Optimizing Dynamic Languages for Analytical Workloads

Toni Mattis

HPI Graduate School
Supervisor: Prof. Dr. Robert Hirschfeld

HPI Workshop

2015-06-30

Analytical Workloads

HPI

Analytical

> "… concerned with the discovery and communication of meaningful patterns in data"

Analytical Workloads

- Process large data volumes
- Data often homogeneous, sometimes high-dimensional
- Data mostly read, side-effect free computations

Analytical Workloads

HPI

Scenarios

- Decision support
- Statistics
- Model fitting
- Simulations
- ...

```
@get(„product/<id>/units_sold“)
def get_sales_total(id):
   units = sum(item.quantity
            for item in SalesItem
            if item.id == id)
   return json.dumps({
            „product“: id
            „units_sold“: units})
```

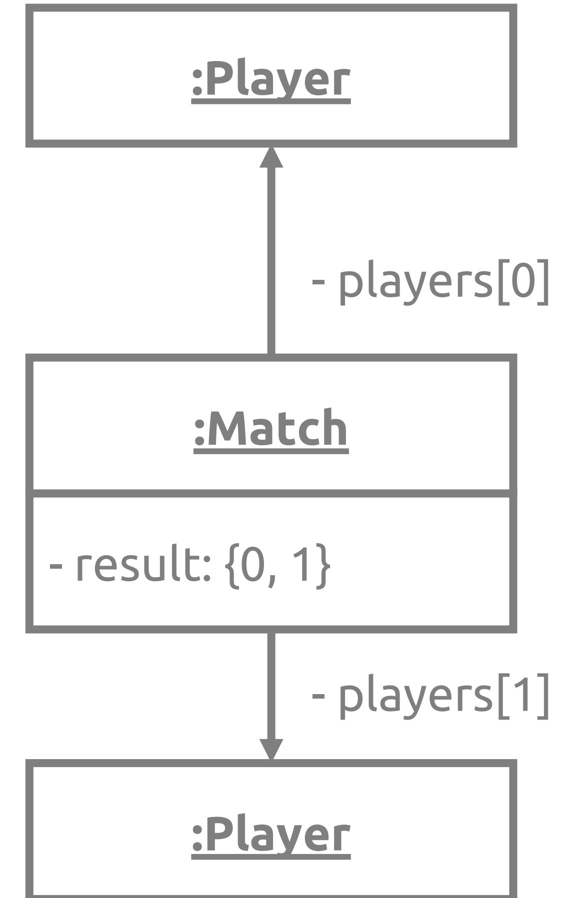
Example

- Scope: ERP web service
- Answer the number of units sold for a specified product

Example: Ranking Players in Games


```
1   scores = {p: 100 for p in players}
    ...
2   for match in matches_played:

3       pred  = predict_result(match, scores)
4       delta = 40 * (match.result - pred)

5       scores[match.players[0]] += delta
6       scores[match.players[1]] -= delta
```

Large data volume
Homogeneous



Dynamic Languages

HPI

Maintainability Requirements

- Conciseness: code communicates exactly the domain logic
- Evolution: domain logic should be able to evolve independently from technology and technical code

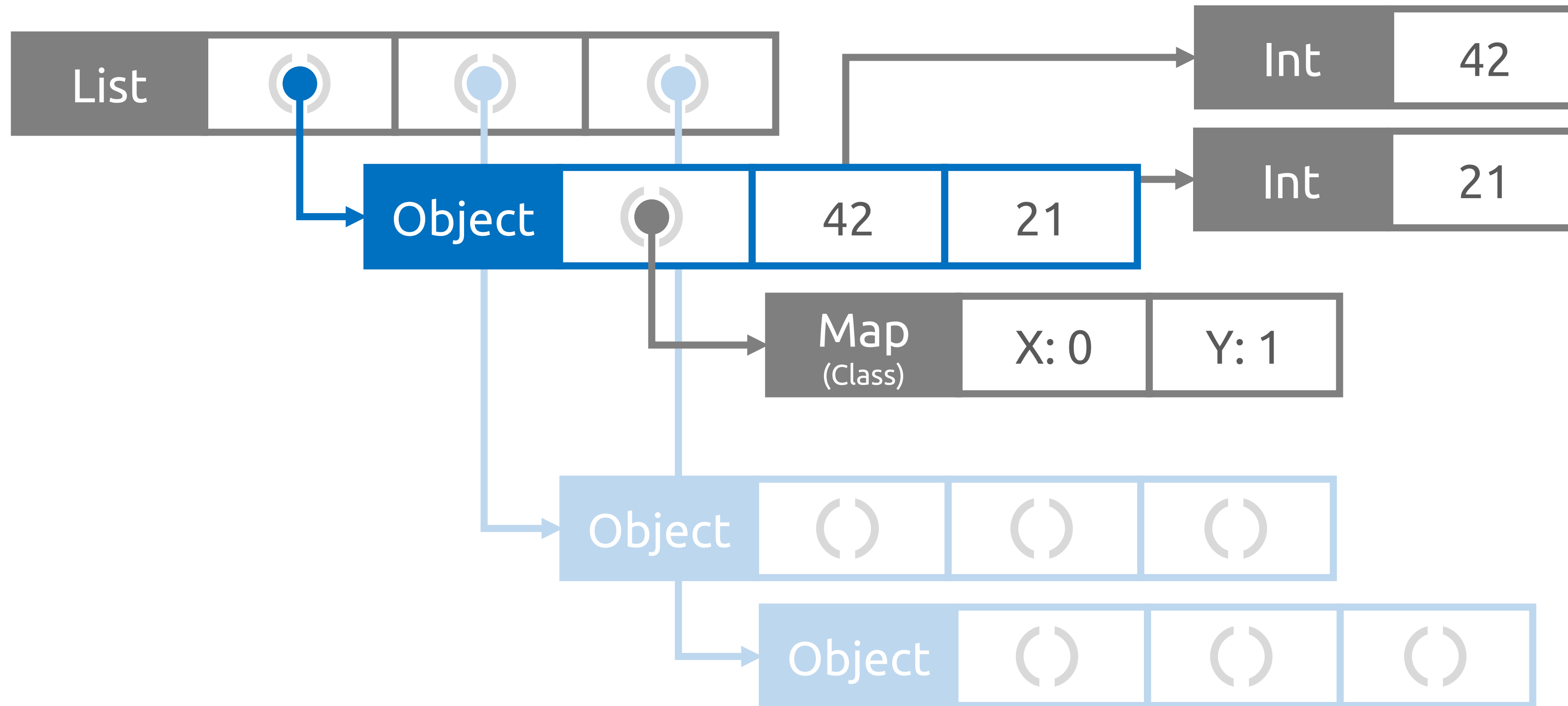
Dynamic Languages

Dynamicity comes at a performance cost

Why?

- Heap structure (objects) build for flexibility
- Analytical data and workloads are quite static

