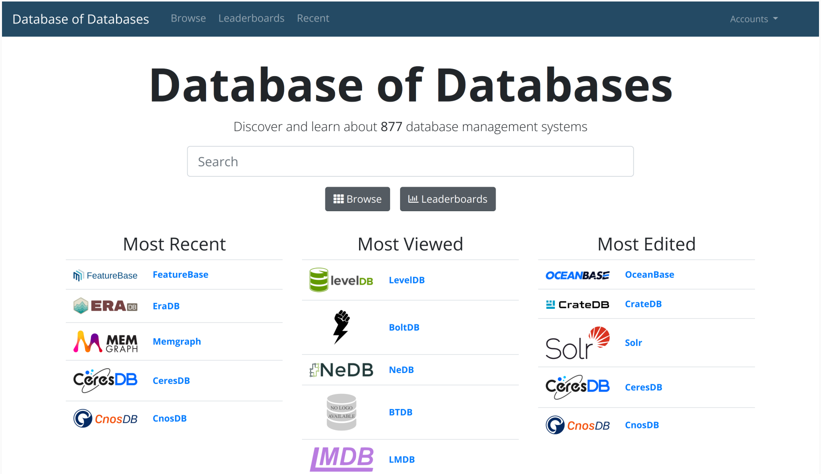
mutable

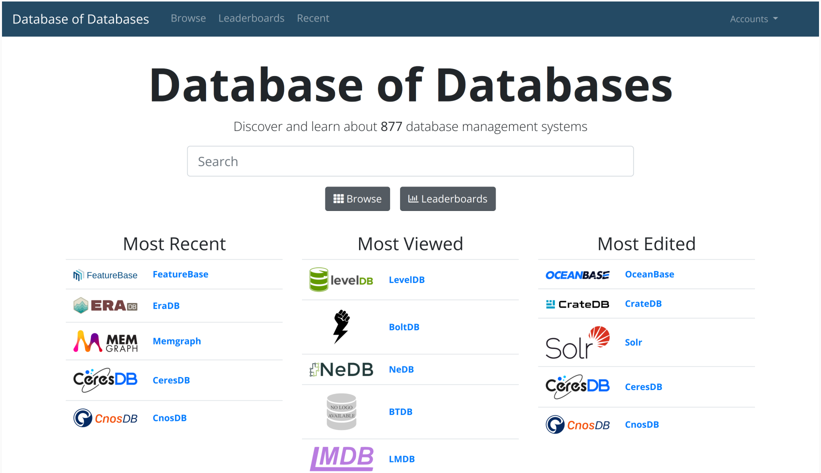
A Modern DBMS for Research and Fast Prototyping

Immanuel Haffner   Jens Dittrich

January 9, 2023 @ CIDR

Saarland University
Saarland Informatics Campus


Database of Databases
Browse
Leaderboards
Recent
Accounts
Database of Databases
Discover and learn about 877 database management systems
Search
Browse
Leaderboards
Most Recent
FeatureBase
EraDB
Memgraph
CeresDB
CnosDB
Most Viewed
LevelDB
BoltDB
NeDB
BTDB
LMDB
Most Edited
OceanBase
CrateDB
Solr
CeresDB
CnosDB



877 database systems listed

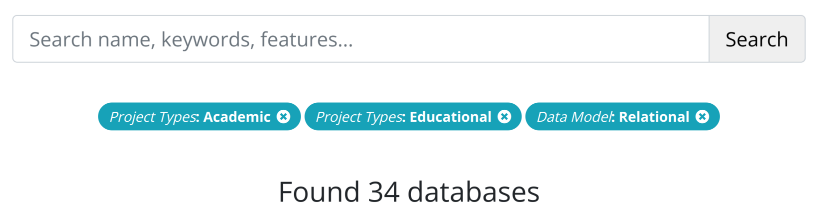
Refine search to **academic** or **educational** projects with a **relational** data model.

Search name, keywords, features... | Search

Project Types: **Academic** ⊗ *Project Types*: **Educational** ⊗ *Data Model*: **Relational** ⊗

Found 34 databases

Refine search to **academic** or **educational** projects with a **relational** data model.



Still contains some popular open-source projects, e.g. PostgreSQL, DuckDB, and NoisePage.

Add **Code Generation** to search criteria.

Search name, keywords, features... | Search

Project Types: **Academic** ⊗ *Project Types*: **Educational** ⊗ *Data Model*: **Relational** ⊗
Query Compilation: **Code Generation** ⊗

Found 2 databases

Add **Code Generation** to search criteria.

Search name, keywords, features... | Search

Project Types: **Academic** *Project Types*: **Educational** *Data Model*: **Relational** *Query Compilation*: **Code Generation**

Found 2 databases

Only NoisePage (open source) and Umbra (closed source) remain.

Status Quo of Open Source DBMSs?

Brief Summary

Status Quo of Open Source DBMSs?

Brief Summary

- PostgreSQL remains the top dog.

Status Quo of Open Source DBMSs?

Brief Summary

- PostgreSQL remains the top dog.
- A few emerging systems bring state-of-the-art technologies to open source, e.g. DuckDB and NoisePage.

