Building Memory-efficient Java Applications: Practices and Challenges

Nick Mitchell, Gary Sevitsky (presenting)
IBM TJ Watson Research Center Hawthorne, NY USA

ACM SIGPLAN PLDI 2009, Dublin, Ireland

Quiz

Small boxes?

Q: What is the size ratio of Integer to int?

a. 1 : 1

b. 1.33 : 1

c. 2 : 1

d. ?

Assume 32-bit platform

Small things?

Q: How many bytes in an 8-character String?

a. 8

b. 16

c. 28

d. ?

Assume 32-bit platform

Bigger? Better?

Q: Which of the following is true about HashSet relative to HashMap

a. does less, smaller

b. does more, smaller

c. similar amount of functionality, same size

d. ?

Small collections?

Q: Put the following 2-element collections in size order: ArrayList, HashSet, LinkedList, HashMap

Collections?

Q: How many live collections in a typical heap?

a. between five and ten

b. tens

c. hundreds

d. ?

Roadmap

Quiz

Background & myths

Memory health

Patterns of memory usage

- Case studies, with JVM background mixed in

Process

Background

Background

- Our group has been diagnosing memory and performance problems in large Java systems for 10 years
- Built diagnosis tools used widely within IBM
  - most recent: Yeti
- Worked with dozens of applications: open source, large commercial applications, software products
  - servers, clients, applications, frameworks, generated code, etc.

The big pile-up

Heaps are getting bigger

- Grown from 500M to 2-3G or more in the past few years
- But not necessarily supporting more users or functions

Surprisingly common:

- requiring 1G memory to support a few hundred users
- saving 500K session state per user
- requiring 2M for a text index per simple document
- creating 100K temporary objects per web hit

Consequences for scalability, power usage, and performance

Common thread

- It is easy to build systems with large memory requirements for the work accomplished
- Overhead of representation of data can be 50-90%
  - Not counting duplicate data and unused data

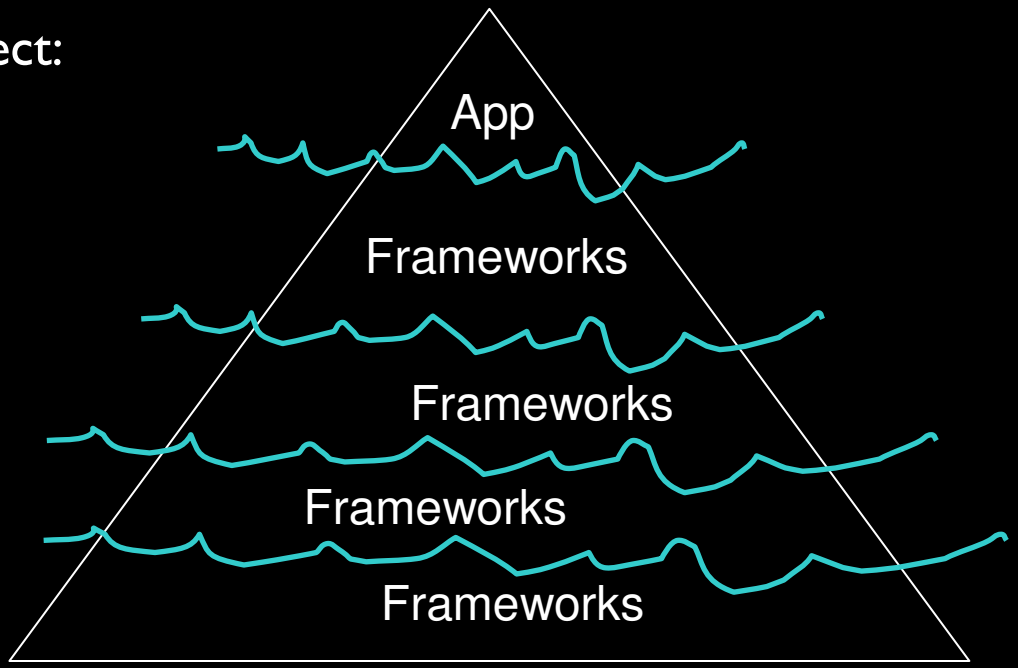
The big pile-up

Not a reflection on the quality of programmers – many are expert

More abstractions = less awareness of costs

- It is easy for costs to pile up, just piecing together building blocks

The iceberg effect:



Myths

Things are fine

Objects (or Strings, HashMaps, …) are cheap

Frameworks are written by experts, so they've been optimized (for my use case!)

