large datasets over network interfaces.

The Arrow Flight libraries provide a development framework for implementing a service that can send and receive data streams. A Flight server supports several basic kinds of requests:

- **Handshake**: a simple request to determine whether the client is authorized and, in some cases, to establish an implementation-defined session token to use for future requests
- **ListFlights**: return a list of available data streams
- **GetSchema**: return the schema for a data stream
- **GetFlightInfo**: return an "access plan" for a dataset of interest, possibly requiring consuming multiple data streams. This request can accept custom serialized commands containing, for example, your specific application parameters.
- **DoGet**: send a data stream to a client
- **DoPut**: receive a data stream from a client
- **DoAction**: perform an implementation-specific action and return any results, i.e. a generalized function call
- **ListActions**: return a list of available action types

The `GetFlightInfo` method could be used to compile a query plan and return the necessary information for receiving the results, for example, followed by calls to `DoGet` on each executor to start receiving the results from the query.

Streaming

It is important that results of a query can be made available as soon as possible and be streamed to the next process that needs to operate on that data, otherwise there would be unacceptable latency involved as each operation would have to wait for the previous operation to complete.

However, some operations require all the input data to be received before any output can be produced. A sort operation is a good example of this. It isn't possible to completely sort a dataset until the whole data set has been received. The problem can be alleviated by increasing the number of partitions so that a large number of partitions are sorted in parallel and then the sorted batches can be combined efficiently using a merge operator.

Custom Code

It is often necessary to run custom code as part of a distributed query or computation. For a single language query engine it is often possible to use the language's built-in serialization mechanism to transmit this code over the network at query execution time which is very convenient during development. Another approach is to publish compiled code to a repository so that it can be downloaded into a cluster at runtime. For JVM based systems, a maven repository could be used. A more general purpose approach is to package all runtime dependencies into a Docker image.

The query plan needs to provide the necessary information to load the user code at runtime. For JVM based systems this could be a classpath and a class name. For C based systems, this could be the path to a shared object. In either case, the user code will need to implement some known API.

Distributed Query Optimizations

Distributed query execution has a lot of overhead compared to parallel query execution on a single host and should only be used when there is benefit in doing so. I recommend reading the paper Scalability! But at what COST for some interesting perspectives on this topic.

Also, there are many ways to distribute the same query so how do we know which one to use?

One answer is to build a mechanism to determine the cost of executing a particular query plan and then create some subset of all possible combinations of query plan for a given problem and determine which one is most efficient.

There are many factors involved in computing the cost of an operation and there are different resource costs and limitations involved.

- **Memory**: We are typically concerned with availability of memory rather than performance. Processing data in memory is orders of magnitude faster than reading and writing to disk.
- **CPU**: For workloads that are parallelizable, more CPU cores means better throughput.
- **GPU**: Some operations are orders of magnitude faster on GPUs compared to CPUs.
- **Disk**: Disks have finite read and write speeds and cloud vendors typically limit the number of I/O operations per second (IOPS). Different types of disk have different performance characteristics (spinning disk vs SSD vs NVMe).
- **Network**: Distributed query execution involves streaming data between nodes. There is a throughput limitation imposed by the networking infrastructure.
- **Distributed Storage**: It is very common for source data to be stored in a distributed file system (HDFS) or object store (Amazon S3, Azure Blob Storage) and there is a cost in transferring data between distributed storage and local file systems.
- **Data Size**: The size of the data matters. When performing a join between two tables and data needs to be transferred over the network, it is better to transfer the smaller of the two tables. If one of the tables can fit in memory than a more efficient join operation can be used.
- **Monetary Cost**: If a query can be computed 10% faster at 3x the cost, is it worth it? That is a question best answered by the user of course. Monetary costs are typically controlled by limiting the amount of compute resource that is available.

Query costs can be computed upfront using an algorithm if enough information is known ahead of time about the data, such as how large the data is, the cardinality of the partition of join keys used in the query, the number of partitions, and so on. This all depends on certain statistics being available for the data set being queried.

Another approach is to just start running a query and have each operator adapt based on the input data it receives. Apache Spark 3.0.0 introduced an Adaptive Query Execution feature that does just this.

Testing

Query engines are complex, and it is easy to inadvertently introduce subtle bugs that could result in queries returning incorrect results, so it is important to have rigorous testing in place.

Unit Testing

An excellent first step is to write unit tests for the individual operators and expressions, asserting that they produce the correct output for a given input. It is also essential to cover error cases.

