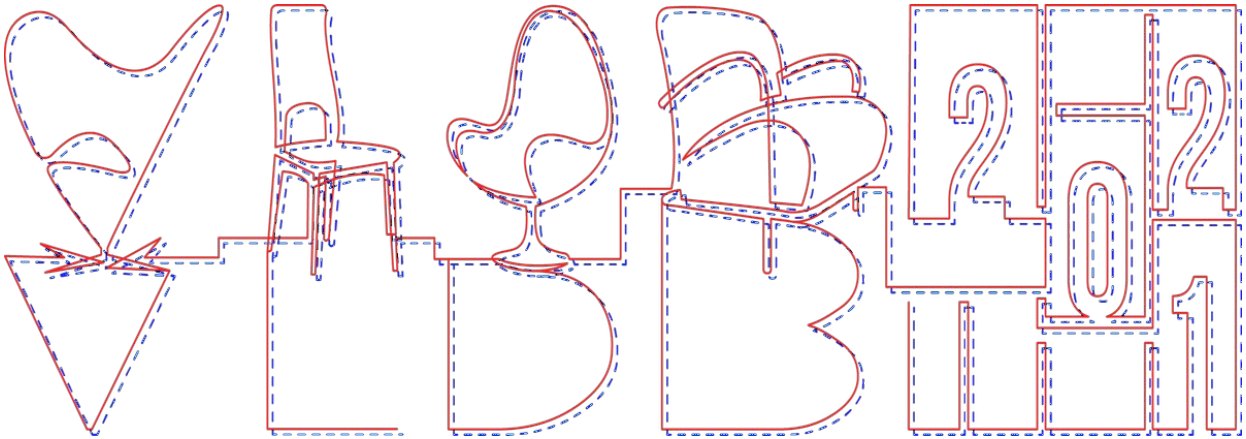


Language-Agnostic Integrated Queries in a Managed Polyglot Runtime

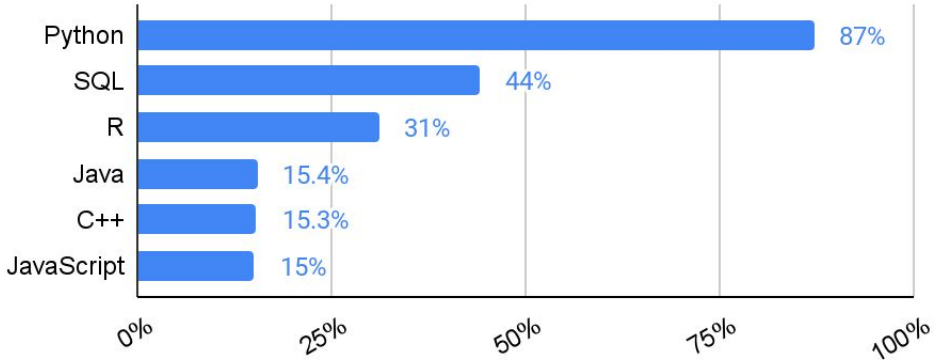
Filippo Schiavio
USI Lugano

Daniele Bonetta
Oracle Labs

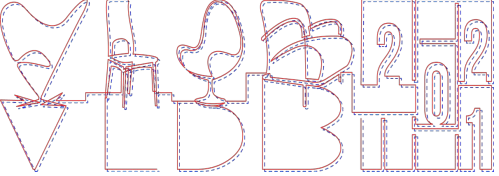
Walter Binder
USI Lugano

Data Analytics is “in-the-language”

- Modern data processing & analytics are often implemented “in the language”
- Adoption of ad-hoc solutions, avoiding “external” runtime systems (RDBMs)
- Data analytics are commonly implemented in dynamic languages [1]