The JIT and GC will fix everything

Things are not fine

I knew foo was expensive; I didn't know it was this expensive!

It's no use: O-O plus Java is always expensive

Efficiency is incompatible with good design

Goals

- Raise awareness of costs
- Give you a way to make informed tradeoffs

For the research audience

- Understand the walls that developers and JITs face
- Many opportunities for improvement

Roadmap

Quiz

Background & myths

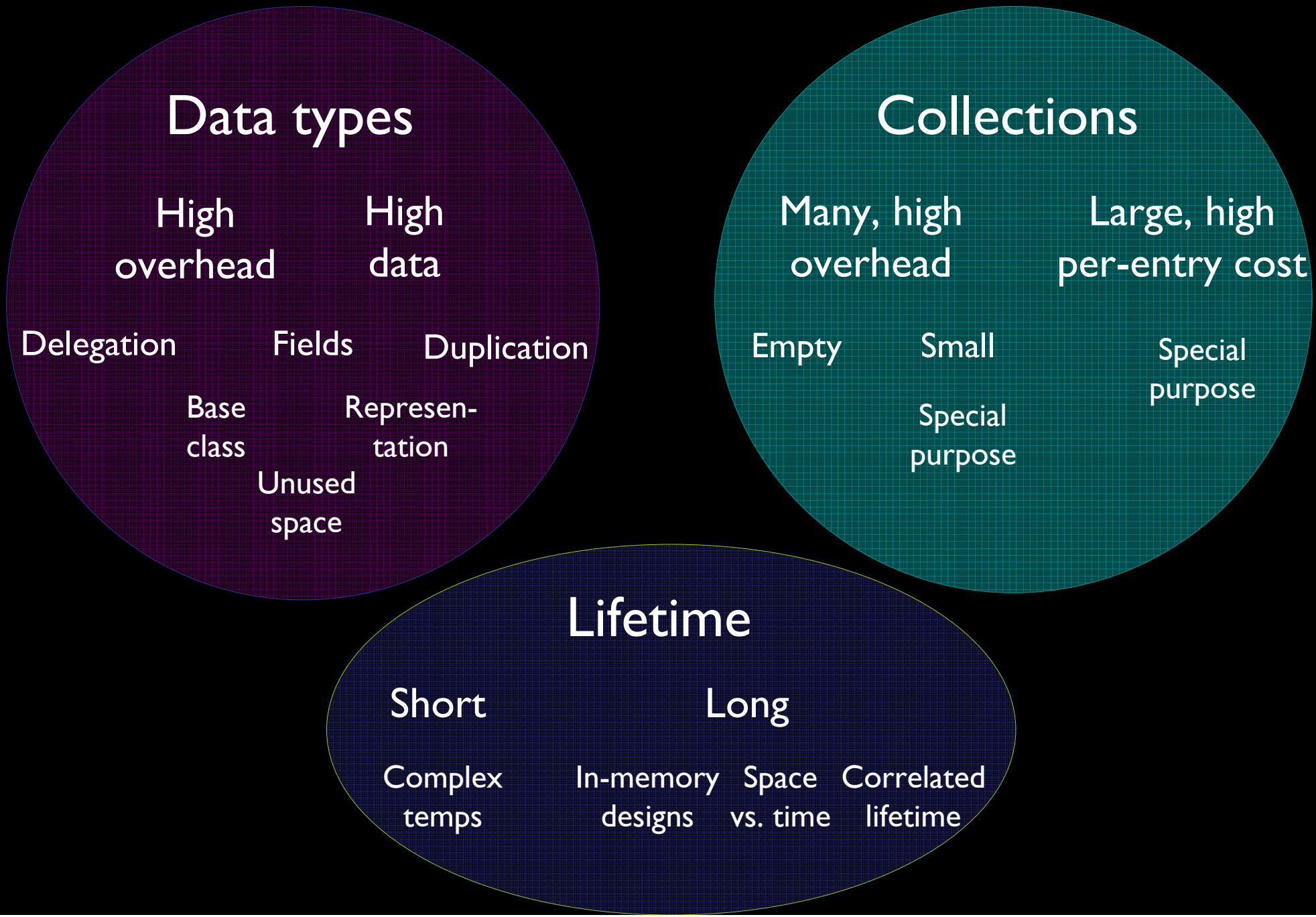
Memory health

Patterns of memory usage

- Case studies, with JVM background mixed in

Process

Patterns of memory usage



