Do Programming Languages need Query Languages?

Jelle Hellings

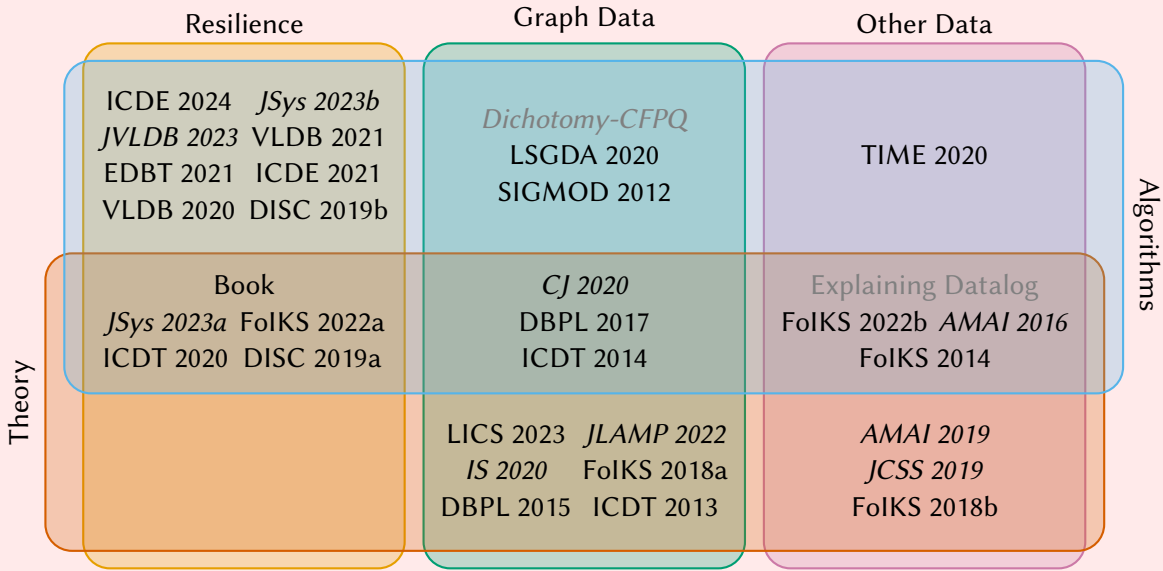
Department of Computing and Software
McMaster University
1280 Main St. W., Hamilton, ON L8S 4L7, Canada

Do Programming Languages need Query Languages?

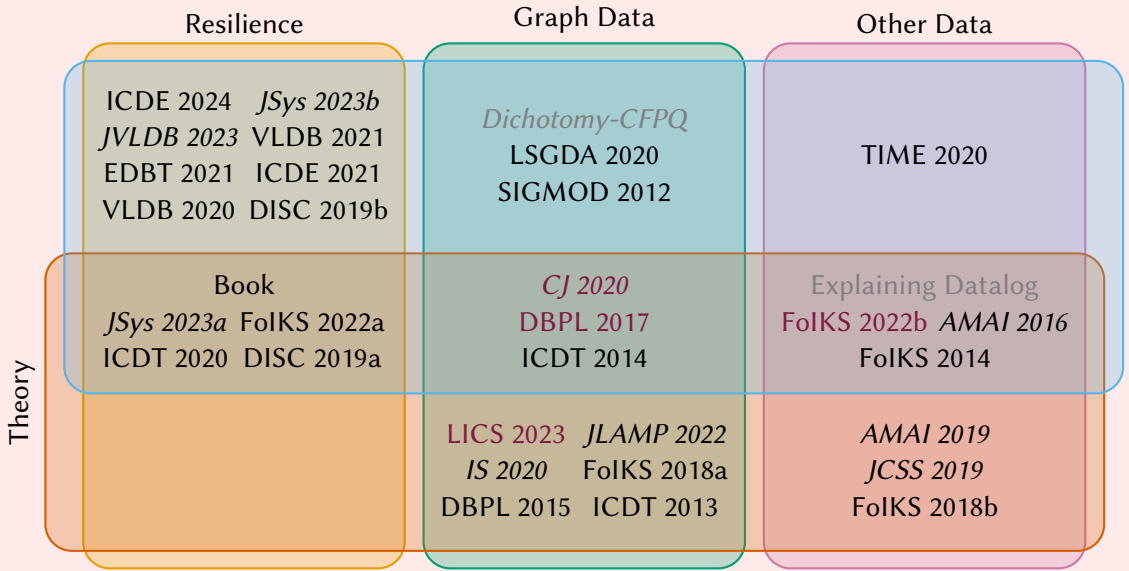
(spoiler alert)

Yes they do!

