Columnar Objects
Improving the Performance of Analytical Applications

Toni Mattis¹, Johannes Henning¹, Patrick Rein¹, Robert Hirschfeld¹², and Malte Appeltauer³

¹Hasso-Plattner-Institute, University of Potsdam, Germany; {first.last}@hpi.uni-potsdam.de
²Communications Design Group (CDG), SAP Labs; Viewpoints Research Institute
³SAP Innovation Center Potsdam, Germany; malte.appeltauer@sap.com

Abstract
Growing volumes of data increase the demand to use it in analytical applications to make informed decisions. Unfortunately, object-oriented runtimes experience performance problems when dealing with large data volumes. Similar problems have been addressed by column-oriented in-memory databases, whose memory layout is tailored to analytical workloads. As a result, data storage and processing are often delegated to such a database. However, the more domain logic is moved to this separate system, the more benefits of object-orientation are lost.

We propose modifications to dynamic object-oriented runtimes to store collections of objects in a column-oriented memory layout and leverage a just-in-time compiler (JIT) to take advantage of the adjusted layout by mapping object traversal to array operations. We implemented our concept in PyPy, a Python interpreter equipped with a tracing JIT. Finally, we show that analytical algorithms, expressed through object-oriented code, are up to three times faster due to our optimizations, without substantially impairing the paradigm. Hopefully, extending these concepts will mitigate some problems originating from the paradigm mismatch between object-oriented runtimes and databases.

Categories and Subject Descriptors D1.5 [Programming Techniques]: Object-oriented Programming; D3.4 [Programming Languages]: Processors—Run-time environments, Optimization; E2 [Data storage representations]: Object representation

Keywords Column-oriented Object Layout, Dynamic Languages, Data Science, Just-in-Time Compilation

1. Introduction
Advances in information technology have lead to an ever increasing amount of available data, creating demand to analyze it using algorithms from a variety of methodologies, e.g., statistics, clustering, and forecasting. Such analyses are typically executed on database systems, as these provide an optimized execution and efficient scaling on the data volume used. For an interactive analysis, the response time is crucial. Thus, the improvement of analytical algorithm performance has been the goal of recent developments in relational database technology, such as columnar in-memory databases [19].

In contrast, dynamic object-oriented execution environments have been optimized for different use-cases, in particular for systems with manifold interactions of elements in a complex domain. Hence, they suffer from suboptimal execution times for analytical algorithms. The requirement of short response times can force programmers to give up on object-oriented principles, like abstractions close to the application domain. Instead, they have to program using languages or libraries with less suited abstractions or switch the programming paradigm altogether.

Relational databases, which are a common approach to data-intensive applications, implement a different paradigm. To take advantage of the functions implemented in the database, we have to incorporate them into our object-oriented application. There are three common options for this: Using an object-relational mapper (ORM), which cancels out most performance benefits gained through the database and eventually reduces maintainability [16], employing Structured Query Language (SQL) directly, which constrains developers to a different paradigm and restricts the set of expressible algorithms, or we can use a stored procedure language, which provides good performance but often lacks concepts to adequately express the application domain.

None of these options are satisfactory regarding performance, expressiveness, and maintainability. An ideal solution would allow us to keep programming with an object-
Listing 1: Example implementation of the Elo chess ranking algorithm in Python with columns. Constructor of class Player omitted.

```python
class Player: pass
class Match:
    def __init__(self, black, white, result):
        self.black = black
        self.white = white
        self.result = result
    def predict_result(self):
        d = self.white.rating - self.black.rating
        return 1. / (1. + 10. ** (d / 40.))
for match in matches:
    expected = match.predict_result()
    delta = 2 * (match.result - expected)
    match.black.rating += delta
    match.white.rating -= delta
```

Listing 2: Elo implementation in the column-oriented stored procedure language "L". Most column and variable declarations omitted; prediction function omitted.

```sql
Column<Int32> playerCol = match_results.getColumn<Int32>("PLAYER");
Column<Int32> opponentCol = match_results.getColumn<Int32>("OPPONENT");
// 16 additional declarations omitted ...
while(row < length) {
    current_player = Size(playerCol[row]);
    current_opponent = Size(opponentCol[row]);
    current_result = matchResultCol[row];
    expected = predict_result(
        ratingCol[current_player],
        ratingCol[current_opponent]);
    delta = 2 * (Double(current_result) - expected);
    ratingCol[current_player] += delta;
    ratingCol[current_opponent] -= delta;
    row = row + 1z;
}
```

2. Background

In this section, we define the scope of our contribution and expand on our motivation by examining related solutions. Additionally, we will explain concepts our approach is based on, namely ideas from in-memory databases as well as JIT technologies.

2.1 Analytical Applications

Our approach targets the execution of data-intensive analytical applications, which involve reporting, data mining, forecasting, and similar algorithms. They typically process large amounts of data in a read-intensive way, free of side-effects, and produce an aggregate value, a model from which patterns and prognoses can be drawn, or similar decision-supporting results.

The processed data is homogeneously structured, i.e. from a database perspective there are a lot of entities per table. Further, the data often has a high dimensionality, which corresponds to many columns per database table. However, most analytical algorithms only use a subset of these columns [19, 20]. We neither target transactional nor write-intensive processing. For our concepts we assume that the data is already available and is primarily read.

2.2 Limitations of Current Approaches

We identified the following approaches for implementing analytical applications:

1. Implement the full application inside a single language and execution environment
2. Move the data to a database, but maintain object-oriented abstractions by using an ORM
3. Move the data to a database, but give up object-oriented abstractions and implement performance-critical logic as stored procedures or SQL

1 PyPy is a meta-traced Python interpreter build in RPython [22]
The first approach is the most preferrable in terms of convenience and maintainability, but performance and memory efficiency is often impractical. We explain the reasons in section 3.