Roadmap

Quiz

Background & myths

Memory health

Patterns of memory usage

- Modeling your data types
- Modeling relationships
- …. break …
- More relationships
- More data type modeling
- Object lifetime

Process

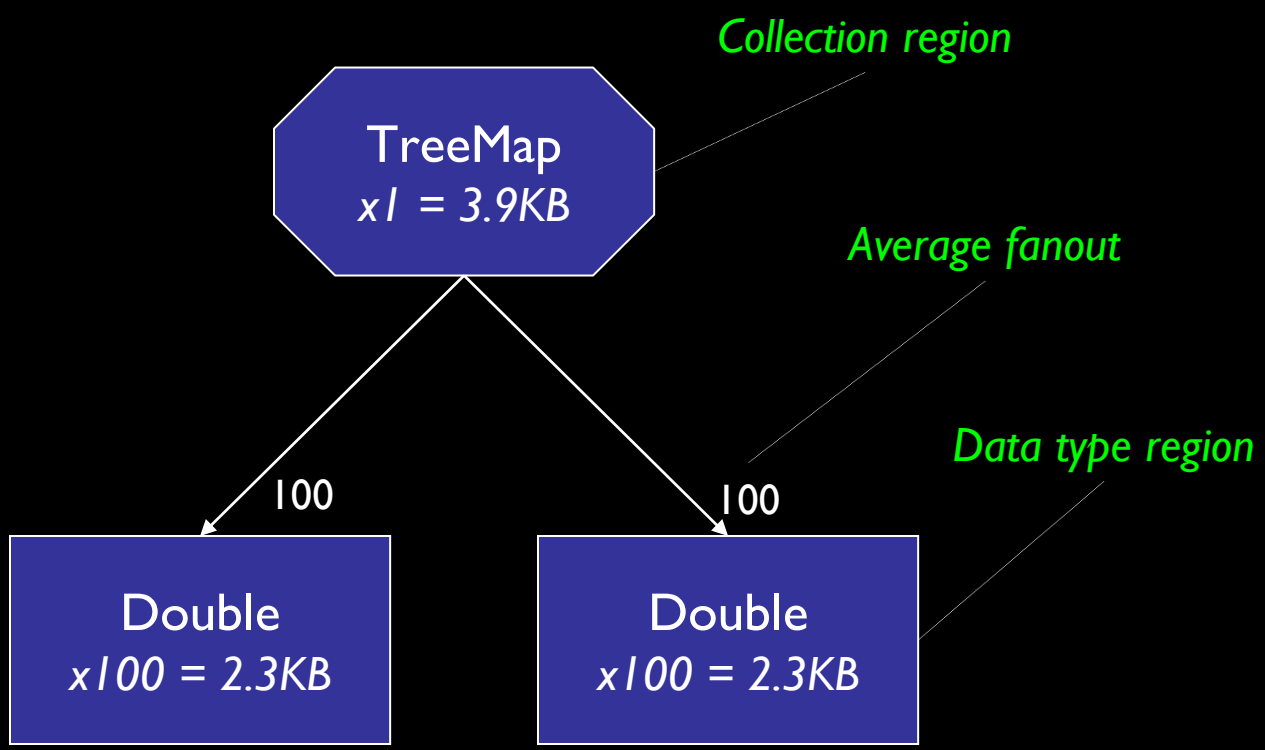
Note about measurements

Measurements shown are estimates obtained from experiments on a sampling of different JVMs. In practice costs will vary across JVMs. Measures we report may be subject to errors due to data collection, analysis tools, or interpretation of results. They are intended only to illustrate the nature of memory problems.