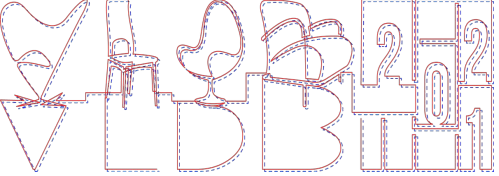


[1] https://www.kaggle.com/kaggle-survey-2019



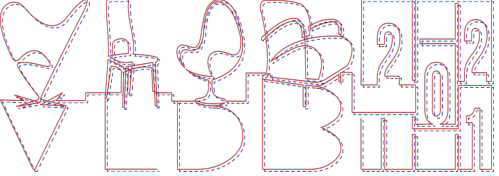
DataFrames & Embedded DB

- DataFrames  
  - Often show suboptimal performance
  - Tabular data structure with typed columns
- Embedded DB 
  - Often they require defining a schema in advance and a data ingestion phase
  - Emerging embedded DBs allow in-situ query execution 
- Limited support for querying arbitrary heterogeneous data structures in-situ



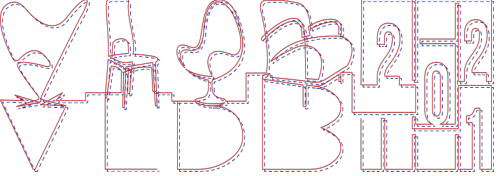
Language Integrated Query

- Language Integrated Query (LINQ) Frameworks
  - Flexible solution which integrates all language features in the query engine
  - In-situ query processing on any iterable, i.e., directly on the heap of the runtime
  - State-of-the-art optimizations leverage static ahead-of-time (AOT) query compilation
- Challenge: compiling queries in dynamic languages
  - How to compile “SELECT COUNT(*) FROM T WHERE x < y” with T collection of dynamic objects?
  - The implementation of the less-than operator depends on the runtime types of x, y



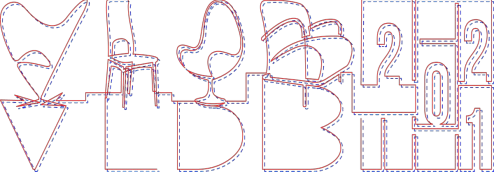
Polyglot Query Engine

- Challenge: compiling queries in **multiple** dynamic languages
- Dynamic languages have different syntax and semantics but share a set of abstract operations
  - E.g., property reads, array accesses, function calls
- A query engine for dynamically typed collections can be designed by abstracting the semantics of a specific language
  - Implementation should focus on query operators and engine optimizations
  - Decoupling them from (language-dependent) data-access operations



Dynamic Compilation to the Rescue!

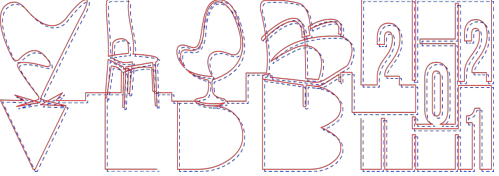
- Query compilers commonly rely on static (AOT) compilation
  - Dynamic languages are not considered suitable for AOT compilation
  - At compilation time the compiler is not aware of runtime types
  - Generated code needs to consider all possible types
- Modern dynamic-language runtimes employ dynamic (JIT) compilation
  - Program execution starts in “interpreted mode”, while the VM collects runtime information (e.g., observed types)
  - Program compilation takes place during interpreted execution, so the compiler is aware observed types
- JIT compilers of VMs can be used to compile SQL queries, not just “regular” user code