The perhaps most convenient way to integrate a database into an object-oriented application is to use an ORM. It maps domain classes to tables of the database. Whenever the developer accesses them, the ORM generates the required SQL commands, executes them and parses the response into objects. However, the performance degradation of this approach outweighs its benefits for data-intensive applications. For instance, when looping over all entries of a table, the ORM will query and materialize each object separately before any computation. The resulting objects take up regular object memory space and offer none of the database optimizations like compression or column-wise traversal.

To take advantage of these optimizations, the performance-critical algorithm needs to be executed close to the data, and therefore is implemented as a stored procedure. While databases provide different stored procedure languages, we have observed that none offers support for in-place execution of dynamically typed object-oriented languages. Stored procedure languages offer abstractions close to the database domain which seldom fit our object-oriented abstractions and do not offer the aforementioned maintainability benefits. All in all, we lose advantages of object-oriented programming by introducing code into our system that is more difficult to maintain, often impossible to debug and needs to be integrated and managed through elaborate database interfaces. A simplified example of a stored procedure that operates on column abstractions is given in listing 2.

### 2.3 Columnar In-Memory Data Storage

In-memory databases perform well executing analytical algorithms. This performance stems from the fact that the data already resides in main memory, as well as from an optimized data layout which mediates issues caused by the memory wall [2, 19]. With regard to the execution of algorithms on large data volumes, dynamic runtimes might benefit from this data layout.

Main memory is a performance bottleneck, as its latency is high in comparison to CPU computations. To mitigate this problem, hardware caches were introduced. As an analytical application accesses large amounts of data, the optimal utilization of the cache determines the overall performance of operations on an in-memory database. For subsequent access, a columnar data layout can improve cache utilization in comparison to row-based storage, by storing all the values of one field for all entities in one sequence (see fig. 1) [1].

As a column stores values which have the same characteristics (e.g. type, range, or distribution), most compression methods become more effective, for example, a run-length encoding. Through these compressions, columnar databases can also store larger amounts of data in memory than row-based databases. Overall, the columnar data layout provides better performance for analytical applications compared to a row-based layout. At the same time, elaborate mechanisms are now needed for write-intensive operations or single entity selects. One example for a columnar database based on the described concepts is SanssouciDB, which influenced the design of SAP HANA [19, 21].

### 2.4 Technology of Meta-Tracing JIT Compilers

Just-in-time compilation is a common mechanism to improve performance in interpreted languages. In contrast to ahead-of-time compilation, JIT compilation arrives at optimization decisions based on the observed types and control flows during program execution. This leads to optimized code which may be very specific to currently processed inputs, but can compete with low-level languages in terms of performance.

#### 2.4.1 Tracing JITs

A tracing JIT is a particular strategy to collect assumptions regarding the control flow of a program. When a code region, e.g., a loop body or method, gets executed very often, a tracer starts to record all operations that are executed during the next run; tracing includes if-branches and descends into method calls. Whenever the trace could have taken another path, it records a guard storing the assumption that leads to this particular trace, e.g., the class handling a polymorphic call or the condition causing a branch. The recorded trace is optimized using common subexpression elimination, constant folding, and the elimination of redundant guards by propagating assumptions (types, non-negativeness, array bounds, etc). The resulting trace undergoes a register allocation step and is compiled to architecture-specific native code [4].

#### 2.4.2 Meta-Tracing

A meta-tracing JIT does not trace the actual user program, but the interpreter running that program. It detects hot paths and loops by observing whether certain parts of the interpreter state repeat, e.g., a repeating program counter may indicate a loop inside the user program. Meta-tracing JITs can be reused across languages. They do not come into direct contact with the high-level dynamicity of the user language but only observe how it is implemented using lower-level
primitives. From a modularity view point, they “scrape” the cross-cutting concerns of JIT compilation (especially tracing) from the actual language implementation. Program design that optimizes for a meta-tracing JIT is likely transferable to other language implementations based on it [4].

2.4.3 Allocation Removal and Escape Analysis

The most important optimization we will address in this paper is the removal of object allocations inside a trace [5]. Dynamic languages usually wrap primitive values, e.g. integers, inside boxes that can be passed around like any other object. However, if a box is only referenced inside a trace and is proven to never escape the trace by a process called escape analysis, there is no need to allocate that box. The surrounding trace can be re-written to directly operate on the primitive and garbage collector invocations are removed completely. Even if the object escapes, its allocation can be deferred to the end of the trace, allowing JITted code to operate on the raw value before it gets wrapped. This optimization can be done recursively for exploding structured objects into register types, i.e. integers, floating point numbers and pointers.

3. A Columnar Object Layout

The following section elaborates on how we adapted the memory layout of objects and contrasts it with common implementations found in dynamic languages. We describe at a conceptual level how these changes help a tracing JIT transforming object-oriented loop code into low-level array operations.

3.1 Common Implementation of Objects

Current object-oriented runtimes represent an object as a continuous block of memory prefixed by a header and followed by the values assigned to its attributes. References to an object are implemented as pointers to the object’s memory [12] 1.

Maps The runtime can read an attribute by loading the object memory at a given offset. Dynamic languages usually link an attribute name to its offset via a structure called map [3, 6, 23] or hidden class [9] referenced by the object header and shared between objects with the same structure. In some implementations (e.g. Dart [23] and PyPy [3]), object header and attribute storage can be separated, so the attribute storage can be relocated in order to grow.