About my reasearch: Data and data management!



About my reasearch: Data and data management!



Programming and data processing

Claims

1. Data processing plays a central role in programming.

Programming and data processing

Claims

1. Data processing plays a central role in programming.

Examples: standard support libraries

1. *Collections* to store data
   binary search trees, hash tables, tuples, dynamic arrays, ...
2. *Algorithms* to operate on data
   sorting, filtering, transforming, set operations, ...

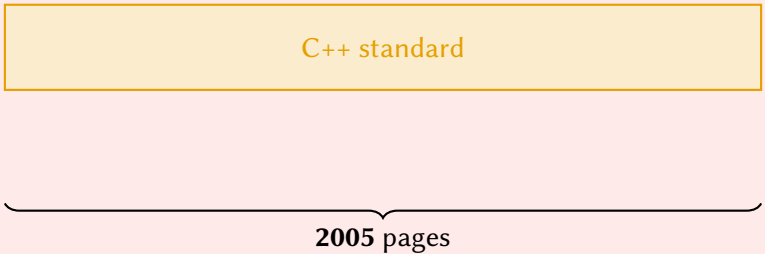
Programming and data processing

Claims

1. Data processing plays a central role in programming.

Working Draft of the C++ standard

(Document Number: N4964, Date 2023-10-15.)



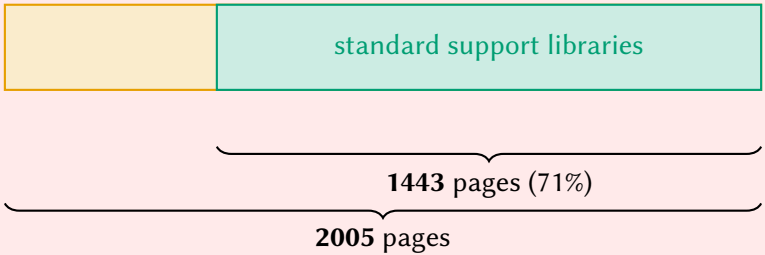
Programming and data processing

Claims

1. Data processing plays a central role in programming.

Working Draft of the C++ standard

(Document Number: N4964, Date 2023-10-15.)



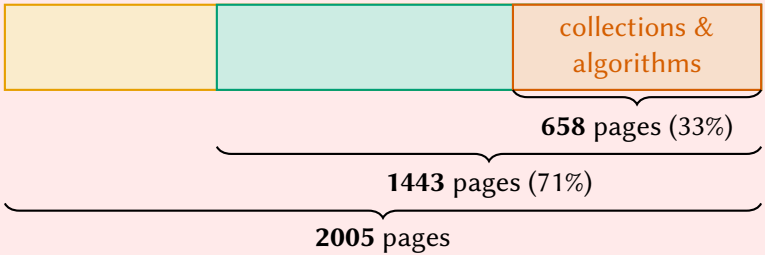
Programming and data processing

Claims

1. Data processing plays a central role in programming.

Working Draft of the C++ standard

(Document Number: N4964, Date 2023-10-15.)



Not counting: numeric, time, formatting, IO, threads, ...

An example: How to *program* relational algebra in C++

An example: How to *program* relational algebra in C++

Data storage: `std::tuple` (single row), `std::set` (rows).

An example: How to *program* relational algebra in C++

Data storage: `std::tuple` (single row), `std::set` (rows).

- **Projection** (π_{columns}): `std::transform`.
- **Selection** ($\sigma_{\text{conditions}}$): `std::copy_if`, `std::views::filter`.
- **Joins** ($\times$, $\bowtie$): loops, `std::views::cartesian_product`.
- **Set operations** ($\cap$, $\cup$, $\setminus$): `std::set_union`, ...

An example: How to *program* relational algebra in C++

Data storage: `std::tuple` (single row), `std::set` (rows).