Here are some suggestions for things to consider when writing unit tests:

- What happens if an unexpected data type is used? For example, calculating `SUM` on an input of strings.
- Tests should cover edge cases, such as using the minimum and maximum values for numeric data types, and NaN (not a number) for floating point types, to ensure that they are handled correctly.
- Tests should exist for underflow and overflow cases. For example, what happens when two long (64-bit) integer types are multiplied?
- Tests should also ensure that null values are handled correctly.

When writing these tests, it is important to be able to construct record batches and column vectors with arbitrary data to use as inputs for operators and expressions. Here is an example of such a utility method.

```
private fun createRecordBatch(schema: Schema,
                              columns: List<List<Any?>>): RecordBatch {

    val rowCount = columns[0].size
    val root = VectorSchemaRoot.create(schema.toArrow(),
                                       RootAllocator(Long.MAX_VALUE))
    root.allocateNew()
    (0 until rowCount).forEach { row ->
        (0 until columns.size).forEach { col ->
            val v = root.getVector(col)
            val value = columns[col][row]
            when (v) {
                is Float4Vector -> v.set(row, value as Float)
                is Float8Vector -> v.set(row, value as Double)
                ...
            }
        }
    }
    root.rowCount = rowCount

    return RecordBatch(schema, root.fieldVectors.map { ArrowFieldVector(it) })
}
```

Here is an example unit test for the "greater than or equals" (`>=`) expression being evaluated against a record batch containing two columns containing double-precision floating point values.

```
@Test
fun `gteq doubles`() {

    val schema = Schema(listOf(
            Field("a", ArrowTypes.DoubleType),
            Field("b", ArrowTypes.DoubleType)
    ))

    val a: List<Double> = listOf(0.0, 1.0,
                                 Double.MIN_VALUE, Double.MAX_VALUE, Double.NaN)
    val b = a.reversed()

    val batch = createRecordBatch(schema, listOf(a,b))

    val expr = GtEqExpression(ColumnExpression(0), ColumnExpression(1))
    val result = expr.evaluate(batch)

    assertEquals(a.size, result.size())
    (0 until result.size()).forEach {
        assertEquals(if (a[it] >= b[it]) 1 else 0, result.getValue(it))
    }
}
```

Integration Testing

Once unit tests are in place, the next step is to write integration tests that execute queries consisting of multiple operators and expressions and assert that they produce the expected output.

There are a few popular approaches to integration testing of query engines:

- **Imperative Testing:** Hard-coded queries and expected results, either written as code or stored as files containing the queries and results.
- **Comparative Testing:** This approach involves executing queries against another (trusted) query engine and asserting that both query engines produced the same results.
- **Fuzzing:** Generating random operator and expression trees to capture edge cases and get comprehensive test coverage.

Fuzzing

Much of the complexity of query engines comes from the fact that operators and expressions can be combined through infinite combinations due to the nested nature of operator and expression trees, and it is unlikely that hand-coding test queries will be comprehensive enough.

Fuzzing is a technique for producing random input data. When applied to query engines, this means creating random query plans.

Here is an example of creating random expressions against a DataFrame. This is a recursive method and can produce deeply nested expression trees, so it is important to build in a maximum depth mechanism.

```
fun createExpression(input: DataFrame, depth: Int, maxDepth: Int): LogicalExpr {
    return if (depth == maxDepth) {
        // return a leaf node
        when (rand.nextInt(4)) {
            0 -> ColumnIndex(rand.nextInt(input.schema().fields.size))
            1 -> LiteralDouble(rand.nextDouble())
            2 -> LiteralLong(rand.nextLong())
            3 -> LiteralString(randomString(rand.nextInt(64)))
            else -> throw IllegalStateException()
        }
    } else {
        // binary expressions
        val l = createExpression(input, depth+1, maxDepth)
        val r = createExpression(input, depth+1, maxDepth)
        return when (rand.nextInt(8)) {
            0 -> Eq(l, r)
            1 -> Neq(l, r)
            2 -> Lt(l, r)
            3 -> LtEq(l, r)
            4 -> Gt(l, r)
            5 -> GtEq(l, r)
            6 -> And(l, r)
            7 -> Or(l, r)
            else -> throw IllegalStateException()
        }
    }
}
```