Boxed Primitives In order to treat everything like an object and refer to it via pointers, primitives like integers, Boolean values and other numbers are boxed. Figure 2 shows the memory structure of a full-fledged object using maps and boxed primitives.

3.2 Problems of Traditional Object Layouts

The flexibility of common object implementations in dynamic languages reduces their efficiency regarding the processing of large collections. The following indirections are some of the problems associated with processing collections of traditionally structured objects:

- Boxed values need unboxing to process the raw value.
- Individually boxed values introduce memory overhead.
- Unboxed values need to be re-wrapped in order to be stored in an object.
- Large objects fill the cache with adjacent attributes that are probably not needed right now.
- Collections store pointers to objects, so traversal switches between collection memory, object memory and boxed value memory all the time, reducing overall memory locality.
- Modern JIT compilers may omit repeated map lookups by compiling the positions directly into native code. However, an object’s attribute layout (map) can change quickly, so the JIT needs frequent tests whether the generated code is still valid by checking whether it is still the same map.

3.3 Columnar Object Layouts

Ideally, traversing through many objects and reading one attribute from each of them should fill the CPU cache with the same, fully unboxed attribute of the next objects and neither pollute the cache with unused attributes nor cause memory accesses to maps and classes. We generate this behavior by storing corresponding values of the same attribute in a consecutive chunk of memory, conceptually known as column in the context of databases. (see Figure 3)

We introduce an additional type of class that explodes the attributes of its instances into arrays and reduces the instance itself to just an offset at which its attributes reside in their respective columns. This idea has been explored before in compiler-based transformations to speed up simulations in Java [17] and Kedama [18] (see sections 6.1.1 and 6.1.3).

Our approach, in contrast to previous approaches, works without modifications to the language or compiler. It heavily relies on a JIT to take full advantage of the vertical object layout. Moreover, we allow mixed compositions of both columnar and non-columnar objects and retain full run-time reflection and metaprogramming capabilities on columnar objects.

3.4 Classes, Identity and Associations

Our model changes the implementation of object identity. Columnar objects are uniquely identified by their class (which refers to the columns) and their offset inside the columns, which we call ID. Referring to an object therefore means referring to a class at a given ID. From this point
3.5 Polymorphism and Encapsulation

A columnar class can implement methods and class members like any non-columnar class; only attribute access will be re-interpreted by the runtime. This way, we do not interfere with encapsulation and information hiding.

We explicitly allow subclassing of columnar classes. A subclass inherits all columnar attributes of its base class and may introduce additional columnar attributes. Inherited columns are shared between instances of all subclasses in a hierarchy and newly introduced columns will have null-values at all offsets not belonging to their declaring class.

Figure 2: Memory layout of an object using maps and boxed primitives

![Figure 2: Memory layout of an object using maps and boxed primitives](image)

We also introduce a class ID column to record the most specific subclass responsible for a particular column offset and to guarantee correct polymorphic message dispatch in co-variant collections and attributes.

3.6 Gradual Typing

In contrast to the fully dynamic nature of Python, columnar attributes should have a type. For value types, e.g. integers, this allows to allocate a plain array without any boxing or run-time type checks. Also, knowing the exact columnar class of a reference allows us to unpack the ID of their instance into a plain integer array and reconstruct the proxy whenever the association is read.

However, types are optional. A column may receive type Object and store boxed values, proxy objects to other columnar classes and arbitrary objects from the language. This may only cause performance issues if the dynamically typed column is frequently read during an analytic computation.

3.7 Leveraging the JIT compiler

Our primary goal is to improve speed despite having proxy objects as additional indirection. This can be achieved by reducing the life time of proxies containing just the class and the ID to a minimum. The more limited an object’s life time and scope, the more effective the allocation removal will be. Also, whenever a proxy object “escapes”, i.e. there are references to the proxy which survive a particular loop or method body, then it cannot be optimized away, but becomes an actual heap object. Life time reduction can be achieved at multiple positions:

- **Iterators** traversing collections of columnar objects always emit a fresh proxy. As long as this proxy is only used inside the loop, allocation removal will explode the proxy into ID and class. Loop code will subsequently be compiled to work with a plain integer ID instead of a proxy object.

- **Collections** should deconstruct inserted proxies into ID and class and reconstruct the proxy on each read access. This saves memory and prevents the proxy from “escaping” due to a reference by the collection.

- When following an association, a new proxy representing the instance of the target class is created. However, if only a primitive attribute is read from that proxy (e.g. `match.black.rating`) it will never be allocated and the
class Player(Columnar, Float('rating'),
    String('name')): pass

class Match(Columnar, Player.one('black'),
    Player.one('white'),
    Integer('result')):
    def predict_result(self):
        d = self.white.rating - self.black.rating
        return 1. / (1. + 10. ** (d / 40.))
    
for match in matches:
    expected = match.predict_result()
    delta = 2 * (match.result - expected)
    match.black.rating += delta
    match.white.rating -= delta

Listing 3: Elo code with columnar attribute definitions added to the classes. The integer ‘result’ is 0 if the black player won or 1 if the white player won.

operation is folded into a nested array lookup (equivalent to evaluating rating_column[black_column[match_id]] without ever allocating a proxy)

Table 1 shows the effects of allocation removal on an iteration over columnar objects in contrast to iterating over unmodified, traditional objects.

4. Implementation

Our implementation consists of a plain Python library which provides an API for working with columnar objects and can be loaded at run-time. We only target the PyPy implementation of Python due to its meta-tracing JIT.

Our prototype uses proxies for each columnar object. The proxies redirect attribute access to the respective columns. We rely on PyPy’s allocation removal and inlining to prevent the proxy from causing any overhead in JIT-compiled code.