Memory health

The idea of health

TreeMap<Double, Double> (100 entries)

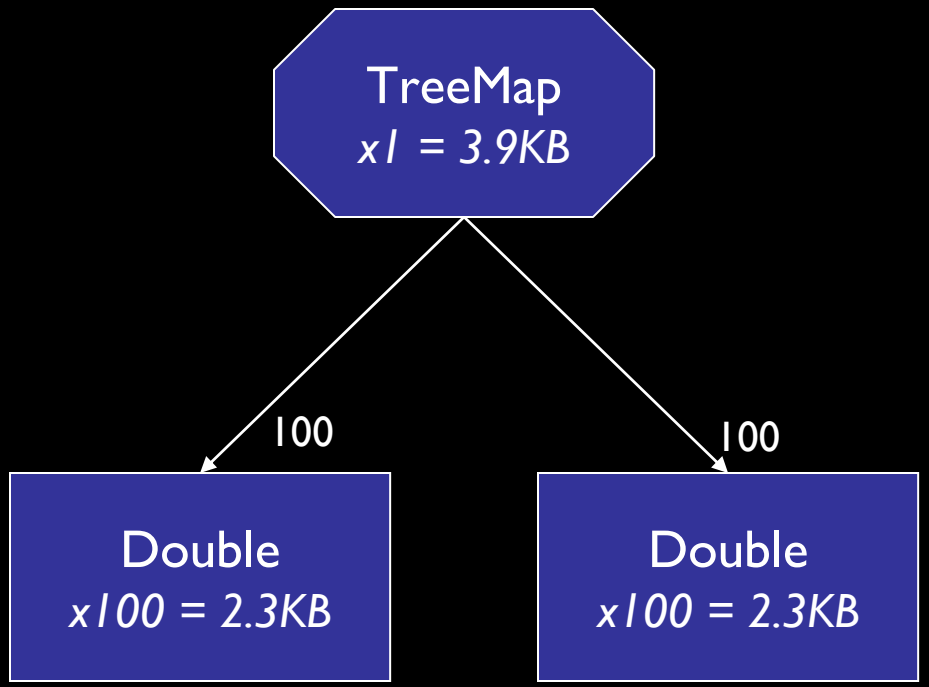


- Schematic of a data structure
- Distinguish data types from collections
- A region includes all of its implementation classes

Note: example based on Java 5 TreeMap

The idea of health

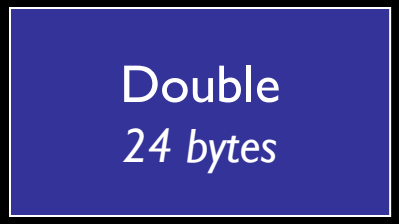
TreeMap<Double, Double> (100 entries)



- Cost: 8.6KB
- What are the bytes accomplishing?
- How much is actual data vs. overhead?

