Introduction to Query Engines

What is a Relational Database System?

Relational Database Management Systems:

- store data as *relations*: set of named tuples

What is a Relational Database System?

Relational Database Management Systems:

- store data as *relations*: set of named tuples
- abstract away from how data is stored and processed (physical data independence)

What is a Relational Database System?

Relational Database Management Systems:

- store data as *relations*: set of named tuples
- abstract away from how data is stored and processed (physical data independence)
- users formulate *declarative* queries to retrieve/compute data, using *relational algebra*

What is a Relational Database System?

Relational Database Management Systems:

- store data as *relations*: set of named tuples
- abstract away from how data is stored and processed (physical data independence)
- users formulate *declarative* queries to retrieve/compute data, using *relational algebra*
- typically use SQL as the query language

What is a Relational Database System?

Relational Database Management Systems:

- store data as *relations*: set of named tuples
- abstract away from how data is stored and processed (physical data independence)
- users formulate *declarative* queries to retrieve/compute data, using *relational algebra*
- typically use SQL as the query language
- *power the business world*

Why isn't one system enough for all?

In most cases, performance of database systems is highly important!

$\Rightarrow$ systems are optimized for

- different workloads

Why isn't one system enough for all?

In most cases, performance of database systems is highly important!

$\Rightarrow$ systems are optimized for

- different workloads
  - **OLTP**: Online Transaction Processing systems handle frequent updates.
    e.g. banking services

Why isn't one system enough for all?

In most cases, performance of database systems is highly important!

$\Rightarrow$ systems are optimized for

- different workloads
  - **OLTP**: Online Transaction Processing systems handle frequent updates.
    e.g. banking services
  - **OLAP**: Online Analytical Processing systems handle analysis of large datasets.
    e.g. business analytics and recommendations for large online shops

Why isn't one system enough for all?

In most cases, performance of database systems is highly important!

$\Rightarrow$ systems are optimized for

- different workloads
    - **OLTP**: Online Transaction Processing systems handle frequent updates.
      e.g. banking services
    - **OLAP**: Online Analytical Processing systems handle analysis of large datasets.
      e.g. business analytics and recommendations for large online shops
    - **HTAP**: Hybrid transactions/analytical processing systems support both workloads.
      e.g. business analytics and recommendations for large online shops on live data

Why isn't one system enough for all?

In most cases, performance of database systems is highly important!

$\Rightarrow$ systems are optimized for

- different workloads
  - **OLTP**: Online Transaction Processing systems handle frequent updates.
    e.g. banking services
  - **OLAP**: Online Analytical Processing systems handle analysis of large datasets.
    e.g. business analytics and recommendations for large online shops
  - **HTAP**: Hybrid transactions/analytical processing systems support both workloads.
    e.g. business analytics and recommendations for large online shops on live data
- different use-cases (e.g., time-series, graph)

Why isn't one system enough for all?

In most cases, performance of database systems is highly important!

$\Rightarrow$ systems are optimized for

- different workloads
  - **OLTP**: Online Transaction Processing systems handle frequent updates.
    e.g. banking services
  - **OLAP**: Online Analytical Processing systems handle analysis of large datasets.
    e.g. business analytics and recommendations for large online shops
  - **HTAP**: Hybrid transactions/analytical processing systems support both workloads.
    e.g. business analytics and recommendations for large online shops on live data
- different use-cases (e.g., time-series, graph)
- different hardware/environments

Sometimes other criteria (e.g., robustness) are more important

What is a Query Engine?

- a large part of database systems

What is a Query Engine?

- a large part of database systems
- takes an SQL query as input

What is a Query Engine?

- a large part of database systems
- takes an SQL query as input
- executes the query

What is a Query Engine?

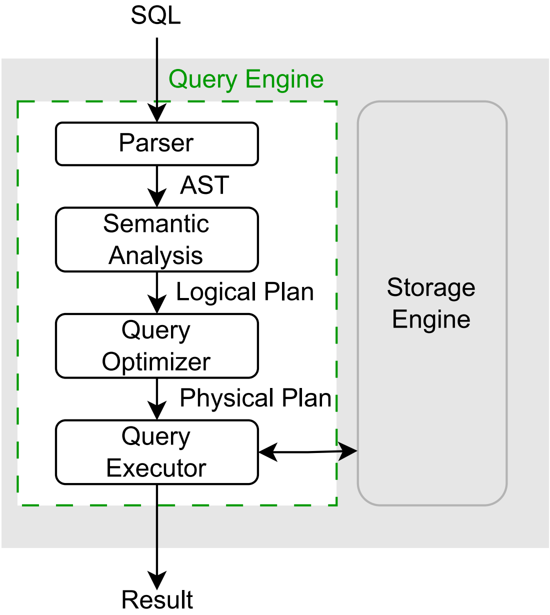
- a large part of database systems
- takes an SQL query as input
- executes the query
- interacts with other components

What is a Query Engine?