Object Layout

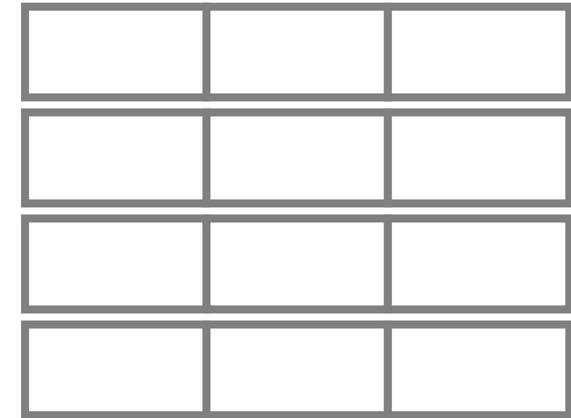


Analytical Databases

HPI

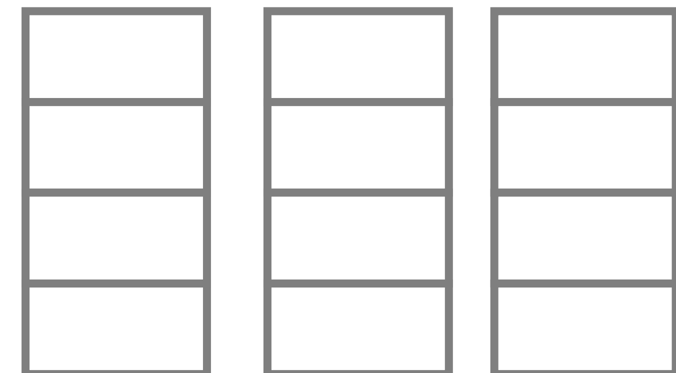
Optimized for

- Data-intensive queries
- Reading
- Tough response time requirements



Prevalent Technology

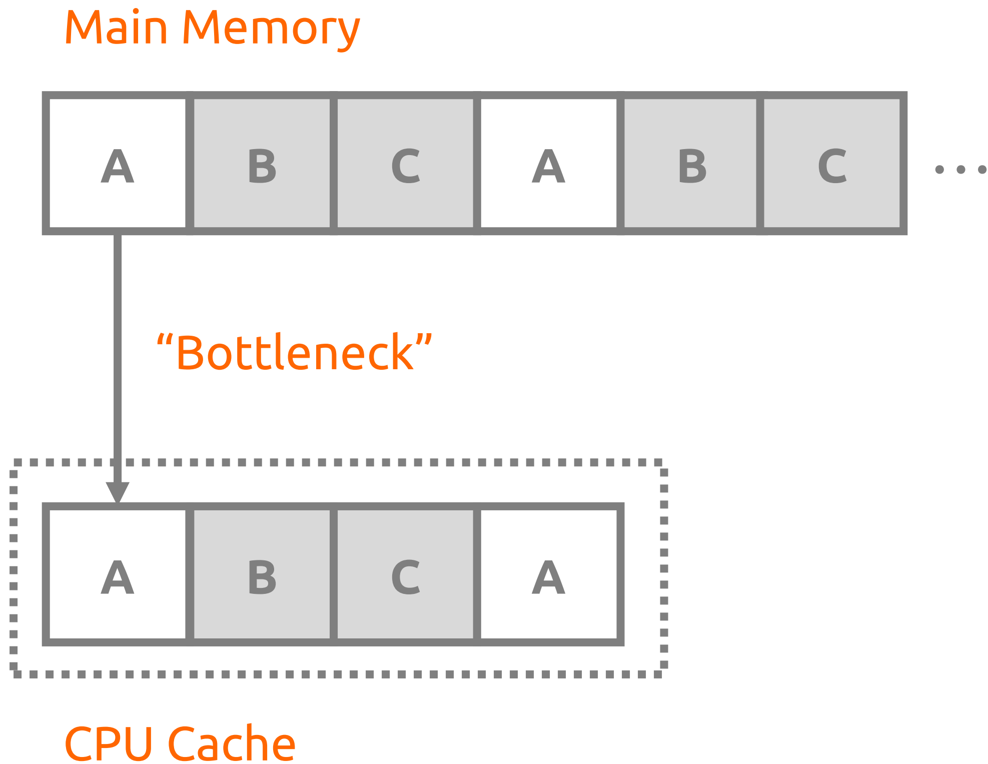
- Main memory based
- Column-oriented storage



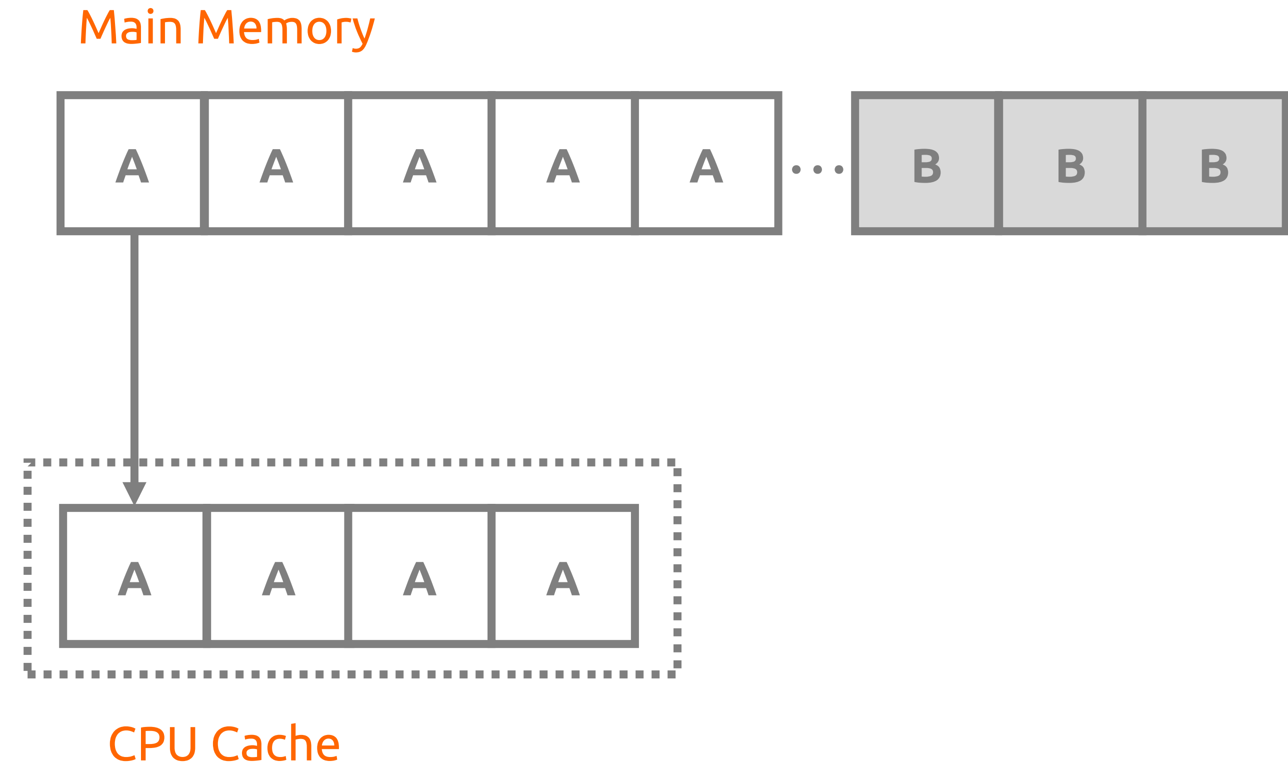
Column-oriented Storage (simplified)