- **Projection** (π_{columns}): `std::transform`.
- **Selection** ($\sigma_{\text{conditions}}$): `std::copy_if`, `std::views::filter`.
- **Joins** ($\times$, $\bowtie$): loops, `std::views::cartesian_product`.
- **Set operations** ($\cap$, $\cup$, $\setminus$): `std::set_union`, ...

- **Sorting**: `std::sort`, `std::stable_sort`.
- **Deduplication**: `std::unique_copy`.
- **Aggregation**: `std::accumulate` and `std::reduce`, `std::ranges::fold_left`.

An example: How to *program* relational algebra in C++

Data storage: `std::tuple` (single row), `std::set` (rows).

- **Projection** (π_{columns}): `std::transform`.
- **Selection** ($\sigma_{\text{conditions}}$): `std::copy_if`, `std::views::filter`.
- **Joins** ($\times$, $\bowtie$): loops, `std::views::cartesian_product`.
- **Set operations** ($\cap$, $\cup$, $\setminus$): `std::set_union`, ...

- **Sorting**: `std::sort`, `std::stable_sort`.
- **Deduplication**: `std::unique_copy`.
- **Aggregation**: `std::accumulate` and `std::reduce`, `std::ranges::fold_left`.

- **Parallelization**: execution policies (algorithms), `std::async` (evaluation), ...
- **Pipelined execution**: `std::ranges`, coroutines using `std::generator`, ...

An example: How to *program* relational algebra in C++

$$\pi_{R.\text{parent}}(\sigma_{R.\text{child}=S.\text{name}}(\rho_R(\text{parentOf}) \times \sigma_{S.\text{place}=\text{“Hamilton”}}(\rho_S(\text{person}))))$$

An example: How to *program* relational algebra in C++

$$\pi_{R.\text{parent}}(\sigma_{R.\text{child}=S.\text{name}}(\rho_R(\text{parentOf}) \times \sigma_{S.\text{place}=\text{“Hamilton”}}(\rho_S(\text{person}))))$$

```
using namespace std::views;

auto where_pred = [](auto l) { return l.place == "Hamilton"; };

auto product_pred = [](auto t) {
  auto [po, p] = t; return po.child == p.name; };

auto join = cartesian_product(parents, persons | filter(where_pred));
for (auto [po, p] : join | filter(product_pred)) {
    std::cout << po.parent << std::endl;
}
```

Programming and data processing

Claims

1. Data processing plays a central role in programming.
2. Programming languages are *bad* at data processing.

Programming and data processing

Claims

1. Data processing plays a central role in programming.
2. Programming languages are *bad* at data processing.

Efficient data processing

"*Complex*" data processing algorithms even for *simple* data processing tasks.
E.g., join algorithms, join ordering, index usage, selection push down, ...

This complexity is delegated to the programmer.

An example: A *more-efficient* query in C++

```
/* parents is a set, ordered on (parent, child).
 * persons is a set, ordered on (name, place). */

auto where_pred = [](auto l) { return l.place == "Hamilton"; };
auto filtered = persons | filter(where_pred)
                        | std::ranges::to<std::vector>();

for (auto& [pname, cname] : parents) {
  person_t p{cname, "Hamilton"};
  bool has_child = std::binary_search(filtered.begin(),
                                      filtered.end(), p);
  if (has_child) {
    std::cout << pname << std::endl;
  }
}
```

An example: A *more-efficient* query in C++

```
/* parents is a set, ordered on (parent, child).
 * persons is a set, ordered on (name, place). */

for (auto& [pname, cname] : parents) {
  person_t p{cname, "Hamilton"};
  bool has_child = persons.contains(p);
  if (has_child) {
    std::cout << pname << std::endl;
  }
}
```

An example: A *more-efficient* query in C++

	Runtime complexity	Memory usage
Original	$\Theta(\lvert\text{parents}\rvert \cdot \lvert\text{persons}\rvert)$	$\Theta(1)$
First variant	$\Theta(\lvert\text{parents}\rvert \log \lvert\text{filtered}\rvert + \lvert\text{persons}\rvert)$	$\Theta(\lvert\text{filtered}\rvert)$
Second variant	$\Theta(\lvert\text{parents}\rvert \log \lvert\text{persons}\rvert)$	$\Theta(1)$

Both “improved” versions can be significantly improved to reduce overheads!