- a large part of database systems
- takes an SQL query as input
- executes the query
- interacts with other components
- returns the result

What is a Query Engine?

- a large part of database systems
- takes an SQL query as input
- executes the query
- interacts with other components
- returns the result

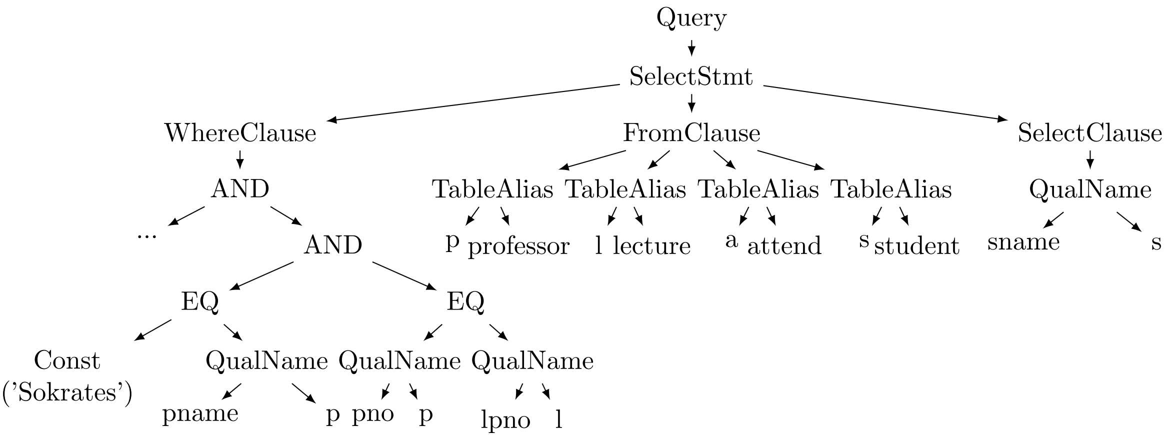


Query Engine: SQL Query

```sql
select s.sname
from student s, attend a, lecture l, professor p
where s.sno = a.asno and a.alno = l.lno and
      l.lpno = p.pno and p.pname ='Sokrates'
```

Query Engine: Parser & Semantic Analysis

- Parser creates abstract syntax tree (AST)



- Semantic Analysis: resolve references, type inference, eliminate syntactic sugar

Query Engine: Logical Operators

- canonical translation into logical operators

Query Engine: Logical Operators

- canonical translation into logical operators
- based on relational algebra

Query Engine: Logical Operators

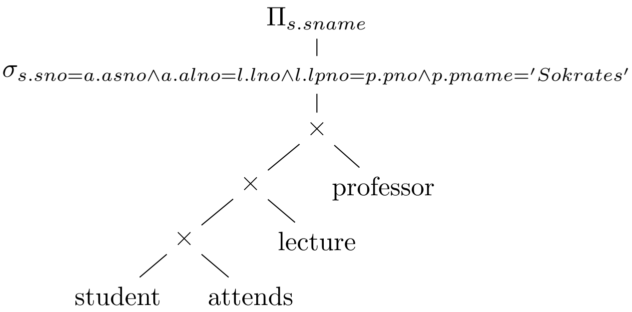
- canonical translation into logical operators
- based on relational algebra
- but: smaller differences (e.g., multi-set semantics)

Query Engine: Logical Operators

- canonical translation into logical operators
- based on relational algebra
- but: smaller differences (e.g., multi-set semantics)
- also new logical operators (e.g., window functions)

Query Engine: Logical Operators

- canonical translation into logical operators
- based on relational algebra
- but: smaller differences (e.g., multi-set semantics)
- also new logical operators (e.g., window functions)



Query Engine: Query Optimizer

- transforms plan into optimized but equivalent plan

Query Engine: Query Optimizer

- transforms plan into optimized but equivalent plan
- common optimizations:

Query Engine: Query Optimizer

- transforms plan into optimized but equivalent plan
- common optimizations:
  - simplifications

Query Engine: Query Optimizer

- transforms plan into optimized but equivalent plan
- common optimizations:
  - simplifications
  - predicate pushdown

Query Engine: Query Optimizer

- transforms plan into optimized but equivalent plan
- common optimizations:
  - simplifications
  - predicate pushdown
  - cost-based reordering of operators