DynQ Overview

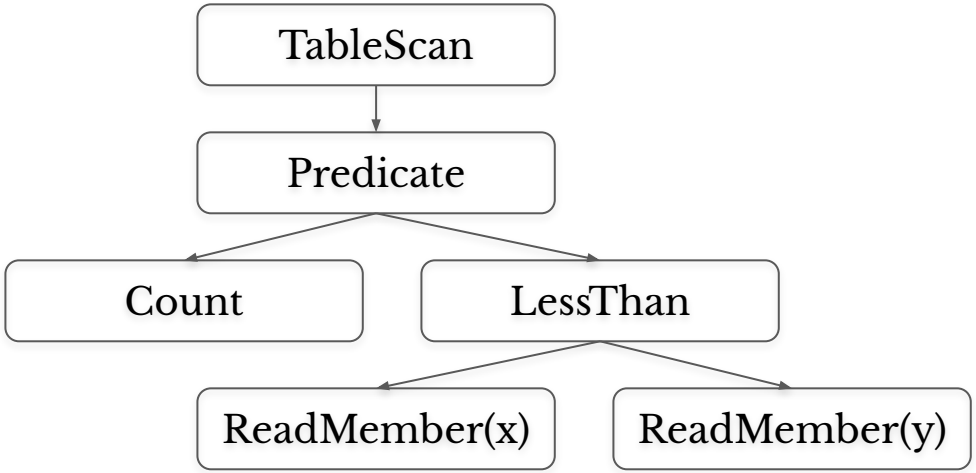
- DynQ is a language-agnostic query-engine integrated in GraalVM
- Simply imported as a library from any language executed on GraalVM
- Implements the backend of a SQL query compiler
  - Built on Truffle language implementation framework [1]
  - Directly interacts with underlying JIT compiler (Graal)
  - Exploits dynamic compilation optimizations (e.g., polymorphic inline caching and loop unrolling)
  - Frontend of the compiler could be any SQL planner (currently Apache Calcite)

[1] T. Würthinger et al. Practical partial evaluation for high-performance dynamic language runtimes. PLDI (2017)



Query on a JavaScript Array

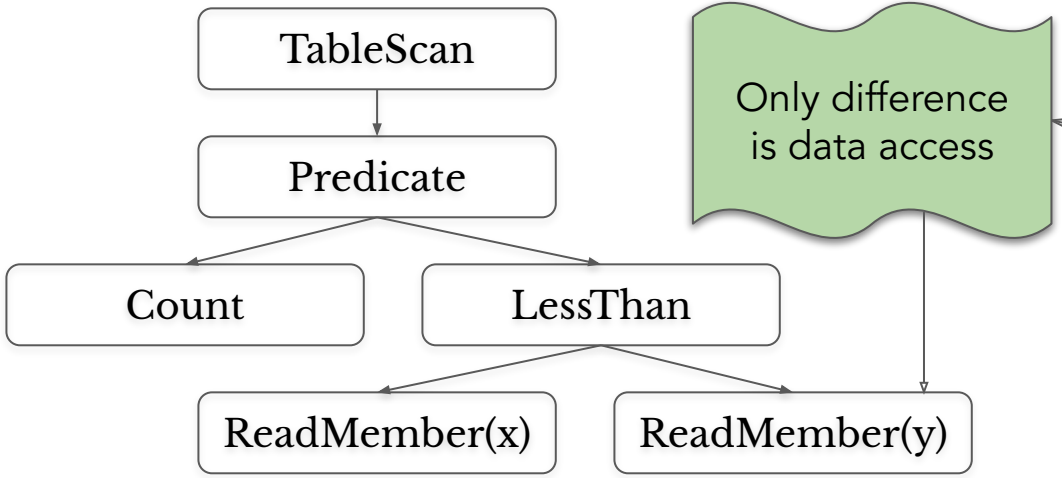
```
T = [{x: 1, y: 2}, {x: 2, y: 1}, … ]; // assuming x, y are integers for all items
result = DynQ('SELECT COUNT(*) FROM T WHERE x < y',T);
```




```
executeMethodAfterJITCompilation() {
  result = 0;
  for(i = 0; i < numElements; i++) {
    row = // read i-th array element
    x = // read property "x" of row
    y = // read property "y" of row
    // Type checking for LessThan
    if(/* x and y are integers */) {
      if(x < y) result++;
    }
    else {
      deoptimize();
    }
  }
  return result;
}
```