HPI

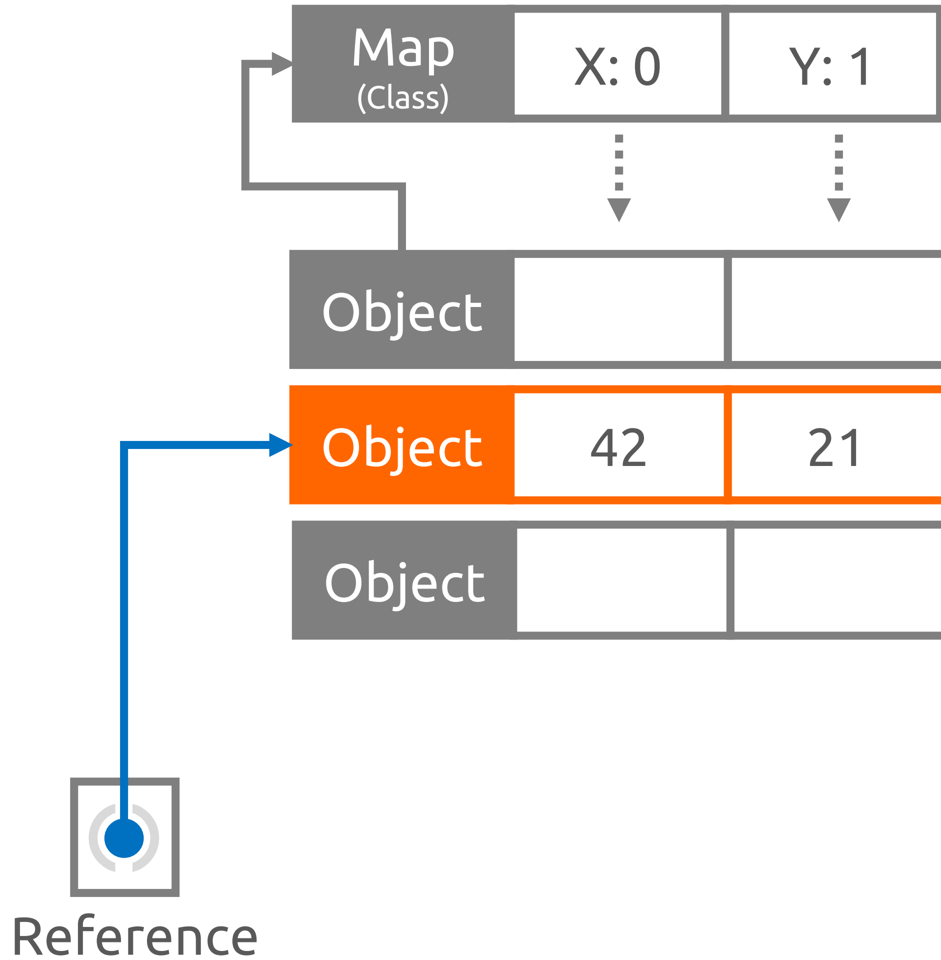
Row-oriented Layout

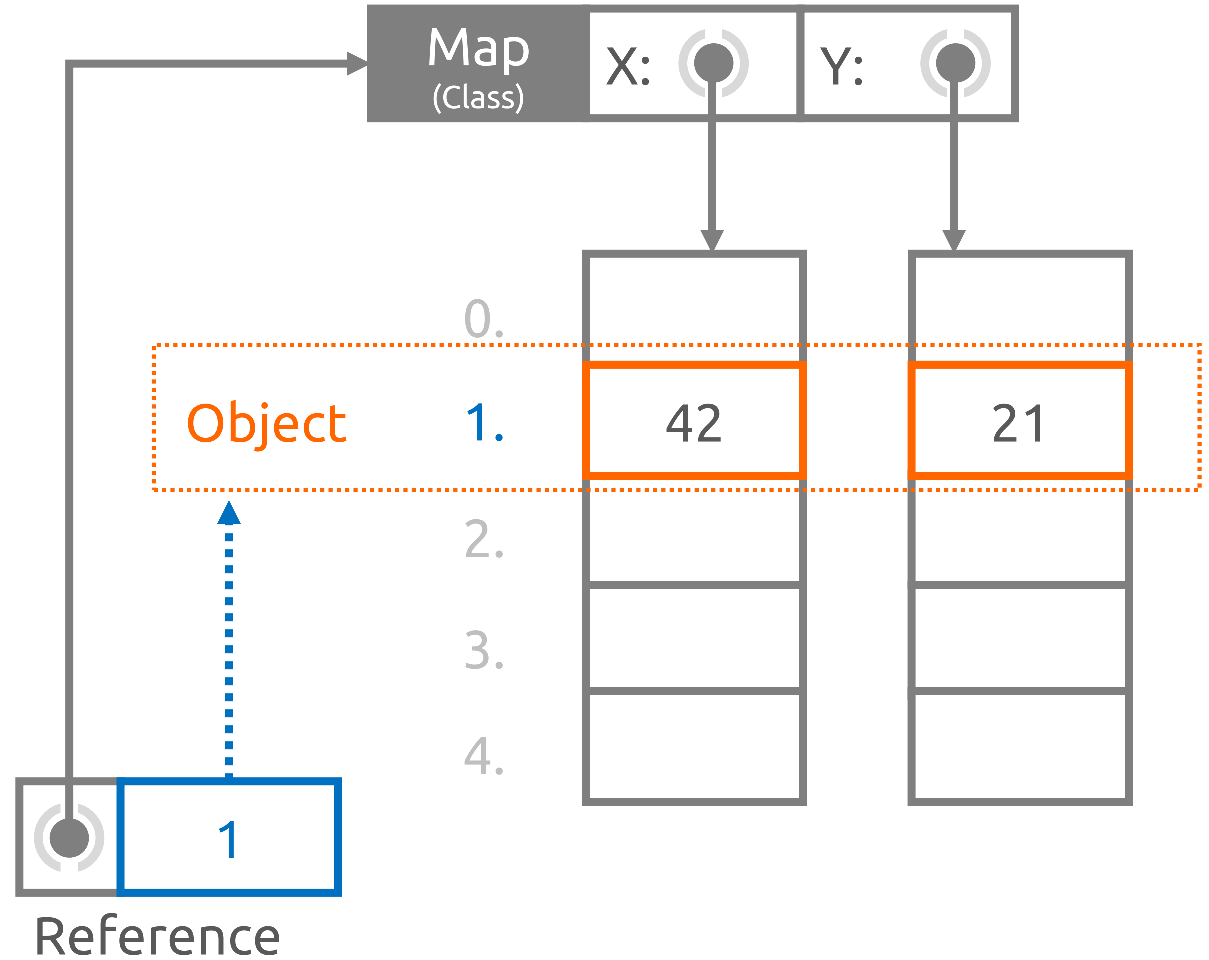


Column-oriented Layout



Transposing Objects





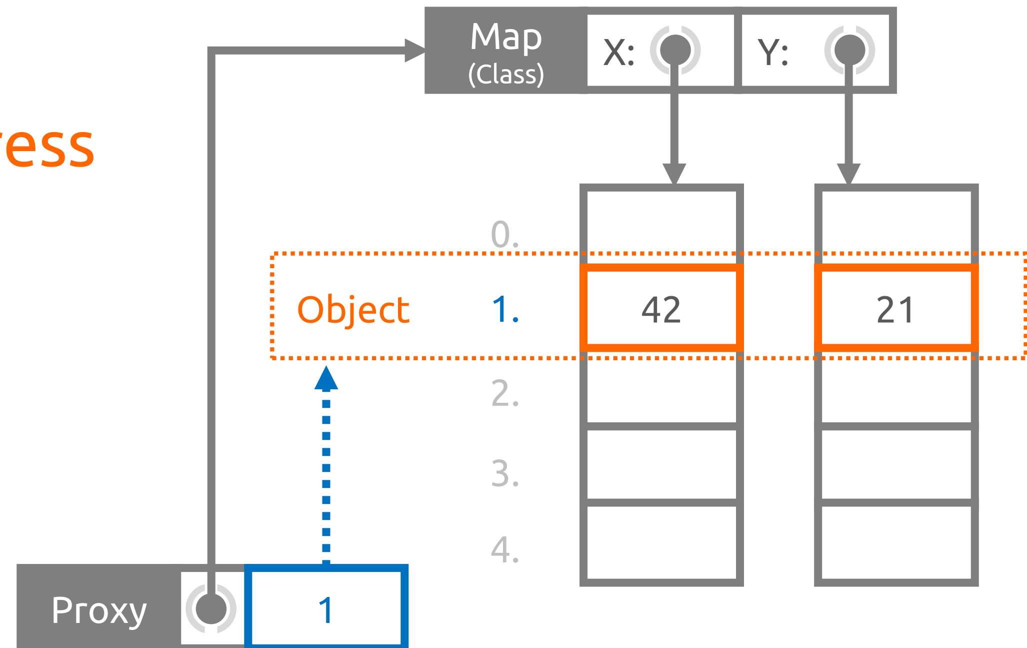
Object Identity by Proxy

HPI

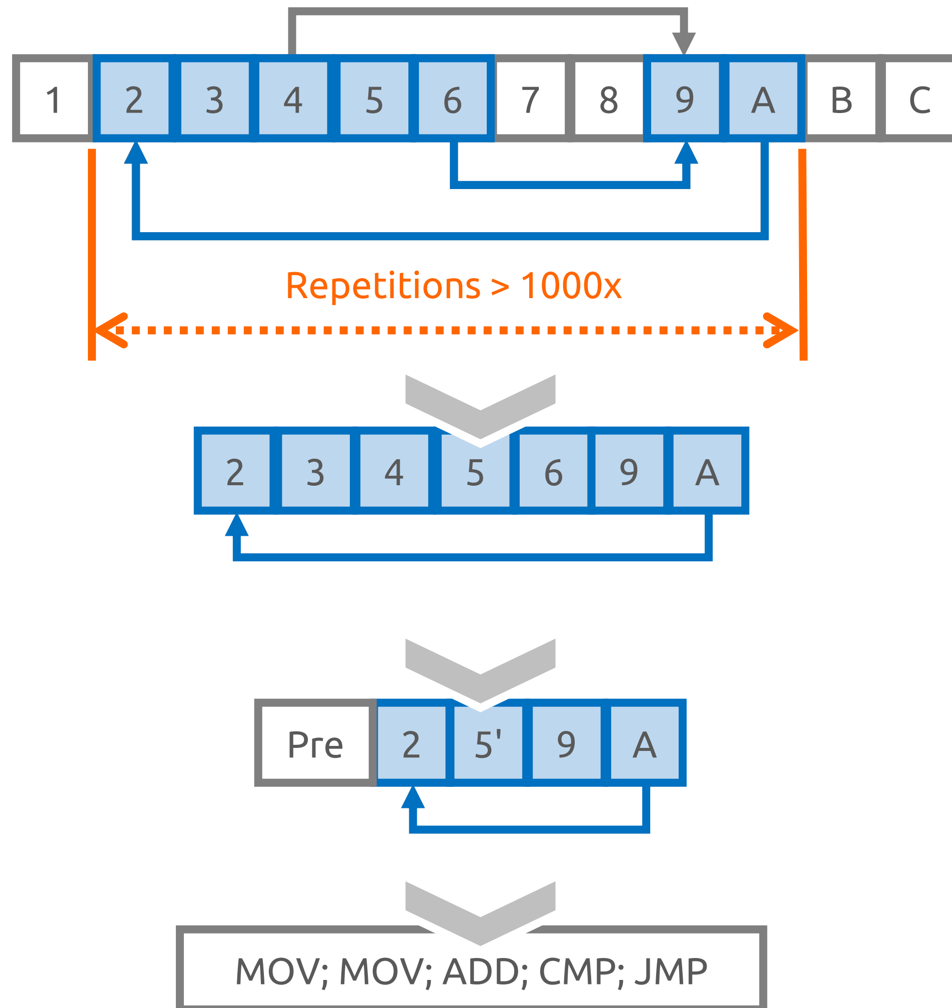
Embedding of *(class, offset)-identity* into *address identity*:

- Use a proxy object with fields class and offset
- Dynamically computes memory address on attribute access

» Overhead can be mitigated by Just-in-time (JIT) compilation



Tracing JIT Compilers (Background)



Instruction Stream (Opcodes)