Status Quo of Open Source DBMSs?

Brief Summary

- PostgreSQL remains the top dog.
- A few emerging systems bring state-of-the-art technologies to open source, e.g. DuckDB and NoisePage.
- Still, much research remains proprietary / closed source, e.g. HyPer and Umbra.

Status Quo of Open Source DBMSs?

Brief Summary

- PostgreSQL remains the top dog.
- A few emerging systems bring state-of-the-art technologies to open source, e.g. DuckDB and NoisePage.
- Still, much research remains proprietary / closed source, e.g. HyPer and Umbra.

Subjective Problem

- It is difficult for researchers and developers to get started with the available open source projects.

Status Quo of Open Source DBMSs?

Brief Summary

- PostgreSQL remains the top dog.
- A few emerging systems bring state-of-the-art technologies to open source, e.g. DuckDB and NoisePage.
- Still, much research remains proprietary / closed source, e.g. HyPer and Umbra.

Subjective Problem

- It is difficult for researchers and developers to get started with the available open source projects.
  - Lack of documentation.
  - Many built-in assumptions or design decisions.

Our Vision

A system that…

Our Vision

A system that…

- is primarily targeted at researchers and developers.

Our Vision

A system that…

- is primarily targeted at researchers and developers.
- may serve as a unifying framework for database research.

Our Vision

A system that...

- is primarily targeted at researchers and developers.
- may serve as a unifying framework for database research.
- imposes as few design decisions on the developer as possible.

Our Vision

A system that…

- is primarily targeted at researchers and developers.
- may serve as a unifying framework for database research.
- imposes as few design decisions on the developer as possible.
- is flexible and can be configured to the developer's demands.

Our Vision

A system that…

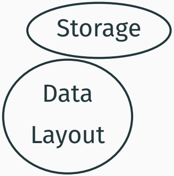
- is primarily targeted at researchers and developers.
- may serve as a unifying framework for database research.
- imposes as few design decisions on the developer as possible.
- is flexible and can be configured to the developer’s demands.
- provides documentation for developers and eases onboarding.

