

# Declarative query processing in imperative managed runtimes

Stratis D. Viglas

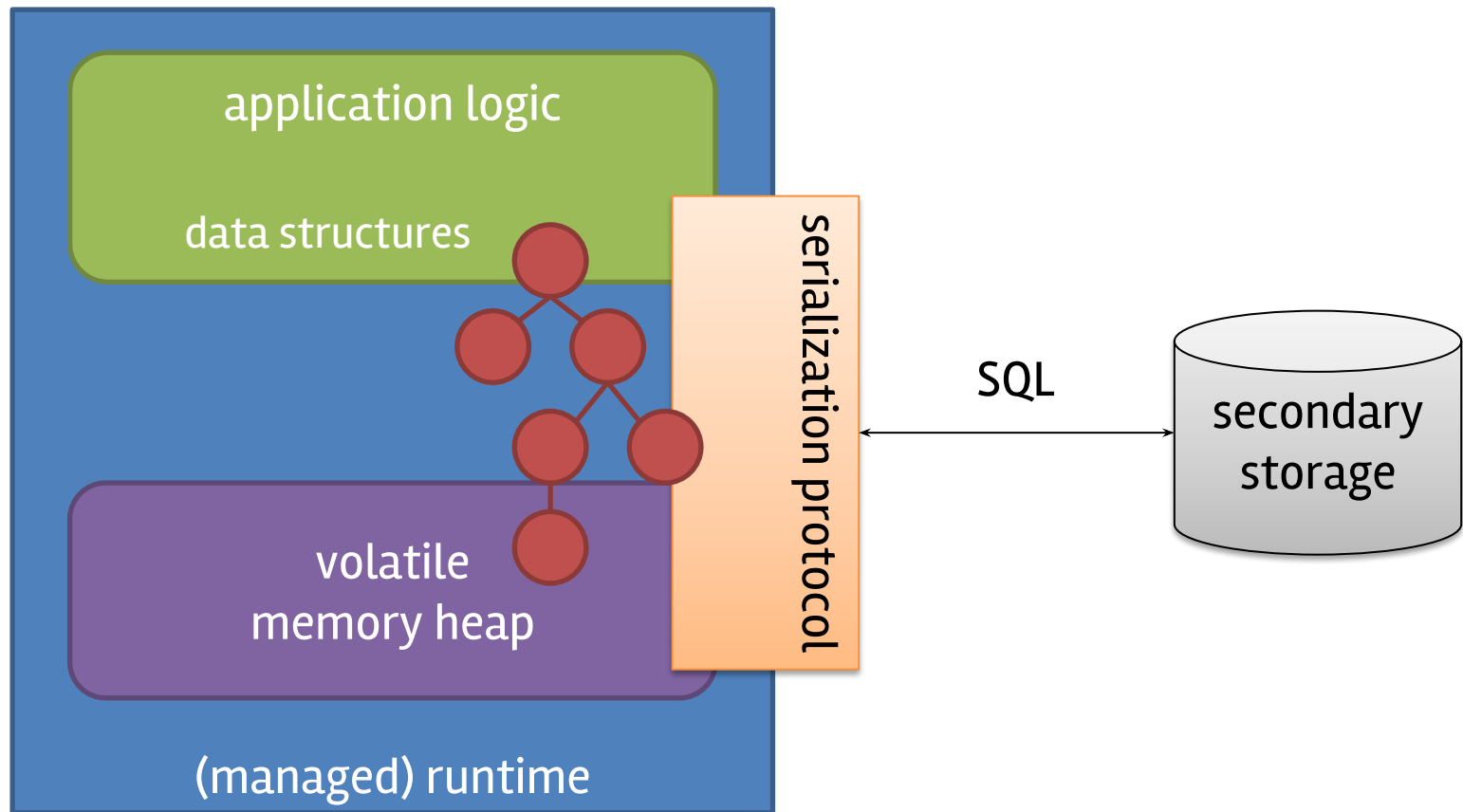
Google, US & School of Informatics, University of Edinburgh, UK  
sviglas@google.com