Hot code detected:
→ start recording operations

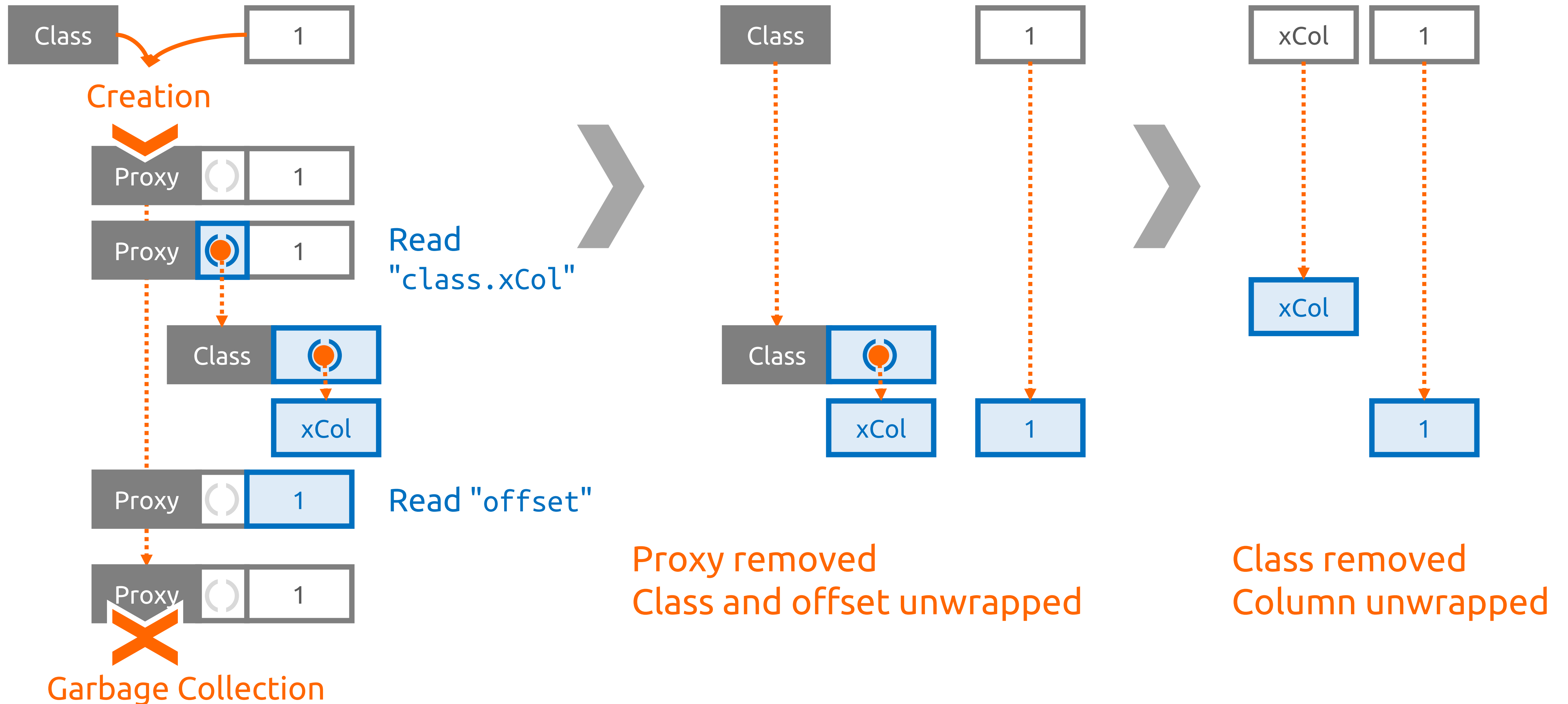
Trace
(flat, no branches, only "guards")

Optimized Trace
(Loop-invariant code motion, allocation removal, …)

Native Code

Allocation Removal (Background)

HPI



JIT + Columns

HPI

» Allocation Removal explodes proxies and classes into offsets and columns

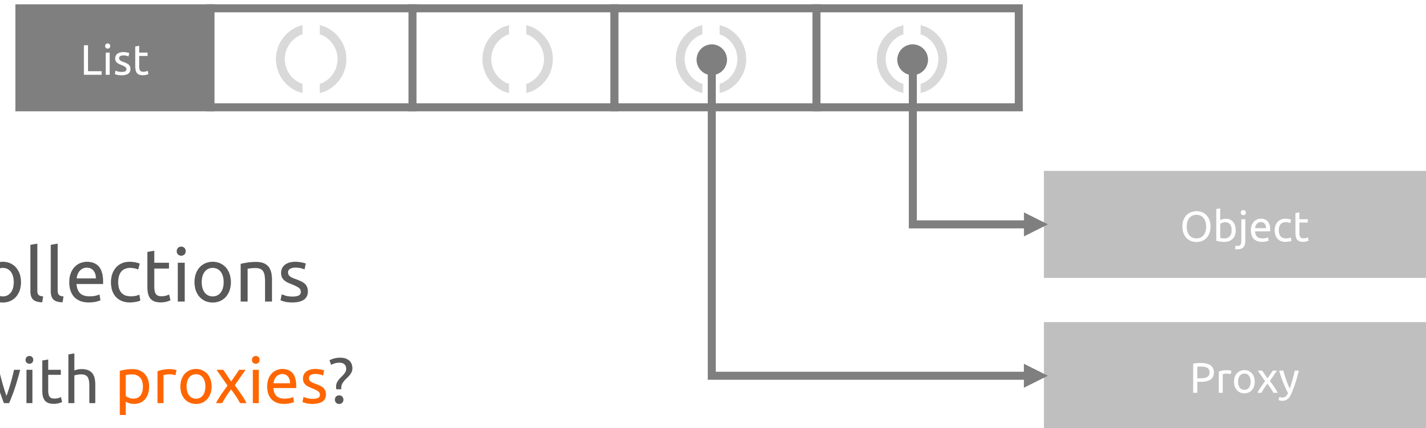
```
for i in LineItem.all():
    total += i.quantity
```

```
n = 0

while n < size:
    i = proxy(LineItem, n)
    col = i.class.columns['quantity']
    offset = i.offset
    value = col[offset]
    total = total + value
    n += 1
```

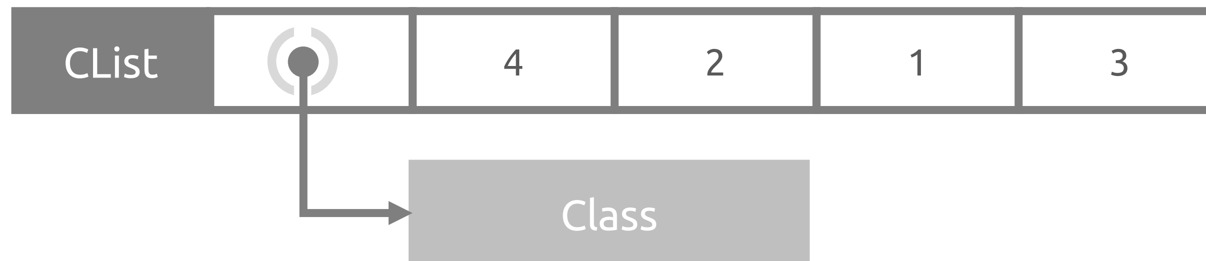
```
n = 0
col = LineItem.columns['quantity']
while n < size:



    value = col[n]
    total = total + value
    n += 1
```

Changes to Collections

- Traditional collections: structure with pointers (addresses)

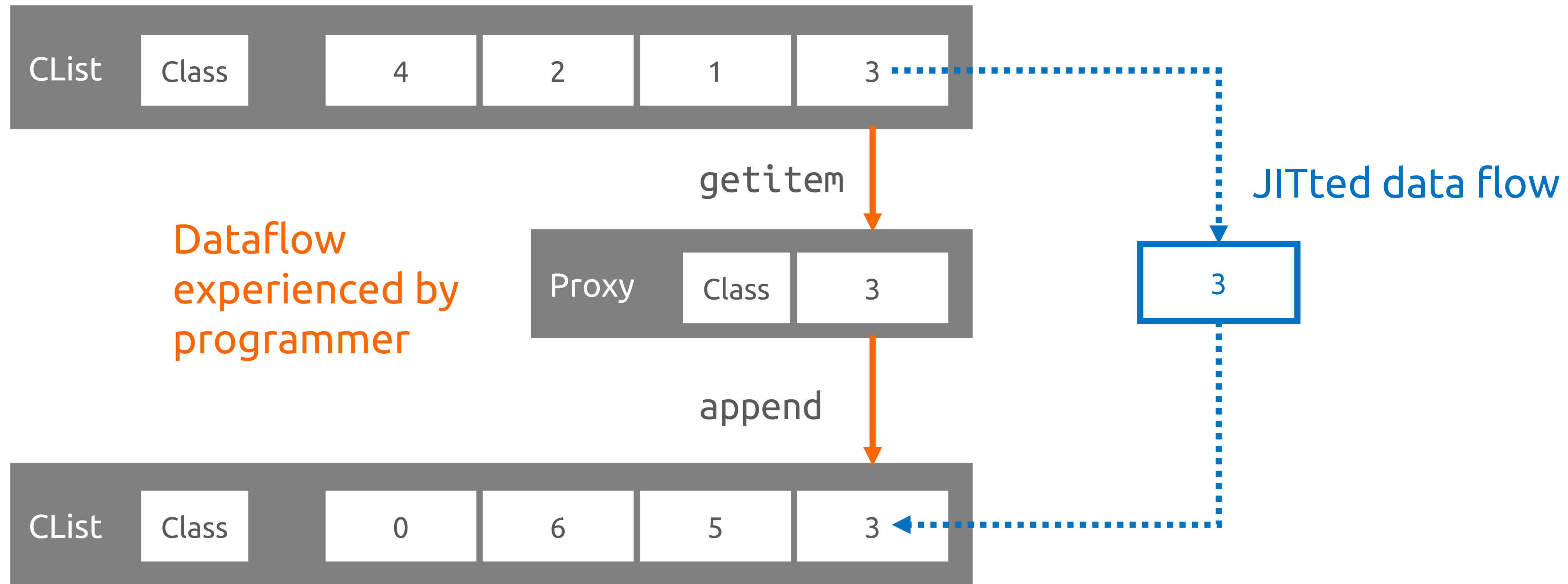


- Columnar collections
  - Structure with proxies?
  - Structure with offsets if items of single class!