An example: A *more-efficient* query in C++

	Runtime complexity	Memory usage
Original	$\Theta(\lvert\text{parents}\rvert \cdot \lvert\text{persons}\rvert)$	$\Theta(1)$
First variant	$\Theta(\lvert\text{parents}\rvert \log \lvert\text{filtered}\rvert + \lvert\text{persons}\rvert)$	$\Theta(\lvert\text{filtered}\rvert)$
Second variant	$\Theta(\lvert\text{parents}\rvert \log \lvert\text{persons}\rvert)$	$\Theta(1)$

Both “improved” versions can be significantly improved to reduce overheads!

What if we want *unique* parents?

An example: A *more-efficient* query in SQL

```
SELECT pname
FROM parents R, persons S
WHERE R.child = S.name AND S.place = "Hamilton";
```

An example: A *more-efficient* query in SQL

```
SELECT DISTINCT pname
FROM parents R, persons S
WHERE R.child = S.name AND S.place = "Hamilton";
```

An example: A *more-efficient* query in SQL

```
SELECT DISTINCT pname
FROM parents R, persons S
WHERE R.child = S.name AND S.place = "Hamilton";
```

Some database systems need a helping hand…

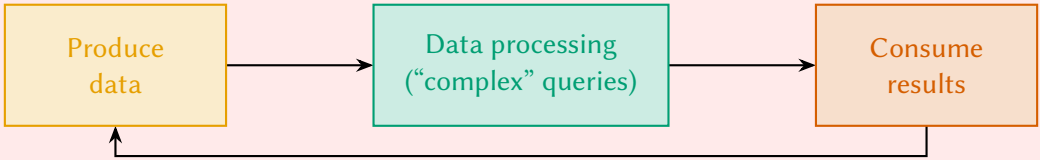
```
SELECT DISTINCT pname
FROM parents
WHERE child IN (SELECT name
                FROM persons
                WHERE place = "Hamilton");
```

Programming and data processing

Claims

1. Data processing plays a central role in programming.
2. Programming languages are *bad* at data processing.
3. We need database technology in our programming languages.

Potential solution?

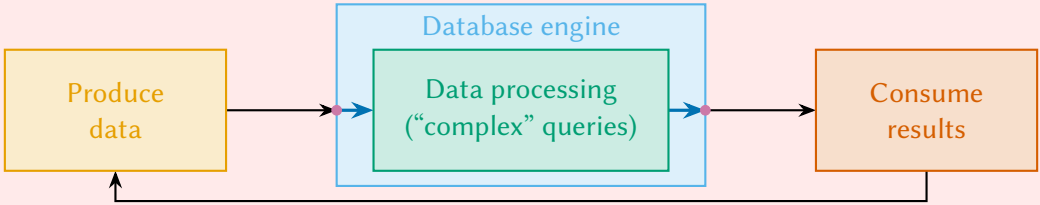


Programming and data processing

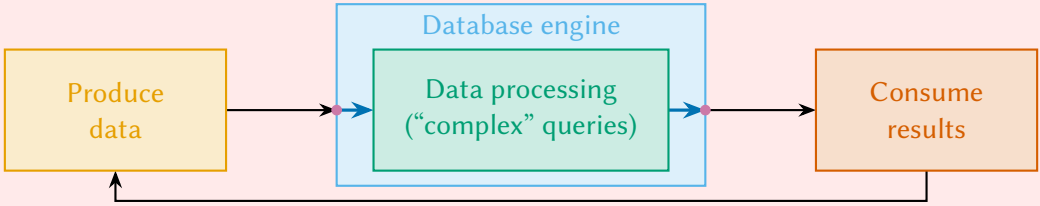
Claims

1. Data processing plays a central role in programming.
2. Programming languages are *bad* at data processing.
3. We need database technology in our programming languages.

Potential solution?