Data type health

One Double

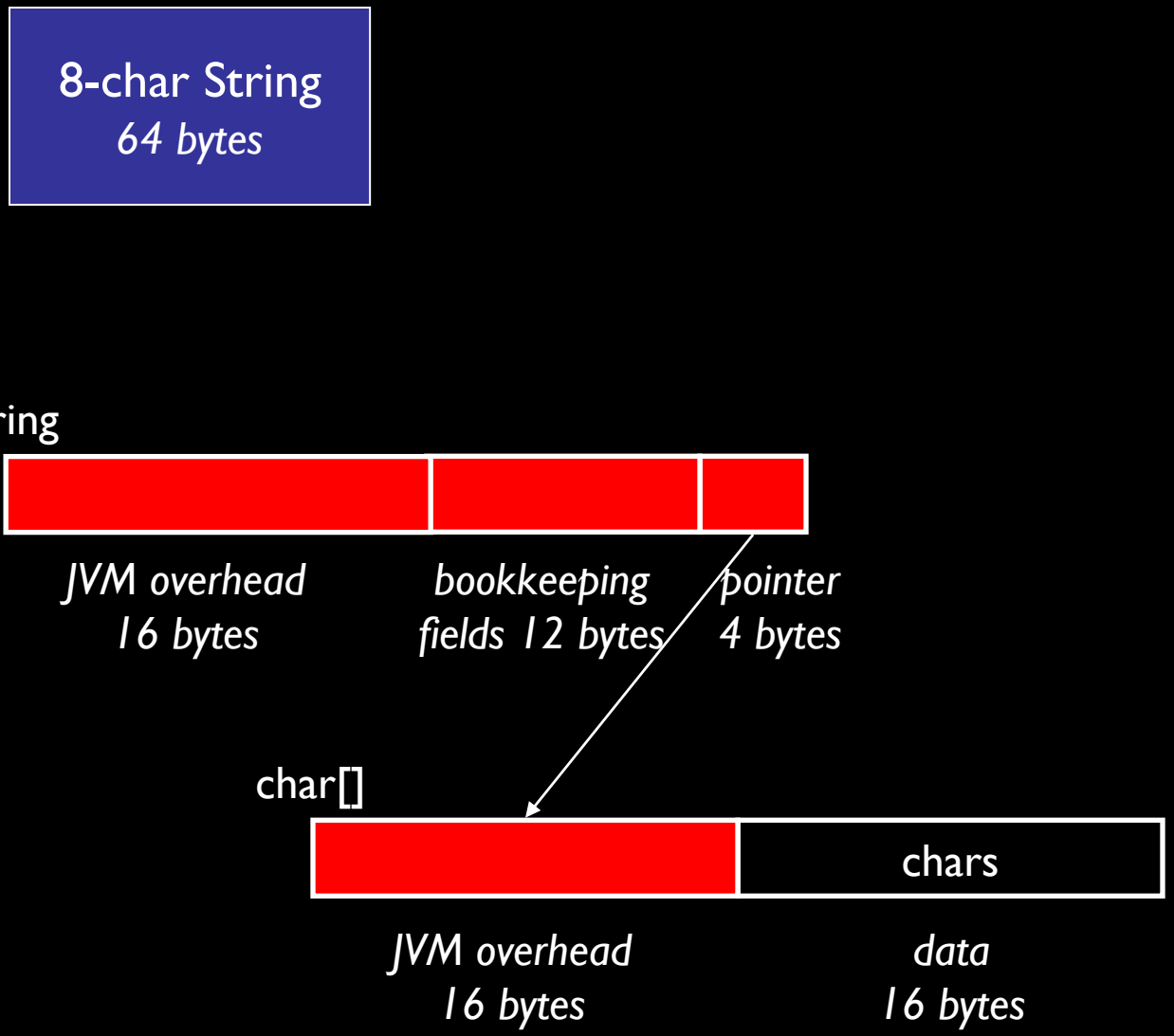


Double

- 33% is actual data
- 67% is the representation overhead
- From one 32-bit JVM. Varies with JVM, architecture.

