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!

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, ...

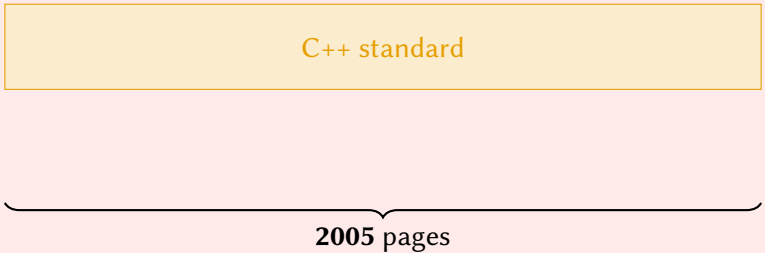
Programming and data processing

Claims

1. Data processing plays a central role in programming.

Working Draft of the C++ standard

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



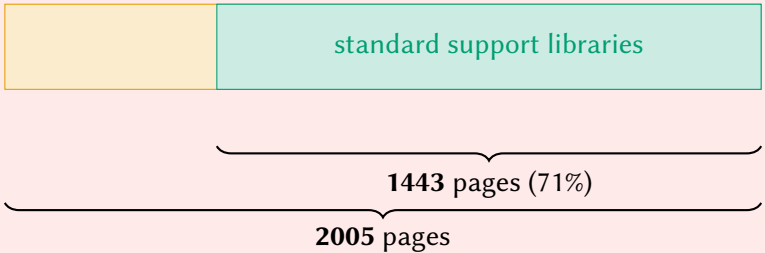
Programming and data processing

Claims

1. Data processing plays a central role in programming.

Working Draft of the C++ standard

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



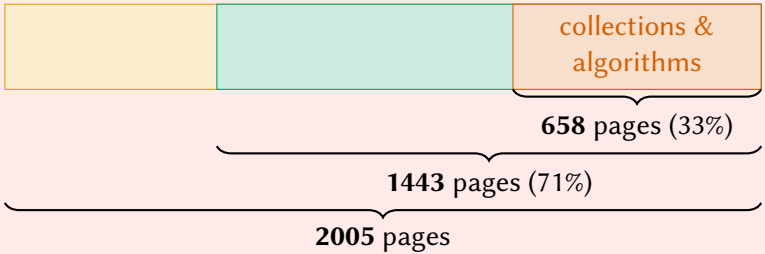
Programming and data processing

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), name is key. */

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 pcname, "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), name is key. */

for (auto& [pname, cname] : parents) {
  person_t pcname, "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");
```