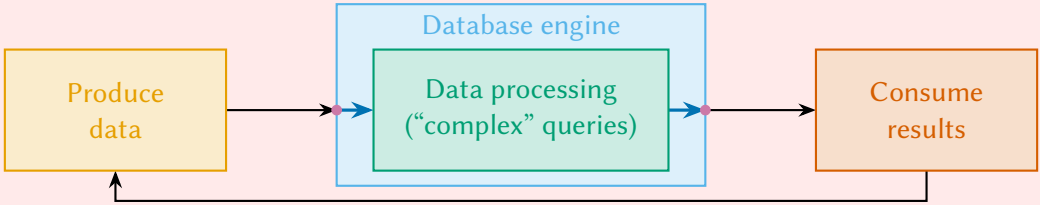
Database systems are *not* the solution



What if we use a database system

For example, PostgreSQL (is free!)

Database systems are *not* the solution



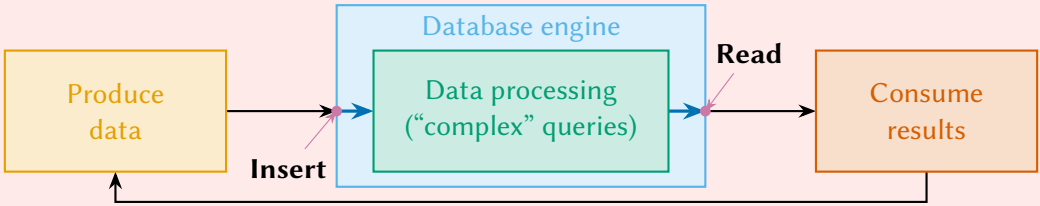
What if we use a database system

For example, PostgreSQL (is free!)

Not all data is always in a database system.

- There might not be a database system.

Database systems are *not* the solution



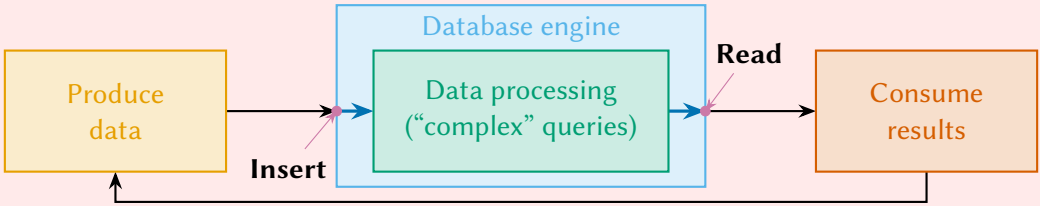
What if we use a database system

For example, PostgreSQL (is free!)

Not all data is always in a database system.

- There might not be a database system.
- Inserting data into and reading data from the system is *not free*:
  Data transfers, necessary code to "translate" between types, ...

Database systems are *not* the solution



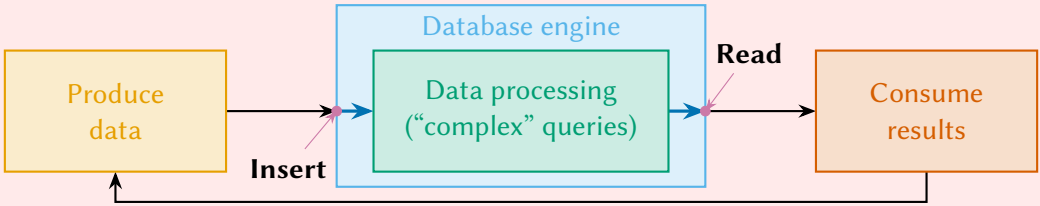
What if we use a database system

For example, PostgreSQL (is free!)

Not all data is always in a database system.

- There might not be a database system.
- Inserting data into and reading data from the system is *not free*:
  Data transfers, necessary code to "translate" between types, ...
- Conceptual mismatches: database code does not mix well with other code!
  For example, type safety (program) versus "no" type safety (query results).

Database systems are *not* the solution



What if we use a database system

For example, ~~PostgreSQL~~ SQLLite (is free!)

Not all data is always in a database system.

- ~~There might not be a database system~~.
- Inserting data into and reading data from the system is *not free*:
  Data transfers, necessary code to "translate" between types, ...
- Conceptual mismatches: database code does not mix well with other code!
  For example, type safety (program) versus "no" type safety (query results).

Proposed solution

What if…

```
#include "my_fancy_library.hpp"
          ⋮
query<"%(parent) :- parents(parent, c),"
      "             persons(c, 'Hamilton')">(dataset);
```

is valid C++ code in which **query** generates an *optimized* C++ algorithm (at compile time).