Our Approach

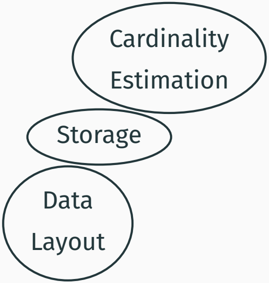
Our Approach

Data Layout

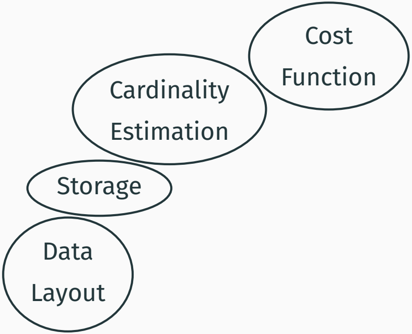
Our Approach



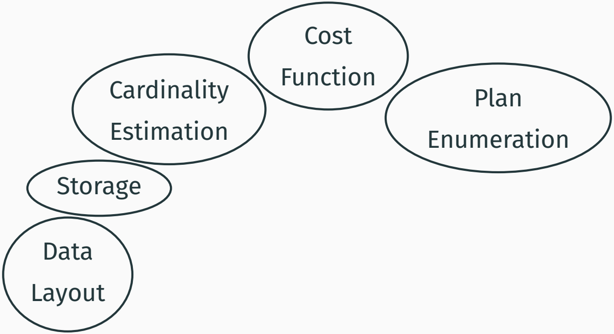
Our Approach



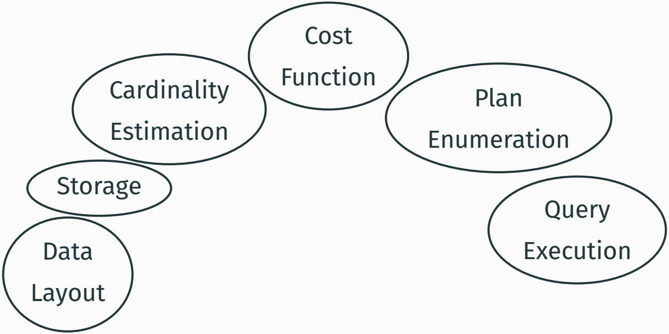
Our Approach



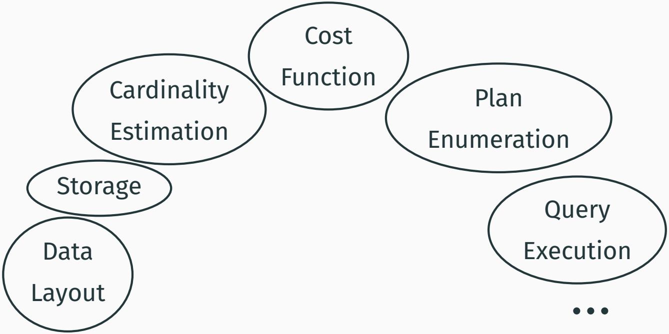
Our Approach



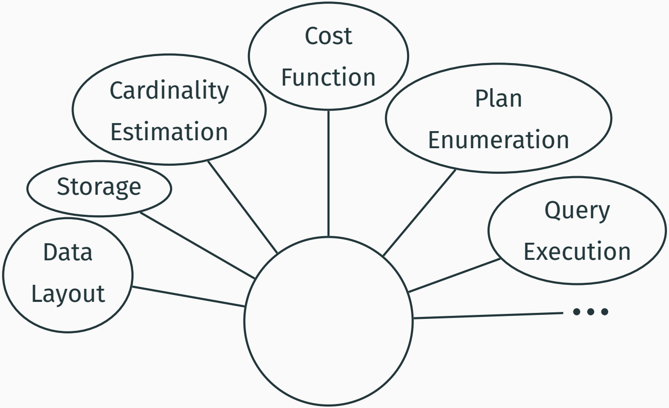
Our Approach



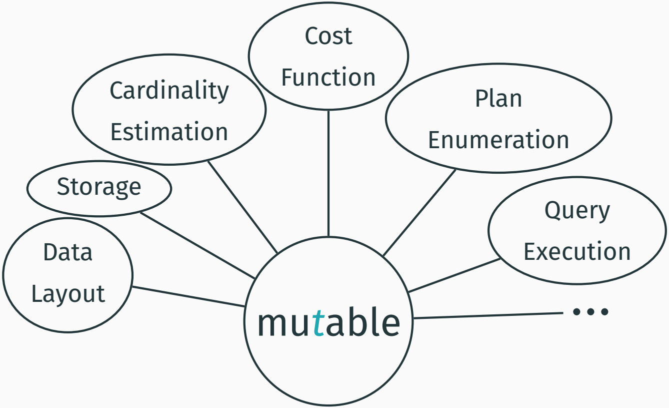
Our Approach



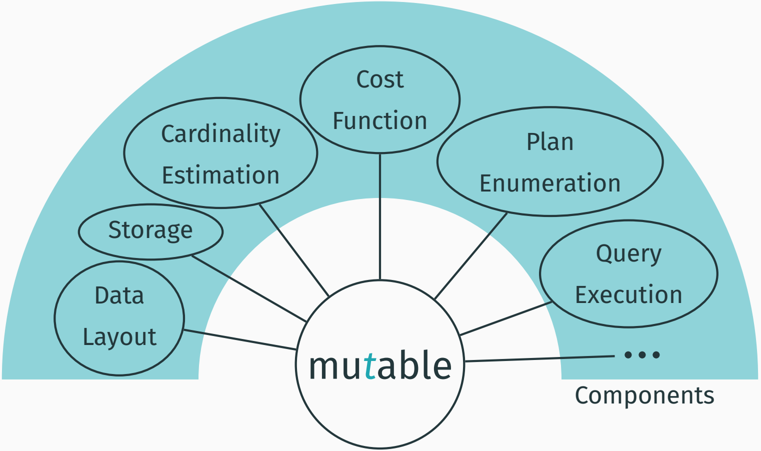
Our Approach



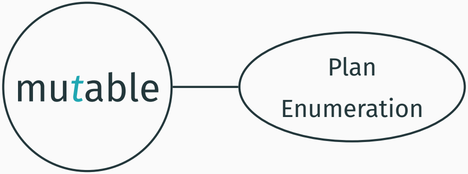
Our Approach



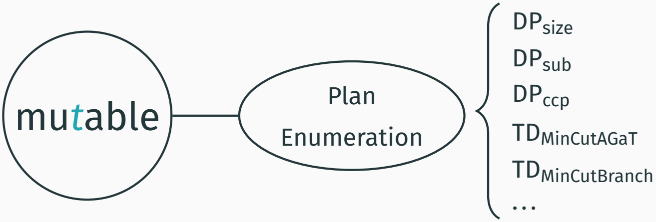
Our Approach



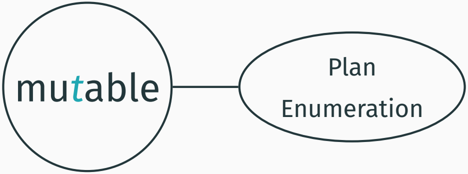
Design Goals: (1) Extensibility



Design Goals: (1) Extensibility



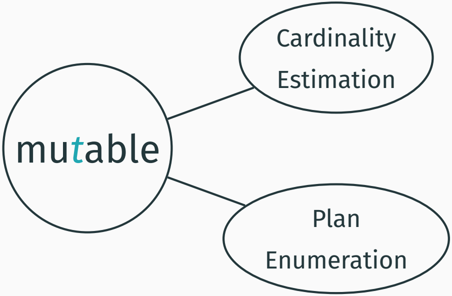
Design Goals: (1) Extensibility



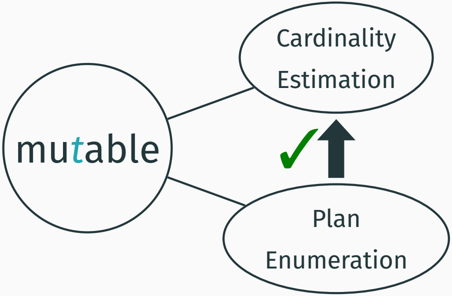
$\mathrm{DP}_{\mathrm{size}}$
$\mathrm{DP}_{\mathrm{sub}}$
$\mathrm{DP}_{\mathrm{ccp}}$
$\mathrm{TD}_{\mathrm{MinCutAGaT}}$
$\mathrm{TD}_{\mathrm{MinCutBranch}}$
...