Query on a ~~JavaScript~~ Array Python

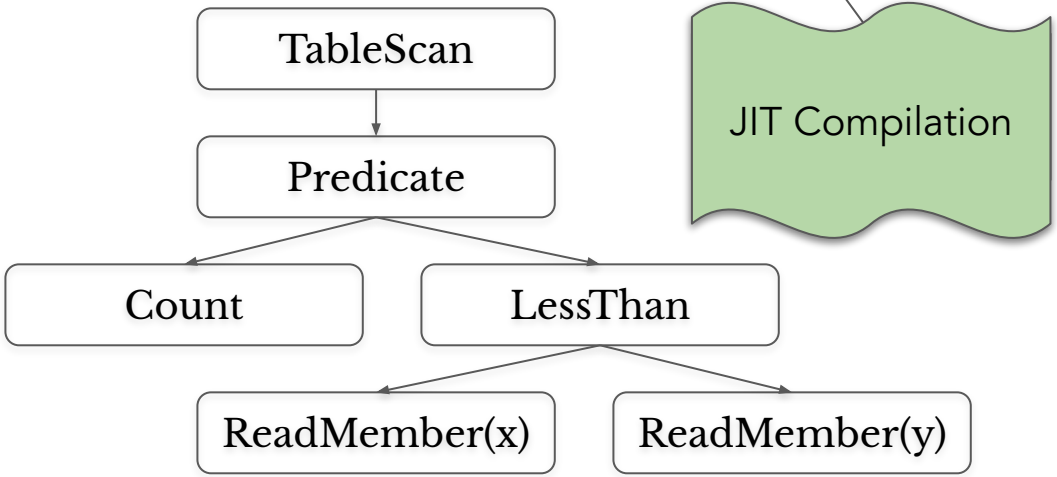
```
T = [{x: 1, y: 2}, {x: 2, y: 1}, … ]; // assuming x, y are integers for all items
result = DynQ('SELECT COUNT(*) FROM T WHERE x < y',T);
```




```
executeMethodAfterJITCompilation() {
  result = 0;
  for(i = 0; i < numElements; i++) {
    row = // read i-th array element
    x = // read property "x" of row
    y = // read property "y" of row
    // Type checking for LessThan
    if(/* x and y are integers */) {
      if(x < y) result++;
    }
    else {
      deoptimize();
    }
  }
  return result;
}
```

Query on a Polymorphic Array

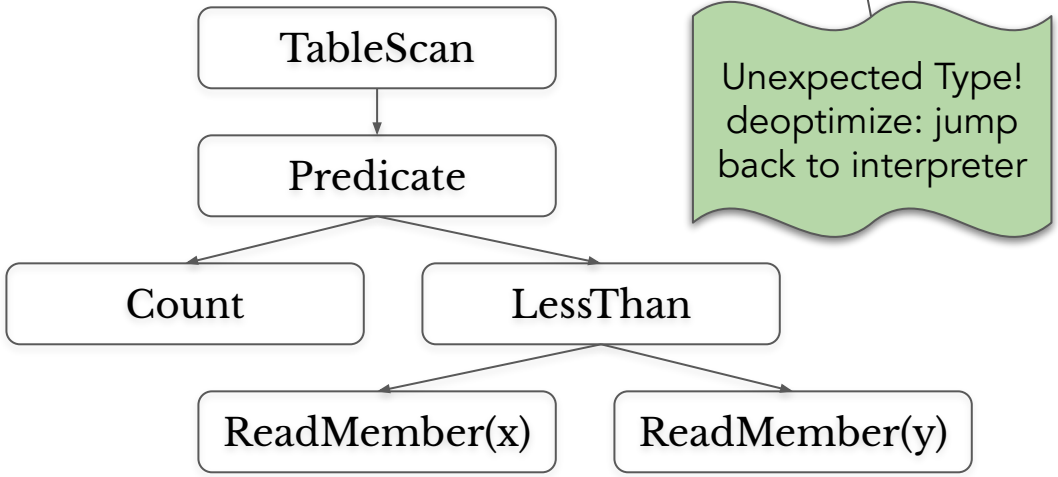
```
T = [{x: 1, y: 2}, {x: 2, y: 1}, … , {x: Date('2000-01-01'), y: Date('2000-01-02')}];
result = DynQ('SELECT COUNT(*) FROM T WHERE x < y',T);
```




```
executeMethodAfterJITCompilation() {
  result = 0;
  for(i = 0; i < numElements; i++) {
    row = // read i-th array element
    x = // read property "x" of row
    y = // read property "y" of row
    // Type checking for LessThan
    if(/* x and y are integers */) {
      if(x < y) result++;
    }
    else {
      deoptimize();
    }
  }
  return result;
}
```