Proposed solution

What if…

```
#include "my_fancy_library.hpp"
          ⋮
query<"%(parent) :- parents(parent, c),"
      "             persons(c, 'Hamilton')">(dataset);
```

is valid C++ code in which **query** generates an *optimized* C++ algorithm (at compile time).

How to do so

Create a C++ library that

- Provides a *domain specific query language* for C++.
- At C++ compile time, the library *compiles these queries* into C++ algorithms: derive result types, derive query evaluation strategy, ….
- At runtime: these query algorithms are *normal C++ functions*.

Proposed solution: Detailed plans

Proposed solution: Detailed plans

1. Design the *domain specific query language*:
   Target: multiset aware Datalog with stratified negation and aggregation.

Proposed solution: Detailed plans

1. Design the *domain specific query language*:
   Target: multiset aware Datalog with stratified negation and aggregation.

4. At runtime: *execute* query plans using *efficient query algorithms*.
   With zero overhead: the *query plan representation is* the algorithm.

Proposed solution: Detailed plans

1. Design the *domain specific query language*:
   Target: multiset aware Datalog with stratified negation and aggregation.

2. A *compile-time parser* for the query language:
   Result: the Datalog program in some *internal representation*.

4. At runtime: *execute* query plans using *efficient query algorithms*.
   With zero overhead: the *query plan representation is* the algorithm.

Proposed solution: Detailed plans

1. Design the *domain specific query language*:
   Target: multiset aware Datalog with stratified negation and aggregation.

2. A *compile-time parser* for the query language:
   Result: the Datalog program in some *internal representation*.

3. A compile-time *query optimizer* and *query plan* compiler:
   Goal: turn the *internal representation* into an efficient query plan.

4. At runtime: *execute* query plans using *efficient query algorithms*.
   With zero overhead: the *query plan representation is* the algorithm.

C++ compile-time support for *domain specific languages*

Almost feature complete (optimistic: publication in 2024?).

C++ compile-time support for *domain specific languages*

What we have

- A high-level grammar language based on *parsing expression grammars*:
  that controls scanning, parsing, and syntax tree generation.
- A C++ compile-time parser generator that turns grammars into compile-time parsers.
- Utilities to transform syntax trees into complex types
  (as part of new compile-time compilers).
- Runtime parser generator support and debugging utilities.

C++ compile-time support for *domain specific languages*

What is still being worked on

- Full integration of optimizations in the generated parsers.
- Memorization support for the runtime parser generator.
- Evaluation.

A compile-time *query optimizer* and *query plan* compiler

We can use a database-style query planner and query optimizer.
A compile time environment is different, however.

Zero-cost principle

Our *abstraction* of a domain specific query language should not introduce costs (when compared to a handwritten algorithm).

A compile-time *query optimizer* and *query plan* compiler

We can use a database-style query planner and query optimizer.
A compile time environment is different, however.

Zero-cost principle

Our *abstraction* of a domain specific query language should not introduce costs (when compared to a handwritten algorithm).

- *Only-once* query optimization (one compile time, many executions).
- Query optimization *without* access to the data.
- At runtime: algorithms can make choices (e.g., `if-else`),
  But there is only limited information: C++ containers do not have heuristics.

A compile-time *query optimizer* and *query plan* compiler

A blank-slate design

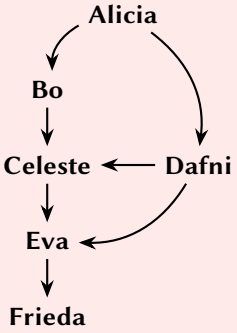
Only-once query optimization: we can use *costly yet effective* optimization techniques!

Inspiration: Automated semi-join rewritings

- In [DBPL 2017, CJ 2020], we showed that
  - almost all node queries expressed in terms of graph navigation can be expressed *without joins* and *without transitive closure*;
  - provided an algorithm to apply such optimizations on (parts) of graph queries.
- In [FoIKS 2022b], we showed that
  - these results *generalize to SQL*: weak-equivalent rewrite rules.
- In [LICS 2023], we showed that
  - *all first-order node queries on trees* are expressible in semi-join-based languages.