Active/HardBD 2017

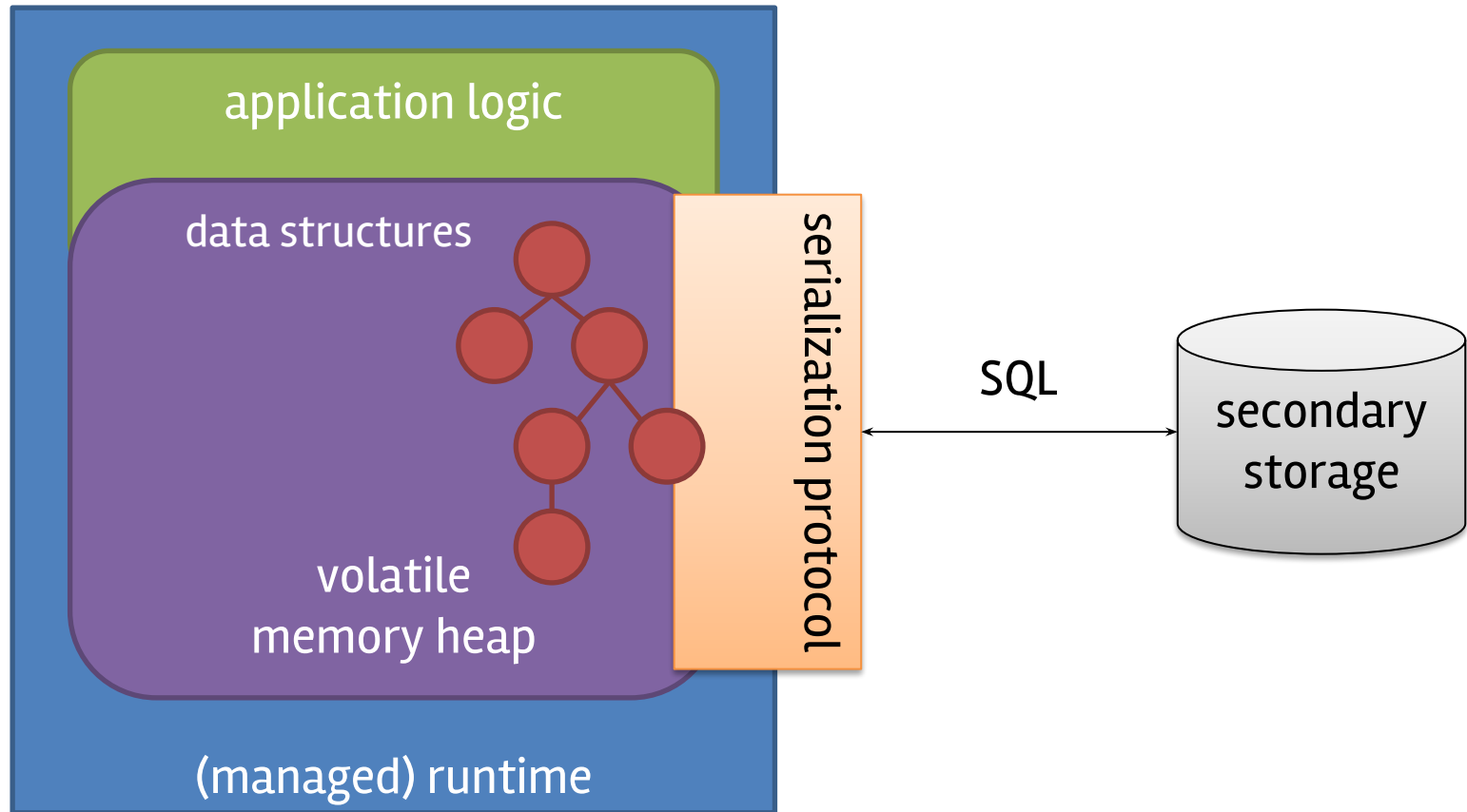


THE UNIVERSITY of EDINBURGH  
**informatics**

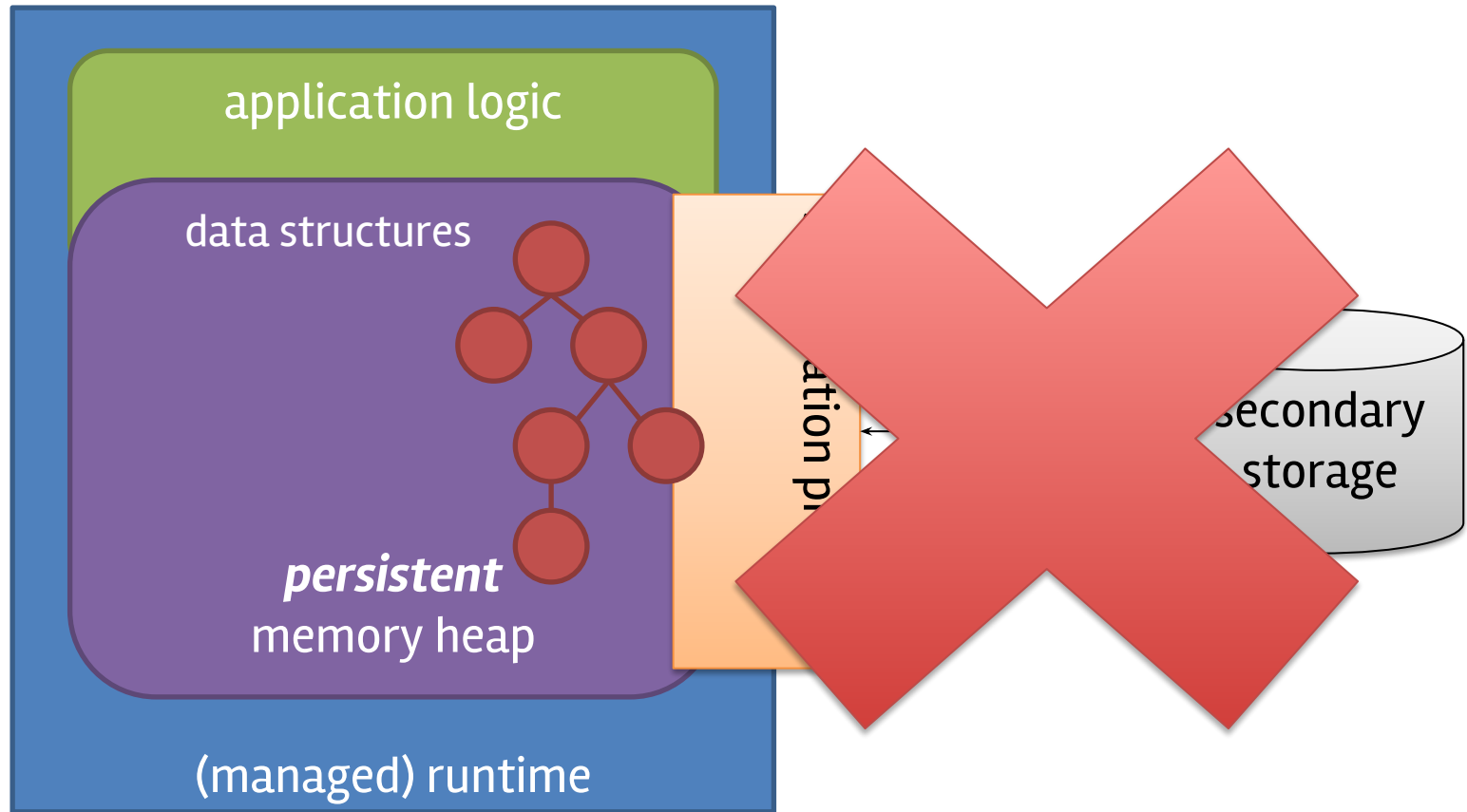
# Multi-tier applications



# What's changing?



# What's changing?



# In this talk

- Code generation for just-in-time query compilation
  - Starting from compiling SQL to C
  - Moving on to managed runtimes and language-integrated queries
- Write-limited algorithms for persistent memory
  - Staging algorithms for query processing
  - API to enable dynamic optimization
  - A runtime to support the API

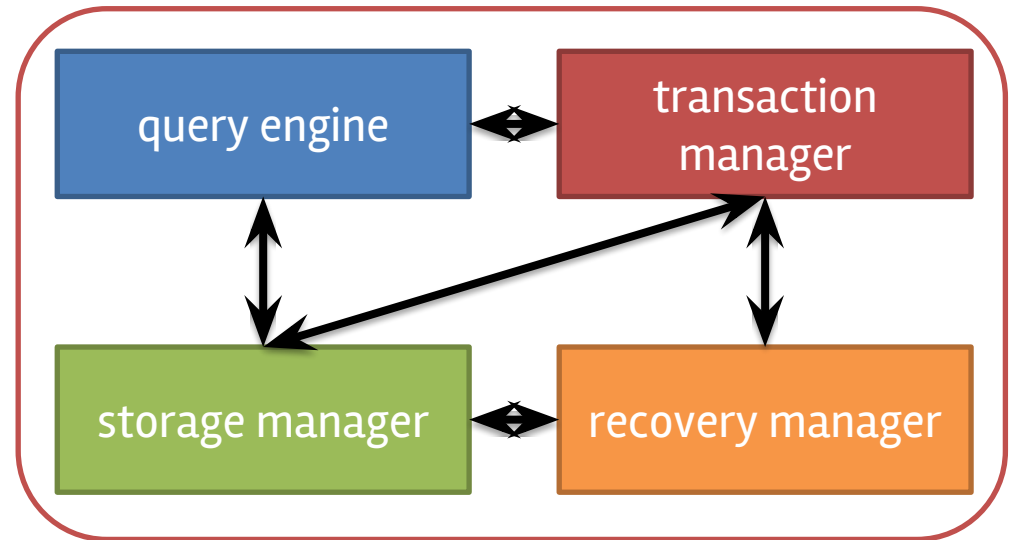
Part I

# Just-in-time code generation for query processing

# Database systems architecture

- Roughly decomposed into four main building blocks

- Query engine
- Storage manager
- Transaction manager
- Recovery manager

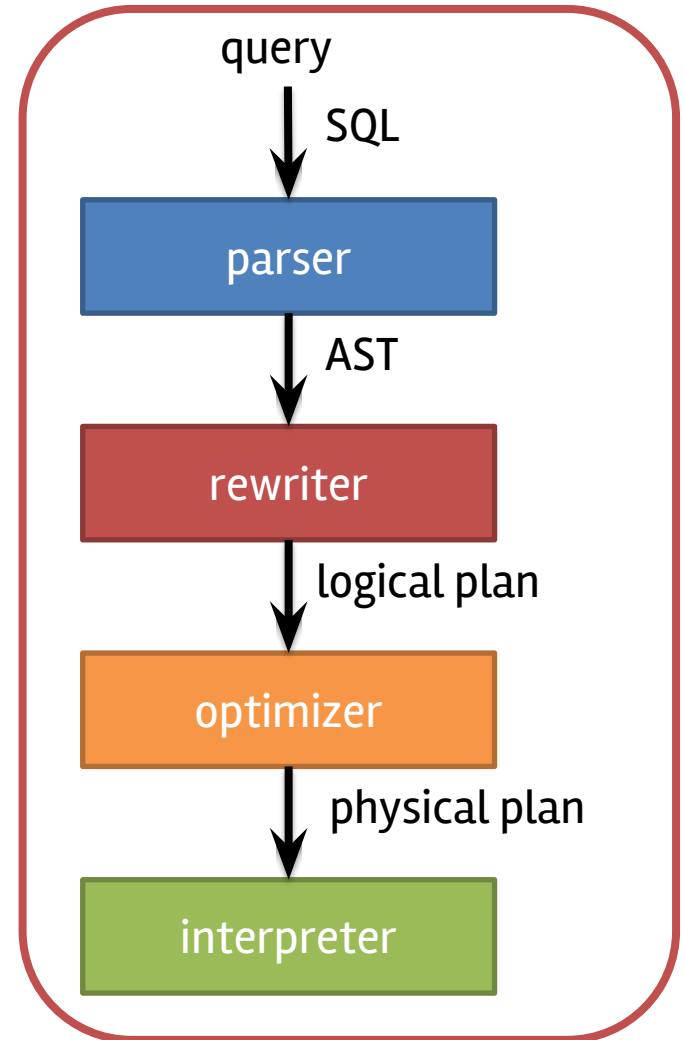


- Relatively orthogonal aspects

- Improvements in one block improve the system overall
- Or, at least, we try to abide by that rule

# Zooming in to query processing

- Queries go through a sequence of transformations
  - Parsing
    - SQL to abstract syntax tree (AST)
  - Rewriting
    - AST to logical plan
    - Potentially more than one rewriting passes
  - Optimization
    - Logical plan to physical plan
- Interpretation-based approach
  - Query engine interprets the query plan to produce results