- Extend mu*t*able by implementations of a component.
- Proper documentation of components.
- Clean component API.

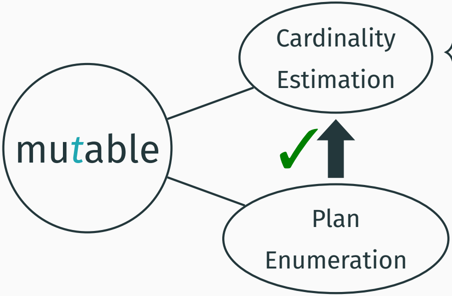
Design Goals: (2) Separation of Concerns



Design Goals: (2) Separation of Concerns

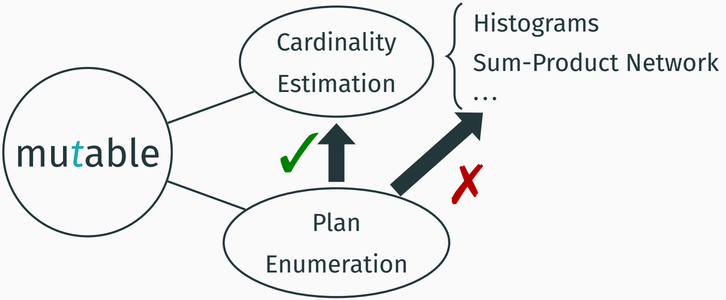


Design Goals: (2) Separation of Concerns

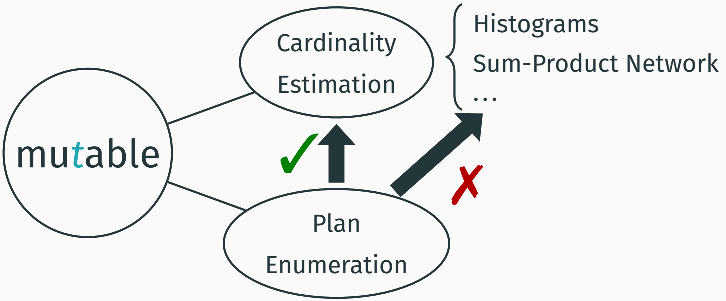


- Histograms
- Sum-Product Network
- ...

Design Goals: (2) Separation of Concerns



Design Goals: (2) Separation of Concerns



- Each component is independent of other components.
- Only knowledge of API of other components may be necessary.
- In mutable, components must appear *stateless* to the outside.

Design Goals: (3) Abstraction...

```
struct PlanEnumerator {
  /** Enumerate feasible plans for query \p G.
   * \param G  graph representation of the query
   * \param CE cardinality estimator component of the
   *           queried database
   * \param CF cost function to minimize
   * \param PT table of best plans found, with one
               entry per feasible partial plan */
  virtual void enumerate_plans(
    const QueryGraph &G,              // value (in)
    const CardinalityEstimator &CE, // component
    const CostFunction &CF,          // component
    PlanTable &PT                     // value (in & out)
  ) const = 0;
};
```

Design Goals: (3) Abstraction...

```
struct PlanEnumerator {
  /** Enumerate feasible plans for query \p G.
   * \param G  graph representation of the query
   * \param CE cardinality estimator component of the
   *           queried database
   * \param CF cost function to minimize
   * \param PT table of best plans found, with one
               entry per feasible partial plan */
  virtual void enumerate_plans(
    const QueryGraph &G,                // value (in)
    const CardinalityEstimator &CE, // component
    const CostFunction &CF,          // component
    PlanTable &PT                   // value (in & out)
  ) const = 0;
};
```

- The `PlanEnumerator` component appears stateless to the outside.

Design Goals: (3) Abstraction...

```
struct PlanEnumerator {
  /** Enumerate feasible plans for query \p G.
   * \param G  graph representation of the query
   * \param CE cardinality estimator component of the
   *           queried database
   * \param CF cost function to minimize
   * \param PT table of best plans found, with one
   *           entry per feasible partial plan */
  virtual void enumerate_plans(
    const QueryGraph &G,               // value (in)
    const CardinalityEstimator &CE, // component
    const CostFunction &CF,          // component
    PlanTable &PT                      // value (in & out)
  ) const = 0;
};
```

- The `PlanEnumerator` component appears stateless to the outside.
- Values go in, values come out.