Inspiration: Graph query evaluation and SQL [FoIKS 2022b]

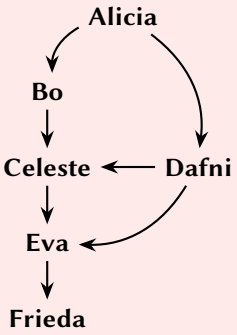
edges	
nfrom	nto



$\pi_1[Edges \circ Edges \circ Edges \circ Edges]$.

Inspiration: Graph query evaluation and SQL [FoIKS 2022b]

edges	
nfrom	nto

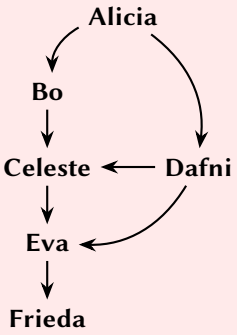


$$\pi_1[Edges \circ Edges \circ Edges \circ Edges].$$

```
SELECT DISTINCT S.nfrom
FROM edges S, edges R, edges T, edges U
WHERE S.nto = R.nfrom AND
      R.nto = T.nfrom AND
      T.nto = U.nfrom;
```

Inspiration: Graph query evaluation and SQL [FoIKS 2022b]

edges	
nfrom	nto



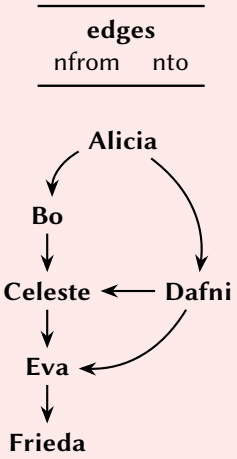
$$\pi_1[Edges \circ Edges \circ Edges \circ Edges].$$

```
SELECT DISTINCT S.nfrom
FROM edges S, edges R, edges T, edges U
WHERE S.nto = R.nfrom AND
      R.nto = T.nfrom AND
      T.nto = U.nfrom;
```

Cost of query plan

Depending on system $\pm\mathcal{O}(|\text{edges}|^3)$–$\mathcal{O}(|\text{edges}|^4)$:
Queries do not complete.

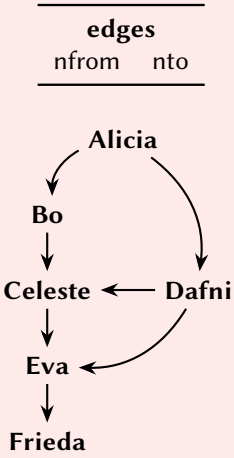
Inspiration: Graph query evaluation and SQL [FoIKS 2022b]



$$\pi_1[Edges \ltimes (Edges \ltimes (Edges \ltimes Edges))].$$

```
SELECT DISTINCT nfrom FROM edges
WHERE nto IN (
  SELECT nfrom FROM edges
  WHERE nto IN (
    SELECT nfrom FROM edges
    WHERE nto IN (
      SELECT nfrom FROM edges)));
```

Inspiration: Graph query evaluation and SQL [FoIKS 2022b]



$$\pi_1[Edges \ltimes (Edges \ltimes (Edges \ltimes Edges))].$$

```
SELECT DISTINCT nfrom FROM edges
WHERE nto IN (
  SELECT nfrom FROM edges
  WHERE nto IN (
    SELECT nfrom FROM edges
    WHERE nto IN (
      SELECT nfrom FROM edges)));
```

Cost of query plan

$\pm\mathcal{O}(|\text{edges}|)$: 90 ms with 75,000 edges.

Inspiration: An SQL-based example [FoIKS 2022b]

Original query

```
SELECT DISTINCT C.cname, P.type
FROM customer C, bought B, product P
WHERE C.cname = B.cname AND B.pname = P.pname AND
        P.type = 'food';
```

Inspiration: An SQL-based example [FoIKS 2022b]

Original query

```
SELECT DISTINCT C.cname, P.type
FROM customer C, bought B, product P
WHERE C.cname = B.cname AND B.pname = P.pname AND
        P.type = 'food';
```

Rewritten query

```
SELECT cname, 'food' AS type
FROM customer WHERE cname IN (
    SELECT cname FROM bought WHERE pname IN (
        SELECT pname FROM product WHERE type = 'food'));
```

Inspiration: An SQL-based example [FoIKS 2022b]

Original query

```
SELECT DISTINCT C.cname, P.type
FROM customer C, bought B, product P
WHERE C.cname = B.cname AND B.pname = P.pname AND
        P.type = 'food';
```

Rewritten query

```
SELECT cname, 'food' AS type
FROM customer WHERE cname IN (
    SELECT cname FROM bought WHERE pname IN (
        SELECT pname FROM product WHERE type = 'food'));
```

Rewrite result

Performance gain of at-least 15%.