Query on a Polymorphic Array

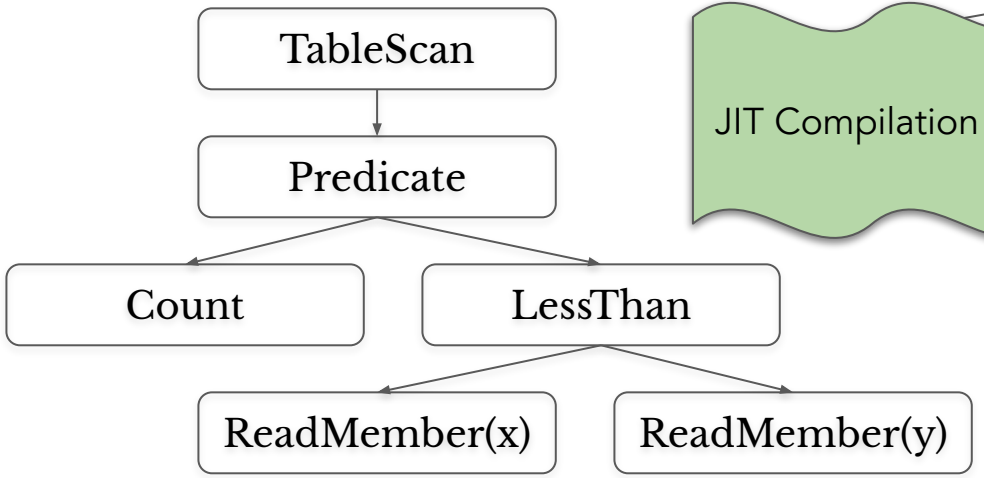
```
T = [{x: 1, y: 2}, {x: 2, y: 1}, … , {x: Date('2000-01-01'), y: Date('2000-01-02')},…];
result = DynQ('SELECT COUNT(*) FROM T WHERE x < y',T);
```




```
executeMethodAfterJITCompilation() {
  result = 0;
  for(i = 0; i < numElements; i++) {
    row = // read i-th array element
    x = // read property "x" of row
    y = // read property "y" of row
    // Type checking for LessThan
    if(/* x and y are integers */) {
      if(x < y) result++;
    }
    else {
      deoptimize();
    }
  }
  return result;
}
```

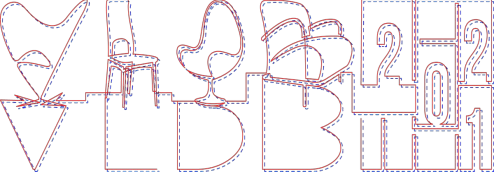
Query on a Polymorphic Array

```
T = [{x: 1, y: 2}, {x: 2, y: 1}, … , {x: Date('2000-01-01'), y: Date('2000-01-02')},…];
result = DynQ('SELECT COUNT(*) FROM T WHERE x < y',T);
```



JIT Compilation (2)

```
executeMethodAfterJITCompilation() {
  result = 0;
  for(i = 0; i < numElements; i++) {
    row = // read i-th array element
    x = // read property "x" of row
    y = // read property "y" of row
    // Type checking for LessThan
    if(/* x and y are integers */) {
      if(x < y) result++;
    }
    else if(/* x and y are dates */) {
      if(x.isBefore(y)) result++;
    } else { deoptimize(); }
  }
  return result;
}
```