Design Goals: (3) Abstraction...

```
struct PlanEnumerator {
  /** Enumerate feasible plans for query \p G.
   * \param G  graph representation of the query
   * \param CE cardinality estimator component of the
   *           queried database
   * \param CF cost function to minimize
   * \param PT table of best plans found, with one
   *           entry per feasible partial plan */
  virtual void enumerate_plans(
    const QueryGraph &G,                 // value (in)
    const CardinalityEstimator &CE, // component
    const CostFunction &CF,          // component
    PlanTable &PT                        // value (in & out)
  ) const = 0;
};
```

- The `PlanEnumerator` component appears stateless to the outside.
- Values go in, values come out.
- `PlanEnumerator` component makes use of other components.

Design Goals: (4) …without Regret

Regret?

Abstraction through dynamic dispatch (e.g. `virtual` methods) may be too much overhead for frequently called, short running functions.

- Particularly true for interpretation-based query execution.

Design Goals: (4) ...without Regret

Regret?

Abstraction through dynamic dispatch (e.g. `virtual` methods) may be too much overhead for frequently called, short running functions.

- Particularly true for interpretation-based query execution.

Code Generation to the Rescue

Design Goals: (4) …without Regret

Regret?

Abstraction through dynamic dispatch (e.g. `virtual` methods) may be too much overhead for frequently called, short running functions.

- Particularly true for interpretation-based query execution.

Code Generation to the Rescue

- Use `template` meta programming for compile-time composition.

Design Goals: (4) …without Regret

Regret?

Abstraction through dynamic dispatch (e.g. `virtual` methods) may be too much overhead for frequently called, short running functions.

- Particularly true for interpretation-based query execution.

Code Generation to the Rescue

- Use `template` meta programming for compile-time composition.
- Use code generation where the former is inapplicable.

Design Goals: (4) …without Regret

Regret?

Abstraction through dynamic dispatch (e.g. `virtual` methods) may be too much overhead for frequently called, short running functions.

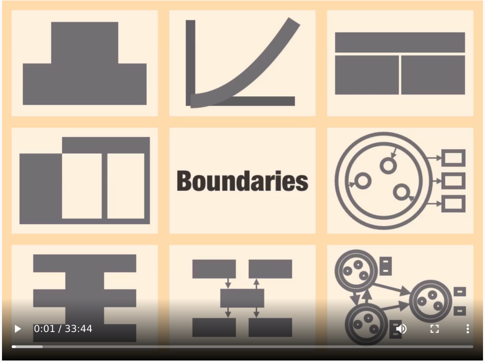
- Particularly true for interpretation-based query execution.

Code Generation to the Rescue

- Use `template` meta programming for compile-time composition.
- Use code generation where the former is inapplicable.
- Provide a deeply-embedded DSL, that mimics C, for easy adaption of code generation.

Achieving Extensibility & Separation of Concerns

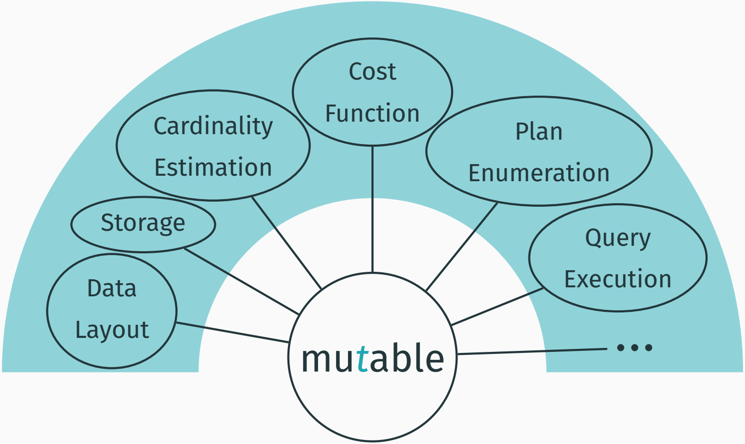
The Value is the Boundary



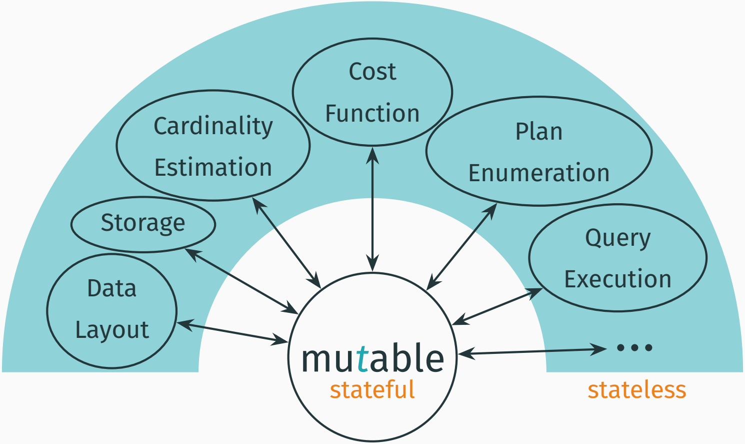
A talk by Gary Bernhardt from SCNA 2012[1]

[1] https://www.destroyallsoftware.com/talks/boundaries

The Value is the Boundary



The Value is the Boundary



SQL Query

```
... WHERE x > 42 ...
```

SQL Query

```
... WHERE x > 42 ...
```

Implementation of Branching Selection

```
IF (compile(this->condition())) {
    Pipeline();
};
```