4.1 Example

Listing 3 shows an implementation of the Elo chess ranking algorithm from listing 1 using our library. The base class Columnar implements instance and proxy creation, Float('rating') and Integer('result') create primitive columnar attributes named rating and result, while Player.one creates an association attribute. There are no changes to methods and to the analytical computation.

4.2 Proxy Implementation

What appears to be an instance of a class, e.g. a Match object, is in fact just a proxy object. In our Python implementation, their state consists of __class__ and __id__ fields, as illustrated in fig. 3 and is managed by the Columnar base class. This is analogous to normal objects, which consist of a __class__ field but have their attributes directly attached to the object. The attributes at a proxy are implemented by Python’s property objects, which instruct the runtime to explicitly invoke getters and setters associated with the respective property.

4.3 Attribute Mixins

The term Integer('result') in the class header of Listing 3 creates a mix-in, whose purpose is to provide an integer column and a property that accesses the column whenever the result attribute is read or written. This way, we can compose our columnar class by inheriting single-attribute mix-ins. As attribute lookup is late-bound, new columnar attributes can still be added and removed from the class dynamically. The following code illustrates the implementation of the Integer() mix-in factory, which spawns integer columns with accessors:

```
def Integer(name):
    # column construction
    column = allocate_int_column()
    
    # getter reads column at instance offset
    def getter(instance):
        return column[instance.__id__]
    
    # setter writes column at instance offset
    def setter(instance, value):
        column[instance.__id__] = value
    
    prop = property(getter, setter)
    
    # type/mixin construction:
    return type(
        name='',
        bases=(),
        dict={name: prop},
    )
```

Listing 4: The Integer factory creating a mixin redirecting attribute access to a column.

The resulting inheritance hierarchies do not impact performance, because the tracing JIT will remove super-class lookups and inline accessors, regardless of where they appear in the hierarchy.

4.4 Inspecting the Trace

Apart from continuous speed measurements, the effectiveness of optimizations can be analyzed by inspecting the trace produced by the JIT. Consider the following microbenchmark counting how often the white player won:

```
def white_player_wins():
    count = 0
    for item in Match.instances:
        # result: 0 = black wins, 1 = white wins
        count += match.result
    return count
```

Listing 5: Example aggregation

Iterating over the instances of a class yields new proxy Match instances for each offset allocated by instances of this
class. After creating and aggregating several millions of instances, the JIT converged to the following set of instructions (operation names and variable names are renamed for better readability, # starts a comment):

```java
iterator = <set up iterator>
max = <get upper limit of iterator>
column = <get result column>
column_len = column.size
c = 0 # the unwrapped count variable
i = 0 # the unwrapped iterator state
loop:
    # --- inlined iterator call ---
guard(i < max)
i = k + 1
    # --- inlined item.quantity lookup ---
guard(k < column_len)
guard(k >= 0)
qty = column[k]
    # --- addition, overflow check ---
s = s + qty
    guard_no_overflow
    # --- write iterator state back ---
cur = wrap_int(k)
iterator.current = cur
i = k
jump(loop)
```

4.5 Associations

An association to another class is represented as a column of IDs. Instead of reading and writing proxies from and to an array, the associated instance’s ID is stored inside the column and the wrapper object is reconstructed when the field is read. This mechanism is exposed to the user by the `one()` class method, which returns the appropriate type to inherit. See the usage of `Player.one()` in the `Match` class in listing 3. Evaluating an expression like `match.black.rating` now results in the following operations:

1. Lookup `id = match.__id__`
2. Lookup the player ID `id2 = Match.black_column[id]`
3. Construct the player proxy `pl = Player[id2]`
4. Lookup `id3 = pl.__id__`
5. Lookup the rating `rating = Player.rating_column[id3]`

The allocation removal of the JIT will prevent the player proxy from being allocated as it only serves the purpose of looking up its rating and would be garbage collected instantly. Instead, the five lookups above will be collapsed into two nested lookups, which do the same as:

```java
rating_column[black_column[id]]
```

where `id` is the fully unwrapped `Match` instance.

4.6 Inheritance

By introducing a `metaclass` for columnar classes, we override class creation. When our metaclass constructor detects that a columnar class is being subclassed, it adds a `class_id` column to the topmost columnar class if not already present, and each class receives an integer representing its ID.

When a proxy is created, which happens in iterators or when following an association, it sets its `__class__` field depending on the value of `class_id` at the instance’s offset. This implements co-variance: Declaring an attribute as `Player.one('black')` also allows subclasses of `Player` to be written to and read from the `black` attribute.
4.7 Collections

When proxies are put into collections, such as lists, the proxy object is usually stored as a full heap object, because it will be referenced by the collection for a long time. This can be avoided by unpacking the proxy ID once it is inserted into the collection and reconstructing the proxy when it is accessed. The resulting memory layout is depicted in Figure 4 and generally improves speed and memory efficiency of collections of columnar instances. In our library, we provide custom data types for lists, sets and dictionaries, which implement this unwrapping behavior.

5. Evaluation

Our motivation for using a column-based object layout is to decrease the execution time of analytical algorithms written in an object-oriented dynamic language. We evaluated our concept by running four analytical algorithms and three microbenchmarks on our prototype.

Our approach also aims to provide an abstraction to program analytical algorithms in an object-oriented fashion. To determine the effects of the changed memory layout on the abstractions available to the programmer, we also qualitatively evaluated our integration of the columnar objects into the object-oriented abstractions of Python.