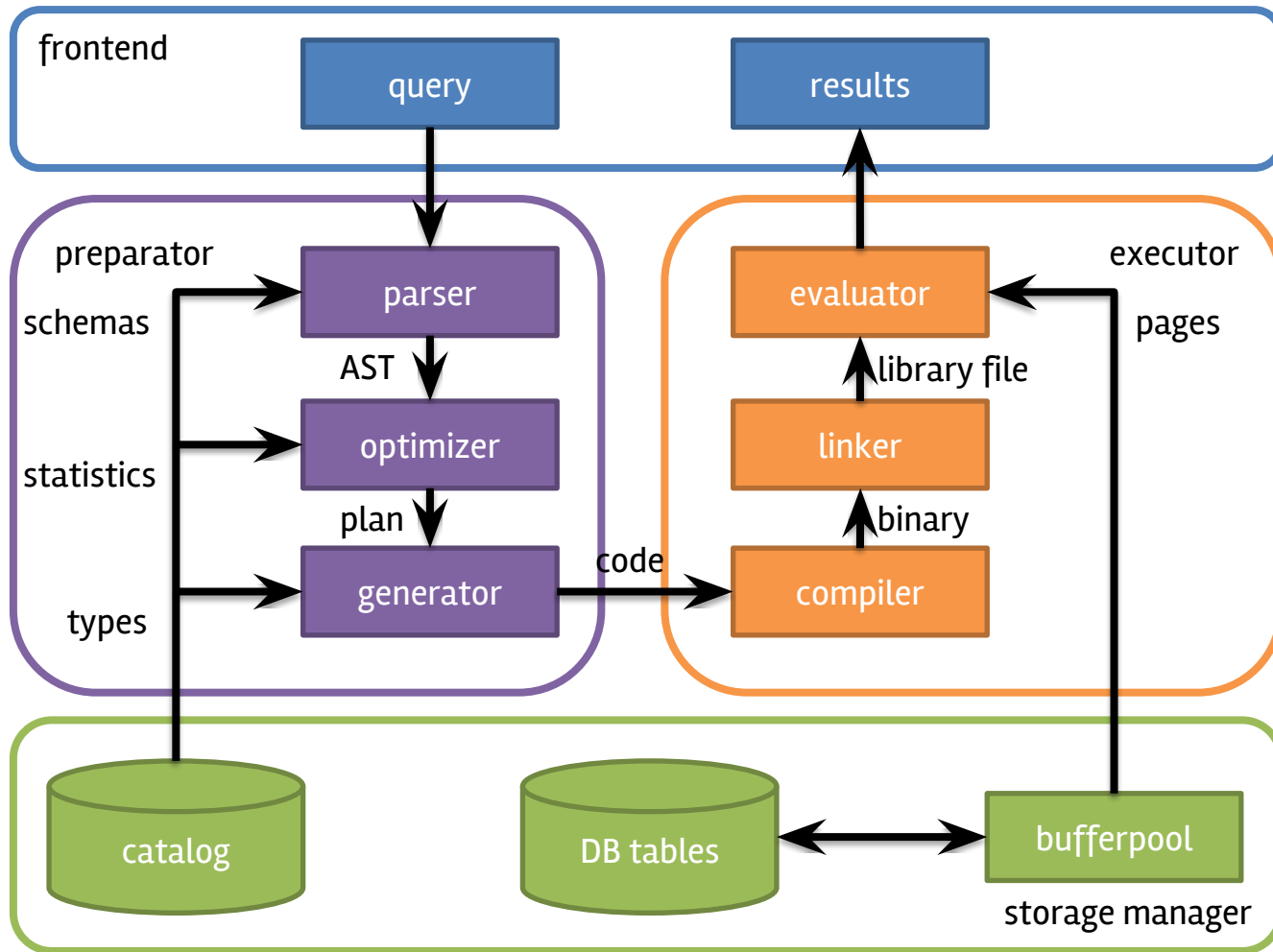




# Holistic techniques

- Template-inspired approach
  - Languages like C++ generate type-specific code in their standard library
  - Reduces bloat of generic implementations
  - Operators are templated and instantiated per query
- At the same time, look at the query holistically
  - Collapse operations when possible
  - Generate type-specific code
  - Eliminate function calls apart from the necessary
  - Source to source transformation: from SQL to C
- Leave orthogonal aspects of the system unaffected
- Treat SQL truly as a managed runtime with just-in-time compilation capability

# HIQUE – the Holistic Integrated Query Engine



# Example generated code

```
/* Inlined code to stage inputs */
hash: /* examine corresponding partitions together */
for (k = 0; k < M; k++) {
  /* algorithm bookkeeping */
  /* loop over pages */
  for (p_1 = start_page_1; p_1 <= end_page_1; p_1++) {
    page_struct *page_1 = read_page(p_1, partition_1[k]);
    for (p_2 = start_page_2; p_2 <= end_page_2; p_2++) {
      page_struct *page_2 = read_page(p_2, partition_2[k]);
      ...
      for (p_m = start_page_m; p_m <= end_page_m; p_m++) {
        page_struct *page_m = read_page(p_m, partition_m[k]);
        /* for each page loop over tuples in the page */
        for (t_1 = 1; t_1 <= page_1->num_tuples; t_1++) {
          void *tuple_1 = page_1->data + t_1 * tuple_size_1;
          for (t_2 = 1; t_2 <= page_2->num_tuples; t_2++) {
            void *tuple_2 = page_2->data + t_2 * tuple_size_2;
            int *t1 = tuple_1 + offset_1;
            int *t2 = tuple_2 + offset_2;
            if (*t1 != *t2) {
              merge: /* update bounds for all loops */
              continue; }
            ...
            for (t_m = 1; t_m <= page_m->num_tuples; t_m++){
              ...
            } ...}}}}...}}}}

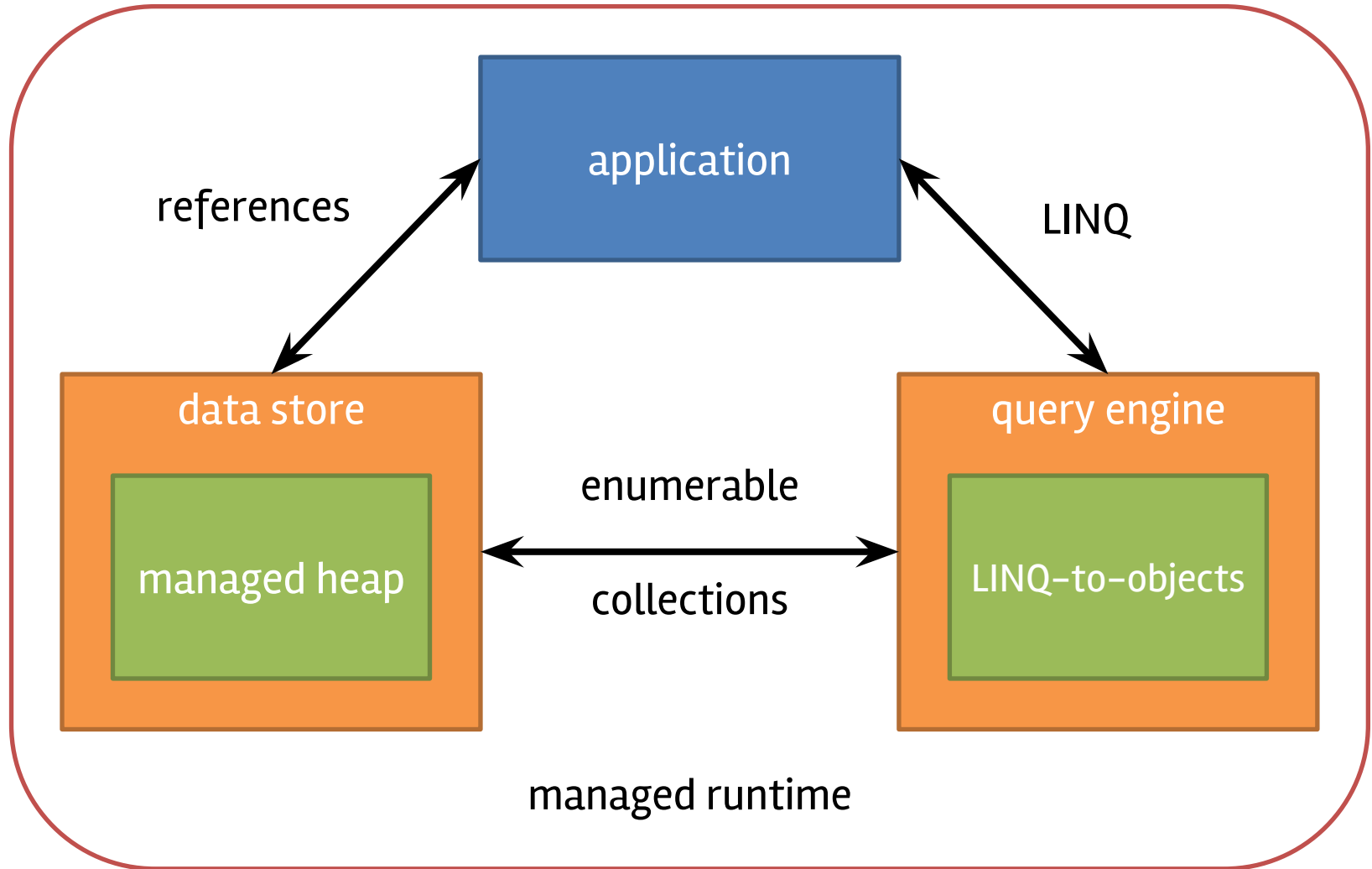
```

nested loops

fixed strides

type-specific computation

# Language-integrated query (C#)



# LINQ-to-objects in more detail

```
List<Order> orders = new List<Order>();
```

```
// Add data elements to order
```

```
var qry_stmt = orders  
    .Where(o => o.orderdate > new DateTime (1/1/1999))  
    .Select(o => o.price * (1 - o.discount));
```

```
foreach (var r in qry_stmt) {
```

```
    // Consume query result
```

```
}
```

query statement  
declaration



query execution



# Standard execution

```
IEnumerable<T> Where<T> (  
    this IEnumerable<T> src,  
    Func<T, bool> pred {  
        foreach (T s in src) {  
            if (pred(s))  
                yield return s;  
        }  
    })
```

foreach

select

where

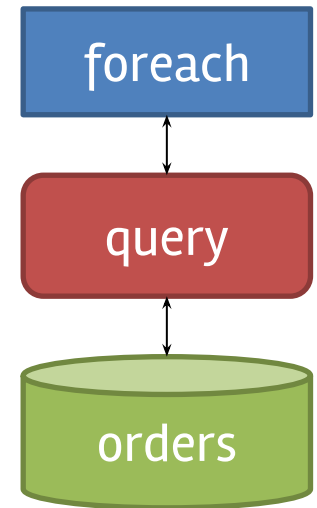
orders

- Virtual function calls to propagate objects through pipeline
- Lambda expression calls to allow generic implementations
- Compiler cannot inline because target not known at compile time