"Proxy with multiple offsets"

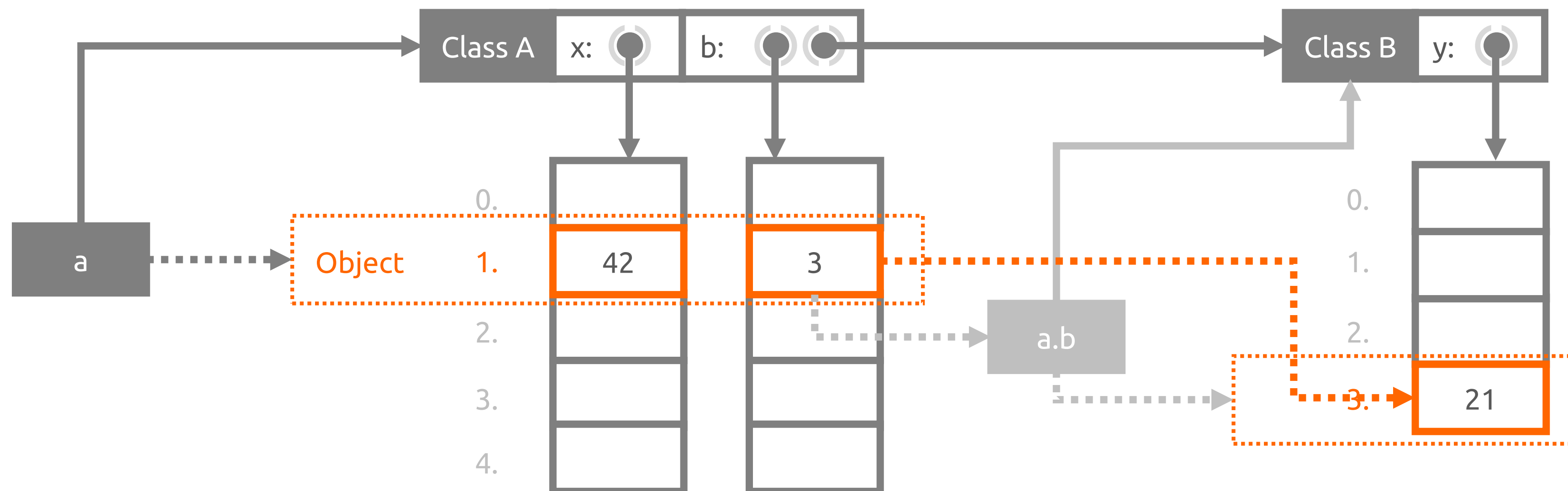
Collections and Allocation Removal



Associations

HPI

» Inspired by foreign keys



» Allocation Removal eliminates intermediate proxies:

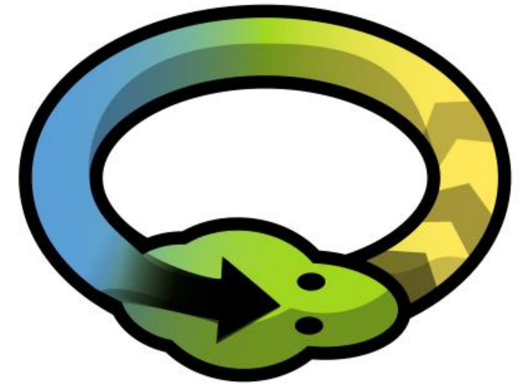
› `a.b.y => y_column[b_column[offset]]` ("Simple Join")

Our Prototype

HPI

PyPy

- Fast Python interpreter
- Tracing JIT-compiler
- Allocation Removal



Our Prototype „Obsidian“

- Implemented on top of PyPy (mostly library code)
- Optimized proxies and collections for PyPy's JIT

Evaluation Setup

HPI

Hypotheses

- Columnar objects outperform traditional objects on analytical code
- Columnar objects are outperformed by commercial in-memory DBs

Competitors

- Baseline: PyPy (as idiomatic Python code)
- PyPy with columnar objects (as idiomatic Python code)
- Commercial in-memory database (as stored procedure)

Platform: 2x 6-core Intel Xeon E5 @ 2.3 GHz (24 threads) | 128 GB RAM | PyPy 2.5.0 | SLES 11.2

Evaluation Scenarios

HPI

ATP: Available-to-Promise Candidate Selection*

- Determine a conflic-free delivery schedule for a set of orders given fixed incoming and outgoing stock changes (always stock ≥ 0)

KM: Kaplan-Meier Estimator*

- Approximate a survival function over time given censored records

Elo: Chess-Player Ranking

- Rank players given only their match outcomes

Balance:

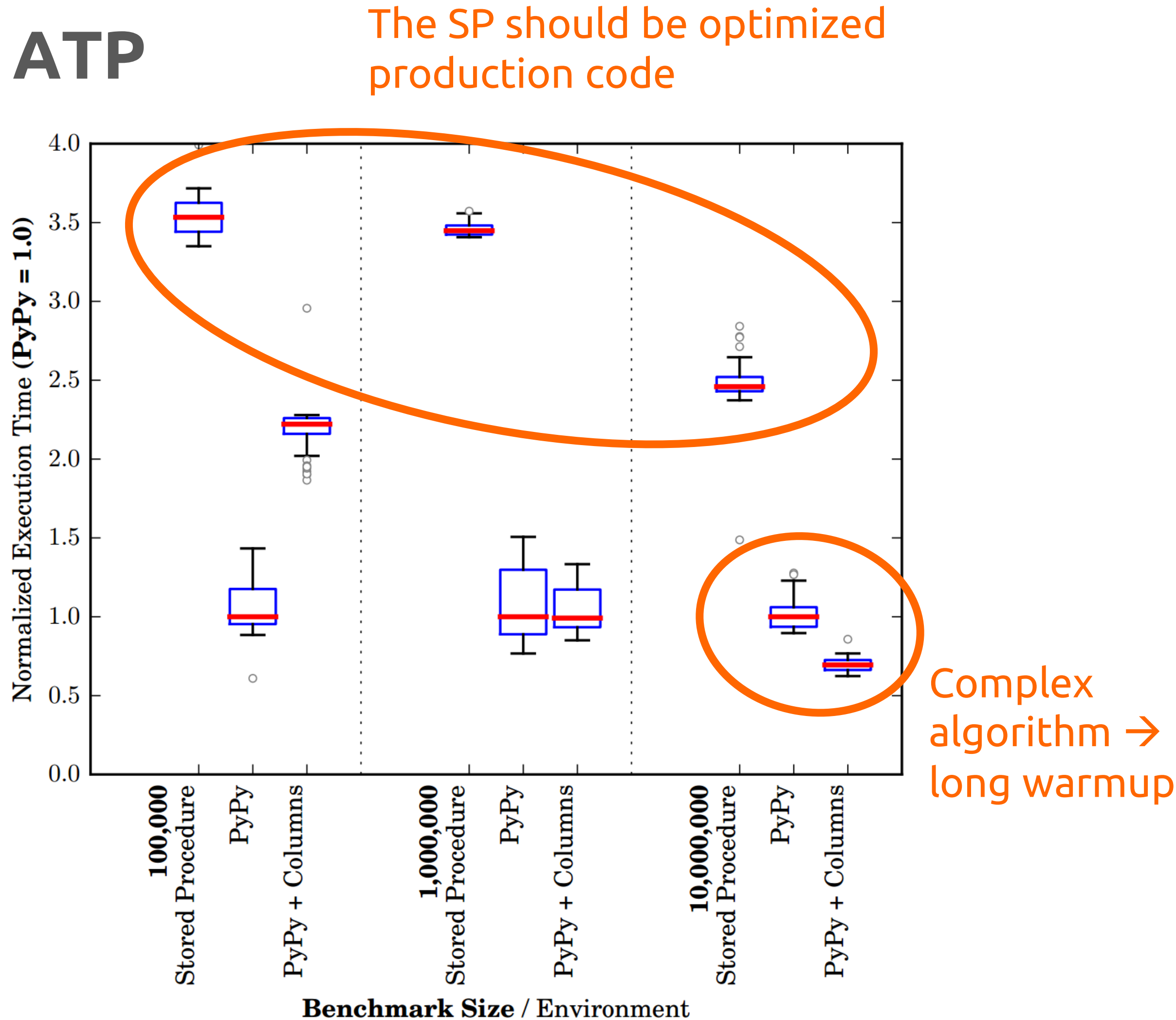
- Compute how long an account was overdrawn given a set of transactions

*) professionally optimized stored procedures were provided by the database vendor