Data type health

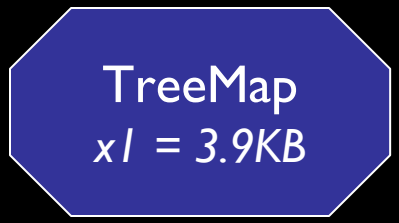
Example: An 8-character String

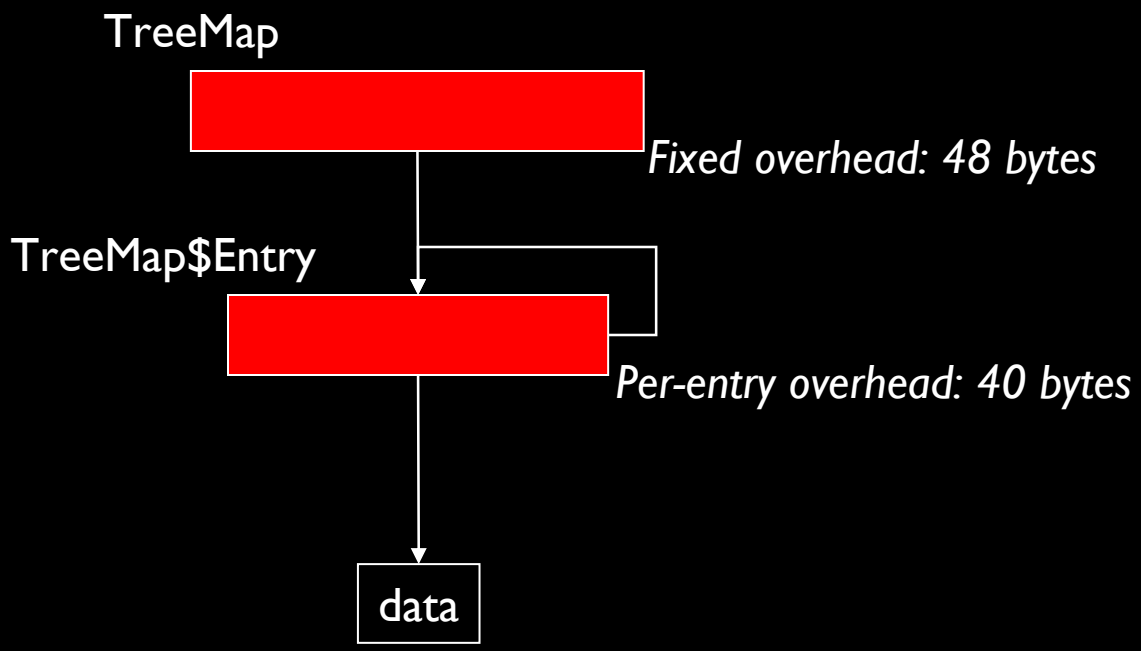


- only 25% is the actual data
- 75% is overhead of representation
- would need 96 characters for overhead to be 20% or less