Query Engine: Query Optimizer

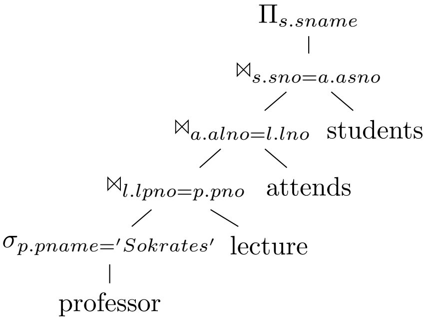
- transforms plan into optimized but equivalent plan
- common optimizations:
  - simplifications
  - predicate pushdown
  - cost-based reordering of operators
  - unnesting of correlated subqueries

Query Engine: Query Optimizer

- transforms plan into optimized but equivalent plan
- common optimizations:
  - simplifications
  - predicate pushdown
  - cost-based reordering of operators
  - unnesting of correlated subqueries
- complex topic on its own which we won't cover here in detail

Query Engine: Query Optimizer

- transforms plan into optimized but equivalent plan
- common optimizations:
  - simplifications
  - predicate pushdown
  - cost-based reordering of operators
  - unnesting of correlated subqueries
- complex topic on its own which we won't cover here in detail



Query Engine: Physical Operators

- in many cases there are different *physical* implementations of *logical* operators

Query Engine: Physical Operators

- in many cases there are different *physical* implementations of *logical* operators
- query optimizer usually select an implementation based on required properties and estimated costs

Query Engine: Physical Operators

- in many cases there are different *physical* implementations of *logical* operators
- query optimizer usually select an implementation based on required properties and estimated costs
- typical variants:

Query Engine: Physical Operators

- in many cases there are different *physical* implementations of *logical* operators
- query optimizer usually select an implementation based on required properties and estimated costs
- typical variants:
  - Index-based: use existing index (e.g., B-tree) to efficiently execute *index*-joins or tablescans

Query Engine: Physical Operators

- in many cases there are different *physical* implementations of *logical* operators
- query optimizer usually select an implementation based on required properties and estimated costs
- typical variants:
  - Index-based: use existing index (e.g., B-tree) to efficiently execute *index*-joins or tablescans
  - Hash-based: use hash-tables for implementing for example *hash*-joins and *hash*-aggregations

Query Engine: Physical Operators

- in many cases there are different *physical* implementations of *logical* operators
- query optimizer usually select an implementation based on required properties and estimated costs
- typical variants:
  - Index-based: use existing index (e.g., B-tree) to efficiently execute *index*-joins or tablescans
  - Hash-based: use hash-tables for implementing for example *hash*-joins and *hash*-aggregations
  - Sorting-based: sorts data and then exploits this for *sortmerge*-joins or *sort-based* aggregations

Query Engine: Volcano/Iterator model

Query Engine: Volcano/Iterator model

```
class Iterator:
    def open()
    def next()

class Output(Iterator):
  def next():
      while True:
          row = child.next()
          if row is None: break
          print(row)

class TableScan(Iterator):
      i = 0
  def next():
      if i >= len(table): return None
      row = table[i]
      i += 1
      return row

class Selection(Iterator):
  def next():
      while True:
          row = child.next()
          if row is None: return None
          if pred(row): return row
```

```
class HashJoin(Iterator):

  def open():
    ht = {}
    # build hashtable with left size
    while True:
      l = left.next()
      if l is None: break
      ht.insert(l)
    q = queue()

  def next():
      # we still have tuples left
      if q: return q.pop()
      # we need to get a new tuple
      while True:
          r = right.next()
          if r is None: return None
          # compute matches for current
          for m in ht.lookup(r):
            q.push(m+r)
          # return first match
          if q: return q.pop()
```

Query Engine: Query Execution

Query Engine: Query Execution

- Iterator model is *one* way of executing queries

Query Engine: Query Execution

- Iterator model is *one* way of executing queries
- Problem: high overhead per tuple

Query Engine: Query Execution

- Iterator model is *one* way of executing queries
- Problem: high overhead per tuple
- Other approaches try to avoid or amortize this overhead

Query Engine: Query Execution

- Iterator model is *one* way of executing queries
- Problem: high overhead per tuple
- Other approaches try to avoid or amortize this overhead

Pull-based vs Push-based

Query Engine: Query Execution

- Iterator model is *one* way of executing queries
- Problem: high overhead per tuple
- Other approaches try to avoid or amortize this overhead

Pull-based vs Push-based

- pull-based (`next()` interface)

Query Engine: Query Execution

- Iterator model is *one* way of executing queries
- Problem: high overhead per tuple
- Other approaches try to avoid or amortize this overhead