# Query compilation

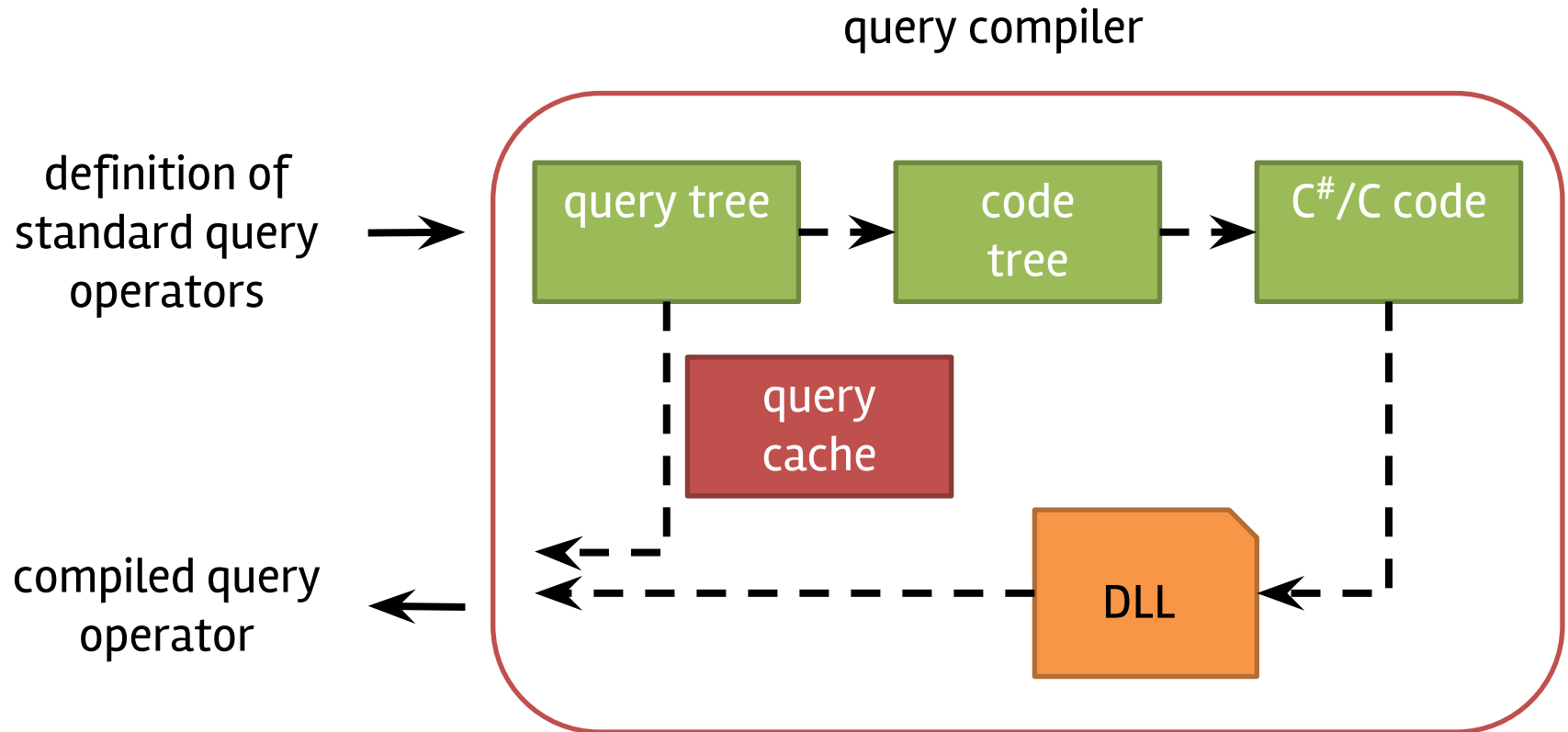
- Dynamically compile queries at run-time
  - Single, specialized operator that evaluates the entire query

```
IEnumerable<Decimal> Query (List<Order> src) {  
    foreach (Order s in src) {  
        if (s.orderdate > new DateTime (1/1/1999))  
            yield return (s.price * (1 - s.discount));  
    }  
}
```



# Compilation architecture

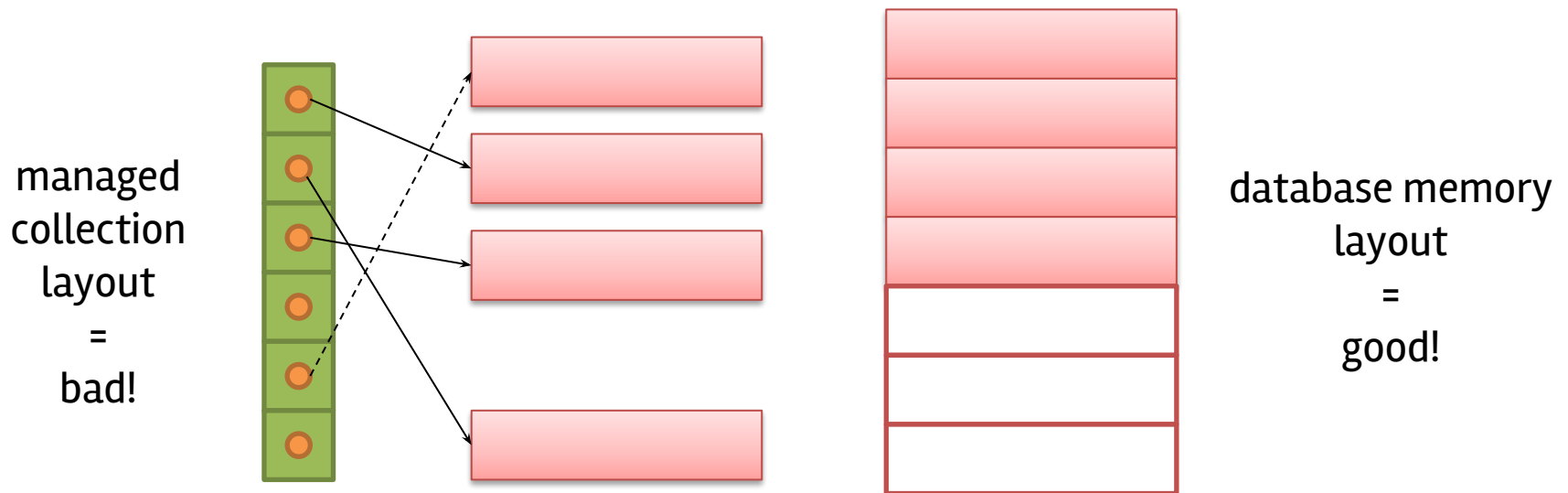
- Query compiler is implemented as a LINQ query provider





# The bad news

- Basic approach is limited by performance of C#
- Relies on (cache) inefficient memory layout dictated by garbage collection

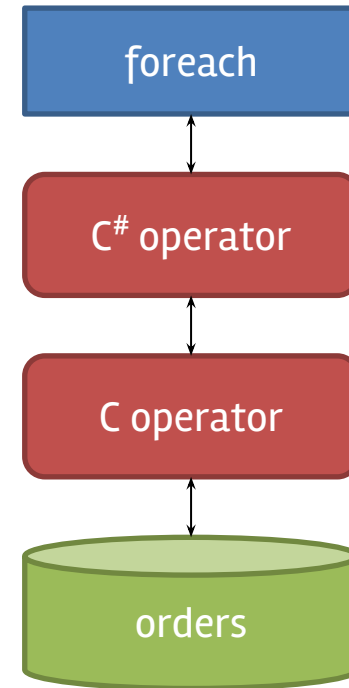


# Planning ahead

- Preferably we would like to perform query processing in native C code and have control over data layout
  - Not possible to access managed objects in C
  - Not possible to control data layout of objects
- But: structs are value types (in C#) and, hence
  - Are not managed by garbage collector
  - Allow some control over data layout

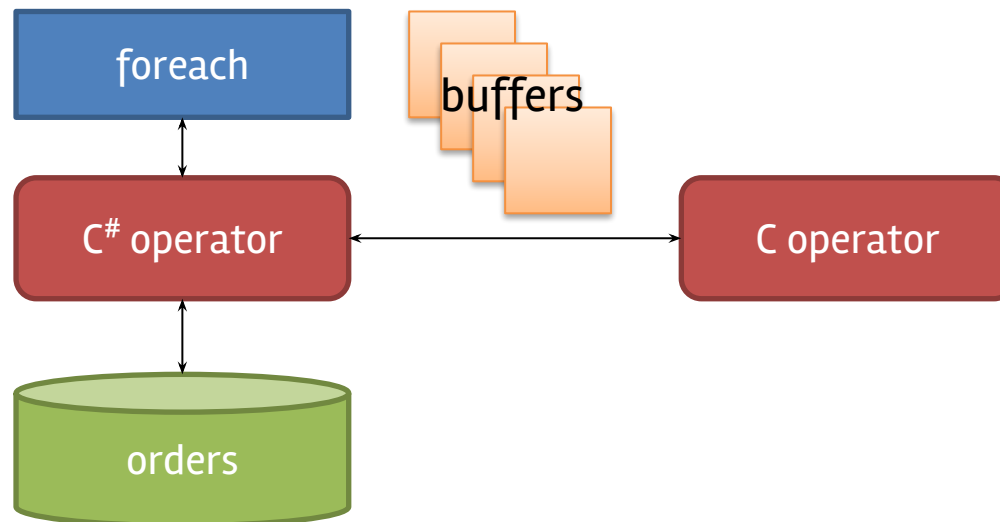
# Adding more C into C#

- Represent dataset as arrays of structs
- Dual operator approach:
  - C# operator interacts with application code by returning query result
  - C operator processes query on arrays of structs