Results

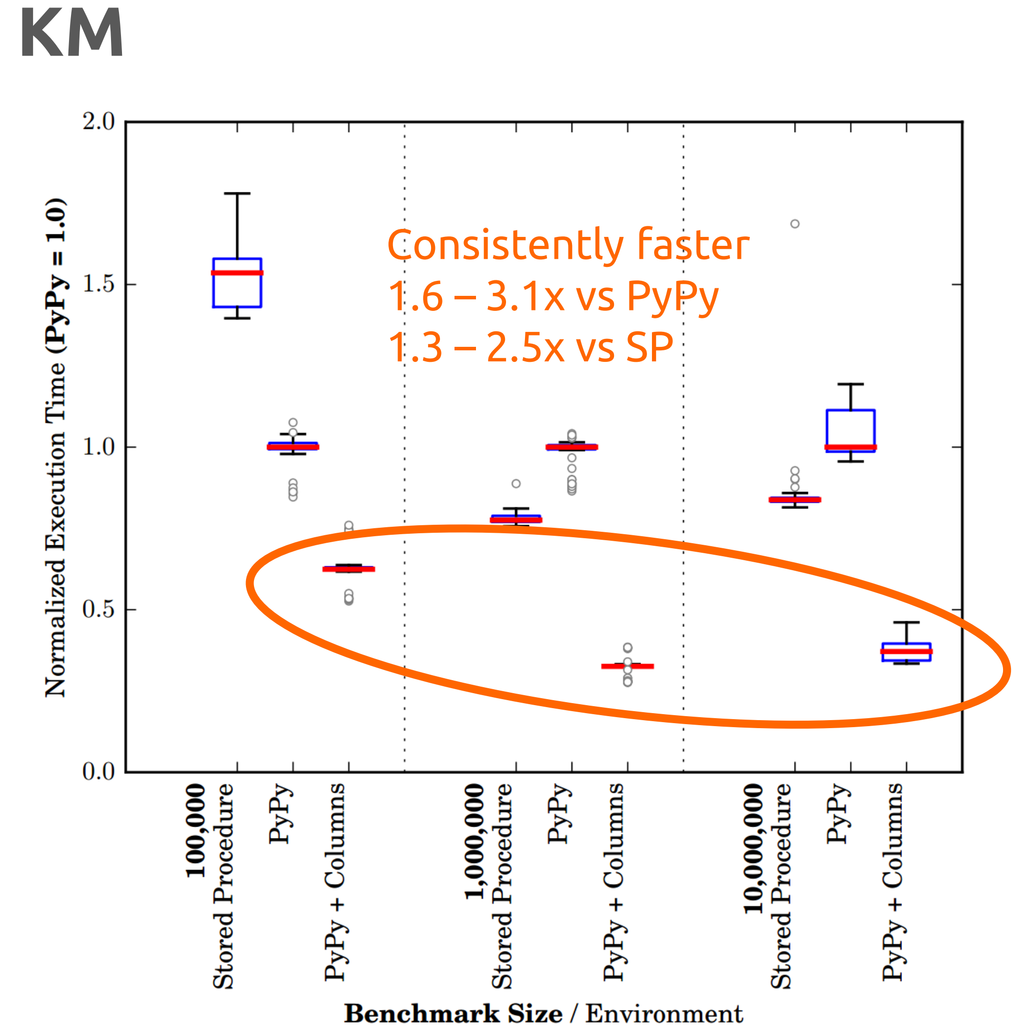
ATP

The SP should be optimized production code



Complex algorithm → long warmup

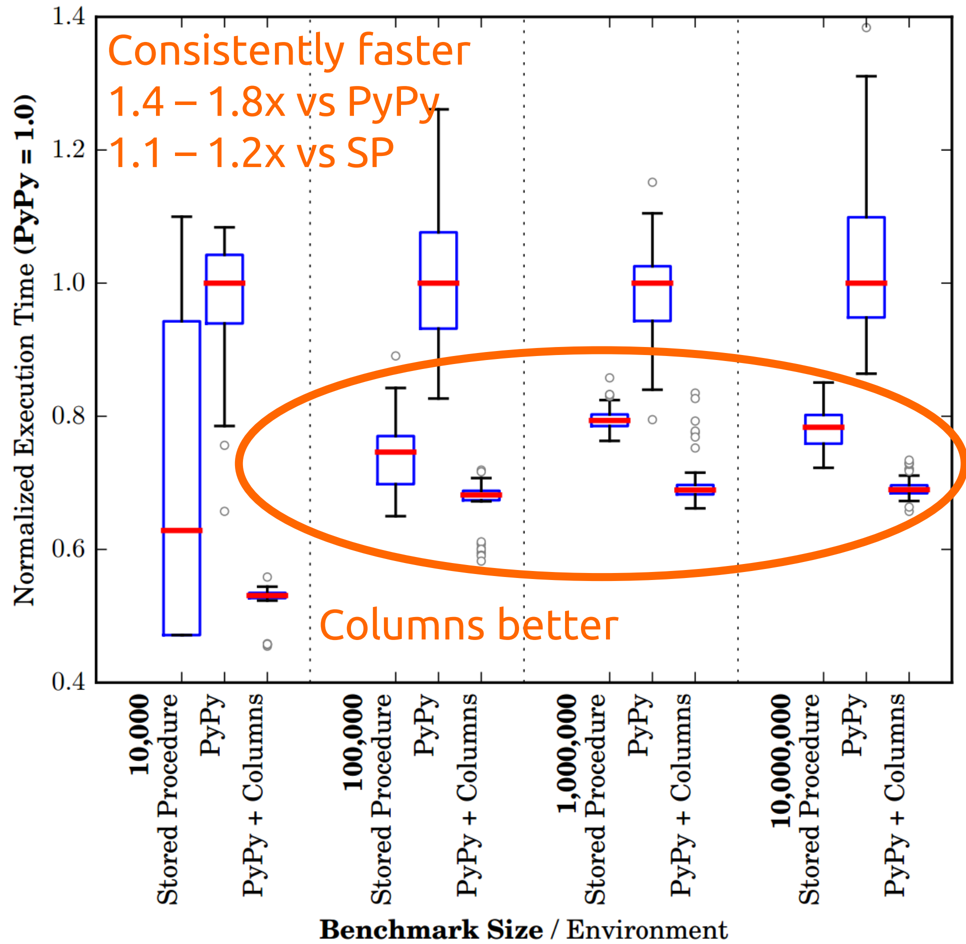
KM



Consistently faster
1.6 – 3.1x vs PyPy
1.3 – 2.5x vs SP

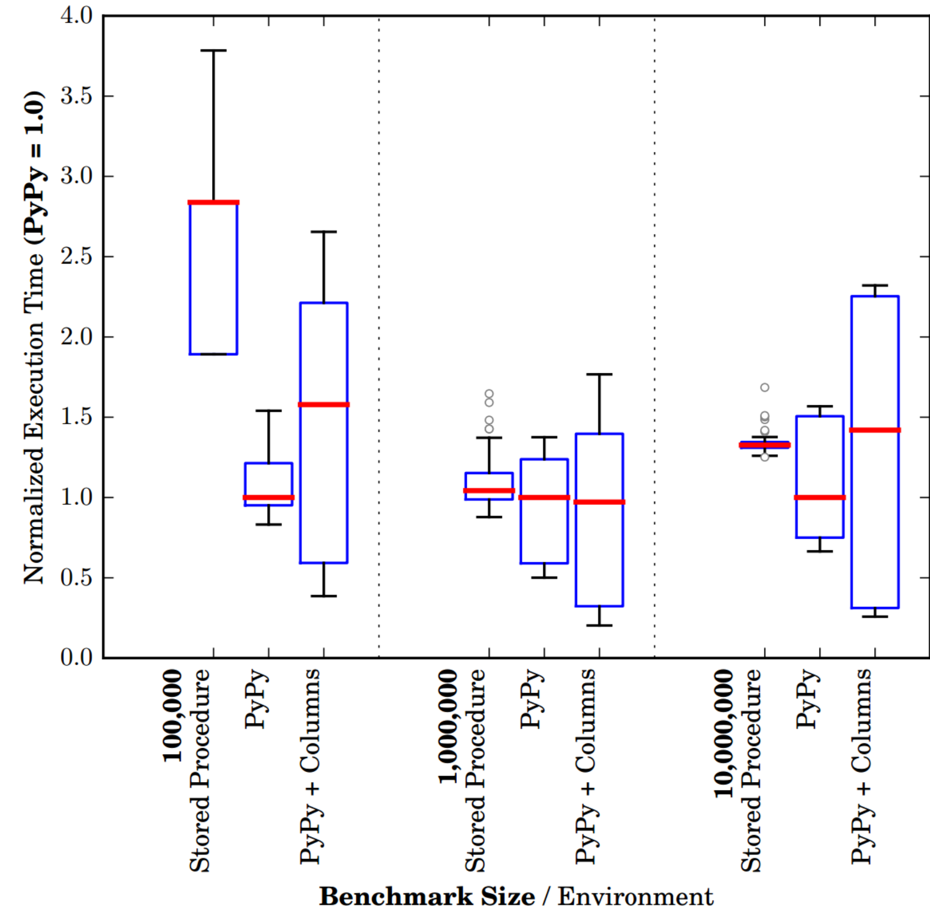
Results

Elo



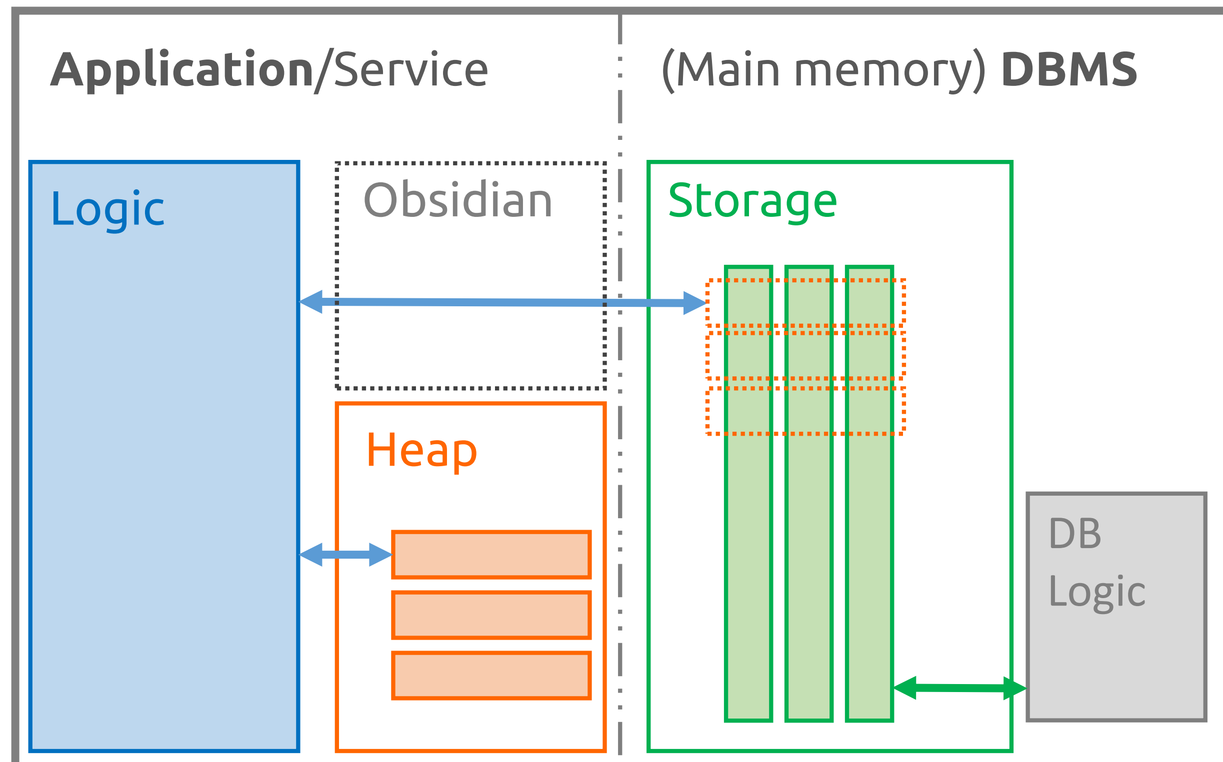
Balance

No significance due to high variance

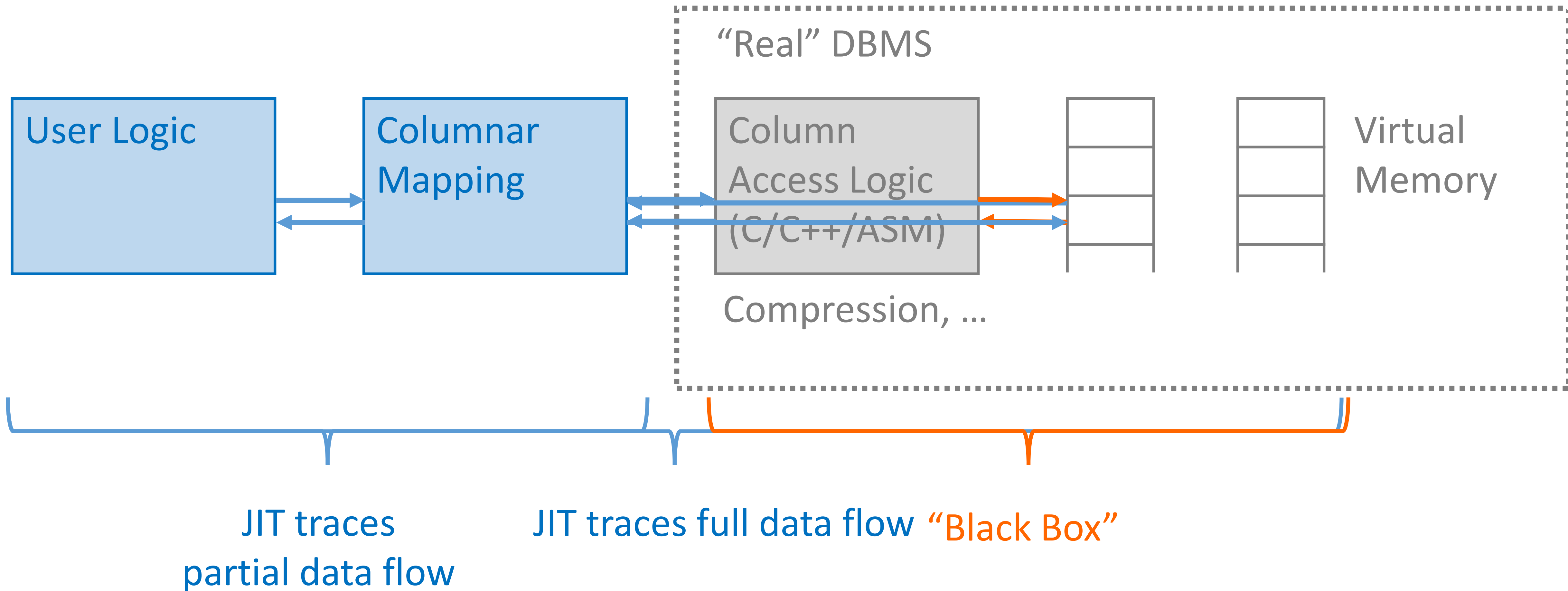


Future Work

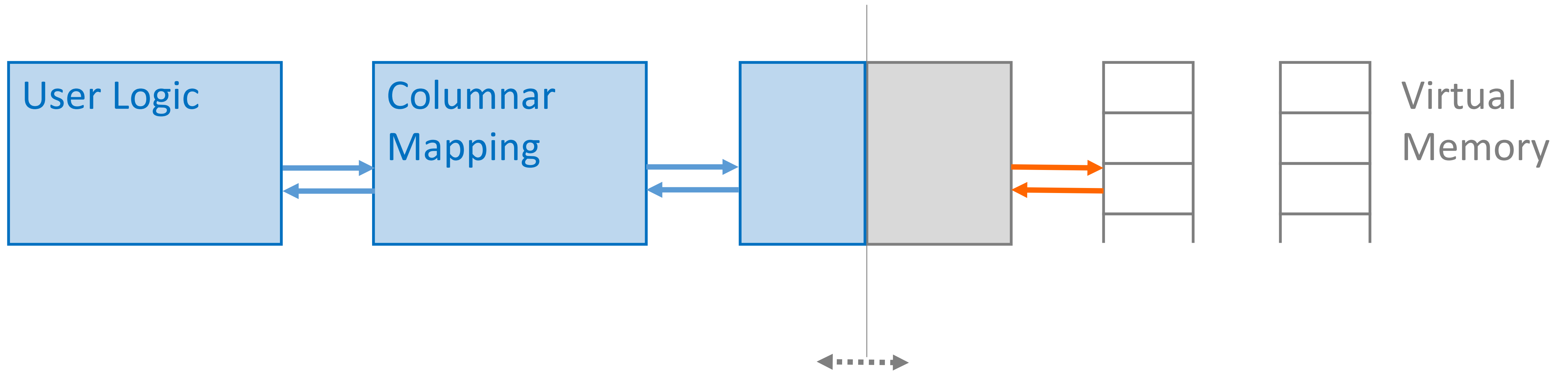