(Instance with 500 customers, 24 077 products, and 100 000 records in *Bought*).

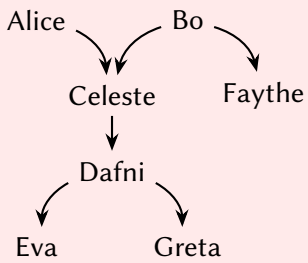
At runtime: *Execute* query plans using *efficient query algorithms*

Central idea

Represent the query plan by a complex type:
use standard *C++ template meta-programming* to generate data structures and algorithms.

Demo

- *person*(id, name) assigns names to nodes.
- *parentOf*(parent, child) relates persons.



Proposed solution: Challenges

Ease-of-use: Writing code

Compiler outputs of megabytes is not acceptable for a single typo!

Proposed solution: Challenges

Ease-of-use: Writing code

Compiler outputs of megabytes is not acceptable for a single typo!
Partial solution. `static_asserts` with sensible error messages in code.

Proposed solution: Challenges

Ease-of-use: Writing code

Compiler outputs of megabytes is not acceptable for a single typo!
Partial solution. `static_asserts` with sensible error messages in code.

Integration with existing code-bases

```
std::set<my_fancy_type, my_fancy_ordering> my_fancy_dataset;
```

- Which fields does `my_fancy_type` define?
- What ordering does `my_fancy_ordering` provide? Keys? Index?

Proposed solution: Challenges

Ease-of-use: Writing code

Compiler outputs of megabytes is not acceptable for a single typo!
Partial solution. `static_asserts` with sensible error messages in code.

Integration with existing code-bases

```
std::set<my_fancy_type, my_fancy_ordering> my_fancy_dataset;
```

- Which fields does `my_fancy_type` define?
- What ordering does `my_fancy_ordering` provide? Keys? Index?

Solution. Sensible defaults where possible, else provide such information via *traits classes.*

Proposed solution: Challenges

Ease-of-use: Writing code

Compiler outputs of megabytes is not acceptable for a single typo!
Partial solution. `static_asserts` with sensible error messages in code.

Integration with existing code-bases

```
std::set<my_fancy_type, my_fancy_ordering> my_fancy_dataset;
```

- Which fields does `my_fancy_type` define?
- What ordering does `my_fancy_ordering` provide? Keys? Index?

Solution. Sensible defaults where possible, else provide such information via *traits classes.*

Reflection (potential future C++ standards) might partly help.

Proposed solution: Challenges

Source: https://xkcd.com/303/ by Randall Munroe.

Evaluation: Two perspectives

First perspective: How do we compare with existing database products.

Evaluation: Two perspectives

First perspective: How do we compare with existing database products.

Second perspective: How do we compare with (C++) *programmers*?

- Performance?
- Readability?
- Ease-of-use?

Evaluation: Two perspectives

First perspective: How do we compare with existing database products.

Second perspective: How do we compare with (C++) *programmers*?

- Performance?
- Readability?
- Ease-of-use?

Central question

Should we switch to *declarative languages* even in traditional *procedural* source code?

Research volunteers needed

To evaluate our research, we will compare with data processing programs written in C++.

You can provide these C++ programs via an online questionnaire.

- Letter of Information https://jhellings.nl/q/loi.pdf.
- Questionnaire at https://jhellings.nl/q/.

The questionnaire takes 30min–120min and participation is entirely voluntarily.

This study has been reviewed by and received ethics clearance from the *McMaster Research Ethics Board* (#6885).

Conclusion

Programming Languages do need Query Languages!

References at https://jhellings.nl/.

Questionnaire at https://jhellings.nl/q/.