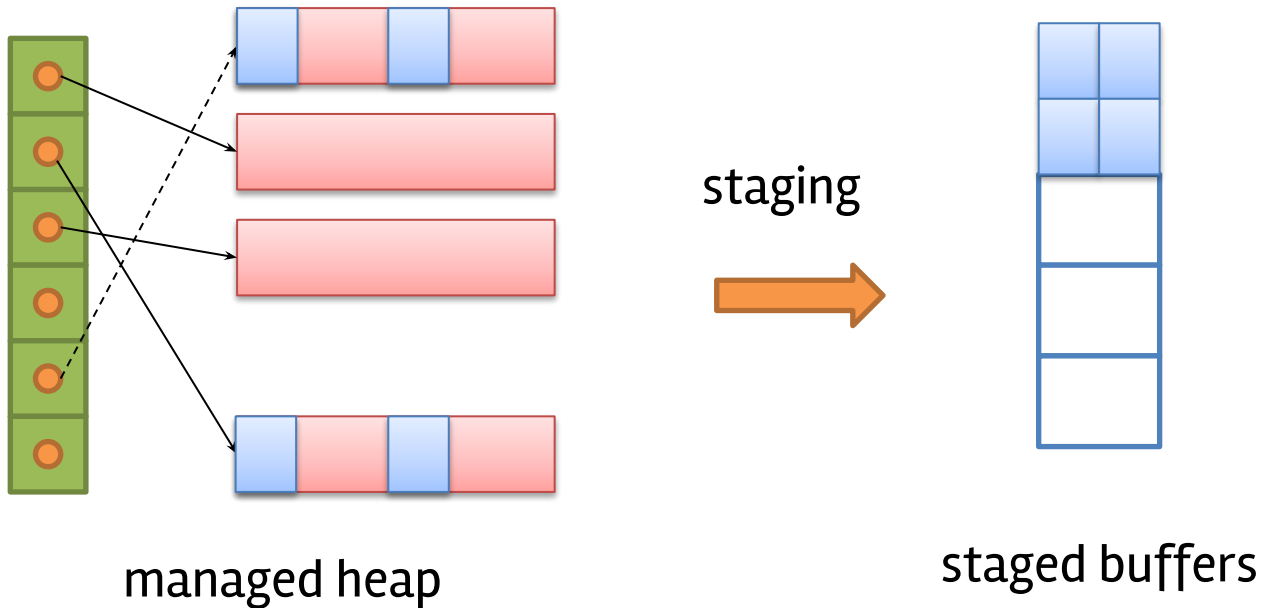
# Staged query processing (from C# to C and back)

- Store data as collections of objects
- Stage data in C# (as arrays of structs) and perform heavy-lifting of query in C on staged data
- Fall back to basic approach for simple operations



# Staging in more detail

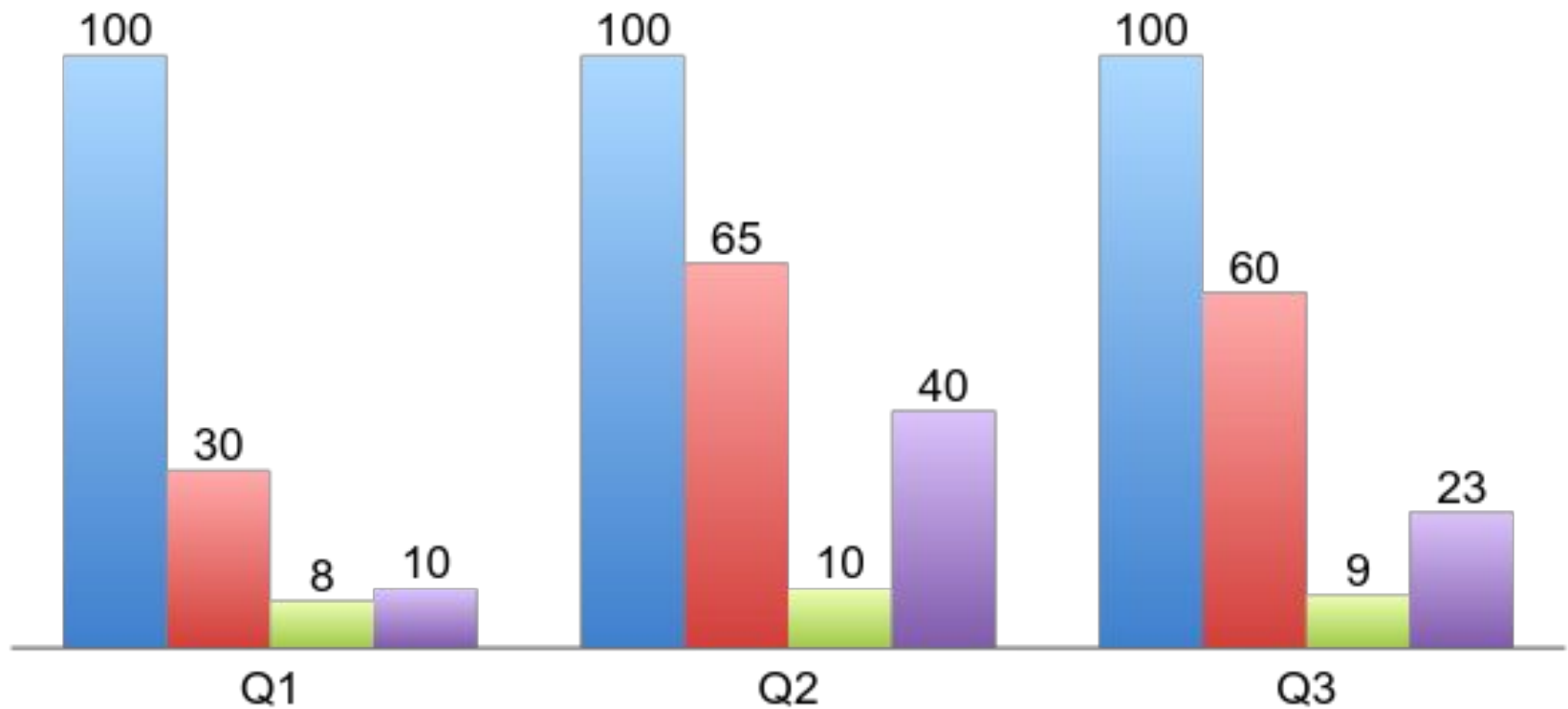
- Apply selections (fewer elements copied)
- Apply implicit projections (fewer fields copied)
- Flatten-out nested objects (removes references)



# Indicative results over TPC-H

## Normalized response time

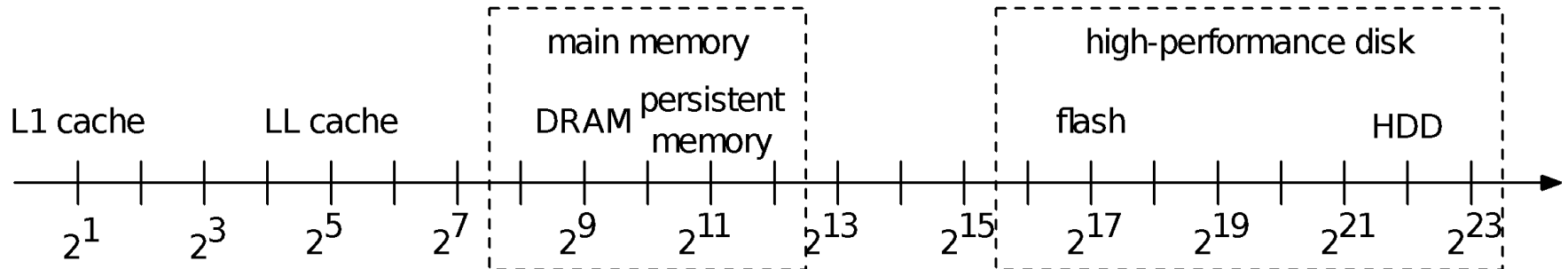
■ LINQ-to-objects ■ Compiled C# ■ Compiled C ■ Staged



Part II

# Write-limited algorithms for persistent memory

# Properties of persistent memory



- Latency comparable to DRAM
  - But not DRAM
- Asymmetry: writes more expensive than reads (up to 15x)
  - Similar to flash memory; much faster overall, but more pronounced asymmetry
- Not a block device
  - Byte-addressable, behaves as memory
  - Potentially accessed through CPU loads and stores
  - Game-changing property



# Incorporating persistent memory

- Persistent memory bridges the gap between disk and memory
    - Universal device, universal optimization objectives
  - But how should it be treated?
    - As byte-addressable, albeit somewhat slower memory?
    - Or as block-addressable but faster persistent storage?
    - Neither? Both?
  - What is the impact on system aspects?
- This work
    - Optimization of fundamental query processing algorithms and a runtime to support them