5.1 Performance Benchmarks Setup

All benchmarks were executed on a server architecture with the following specification:

- **CPU**: 2 Hexa core Intel Xeon E5-2630 (24 logical cores), maximal clock speed of 2301 MHz
- **Memory**: 128GB Main memory consisting of 8GB DDR3 modules with 1333 MHz clock speed
- **System software**: SUSE Linux Enterprise Server 11 (Kernel version 3.0.80-0.7-default)
- **Runtimes and compilers**: PyPy version 2.5.0-alpha0 and gcc version 4.3.4 revision 152973

We measured the execution time of a benchmark by wrapping the benchmark in a function and measuring the time between calling the function and it returning. Each benchmark configuration was measured 60 times. To correct systematic bias created by our choice of input data, each run used a different seed to generate random test data. All PyPy measurements are conducted using the standard PyPy JIT configuration and with a warm JIT. This means that with an input data size smaller than one million the benchmark was run 100 times before a measurement was taken. Above an input data size of one million the benchmark was run once and then a measurement was taken. This is sufficient as for the PyPy JIT compiler only the total number of loop iterations influences the optimizations.

5.2 Benchmarked Algorithms

There are established benchmark suites for object-oriented runtime environments and analytical database systems, but none of them fit the perspective from which we approached application development. Typical benchmarks for object-oriented and mixed-paradigm languages, e.g. “Richards” and “Deltablue”, are computation-intensive rather than data-intensive and often involve a significant portion of writing and side-effects, which makes them unrepresentative for analytical scenarios. Common database benchmarks, e.g. “TPC-H”, are expressed in SQL, which does not directly map to Python constructs.

We therefore acquired implementations of algorithms that are used as stored procedures in production business applications (“ATP”, “KM”), added other benchmarks covering the spectrum between data- and computation-intensive analytical algorithms (“Elo”, “Balance”), and ran them on differently sized collections of objects up to 10,000,000 items. Except for the “Balance” benchmark, all these benchmarks access several different attributes of each instance.

**Available to Promise Benchmark (ATP)** Available-to-promise answers the question whether a customer order can be fulfilled at a specific due date regarding given past and future stock changes and other customer requests. Our implementation of an ATP algorithm has a set of fixed stock changes and a set of orders, both include a time and an amount of stock. The algorithm checks availability in an iterative, backtracking fashion. It simulates the progress in time and correspondingly applies the fixed changes. When an order is due it tries to satisfy it as soon as possible. If it is satisfied and it later turns out that there is a future fixed...
stock change which is rendered impossible by this order, the order is revoked and simulation starts again from the time of the order.

**Kaplan-Meier Estimator (KM)** The Kaplan-Meier estimator estimates the survival curve of a population based on a sample of lifetimes. Amongst other applications, it is used in medical research to determine the survival rates of patients after a specific treatment based on observed survival times. The estimator is mathematically defined and has a straightforward implementation as a product [11].

**Elo-Ranking** Given a set of competing players and a large amount of data recording which player or strategy outperformed or defeated an opponent, the Elo rating [7] puts a rating on each competitor, quantifying its overall performance. Using the rating of two competing players, a win chance can be predicted beforehand. The algorithm is used in competitive Chess and Go, but also for matchmaking in online games, where it needs to quantify player performances live to assign equally skilled opponents.

**Balance Aggregation** Sequential aggregations are often implemented by looping over an input set, modifying an internal state at each iteration. Our example involves computing an account balance while the input is only a set of transactions with positive and negative balance changes, with the addition that days with negative balance are counted. The additional criterion makes the algorithm difficult to express in terms of relational operators, as the decision whether to count the day or not depends on all records before this day.

Test transactions are drawn from a uniform distribution, e.g. over the interval $[-100, 100]$.

**Micro Benchmarks** We used three basic list traversal operations to compare the execution on columnar objects with the execution on arrays: aggregating a sum (Aggregate Sum), adding a fixed number to all elements (Map Addition), and extracting elements which fulfill a simple condition into a new list (Filter).

### 5.3 Benchmark Results

**Statistical Methods** We make no assumptions on the underlying distribution and provide normalized Tukey boxplots in Figure 5 to visualize the median and variation of the measured timings compared to unoptimized PyPy.

Exact median timings are given in table 2. Speedups are computed by dividing the median execution time from the respective platform by the median execution time of our columnar implementation. Confidence bounds of this statistic are given by the 2.5-th and 97.5-th percentile of the bootstrap distribution of the computed ratio.

**Analysis** From the benchmark results, we see that the columnar layout outperforms ordinary objects consistently for 1,000,000 or more instances. However, it is not faster when dealing with small input sizes (below 100,000 traversed records), but, except for the balance benchmark, this meets our expectations exactly and reflects what the columnar layout was intended for.

The balance benchmark is a particularly difficult case for control-flow observing optimizations like those in a tracing JIT, as its loop contains a condition with an activation likelihood which varies a lot while traversing the input. Assumptions on the probability of a certain branch being traversed are often ineffective. Also, non-negativity assumptions can hold for the beginning while being violated frequently at later stages of the input data. We see a wide confidence interval, which indicates high data-dependent variance with a considerable chance of improvement in certain situations, but not in general. We assume that, among others, two types of applications would suffer from switching to our layout: transactional applications, which select a few single objects and use most of their attributes; and technical modules, like web frameworks, which create heterogeneous object graphs. However, at this point we can not back these claims and further evaluation is needed.

The **microbenchmarks** provide clear evidence that the columnar runtime scales better than ordinary objects with increasing improvements over larger numbers of objects. The highest potential shows in the map operation, which effectively updates a full column without having an if-condition or maintaining an aggregate across multiple loop runs.