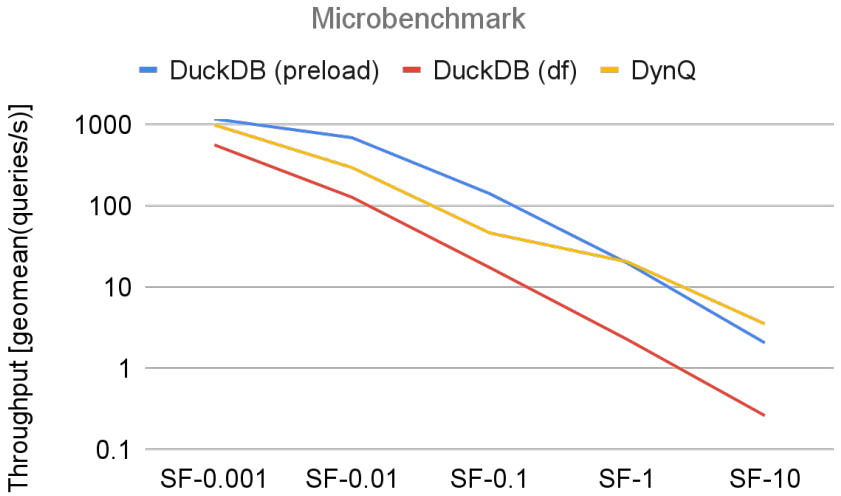
DynQ Evaluation

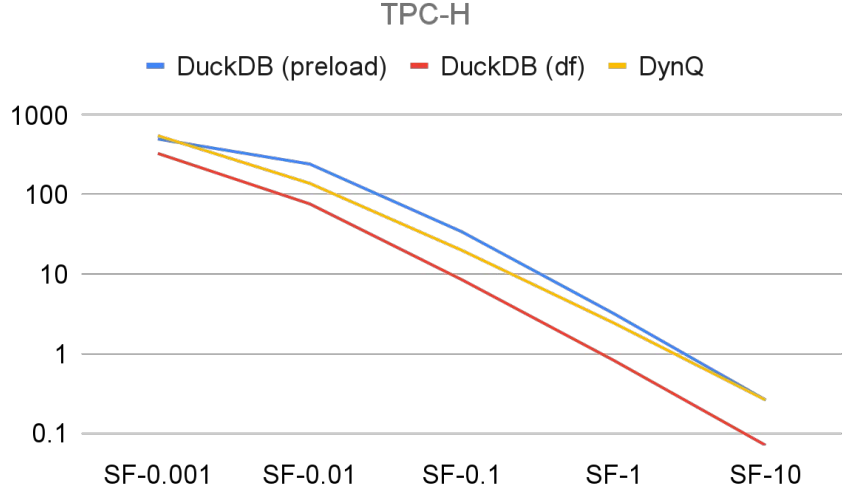
- Evaluation on multiple programming languages and different settings
- Realistic benchmark (TPC-H)
- Micro benchmark - simple queries on TPC-H dataset - from stream-fusion [1]
- Evaluation on RLang against
  - DuckDB 0.2.0 R package [2] (df / preload)
  - data.table R package [3] (see paper)
- Evaluation on JS against
  - AfterBurnerDB [4]
  - Lodash library [5] (see paper)

[1] S. Amir, M. Dashti, C. Koch. Push versus pull-based loop fusion in query engines. Journal of Functional Programming (2018)
[2] H. Mühleisen, M. Raasveldt and DuckDB Contributors. DuckDB: DBI Package for the DuckDB Database Management System (2021)
[3] M. Dowle, A. Srinivasan. data.table: Extension of data.frame. CRAN.R (2021)
[4] E.G. Kareem, J. Lin. In-browser Interactive SQL Analytics with AfterBurner. SIGMOD (2017)
[5] Lodash Team. Lodash. https://lodash.com/ (2020)

DynQ Evaluation (R - Latency)