Here is example of an expression generated with this method. Note that column references are represented here with an index following a hash, e.g. `#1` represents column at index 1. This expression is almost certainly invalid (depending on the query engine implementation), and this is to be expected when using a fuzzer. This is still valuable because it will test error conditions that otherwise would not be covered when manually writing tests.

```
#5 > 0.5459397414890019 < 0.3511239641785846 OR 0.9137719758607572 > -6938650321297559787 < #0 AND #3 < #4
AND 'qn0NN' OR '1gS46UuarGz2CdeYDJDEW3Go6ScMmRhA3NgPJWMpgZCcML1Ped8haRxOkM9F' >= -8765295514236902140 <
4303905842995563233 OR 'IAseGJesQMOI5OG4KrkitichlFduZGtjXoNkVQI0Alaf2ELUTTIci' = 0.857970478666058 >=
0.8618195163699196 <= '9jaFR2kDX88qrKCh2BSArLq517cR8u2' OR 0.28624225053564 <= 0.6363627130199404 >
0.19648131921514966 >= -567468767705106376 <= #0 AND 0.6582592932801918 =
'OtJ0ryPUeSJCcMnaLngBDBfIpJ9SbPb6hC5nWqeAP1rWbozfkPjcKdaelzc' >= #0 >= -2876541212976899342 = #4 >=
-3694865812331663204 = 'gWkQLswcU' != #3 > 'XiXzKNrwrWnQmr3JYojCVuncW9YaeFc' >= 0.5123788261193981 >= #2
```

A similar approach can be taken when creating logical query plans.

```
fun createPlan(input: DataFrame,
               depth: Int,
               maxDepth: Int,
               maxExprDepth: Int): DataFrame {

    return if (depth == maxDepth) {
        input
    } else {
        // recursively create an input plan
        val child = createPlan(input, depth+1, maxDepth, maxExprDepth)
        // apply a transformation to the plan
        when (rand.nextInt(2)) {
            0 -> {
                val exprCount = 1.rangeTo(rand.nextInt(1, 5))
                child.project(exprCount.map {
                    createExpression(child, 0, maxExprDepth)
                })
            }
            1 -> child.filter(createExpression(input, 0, maxExprDepth))
            else -> throw IllegalStateException()
        }
    }
}
```

Here is an example of a logical query plan produced by this code.

```
Filter: 'VejBmVBpYp7gHxHIUB6UcGx' OR 0.7762591612853446
  Filter: 'vHGbOKKqR' <= 0.41876514212913307
    Filter: 0.9835090312561898 <= 3342229749483308391
      Filter: -5182478750208008322 < -8012833501302297790
        Filter: 0.3985688976088563 AND #1
          Filter: #5 OR 'WkaZ54spnoI4MBtFpQaQgk'
            Scan: employee.csv; projection=None
```

This straightforward approach to fuzzing will produce a high percentage of invalid plans. It could be improved to reduce the risk of creating invalid logical plans and expressions by adding more contextual awareness. For example, generating an `AND` expression could generate left and right expressions that produce a Boolean result. However, there is a danger in only creating correct plans because it could limit the test coverage. Ideally, it should be possible to configure the fuzzer with rules for producing query plans with different characteristics.

This book is also available for purchase in ePub, MOBI, and PDF format from https://leanpub.com/how-query-engines-work

Benchmarks

Each query engine is unique in terms of performance, scalability, and resource requirements, often with different trade-offs. It is important to have good benchmarks to understand the performance and scalability characteristics.

Measuring Performance

Performance is often the simplest characteristic to measure and usually refers to the time it takes to perform a particular operation. For example, benchmarks can be built to measure the performance of specific queries or categories of query.

Performance tests typically involve executing a query multiple times and measuring elapsed time.

Measuring Scalability

Scalability can be an overloaded term and there are many different types of scalability. The term scalability generally refers to how performance varies with different values for some variable that affects performance.

One example would be measuring scalability as total data size increases to discover how performance is impacted, when querying 10 GB of data versus 100 GB or 1 TB. A common goal is to demonstrate linear scalability, meaning that querying 100 GB of data should take 10 times as long as querying 10 GB of data. Linear scalability makes it easy for users to reason about expected behavior.

Other examples of variables that affect performance are:

- Number of concurrent users, requests, or queries.
- Number of data partitions.
- Number of physical disks.
- Number of cores.
- Number of nodes.
- Amount of RAM available.
- Type of hardware (Raspberry Pi versus Desktop, for example).

Concurrency

When measuring scalability based on number of concurrent requests, we are often more interested in throughput (total number of queries executed per period of time) rather than the duration of individual queries, although we typically would collect that information as well.

Automation

Benchmarks are often very time-consuming to run and automation is essential so that the benchmarks can be run often, perhaps once per day or once per week, so that any performance regressions can be caught early.