### 5.4 Integration with Object-Orientation

To evaluate the integration of the columnar layout into Python, we qualitatively describe the features of object-orientation supported by the new layout. We will thereby distinguish between working features of our prototype, limitations of the prototype and limitations of our concept. This does not cover all features of Python but focuses on features which are affected by our changed layout.

**5.4.1 Features of the Prototype**

**Object Identity** The identity of an object can be obtained via the `id()` function (usually an integer representing the memory address). We can override the `id()` function to return objects which compare equal for proxies representing the same instance (e.g. a tuple of class and instance ID).

**State and Methods** Our implementation uses the Python attribute facilities to translate attribute access. Therefore, columnar objects exhibit the same lookup behavior as ordinary Python objects. Both, attributes and methods, are defined in the class of an object.

**Inheritance and Polymorphism** Python supports multiple inheritance and polymorphism. Our prototype supports multiple inheritance of behavior in the style of traits, meaning that at most one columnar class may appear as base class and the rest is required to carry mere behavior. The steady construction and destruction of proxies retains the correct class relation and polymorphism works as expected.

---

205
Figure 5: A box plot of the performance benchmark results, normalized to PyPy median. The normalization is the quotient of the execution time of our approach and the time of a PyPy solution. Generally speaking, if the second box is higher than the first box, our approach is slower, if it is lower our approach is faster. The red line indicates the median, the upper and lower edges of the box are the second and third quartile and the end of the whiskers are the most outlying values in a 1.5 inter-quartile range distance from the second and third quartile.
We see no urgent need to support full dynamicity. We merely objects. However, as our scenarios involve homogenous data, significant difference to dynamically typed ordinary Python objects. This is the most significant difference to dynamically typed ordinary Python objects. Also, these types have to be explicitly stated in the form of mix-ins as it can be seen in listing 3. This is the most significant difference to dynamically typed ordinary Python objects. However, as our scenarios involve homogenous data, we see no urgent need to support full dynamicity. We merely provide a contract between programmer and runtime stating that a family of objects are in fact homogenous in attribute types. We could also use types observed during run-time instead of manual type annotations (see section 6.2).

Object Identity Another way to determine object identity is to compare objects using the is operator. We cannot adapt the is operator without modifying the VM. Therefore, proxies may compare non-identical using the is operator despite representing the same columnar instance.

Object-Specific State Object-specific attributes and methods are only defined on proxies and not stored in columns. Therefore, they are lost when the proxy is garbage collected. However, per-instance attributes and methods can be implemented using a global mapping from the object identity to a mapping from attribute names to state or functions.

Associations Associations are usually constructed in an ad-hoc fashion in Python, e.g. if an instance needs references to multiple other instances, some method creates a list where those references are stored. In contrast, our model requires to define the association and its multiplicity in the class definition (see listing 3). One-to-many associations produce a read-only (but not immutable) collection on instance-side, which cannot be replaced by an externally provided collection, but modified through methods with side-effects like append(). However, the interface and run-time complexity of that framework-provided collection can be influenced by specifying whether it should behave like a set, list or dictionary, thus the impact on code operating on these collections can be mitigated well. It is possible to store a columnar object in an ordinary Python object. The inverse is also possible, given the columnar class declares an Object attribute.

### 5.4.2 Limitations of the Prototype

**Typed Fields** We require that the same attribute of all instances of a class has the same primitive or columnar type. Also these types have to be explicitly stated in the form of mix-ins as it can be seen in listing 3. This is the most significant difference to dynamically typed ordinary Python objects. However, as our scenarios involve homogenous data, we see no urgent need to support full dynamicity. We merely provide a contract between programmer and runtime stating that a family of objects are in fact homogenous in attribute types. We could also use types observed during run-time instead of manual type annotations (see section 6.2).

### Table 2: Analytical algorithm benchmarks on the left side of the table and microbenchmarks on the right side. All median benchmark timings in milliseconds. Speedups given as ratio of medians with 95% confidence intervals

<table>
<thead>
<tr>
<th>benchmark</th>
<th>size</th>
<th>PyPy</th>
<th>Col.</th>
<th>timing [ms]</th>
<th>speedup</th>
<th>PyPy</th>
<th>Col.</th>
<th>timing [ms]</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ATP</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>1.32</td>
<td>28.05</td>
<td>0.05</td>
<td></td>
<td>1.41</td>
<td>1.44</td>
<td>1.49</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100000</td>
<td>21.79</td>
<td>48.41</td>
<td>0.45</td>
<td></td>
<td>1.41</td>
<td>1.44</td>
<td>1.49</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000000</td>
<td>235.71</td>
<td>228.91</td>
<td>1.03</td>
<td></td>
<td>1.41</td>
<td>1.44</td>
<td>1.49</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000000</td>
<td>2739.53</td>
<td>1902.78</td>
<td>2.69</td>
<td></td>
<td>1.41</td>
<td>1.44</td>
<td>1.49</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Aggregate Sum</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>0.59</td>
<td>2.99</td>
<td>0.20</td>
<td></td>
<td>1.41</td>
<td>1.44</td>
<td>1.49</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100000</td>
<td>28.65</td>
<td>17.89</td>
<td>1.00</td>
<td></td>
<td>1.41</td>
<td>1.44</td>
<td>1.49</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000000</td>
<td>535.12</td>
<td>174.31</td>
<td>3.07</td>
<td></td>
<td>1.41</td>
<td>1.44</td>
<td>1.49</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000000</td>
<td>4393.76</td>
<td>1631.58</td>
<td>2.69</td>
<td></td>
<td>1.41</td>
<td>1.44</td>
<td>1.49</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Elo</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>6.36</td>
<td>3.38</td>
<td>1.88</td>
<td></td>
<td>1.41</td>
<td>1.44</td>
<td>1.49</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100000</td>
<td>41.54</td>
<td>28.32</td>
<td>1.47</td>
<td></td>
<td>1.41</td>
<td>1.44</td>
<td>1.49</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000000</td>
<td>359.06</td>
<td>247.49</td>
<td>1.45</td>
<td></td>
<td>1.41</td>
<td>1.44</td>
<td>1.49</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000000</td>
<td>3506.49</td>
<td>2418.84</td>
<td>2.69</td>
<td></td>
<td>1.41</td>
<td>1.44</td>
<td>1.49</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Balance</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>0.11</td>
<td>0.16</td>
<td>0.70</td>
<td></td>
<td>1.41</td>
<td>1.44</td>
<td>1.49</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100000</td>
<td>1.06</td>
<td>1.67</td>
<td>0.63</td>
<td></td>
<td>1.41</td>
<td>1.44</td>
<td>1.49</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000000</td>
<td>18.22</td>
<td>17.70</td>
<td>1.03</td>
<td></td>
<td>1.41</td>
<td>1.44</td>
<td>1.49</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000000</td>
<td>119.85</td>
<td>170.17</td>
<td>0.70</td>
<td></td>
<td>1.41</td>
<td>1.44</td>
<td>1.49</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Metaprogramming** It is still possible to use metaprogramming without degrading performance, e.g. if a user-supplied attribute needs to be read, we can use the function `getattr(obj, user_attr)` and it will be eliminated during optimization if the attribute name stays constant inside a long-running loop.