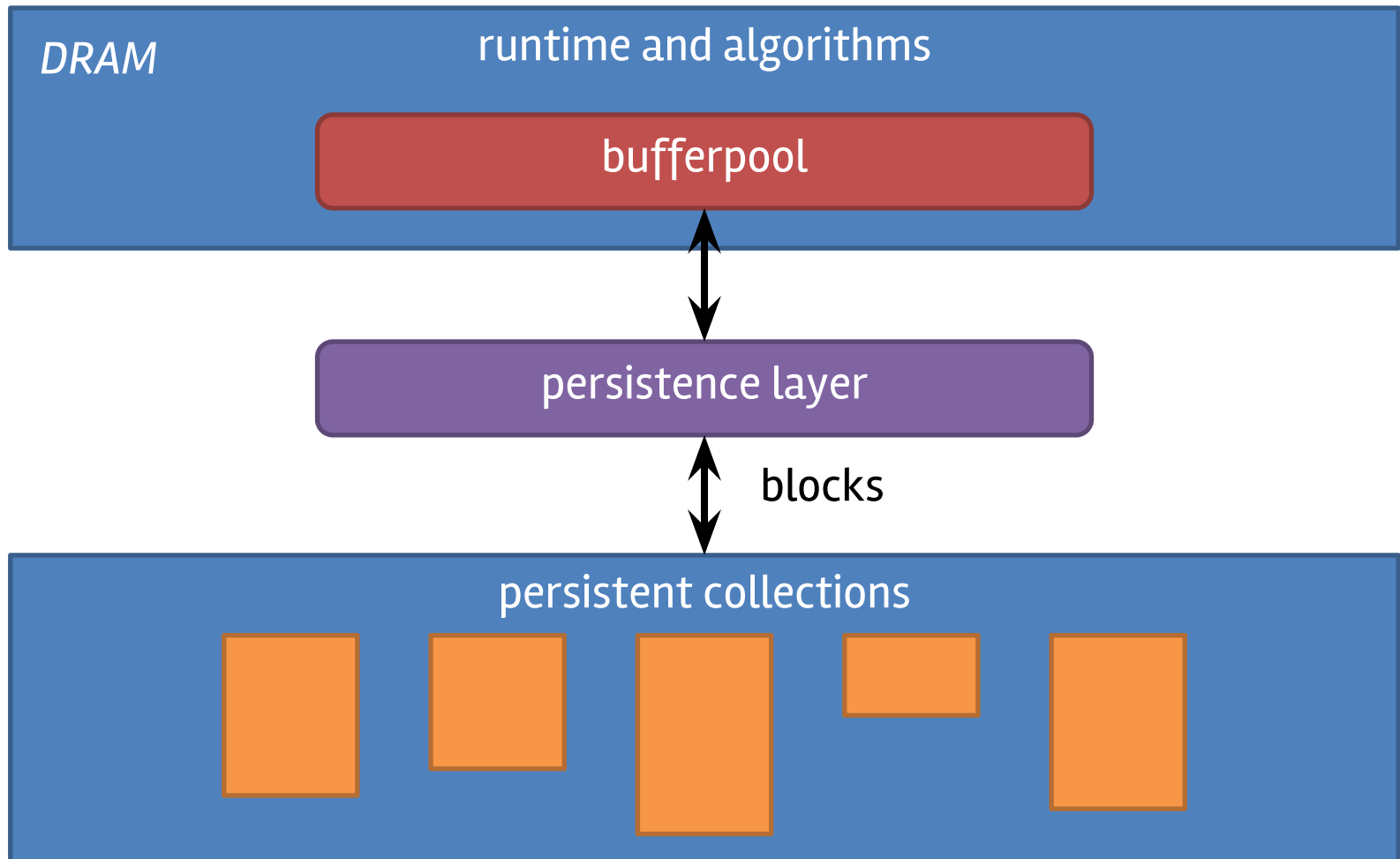
# In more detail

- Design and implementation of persistent-memory-friendly algorithms for query processing
  - And a runtime to support them
- Focus on two fundamental operations
  - Sorting and join processing
- Why these two?
  - Well, we are doing databases after all!
  - But the goal is farther-reaching
- Write-limited algorithms
  - Trade writes for reads with tunable write-intensity
  - Guarantee when they outperform existing algorithms

# General setup

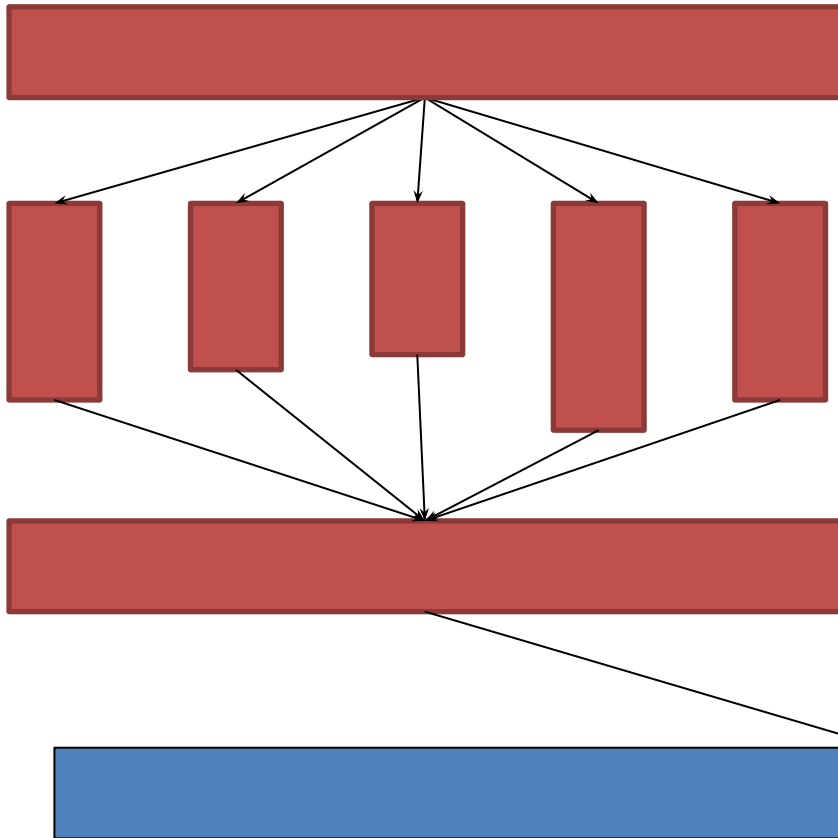
- Overarching goal: trade writes for reads
- Persistent memory I/O takes place in cacheline-sized units (termed buffers)
- Under the assumption there is a ratio  $\lambda = w/r$  where  $w$  is the write cost of the medium;  $r$  is the read cost;  $\lambda > 1$
- Two general classes of algorithms
  - Split processing into a write-incurring and a write-limited part; or
  - Process lazily by performing extra reads and incur writes only when the accumulated read cost is too high

# System overview

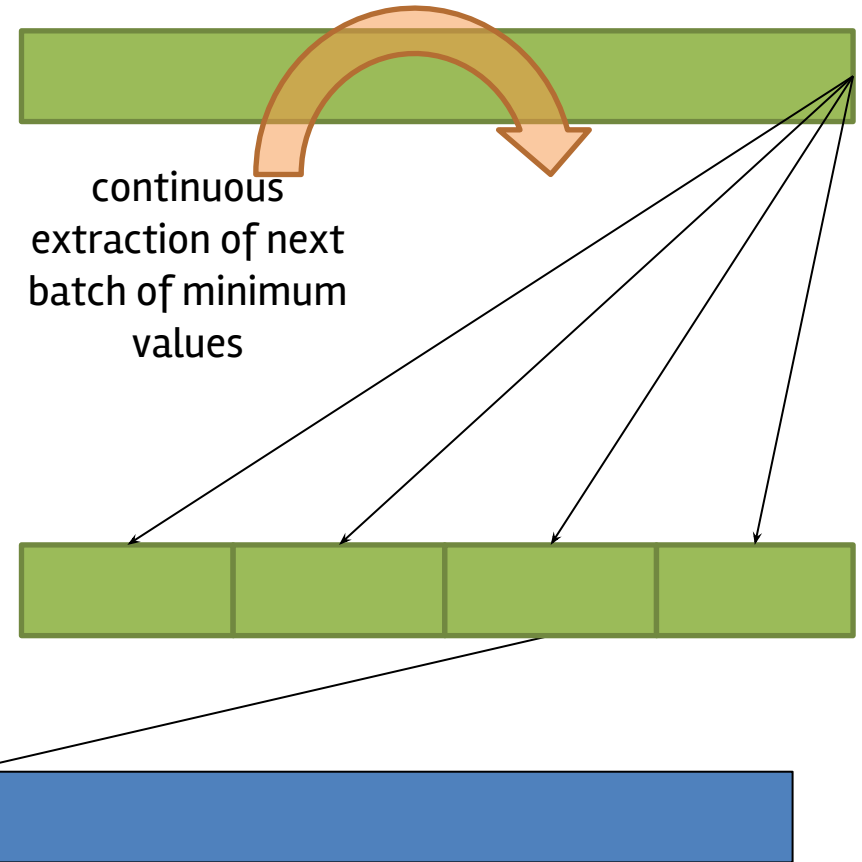


# Limiting writes in sorting: segment sort

write-incurring mergesort on  $x\%$

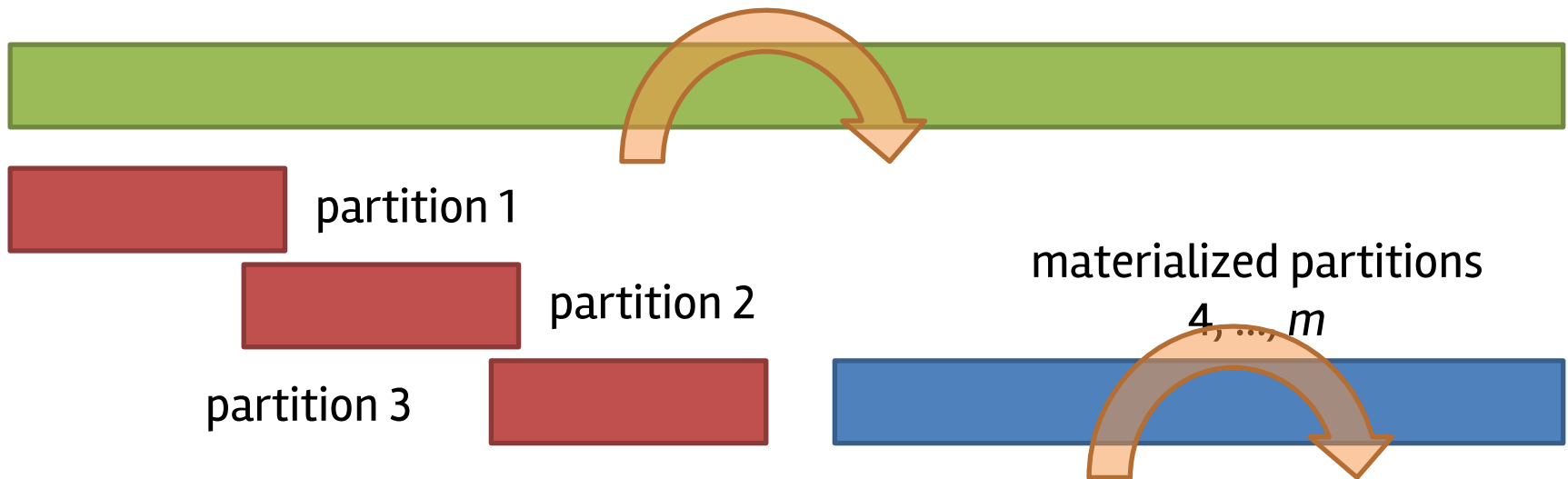


read-only selection sort on  $(1-x)\%$



# Limiting writes in join processing: lazy join

- Objective: process input one hash partition at a time
- Instead of scanning and materializing the partitioned input
  - Extract each partition by rescanning the entire input
  - Keep track of saved cost (by not writing) and penalty (by rescanning)
  - Materialize when cost exceeds savings