Pull-based vs Push-based

- pull-based (`next()` interface)
- push-based (`consume()` interface)

Query Engine: Query Execution

- Iterator model is *one* way of executing queries
- Problem: high overhead per tuple
- Other approaches try to avoid or amortize this overhead

Pull-based vs Push-based

- pull-based (`next()` interface)
- push-based (`consume()` interface)

Interpretation vs Compilation

Query Engine: Query Execution

- Iterator model is *one* way of executing queries
- Problem: high overhead per tuple
- Other approaches try to avoid or amortize this overhead

Pull-based vs Push-based

- pull-based (`next()` interface)
- push-based (`consume()` interface)

Interpretation vs Compilation

- interpretation: virtual function calls

Query Engine: Query Execution

- Iterator model is *one* way of executing queries
- Problem: high overhead per tuple
- Other approaches try to avoid or amortize this overhead

Pull-based vs Push-based

- pull-based (`next()` interface)
- push-based (`consume()` interface)

Interpretation vs Compilation

- interpretation: virtual function calls
- compilation: generate *specialized* code for each query

Query Engine: Query Execution

- Iterator model is *one* way of executing queries
- Problem: high overhead per tuple
- Other approaches try to avoid or amortize this overhead

Pull-based vs Push-based

- pull-based (`next()` interface)
- push-based (`consume()` interface)

Interpretation vs Compilation

- interpretation: virtual function calls
- compilation: generate *specialized* code for each query

How many tuples at a time?

Query Engine: Query Execution

- Iterator model is *one* way of executing queries
- Problem: high overhead per tuple
- Other approaches try to avoid or amortize this overhead

Pull-based vs Push-based

- pull-based (`next()` interface)
- push-based (`consume()` interface)

Interpretation vs Compilation

- interpretation: virtual function calls
- compilation: generate *specialized* code for each query

How many tuples at a time?

- tuple-at-a-time

Query Engine: Query Execution

- Iterator model is *one* way of executing queries
- Problem: high overhead per tuple
- Other approaches try to avoid or amortize this overhead

Pull-based vs Push-based

- pull-based (`next()` interface)
- push-based (`consume()` interface)

Interpretation vs Compilation

- interpretation: virtual function calls
- compilation: generate *specialized* code for each query

How many tuples at a time?

- tuple-at-a-time
- vector-at-a-time

Query Engine: Query Execution

- Iterator model is *one* way of executing queries
- Problem: high overhead per tuple
- Other approaches try to avoid or amortize this overhead

Pull-based vs Push-based

- pull-based (`next()` interface)
- push-based (`consume()` interface)

Interpretation vs Compilation

- interpretation: virtual function calls
- compilation: generate *specialized* code for each query

How many tuples at a time?

- tuple-at-a-time
- vector-at-a-time
- column-at-a-time

Query Engine: Query Execution

- Iterator model is *one* way of executing queries
- Problem: high overhead per tuple
- Other approaches try to avoid or amortize this overhead
- Examples:
  - Postgres (pull, interpret, tuple-at-a-time)
  - DuckDB (push, interpret, vector-at-a-time)
  - Hyper (push, compile, tuple-at-a-time)
  - ...

Pull-based vs Push-based

- pull-based (`next()` interface)
- push-based (`consume()` interface)

Interpretation vs Compilation

- interpretation: virtual function calls
- compilation: generate *specialized* code for each query

How many tuples at a time?

- tuple-at-a-time
- vector-at-a-time
- column-at-a-time

Query Engine: Query Execution

- Iterator model is *one* way of executing queries
- Problem: high overhead per tuple
- Other approaches try to avoid or amortize this overhead
- Examples:
  - Postgres (pull, interpret, tuple-at-a-time)
  - DuckDB (push, interpret, vector-at-a-time)
  - Hyper (push, compile, tuple-at-a-time)
  - ...
- **We will learn more about this in the seminar**

Pull-based vs Push-based

- pull-based (`next()` interface)
- push-based (`consume()` interface)

Interpretation vs Compilation

- interpretation: virtual function calls
- compilation: generate *specialized* code for each query

How many tuples at a time?

- tuple-at-a-time
- vector-at-a-time
- column-at-a-time

Challenges of building query engines

Challenges of building query engines

Complex Systems

Challenges of building query engines

Complex Systems

- Many (non-standard) operators (e.g, SemiJoins)

Challenges of building query engines

Complex Systems

- Many (non-standard) operators (e.g, SemiJoins)
- SQL is complex!