SQL Query

```
... WHERE x > 42 ...
```

Implementation of Branching Selection

```
IF (compile(this->condition())) {
    Pipeline();
};
```

Generated WebAssembly Code

```
(br_if
  (i32.le_s          (; x <= 42 ;)
    (get_local $3)
    (i32.const 42)
  )
  (; Pipeline goes here ;)
)
```

Generated WebAssembly Code

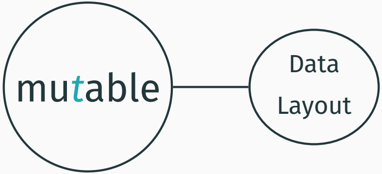
```
(br_if
  (i32.le_s         (; x <= 42 ;)
    (get_local $3)
    (i32.const 42)
  )
  (; Pipeline goes here ;)
)
```

Generated WebAssembly Code

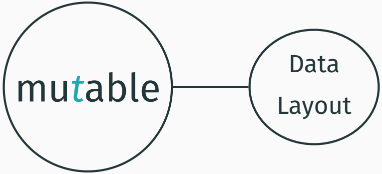
```
(br_if
  (i32.le_s              (; x <= 42 ;)
    (get_local $3)
    (i32.const 42)
  )
  (; Pipeline goes here ;)
)
```

- Google's JavaScript & WebAssembly engine
- Performs JIT compilation to x86
- Tiered compilation, adaptive execution

Components Overview: Data Layout

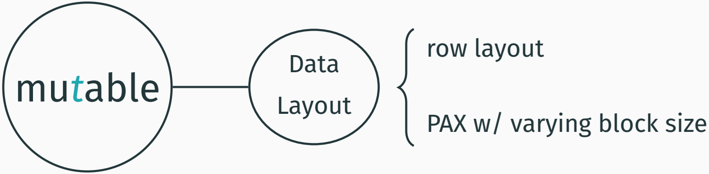


Components Overview: Data Layout



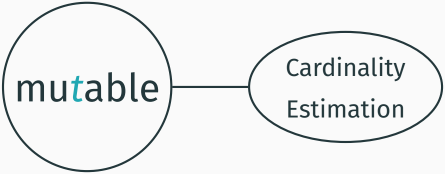
Generic framework to express arbitrary data layouts.

Components Overview: Data Layout

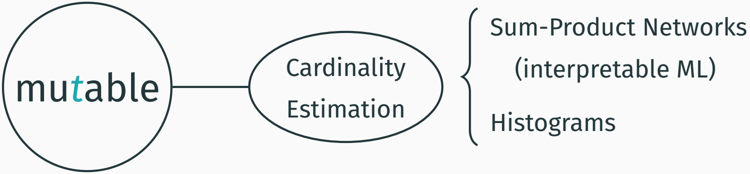


Generic framework to express arbitrary data layouts.

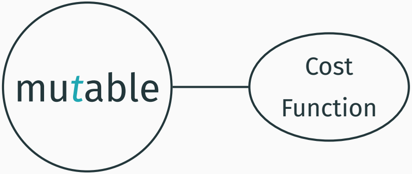
Components Overview: Cardinality Estimation



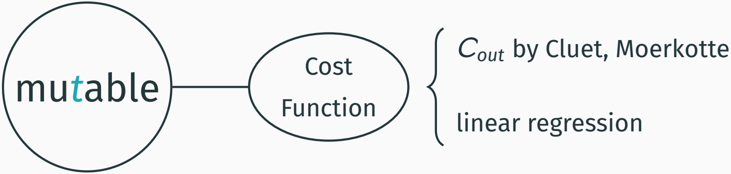
Components Overview: Cardinality Estimation



Components Overview: Cost Function

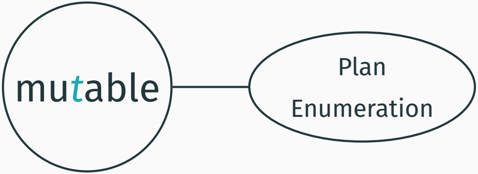


Components Overview: Cost Function

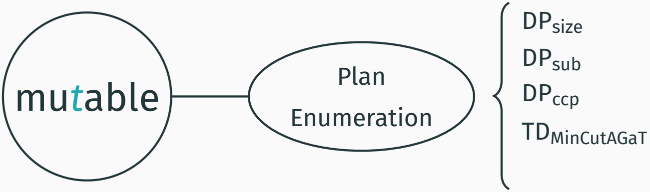


- C_{out} for logical / algebraic join ordering
- linear regression trained with automatic benchmarks on physical operators, used for selecting phys. operators