Automation is also important for ensuring that benchmarks are executed consistently and that results are collected with all relevant details that might be needed when analyzing the results.

Here are some examples of the type of data that should be collected when executing benchmarks:

Hardware Configuration

- Type of hardware
- Number of CPU cores

- Available memory and disk space
- Operating system name and version

Environment

- Environment variables (being careful not to leak secrets)

Benchmark Configuration

- Version of benchmark software used
- Version of software under test
- Any configuration parameters or files
- Filenames of any data files being queried
- Data sizes and checksums for the data files
- Details about the query that was executed

Benchmark Results

- Date/time benchmark was started
- Start time and end time for each query
- Error information for any failed queries

Comparing Benchmarks

It is important to compare benchmarks between releases of the software so that changes in performance characteristics are apparent and can be investigated further. Benchmarks produce a lot of data that is often hard to compare manually, so it can be beneficial to build tooling to help with this process.

Rather than comparing two sets of performance data directly, tooling can perform a "diff" of the data and show percentage differences between two or more runs of the same benchmark. It is also useful to be able to produce charts showing multiple benchmark runs.

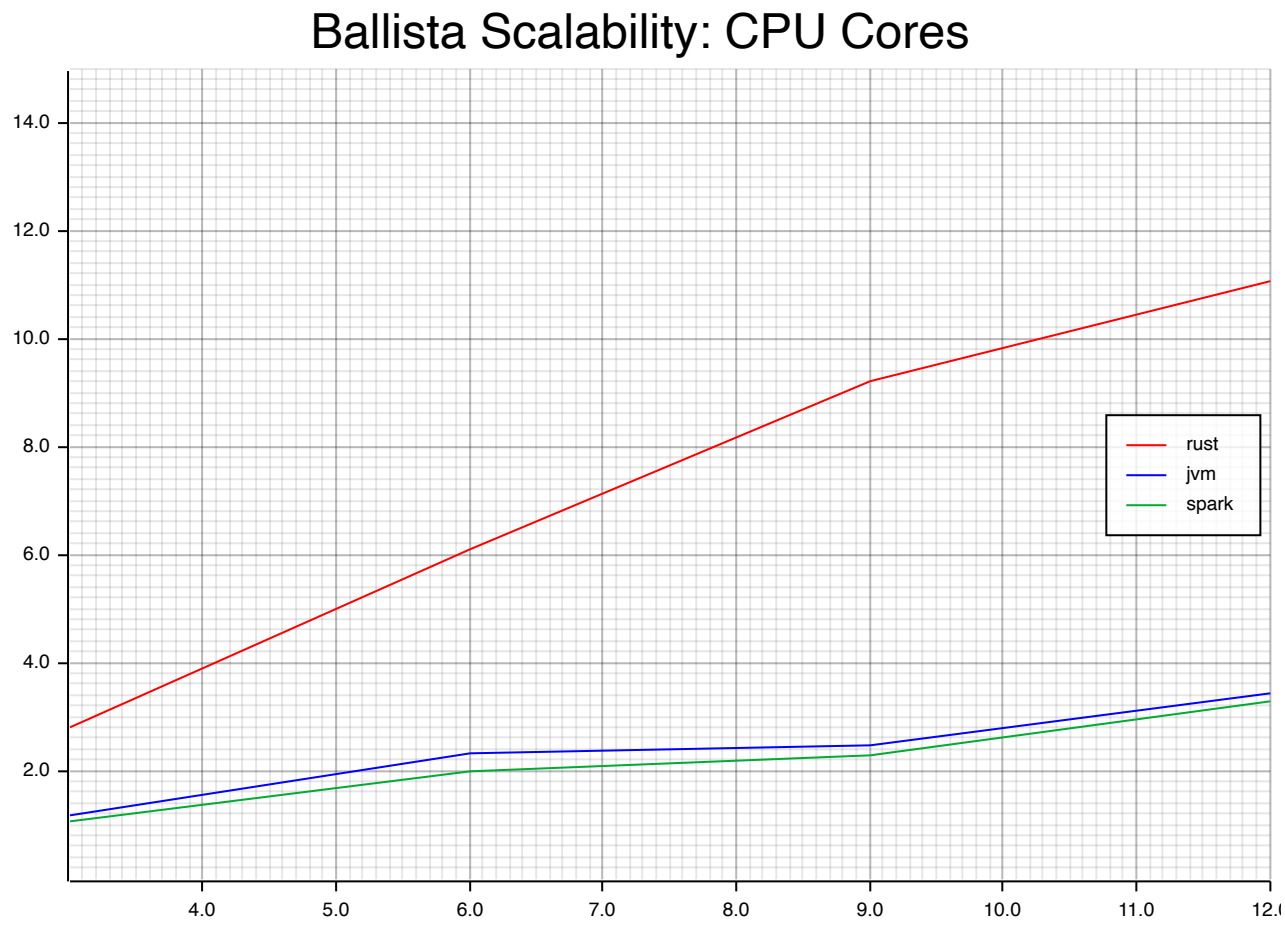
Publishing Benchmark Results

Here is an example of some real benchmark results, comparing query execution time for the Rust and JVM executors in Ballista, compared to Apache Spark. Although it is clear from this data that the Rust executor is performing well, the benefit can be expressed much better by producing a chart.

CPU Cores	Ballista Rust	Ballista JVM	Apache Spark
3	21.431	51.143	56.557
6	9.855	26.002	30.184
9	6.51	24.435	26.401
12	5.435	17.529	18.281

Rather than chart the query execution times, it is often better to chart the throughput. In this case, throughput in terms of queries per minute can be calculated by dividing 60 seconds by the execution time. If a query takes 5 seconds to execute on a single thread, then it should be possible to run 12 queries per minute.

Here is an example chart showing the scalability of throughput as the number of CPU cores increases.



Transaction Processing Council (TPC) Benchmarks

The Transaction Processing Council is a consortium of database vendors that collaborate on creating and maintaining various database benchmark suites to allow for fair comparisons between vendor's systems. Current TPC member companies include Microsoft, Oracle, IBM, Hewlett Packard Enterprise, AMD, Intel, and NVIDIA.

The first benchmark, TPC-A, was published in 1989 and other benchmarks have been created since then. TPC-C is a well known OLTP benchmark used when comparing traditional RDBMS databases, and TPC-H (discontinued) and TPC-DS are often used for measuring performance of "Big Data" query engines.