As everything is manifested in application-level Python structures, full reflection over objects and classes is retained. The underlying columnar model can be inspected by accessing the column_attributes dictionary of the class, which adds to existing reflection capabilities.

**Tooling** The Python ecosystem provides tooling for several purposes, e.g. debuggers or editors with auto-completion. As the runtime was not significantly altered but only extended with a user library, these mechanisms still work for any Python code. They also work for columnar code as the new physical layout was merely introduced by using existing Python meta-programming features that are recognized by most tools.
5.4.3 Conceptual Limitations

Garbage-Collection (GC)  As classes are considered collections of their instances, an instance needs to be explicitly removed from that collection to be removed from the system. This means that our columnar objects are effectively excluded from automatic GC and need a separate algorithm.

Any GC algorithm for columnar objects suffers from a conceptual problem. To preserve performance, the GC algorithm has to avoid “holes” in columns caused by invalidated objects. Therefore, we have to reorder objects in columns and as a result we need to update the corresponding proxies. To be able to update the proxies we need to keep a reference on them which hinders the allocation removal optimization.

Further work is needed to find an algorithm which creates continuous sequences of living objects in columns on-the-fly while preserving lookup performance.

Generalization of the Approach  For our approach, we only considered a meta-tracing JIT. It allows us to use metaprogramming instead of an interpreter modification without losing performance, because only the resulting low-level control flow is considered. Therefore, we are confident that the approach generalizes to other meta-tracing runtimes, such as HippyVM4 for PHP or Topaz for Ruby4. A normal tracing JIT or method-based JIT could cause a much higher implementation effort, as the JIT itself might need to be aware of the new memory layout.

5.5 Summary of Evaluation

Our performance measurements show that analytical algorithms can generally benefit from a columnar object layout when traversing large amounts of data ($\geq$ 1,000,000 objects). As the optimization does not apply to all object-oriented algorithms, the programmer has to actively decide when a columnar layout is adequate. The qualitative analysis of our concept shows that most object-oriented features are supported and limitations of the prototype can be overcome by integrating these features into the virtual machine. A comparison of listing 3 and listing 1 illustrates how the JIT currently has to adjust the code to provide the type information required by the columnar layout. Regarding seamless integration of our columnar object layout into a dynamically-typed object-oriented language, the interface available to the programmer leaves room for improvement.

Overall, the results suggest that a columnar object layout provides performance benefits for analytical algorithms expressed with object-oriented abstractions of the application domain.

6. Related and Future Work

6.1 Related Work

The idea of columnar objects has already been applied to other types of runtimes and libraries.

6.1.1 Kedama

The Kedama [18] educational parallel programming system allows users to program “turtles”, a sort of agents that interact with their environment, organized as a grid. It is integrated into the eToys system, which itself is build on top of Squeak [10], a Smalltalk system. A group of turtles (“breed”) that share the same properties can be instructed to perform a collective action at the same time, implementing what is known as Single Instruction, Multiple Data (SIMD) in parallel programming.

In order to efficiently modify the state of a breed, the turtle properties are stored in columns. A property update on a breed gets compiled to a vectorized operation, which uses functions implemented in C (“primitives”) to process arrays. For example, Kedama provides primitives for arithmetic operations, such as adding two arrays. These functions are platform-level code and not editable by application developers, thus limiting them to the types supported by the primitives. This works well for the domain of simulations for educational purposes. In comparison, our approach targets applications in general and therefore allows the developer to use arbitrary classes and the JIT compiler optimizes the code to an efficient array operation automatically.

6.1.2 OOPAL

The OOPAL model [15] aims to extend the object-oriented model with concepts from array programming, as found for example in APL. In particular, the model extends the message dispatch to allow the expression of operations on sets of objects without explicitly stating any form of iteration. This concept is evaluated through an implementation in F-Script. This implementation is optimized e.g. by using so called ”smart arrays” which change their representation according to their content, e.g. double-precision numbers are stored in their native platform representation. As a result, method calls to elements of such an array can directly be mapped to native operations.