# Runtime support: procrastination is bliss

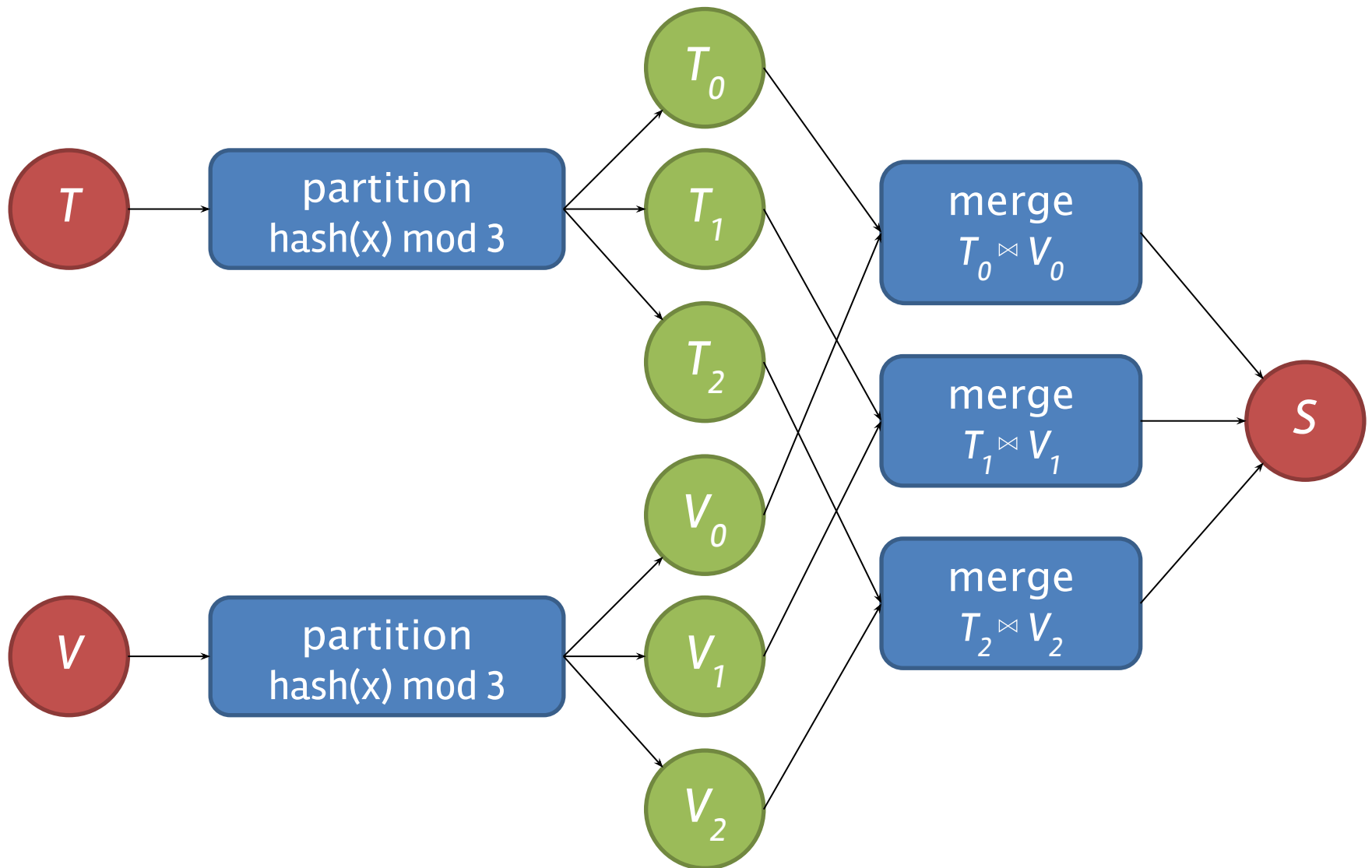
- Each operator belongs to an operator context
- Express algorithms in terms of a common API
  - Record the workflow in a control flow graph
- Do not materialize any collection until it is accessed
  - Upon access, `assess()` it to see if it should be materialized
  - If collection is to be materialized, `produce()` it by walking the control flow graph
  - If not, go to the last materialized parent and apply recorded operations dynamically to produce

# An API for recording algorithmic workflow

- $\text{split}(T, n, T_l, T_h)$ 
  - Split collection  $T$  at position  $n$  into  $T_l$  and  $T_h$
- $\text{partition}(T, h(), k, [T_i], [s_i] = |T|/k)$ 
  - Partition collection  $T$  into  $k$  partitions  $T_1$  to  $T_k$  using  $h()$  as the partitioning function
  - Size of each partition expected to be  $s_1$  to  $s_k$
  - Last argument optional and reverts to  $|T|/k$
- $\text{filter}(T, p(), f, T_p)$ 
  - Filter collection  $T$  into  $T_p$  using predicate  $p()$
  - Output size expected to be  $f|T|$  (where  $f \in [0, 1]$ )
- $\text{merge}(T_l, T_r, m(), T)$ 
  - Merge collections  $T_l$  and  $T_r$  into  $T$  using  $m()$  as the merging function

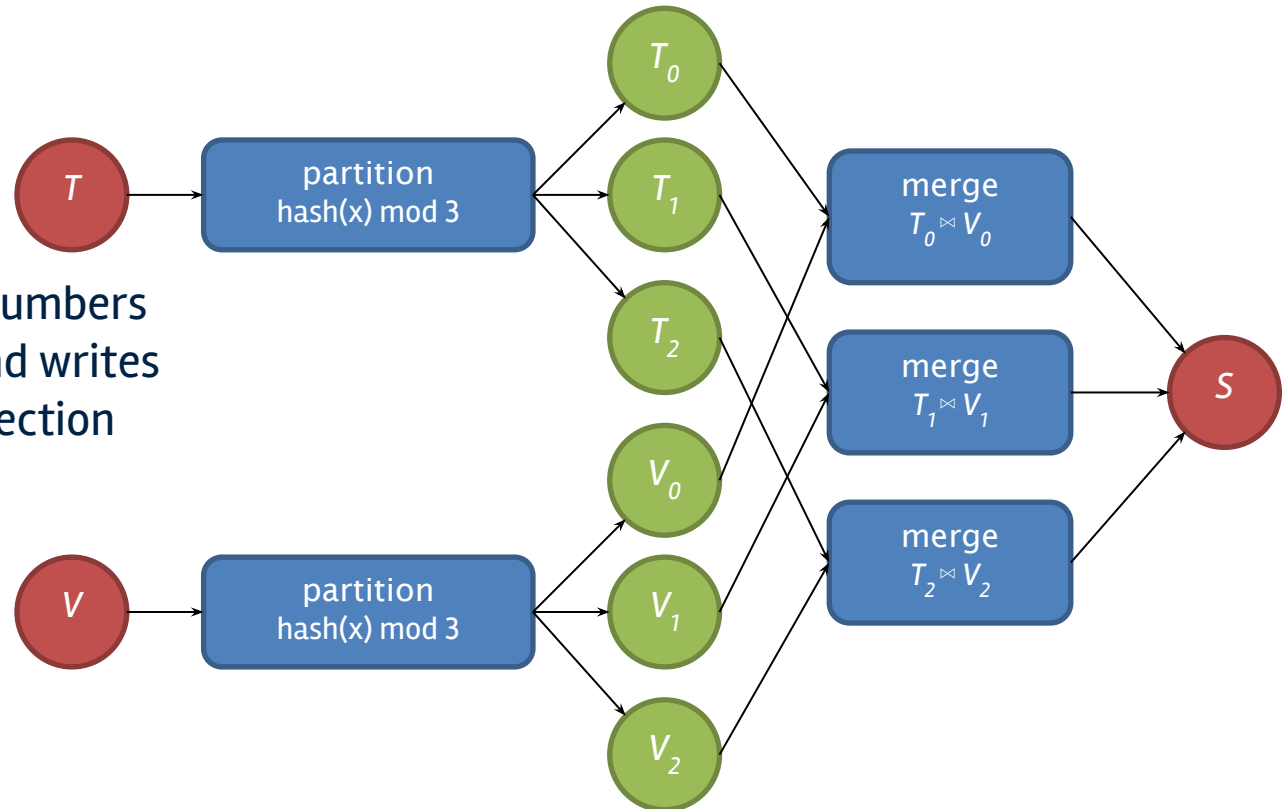


# Example control flow graph



# Optimizing the workflow

- Track accumulated numbers of cache line reads and writes per materialized collection