- Run obsidian on the same columns as an in-memory database
  - Ongoing research in our group with promising preliminary results



“Jit-compiling” to a Real Database



“Jit-compiling” to a Real Database



Where should we place the boundary?

← Guide JIT to better “inline” C/C++ code

→ Re-implement DB in Python to allow JITting

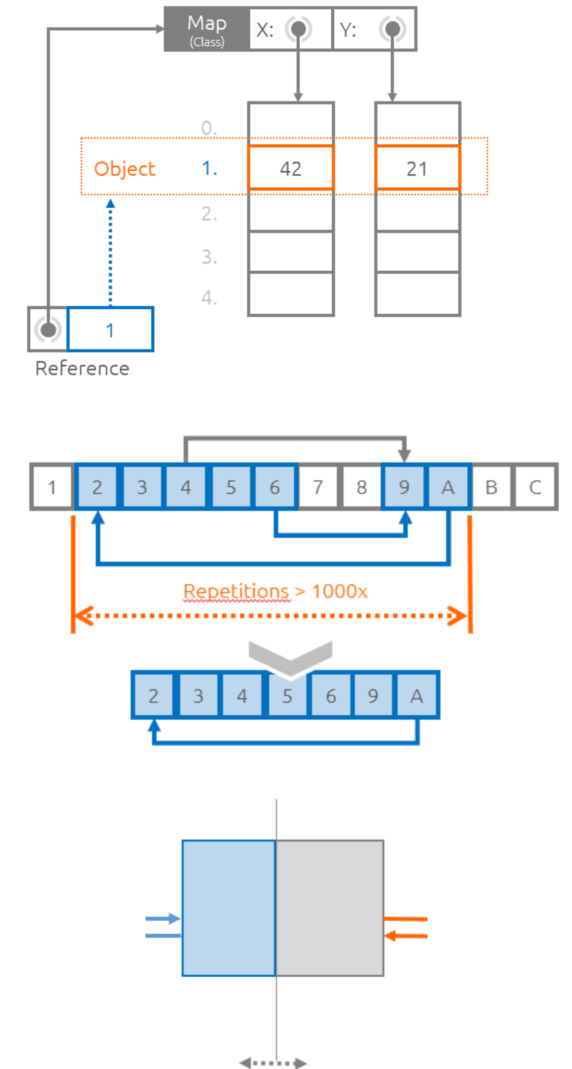
Ongoing research by Johannes Henning, HPI

Conclusion

» Columns can bring performance benefits to dynamic language implementations in analytical scenarios