While an array programming interface would be a suitable extension to our approach, we aim to improve performance for large data sets without changing the dynamic object-oriented model. Nevertheless, our approach indirectly makes use of similar optimization techniques, as the smart arrays of the OOPAL implementation are similar to the storage strategies for collections in the PyPy JIT compiler.

6.1.3 Exploded Java Classes

Noth [17] proposed a modification to the Java language introducing the exploded keyword. Classes attributed with this keyword store their properties inside columnar arrays. Ex-
ploded objects can be used in specialized collections and it-
erators, which are generated by instantiating code templates
at compile-time. Field access, instance creation, and it-
eration on exploded classes and specialized collections also
undergo a source-to-source transformation to reflect column
access. Subclass polymorphism is implemented by maintain-
ing a type ID for each exploded instance and generating a
\texttt{switch} block with one case for each possible method im-
plementation. This implementation strategy results in a dis-
crepancy between the code specified by the programmer and
the code actually executed at run-time. In particular, this
might become an issue when debugging an application us-
ing exploded objects. Our approach does not alter the source
code before execution, but improves the performance by
re-interpreting the object-oriented execution as array-based
computation at run-time. Thus, during debugging or pure
interpretation, the code is executed as written down by the
programmer. Further, an exploded class must neither con-
tain associations to ordinary Java classes nor inner classes,
thereby limiting their composability with the Java object
model. Also, reflection and metaprogramming on exploded
instances are not supported.

\subsection{Bcolz}
The \texttt{bcolz} [2] Python library implements array and table ab-
stractions for in-memory analyses of large bulks of struc-
tured data. They make use of a columnar data layout and
column-wise compression to save memory and CPU cache
space and subsequently speed up read-intensive algorithms.
A special query interface can be used to execute some com-
putations with highly optimized compression-aware algo-
rithms. However, the library does not integrate with object-
oriented abstractions and does not use JIT-based optimiza-
tions.

\subsection{GemStone}
The \texttt{GemStone/S} system [13] is an object database which is
capable of running a full application. Due to seamless inte-
gration with the Smalltalk-80 language, there is no bound-
ary between application logic and database: Persisted ob-
jects can be queried using the Smalltalk collection proto-
col and handled inside domain logic as if they were native
heap objects. To our knowledge, there are no publications on
the fundamental implementation of Gemstone, and thus we
cannot provide an appropriate comparison to our approach.
However, instead of also providing database features, like
persisting objects, establishing transaction boundaries and
versioning, we are merely focused on improving analytical
algorithm performance.

\section{Future Work}
Based on our proposal to implement a columnar object lay-
out, we see several remaining opportunities for improving
runtimes with database technology.

\subsection{Optimized Collection Protocols.}
Python’s collection protocol, including built-in operations
like \texttt{map, filter, reduce}, and list comprehensions, can be
optimized for columnar data. Thus, we are currently working
on a prototype collection protocol that transforms the inner
Python expressions into a query plan. We can optimize this
plan similar to the optimizations applied by an SQL query
optimizer. Based on this, we can map the resulting algorithm
onto faster operations working directly on the columns.

\subsection{Sharing Columns with an In-Memory Database}
Our efforts to improve runtime performance for data-heavy
algorithms are also part of a project concerned with im-
proving the interface between databases and object-oriented
runtimes. For instance, if objects can be implemented in an
object-oriented runtime in a similar structure as data is stored
inside an in-memory database, the interface between them
could be vastly different from the current ones (i.e. ORM or
stored procedures). For example, shared memory between
runtime and the database could allow the runtime to map
objects to native database data directly, given that security
and transactional properties can still be maintained.

Therefore, shared data allows one to manipulate database
data through objects, while also using optimized database
operations. For example, when filtering a set of objects, in-
stead of using the built-in generic \texttt{filter} operation, the ex-
ecution environment could map it to the database operation,
optimized for the database data layout.

\subsection{Columnar Runtimes}
Another opportunity is a dynamic object-oriented execution
environment based completely on a columnar object layout.
In particular, it will be interesting to see the performance
trade-offs resulting from such an approach. To assess this,
we need detailed benchmarks on the impact of a columnar
layout on the performance of typical object-oriented appli-
cations. Suitable benchmarks are “Richards” or “Deltablue”.

A hybrid solution might create interesting opportunities.
Currently the programmer has to decide which object layout
fits the anticipated access patterns best. Manual optimiza-
tion is an extra effort and should be offloaded to the runtime
whenever possible. The question remains whether it is fea-
sible for the execution environment to automatically switch
between object layouts, based on observed access patterns.

\subsection{Transactional Object-Oriented Runtimes with
Persistence}
There has been progress towards software-transactional
memory (STM) in PyPy [14], which could be the founda-
tion for a scalable execution environment for large data sets.
We might be able to move important database functionality
into the runtime itself. One major challenge is an implemen-
tation of a transactional persistence on top of transactional
objects, only causing minimal overhead. The ideal execution
environment would merge the functionality of databases and
traditional runtimes, resulting in a system similar to Gemstone/S. As a result, the programmer would not have to switch the paradigm at all and can program in one development environment, i.e. one language and one set of tools.

7. Conclusion

To mitigate the performance deficiencies of dynamic object-oriented runtimes regarding analytical workloads, we introduced a column-oriented object layout which leverages tracing JIT technology to execute object-oriented code on columnar data structures. We developed an interpretation of object identity, associations, attribute access, and collections in terms of the columnar object layout. We have demonstrated the feasibility of our approach with a prototype implemented in PyPy. Performance measurements with this prototype showed that analytical algorithms running on columnar objects perform significantly better than running on native objects. The dynamic object-oriented mechanisms and concepts remain largely unchanged. Overall, our approach contributes to the ways programmers can be relieved of the task of manual optimization.

References