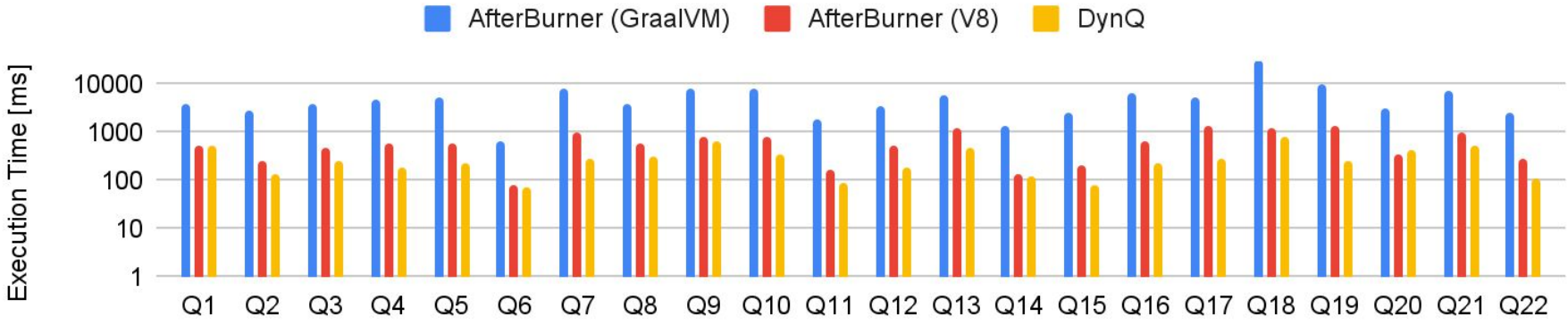
- On small datasets, query interpretation is usually faster than compilation
- Thanks to dynamic compilation, query compilation overhead is not an upfront performance penalty

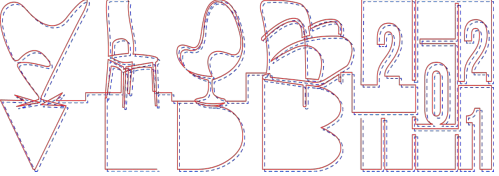




DynQ Evaluation (JavaScript)

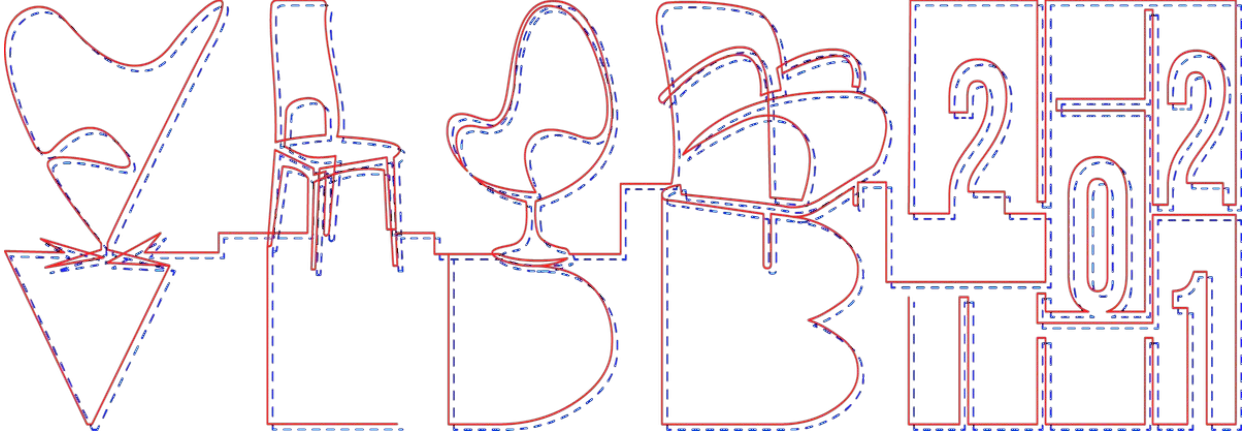
- Evaluation against AfterBurnerDB using a DynQ adapter for AfterBurner columnar layout (<1k lines of code)
- Faster than AfterBurnerDB on its own memory layout in most of the queries





Conclusions

- We presented DynQ, a novel query engine based on JIT compilation
- To the best of our knowledge, DynQ is the first system which executes queries in-situ on object collections from (multiple) dynamic languages
- Our evaluation shows that the flexibility of DynQ does not impair performance
- Future work: DynQ as a standalone library
  - Integrating DynQ in existing data-processing systems



Thanks!

Language-Agnostic Integrated Queries in a Managed Polyglot Runtime

Filippo Schiavio
filippo.schiavio@usi.ch