» Tracing JIT compilers and columns syngergize well

» Unexplored opportunities in the database domain



Stored Procedures

HPI

```
1  CREATE FUNCTION "membership_weight_with_skipping"
2    ( "area_id_p" "area"."id"%TYPE,
3    "member_id_p" "member"."id"%TYPE,
4    "skip_member_ids_p" INT4[] ) -- "m
5    RETURNS INT4
6    LANGUAGE 'plpgsql' STABLE AS $$
7    DECLARE
8      "sum_v" INT4;
9      "delegation_row" "area_delegatio
10   BEGIN
11     "sum_v" := 1;
12     FOR "delegation_row" IN
13       SELECT "area_delegation".*
14       FROM "area_delegation" LEFT JO
15       ON "membership"."area_id" = "a
16       AND "membership"."member_id" =
17       WHERE "area_delegation"."area_
18       AND "area_delegation"."trustee
19       AND "membership"."member_id" I
20     LOOP
21       IF NOT
22         "skip_member_ids_p" @> ARRAY
23       THEN
24         "sum_v" := "sum_v" + "member
25           "area_id_p",
26           "delegation_row"."truster_
27           "skip_member_ids_p" || "de
28         );
```

```
23 Void removeCandidates(CANDIDATE_T & candidates,
24                       QTY_T accumulated)
25 {
26     COL_DATE_T dates = candidates.getColumn<DATE_T>("DDATE");
27     COL_QTY_T values = candidates.getColumn<QTY_T>("QTY01");
28     Size currSize = Size(dates.getSize());
29     QTY_T lastValue;
30     QTY_T zero;
31     while ((accumulated < zero) && (currSize > 0z)) {
32         lastValue = values[currSize - 1z];
33         if ((lastValue + accumulated) <= zero)
34         {
35             accumulated = accumulated + lastValue;
36             if (currSize == 1z) { currSize = 0z; } else { currSize = currSize - 1z; }
37             dates.setSize(currSize);
38             values.setSize(currSize);
39         }
40         else
41         {
42             values[currSize - 1z] = values[currSize - 1z] + accumulated;
43             accumulated = zero;
44         }
45
46     }
47
48 }
```

Interacting with a Database

HPI

Object-relational mapping

- \+ Object-oriented abstractions
- \- Limited by underlying protocol (SQL, libpg, ...)
- \- Copying

Stored Procedures

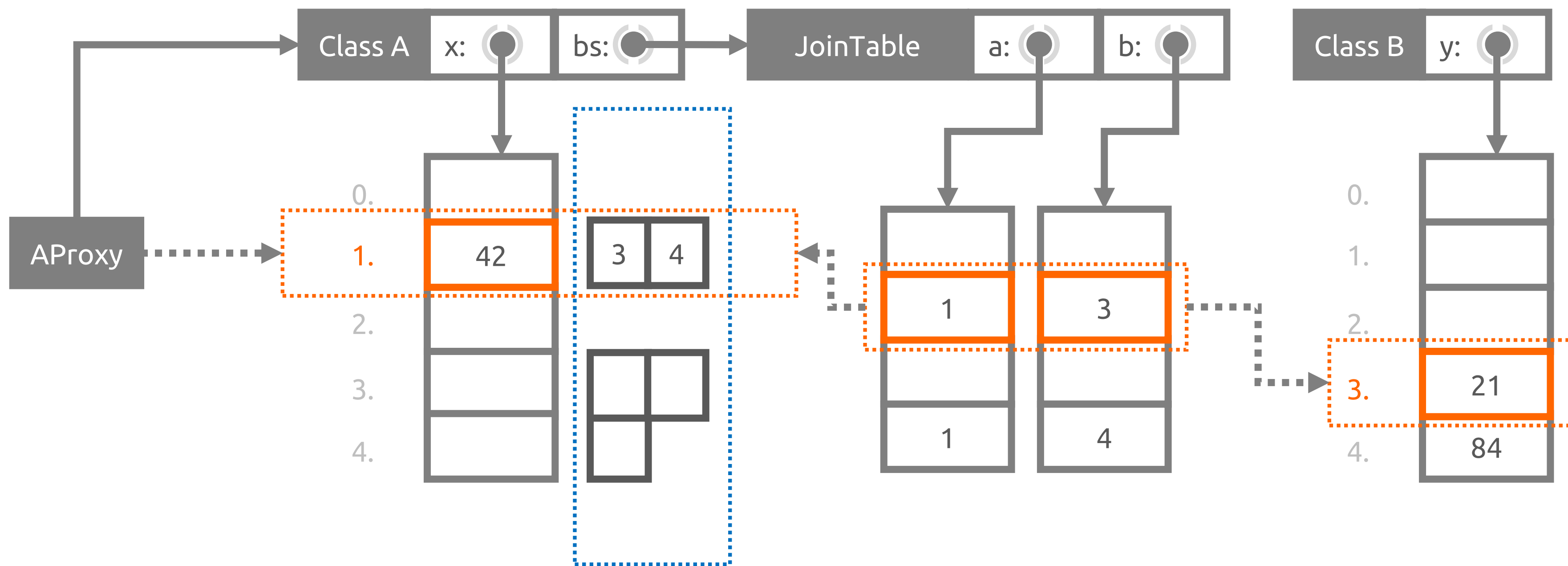
- \+ Performance
- \- Split logic (split tooling, split lifecycles)
- \- Technical abstactions (obscuring domain logic)

It may be a good idea to stay inside a single execution environment

Associations of higher Multiplicity

HPI

» Indexed Join-tables



Quasi-Index [Optional]

Configurable: Consists of per-instance Lists/Sets/Dicts

Collection Operations

» Many languages have special collection operators:

› Python: `reduce, map, [f(a, b) for a in A for b in B]...`

› ST-80: `collection inject:into:, c collect:, c gather:, ...`

› C#: `collection.Reduce(func), c.Select(func), ...`

» Example in Python:

```
sum(i.quantity * i.unit_price
    for i in order.items
    for order in customer.orders
    if order.year >= 2015)
```

Optimizing Collection Operations

- Defer evaluation
  - Apply optimizations over the full operation
  - Prevent intermediate proxies
  - Leave *"Column World"* as late as possible
- Infer result types
  - Allows to allocate result columns instead of ordinary objects

Deferred Evaluation: Plan Construction

```
sum(i.quantity * i.unit_price
    for i in order.items
    for order in customer.orders
    if order.year >= 2015)
```

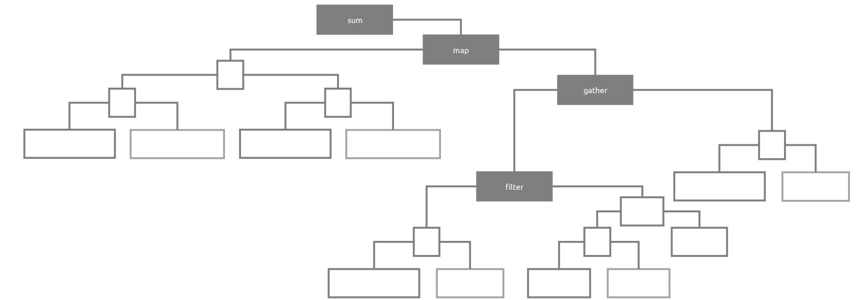
Python OpCodes

Abstract Interpretation

sum
map
*
.
arg[0]
quantity
.
arg[0]
unit_price
gather
.
arg[0]
items
filter
.
customer
orders
>=
.
arg[0]
year
2015

Plan Optimization

HPI



Type inference

- LINQ type system by Erik Meijer
  (handles Python's dynamicity well)
- Create anonymous classes with result columns
- Warn user on failure, continue un-optimized

Tree transformations

- Move filters down the hierarchy
- Replace gather by relational (hash-)join
- ...

Compile to new OpCode and run if needed

Results: Exact Timings

HPI

		platform timings [ms]			speedup	
benchmark	**size**	**PyPy**	**SP**	**Col.**	vs. **PyPy**	vs. **SP**
ATP	10 000	1.32	7.0	28.05	0.05 [0.04 – 0.05]	0.25 [0.25 – 0.29]
	100 000	21.79	77.0	48.41	0.45 [0.44 – 0.46]	**1.59** [1.56 – 1.62]
	1 000 000	235.71	751.0	228.91	**1.03** [0.93 – 1.16]	**3.28** [3.2 – 3.38]
	10 000 000	2739.53	6736.5	1902.78	**1.44** [1.4 – 1.49]	**3.54** [3.48 – 3.61]
KM	10 000	0.59	4.0	2.99	0.2 [0.2 – 0.2]	**1.34** [1.33 – 1.52]
	100 000	28.65	44.0	17.89	**1.6** [1.59 – 1.62]	**2.46** [2.4 – 2.52]
	1 000 000	535.12	415.0	174.31	**3.07** [3.06 – 3.08]	**2.38** [2.37 – 2.4]
	10 000 000	4393.76	3682.5	1631.58	**2.69** [2.53 – 2.91]	**2.26** [2.13 – 2.4]
Elo	10 000	6.36	4.0	3.38	**1.88** [1.8 – 1.95]	**1.18** [1.18 – 1.48]
	100 000	41.54	31.0	28.32	**1.47** [1.43 – 1.51]	**1.09** [1.09 – 1.12]
	1 000 000	359.06	285.0	247.49	**1.45** [1.41 – 1.47]	**1.15** [1.14 – 1.16]
	10 000 000	3506.49	2747.5	2418.84	**1.45** [1.41 – 1.49]	**1.14** [1.12 – 1.15]
Balance	10 000	0.11	0.0	0.16	0.7 [0.58 – 0.88]	0.0 [0.0 – 0.0]
	100 000	1.06	3.0	1.67	0.63 [0.54 – 0.94]	**1.8** [1.53 – 2.66]
	1 000 000	18.22	19.0	17.7	**1.03** [0.61 – 1.91]	**1.07** [0.81 – 2.03]
	10 000 000	119.85	159.0	170.17	0.7 [0.43 – 3.48]	0.93 [0.67 – 4.0]