Challenges of building query engines

Complex Systems

- Many (non-standard) operators (e.g, SemiJoins)
- SQL is complex!
- optimizations add further operators (e.g., GroupJoin)

Challenges of building query engines

Complex Systems

- Many (non-standard) operators (e.g, SemiJoins)
- SQL is complex!
- optimizations add further operators (e.g., GroupJoin)

Performance Focused

Challenges of building query engines

Complex Systems

- Many (non-standard) operators (e.g, SemiJoins)
- SQL is complex!
- optimizations add further operators (e.g., GroupJoin)

Performance Focused

- Get most out of the hardware

Challenges of building query engines

Complex Systems

- Many (non-standard) operators (e.g, SemiJoins)
- SQL is complex!
- optimizations add further operators (e.g., GroupJoin)

Performance Focused

- Get most out of the hardware
- Exploit Parallelism

Challenges of building query engines

Complex Systems

- Many (non-standard) operators (e.g, SemiJoins)
- SQL is complex!
- optimizations add further operators (e.g., GroupJoin)

Performance Focused

- Get most out of the hardware
- Exploit Parallelism
- SIMD

Challenges of building query engines

Complex Systems

- Many (non-standard) operators (e.g, SemiJoins)
- SQL is complex!
- optimizations add further operators (e.g., GroupJoin)

Performance Focused

- Get most out of the hardware
- Exploit Parallelism
- SIMD
- GPUs

Challenges of building query engines

Complex Systems

- Many (non-standard) operators (e.g, SemiJoins)
- SQL is complex!
- optimizations add further operators (e.g., GroupJoin)

Performance Focused

- Get most out of the hardware
- Exploit Parallelism
- SIMD
- GPUs
- avoid frequent allocations

Benchmarking

- Focus on Performance $\rightarrow$ benchmarks

Benchmarking

- Focus on Performance $\rightarrow$ benchmarks
- Different goals depending on workload:

Benchmarking

- Focus on Performance $\rightarrow$ benchmarks
- Different goals depending on workload:
    - OLAP: low latency

Benchmarking

- Focus on Performance $\rightarrow$ benchmarks
- Different goals depending on workload:
    - OLAP: low latency
    - OLTP: high throughput

Benchmarking

- Focus on Performance $\rightarrow$ benchmarks
- Different goals depending on workload:
  - OLAP: low latency
  - OLTP: high throughput
- standardized benchmarks provided by TPC (Transaction Processing Council)

Benchmarking

- Focus on Performance $\rightarrow$ benchmarks
- Different goals depending on workload:
  - OLAP: low latency
  - OLTP: high throughput
- standardized benchmarks provided by TPC (Transaction Processing Council)
  - TPC-H: 22 OLAP queries

Benchmarking

- Focus on Performance → benchmarks
- Different goals depending on workload:
  - OLAP: low latency
  - OLTP: high throughput
- standardized benchmarks provided by TPC (Transaction Processing Council)
  - TPC-H: 22 OLAP queries
  - TPC-DS: ≈ 100 OLAP queries

Benchmarking

- Focus on Performance $\rightarrow$ benchmarks
- Different goals depending on workload:
  - OLAP: low latency
  - OLTP: high throughput
- standardized benchmarks provided by TPC (Transaction Processing Council)
  - TPC-H: 22 OLAP queries
  - TPC-DS: $\approx$ 100 OLAP queries
  - TPC-C: OLTP benchmark

Benchmarking

- Focus on Performance → benchmarks
- Different goals depending on workload:
  - OLAP: low latency
  - OLTP: high throughput
- standardized benchmarks provided by TPC (Transaction Processing Council)
  - TPC-H: 22 OLAP queries
  - TPC-DS: ≈ 100 OLAP queries
  - TPC-C: OLTP benchmark
- A lot more benchmarks exist with a different focus (e.g., JOB, SSB, clickbench)

Query Engines @ TUM

TUM's database group has been building query engine for a long time

Query Engines @ TUM

TUM's database group has been building query engine for a long time

Hyper

- started $\approx$ 2010
- 2015: startup
- 2016: sold to Tableau
- 2020: Tableau sold to Salesforce

Query Engines @ TUM

TUM's database group has been building query engine for a long time

Hyper

- started $\approx$ 2010
- 2015: startup
- 2016: sold to Tableau
- 2020: Tableau sold to Salesforce

Umbra

- started $\approx$ 2018
- large parts of our group work on/with Umbra
- 2024: CedarDB

Query Engines @ TUM

TUM's database group has been building query engine for a long time

Hyper

- started ≈ 2010
- 2015: startup
- 2016: sold to Tableau
- 2020: Tableau sold to Salesforce

Umbra

- started ≈ 2018
- large parts of our group work on/with Umbra
- 2024: CedarDB

LingoDB

- started 2021
- open source