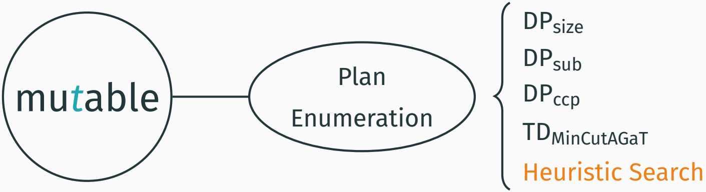
Components Overview: Plan Enumeration

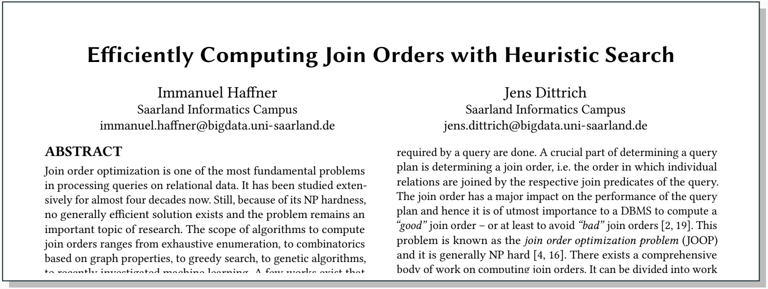


Components Overview: Plan Enumeration



Components Overview: Plan Enumeration





Efficiently Computing Join Orders with Heuristic Search

Immanuel Haffner
Saarland Informatics Campus
immanuel.haffner@bigdata.uni-saarland.de

Jens Dittrich
Saarland Informatics Campus
jens.dittrich@bigdata.uni-saarland.de

ABSTRACT

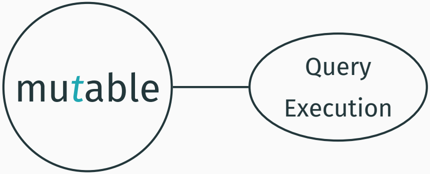
Join order optimization is one of the most fundamental problems in processing queries on relational data. It has been studied extensively for almost four decades now. Still, because of its NP hardness, no generally efficient solution exists and the problem remains an important topic of research. The scope of algorithms to compute join orders ranges from exhaustive enumeration, to combinatorics based on graph properties, to greedy search, to genetic algorithms, to recently investigated machine learning. A few works exist that

required by a query are done. A crucial part of determining a query plan is determining a join order, i.e. the order in which individual relations are joined by the respective join predicates of the query. The join order has a major impact on the performance of the query plan and hence it is of utmost importance to a DBMS to compute a *"good"* join order – or at least to avoid *"bad"* join orders [2, 19]. This problem is known as the *join order optimization problem* (JOOP) and it is generally NP hard [4, 16]. There exists a comprehensive body of work on computing join orders. It can be divided into work

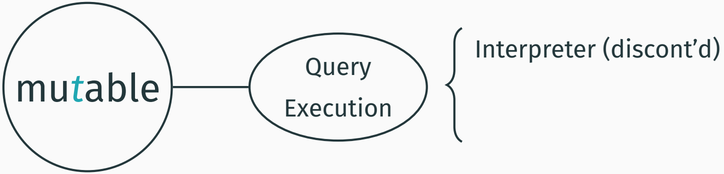
SIGMOD'23

Up to 1000x faster than state of the art (DP_{ccp})

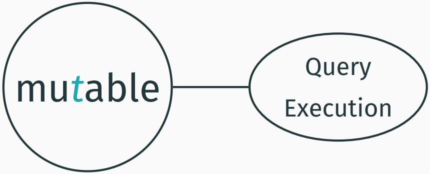
Components Overview: Query Execution



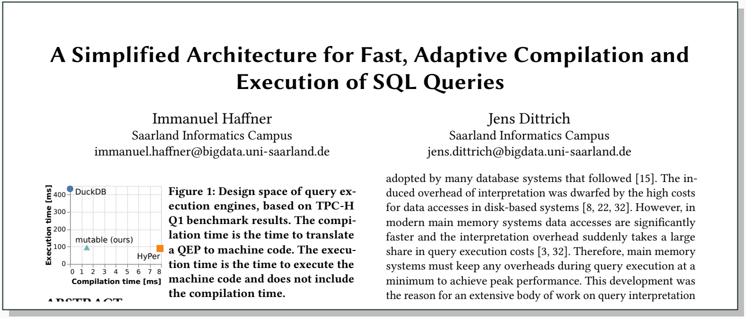
Components Overview: Query Execution



Components Overview: Query Execution



- Interpreter (discont'd)
- WebAssembly & V8



EDBT'23

A Simplified Architecture for Fast, Adaptive Compilation and Execution of SQL Queries

Immanuel Haffner
Saarland Informatics Campus
immanuel.haffner@bigdata.uni-saarland.de

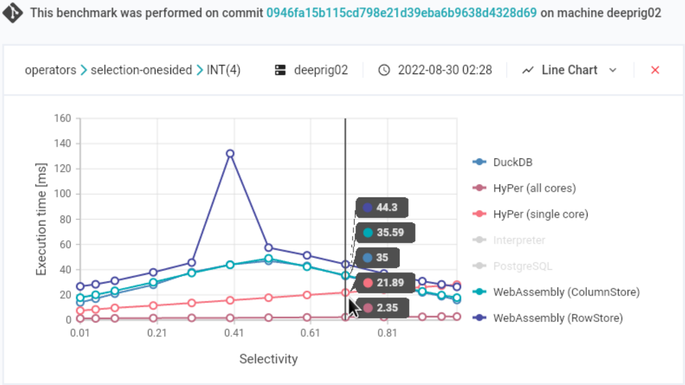
Jens Dittrich
Saarland Informatics Campus
jens.dittrich@bigdata.uni-saarland.de