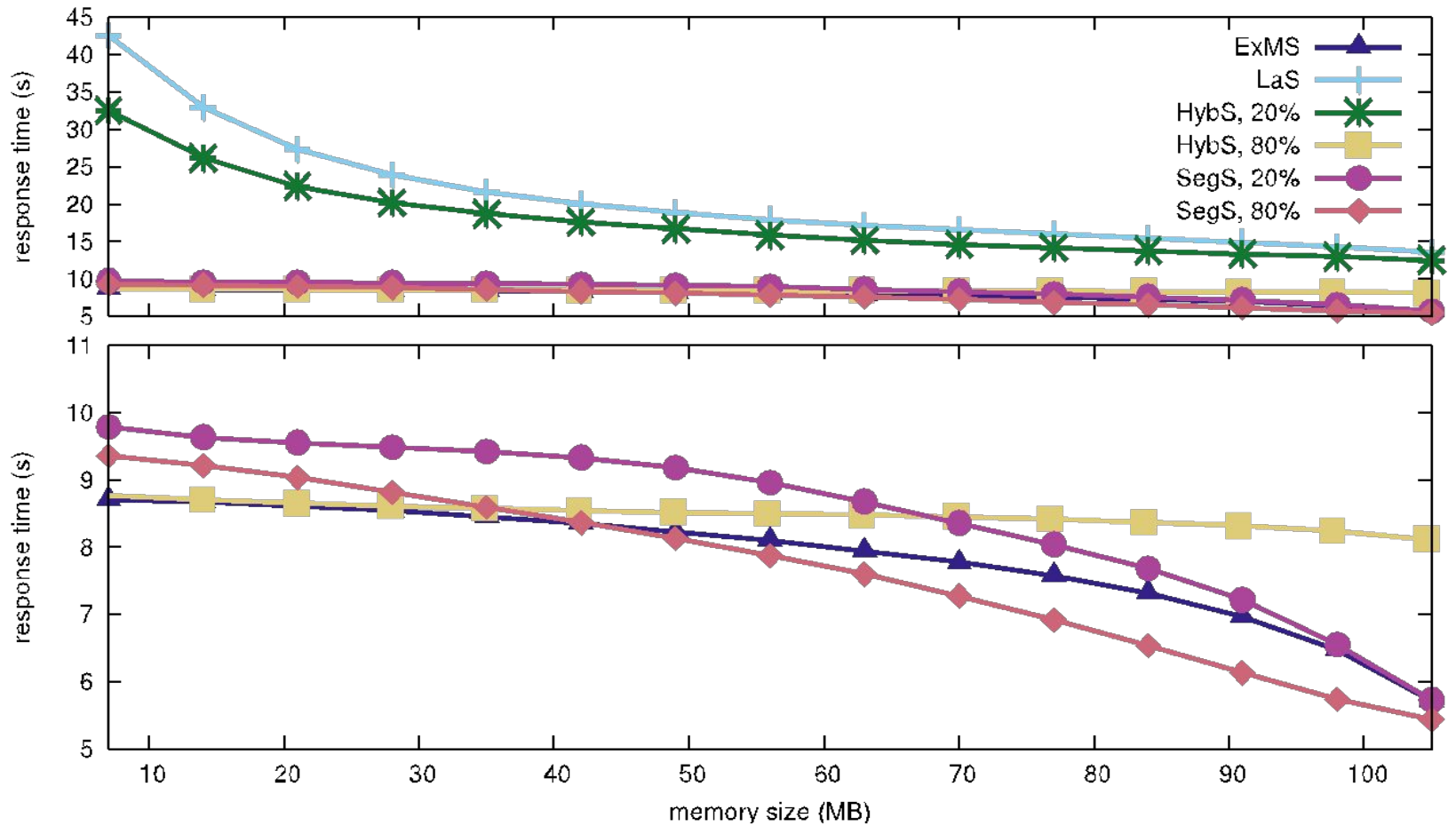


- Use the sum to decide whether cheaper to keep subsequent collection deferred or materialize
- Trigger materialization using rules based on heuristics for access pattern detection

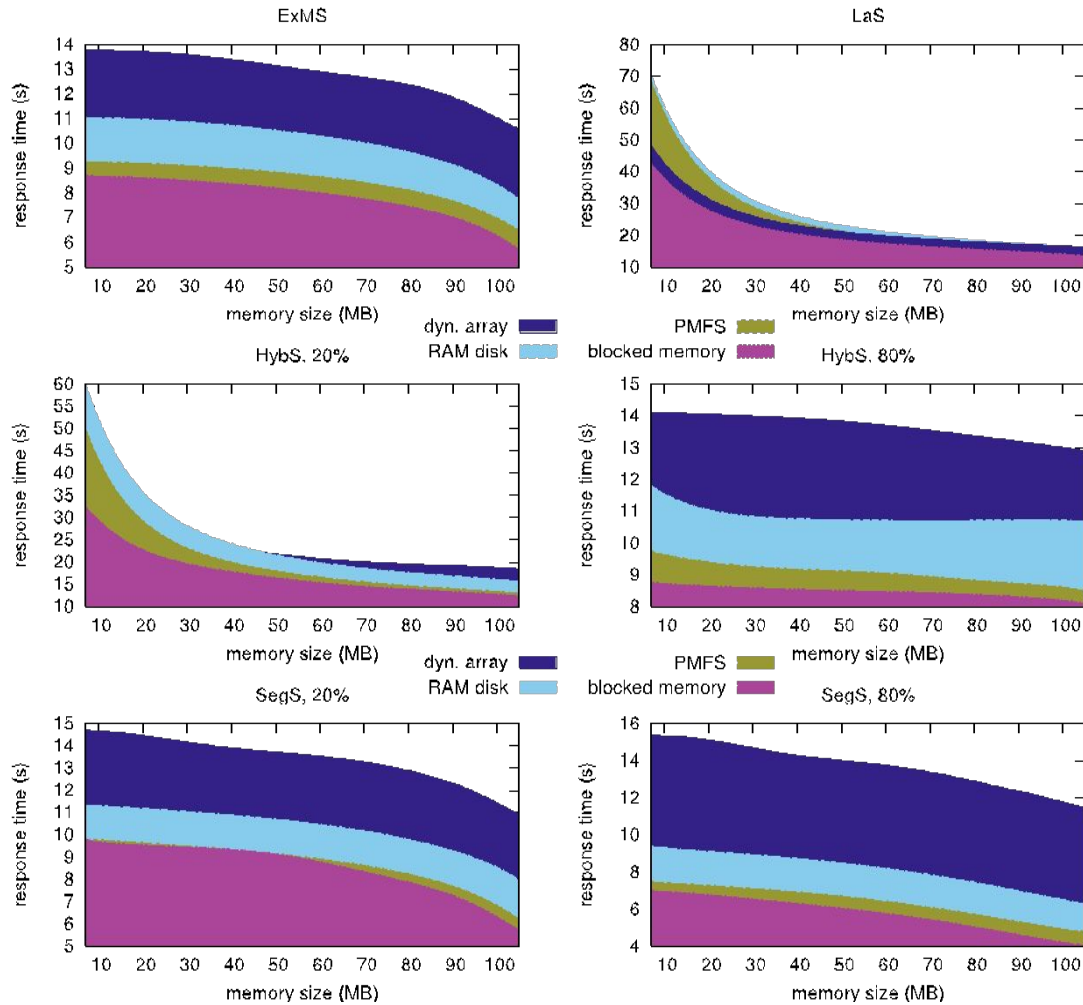
# Implementation alternatives

- Four alternatives for incorporating persistent memory into the hierarchy
  - RAM disk: a full-blown file system running on top of main memory (with true file system overheads)
  - PMFS: a persistent memory file system, optimized for byte-addressable storage
  - Dynamic array: the typical collection one would use for expandable arrays when programming for main-memory
  - Blocked memory: an optimized blocked memory implementation of expandable arrays

# Indicative results: sorting 1M records



# Sorting 1M records: implementation alternatives



# Summary

- Large memories means that data processing will likely be memory-bound
  - No need for separate runtimes for application logic and data management
  - Data processed in the managed runtime, using language-integrated querying
  - *Just-in-time code generation for query processing*
- Memories not only large, but also non-volatile
  - With different performance characteristics
  - *Write-limited algorithms and a dynamic runtime to optimize performance*
- Management at all levels
  - Different applications require different representations for the same data
  - *Workload-driven dynamic data placement*

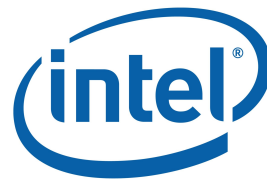
# Acknowledgments

- Students

- Konstantinos Krikellas
- Ioannis Koltsidas
- Vasilis Vasaitis
- Fabian Nagel
- Andreas Chatzistergiou
- Arpit Joshi
- Michail Basios
- Matt Pugh
- Chris Perivolaropoulos

- Collaborators

- Gavin Bierman (Oracle)
- Marcelo Cintra (Intel)
- Aleksandar Dragojevic (MSR)
- Boris Grot (UoE)
- Vijay Nagarajan (UoE)



# Summary

- Large memories means that data processing will likely be memory-bound
  - No need for separate runtimes for application logic and data management
  - Data processed in the managed runtime, using language-integrated querying
  - *Just-in-time code generation for query processing*
- Memories not only large, but also non-volatile
  - With different performance characteristics
  - *Write-limited algorithms and a dynamic runtime to optimize performance*
- Management at all levels
  - Different applications require different representations for the same data
  - *Workload-driven dynamic data placement*