TPC benchmarks are seen as the "gold standard" in the industry and are complex and time consuming to implement fully. Also, results for these benchmarks can only be published by TPC members and only after the benchmarks have been audited by the TPC. Taking TPC-DS as an example, the only companies to have ever published official results at the time of writing are Alibaba.com, H2C, SuperMicro, and Databricks.

However, the TPC has a Fair Use policy that allows non-members to create unofficial benchmarks based on TPC benchmarks, as long as certain conditions are followed, such as prefixing any use of the term TPC with "derived from TPC". For example, "Performance of Query derived from TPC-DS Query 14". TPC Copyright Notice and License Agreements must also be maintained. There are also limitations on the types of metrics that can be published.

Many open source projects simply measure the time to execute individual queries from the TPC benchmark suites and use this as a way to track performance over time and for comparison with other query engines.

This book is also available for purchase in ePub, MOBI, and PDF format from https://leanpub.com/how-query-engines-work

Further Resources

I hope that you found this book useful and that you now have a better understanding of the internals of query engines. If there are topics that you feel haven't been covered adequately, or at all, I would love to hear about it so I can consider adding additional content in a future revision of this book.

Feedback can be posted on the public forum on the Leanpub site, or you can message me directly via twitter at @andygrove_io.

Open-Source Projects

There are numerous open-source projects that contain query engines and working with these projects is a great way to learn more about the topic. Here are just a few examples of popular open-source query engines.

- Apache Arrow
- Apache Calcite
- Apache Drill
- Apache Hadoop
- Apache Hive
- Apache Impala
- Apache Spark
- Facebook Presto
- NVIDIA RAPIDS Accelerator for Apache Spark

YouTube

I only recently discovered Andy Pavlo's lecture series, which is available on YouTube (here). This covers much more than just query engines, but there is extensive content on query optimization and execution. I highly recommend watching these videos.

Sample Data

Earlier chapters reference the New York City Taxi & Limousine Commission Trip Record Data data set. The yellow and green taxi trip records include fields capturing pick-up and drop-off dates/times, pick-up and drop-off locations, trip distances, itemized fares, rate types, payment types, and driver-reported passenger counts. The data is provided in CSV format. The KQuery project contains source code for converting these CSV files into Parquet format.

Data can be downloaded by following the links on the website or by downloading the files directly from S3. For example, users on Linux or Mac can use `curl` or `wget` to download the January 2019 data for Yellow Taxis with the following command and create scripts to download other files based on the file naming convention.

```
wget https://s3.amazonaws.com/nyc-tlc/trip+data/yellow_tripdata_2019-01.csv
```