Figure 1: Design space of query execution engines, based on TPC-H Q1 benchmark results. The compilation time is the time to translate a QEP to machine code. The execution time is the time to execute the machine code and does not include the compilation time.

adopted by many database systems that followed [15]. The induced overhead of interpretation was dwarfed by the high costs for data accesses in disk-based systems [8, 22, 32]. However, in modern main memory systems data accesses are significantly faster and the interpretation overhead suddenly takes a large share in query execution costs [3, 32]. Therefore, main memory systems must keep any overheads during query execution at a minimum to achieve peak performance. This development was the reason for an extensive body of work on query interpretation

JIT compilation, tiered compilation, and adaptive execution

Continuous Benchmarking



- automated benchmarking (nightly)
- automatic detection of performance anomalies

Questions?

github.com/mutable-org/mutable

Backup Slides

Components Overview

- Data Layouts:
  - generic framework, arbitrary layouts; implemented row, PAX with varying block size
- Cardinality Estimation:
  - Sum-Product Networks (interpretable ML), Histograms
- Cost Functions:
  - for algebraic/logical optimization: C_{out} by Cluet, Moerkotte
  - for physical optimization: linear regression trained on autom. benchmarks
- Plan Enumeration:
  - DP_{size}, DP_{sub}, DP_{ccp}, $\text{TD}_{\text{MinCutAGaT}}$
  - our Heuristic Search, published @SIGMOD'23 (up to 1000x faster than DP_{ccp})
- Query Execution:
  - query compilation to WebAssembly, tiered compilation & adaptive execution in Google's V8, published @EDBT'23 (similar Umbra's Tidy Tuples & Flying Start)

Upcoming Papers

EDBT'23

A Simplified Architecture for Fast, Adaptive Compilation and Execution of SQL Queries

Immanuel Haffner
Saarland Informatics Campus
immanuel.haffner@bigdata.uni-saarland.de

Jens Dittrich
Saarland Informatics Campus
jens.dittrich@bigdata.uni-saarland.de

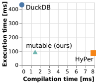


Figure 1: Design space of query execution engines, based on TPC-H Q1 benchmark results. The compilation time is the time to translate a QEP to machine code. The execution time is the time to execute the machine code and does not include the compilation time.

ABSTRACT

Query compilation is
In the past decade, we
this field, including c
ing from interpretatio
switching from non-
ideas aim to reduce lat
these approaches requ
erable part of which i
techniques from the c
ister allocation or mac
in this field for decad

In this paper, we ar
engines conceptually
the compiler constru
in the long run – we
tion techniques shoul
rather than being rein
choosing a suitable c
are able to get just-in-
range from non-opti
adaptive execution in
at runtime – for free!
piler construction com
benefit from future in
neering effort. We pr
WebAssembly and V8
this architecture as pa
provide an extensive

adopted by many database systems that followed [15]. The induced overhead of interpretation was dwarfed by the high costs for data accesses in disk-based systems [8, 22, 32]. However, in modern main memory systems data accesses are significantly faster and the interpretation overhead suddenly takes a large share in query execution costs [3, 32]. Therefore, main memory systems must keep any overheads during query execution at a minimum to achieve peak performance. This development was the reason for an extensive body of work on query interpretation

SIGMOD'23

Efficiently Computing Join Orders with Heuristic Search

Immanuel Haffner
Saarland Informatics Campus
immanuel.haffner@bigdata.uni-saarland.de

Jens Dittrich
Saarland Informatics Campus
jens.dittrich@bigdata.uni-saarland.de

ABSTRACT

Join order optimization is one of the most fundamental problems in processing queries on relational data. It has been studied extensively for almost four decades now. Still, because of its NP hardness, no generally efficient solution exists and the problem remains an important topic of research. The scope of algorithms to compute join orders ranges from exhaustive enumeration, to combinatorics based on graph properties, to greedy search, to genetic algorithms, to recently investigated machine learning. A few works exist that use heuristic search to compute join orders. However, a theoretical argument why and how heuristic search is applicable to join order optimization is lacking.

In this work, we investigate join order optimization via heuristic search. In particular, we provide a strong theoretical framework

required by a query are done. A crucial part of determining a query plan is determining a join order, i.e. the order in which individual relations are joined by the respective join predicates of the query. The join order has a major impact on the performance of the query plan and hence it is of utmost importance to a DBMS to compute a *"good"* join order – or at least to avoid *"bad"* join orders [2, 19]. This problem is known as the *join order optimization problem* (JOOP) and it is generally NP hard [4, 16]. There exists a comprehensive body of work on computing join orders. It can be divided into work on computing optimal join orders [4, 7, 12, 13, 16, 22, 32], work on greedy computation of potentially suboptimal join orders [10, 24, 25, 37], work on adaptive re-optimization of join orders [17, 26, 28, 38], and recent work based on machine learning [20, 21, 23].

Ono and Lohman [27] derive analytically the number of distinct