Collection health

A 100-entry TreeMap

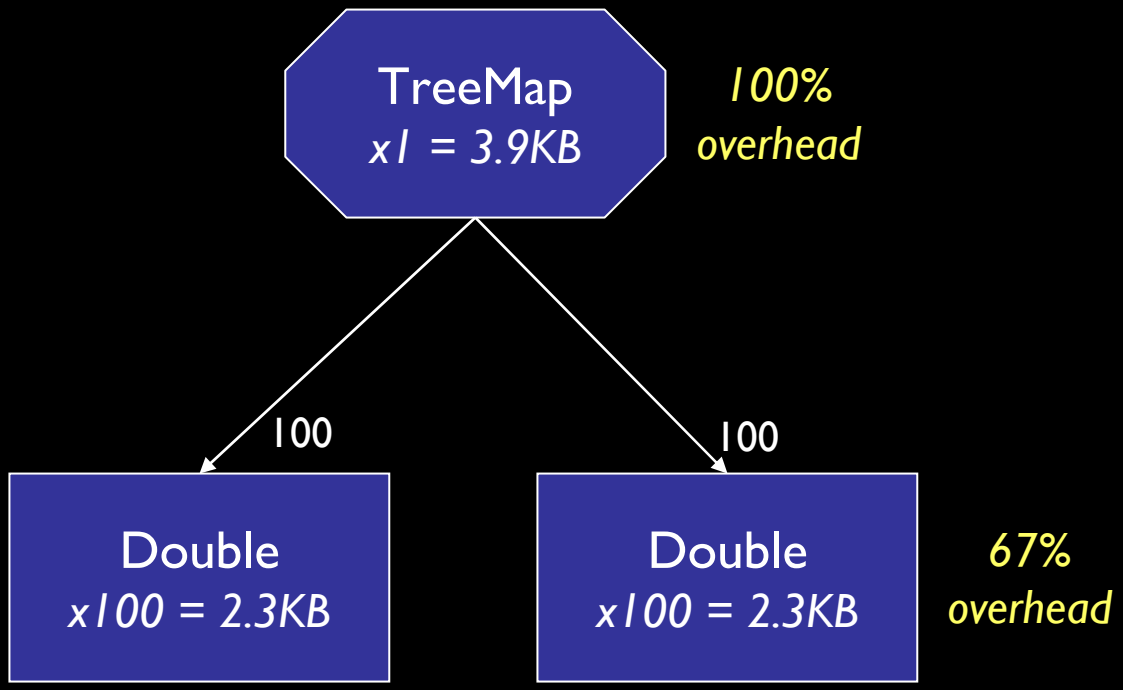




- How does a TreeMap spend its bytes?
- Collections have fixed and variable costs

Data structure health

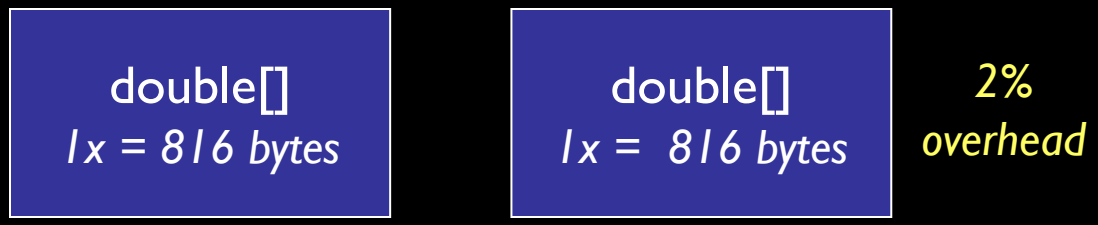
TreeMap<Double, Double> (100 entries)



- 82% overhead overall
- Design enables updates while maintaining order
- Is it worth the price?

Data structure health

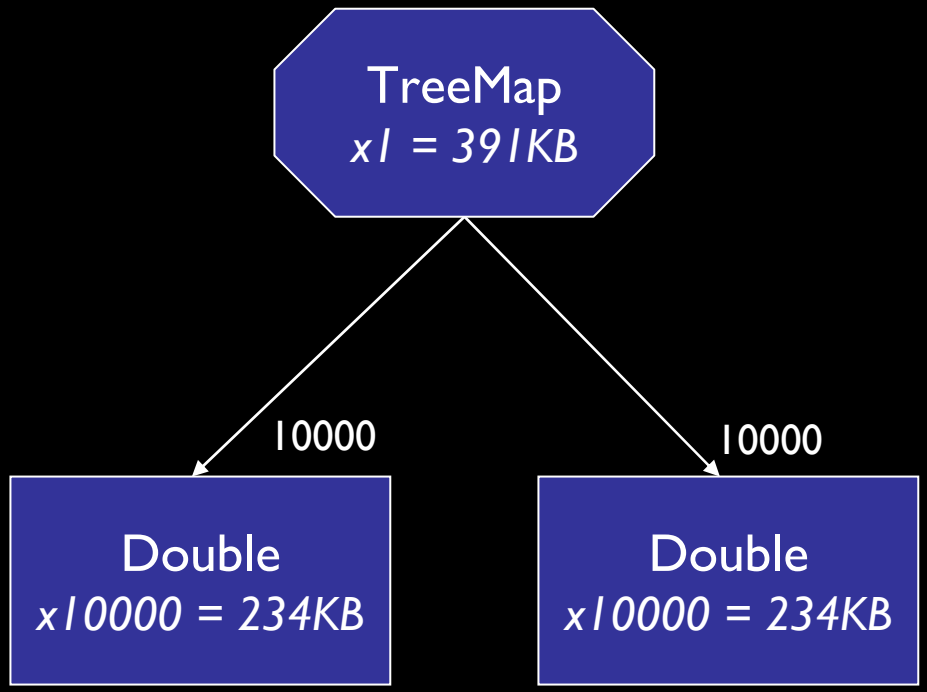
Alternative implementation (100 entries)



- Binary search against sorted array
- Less functionality – suitable for load-then-use scenario
- 2% overhead

Health as a gauge of scalability

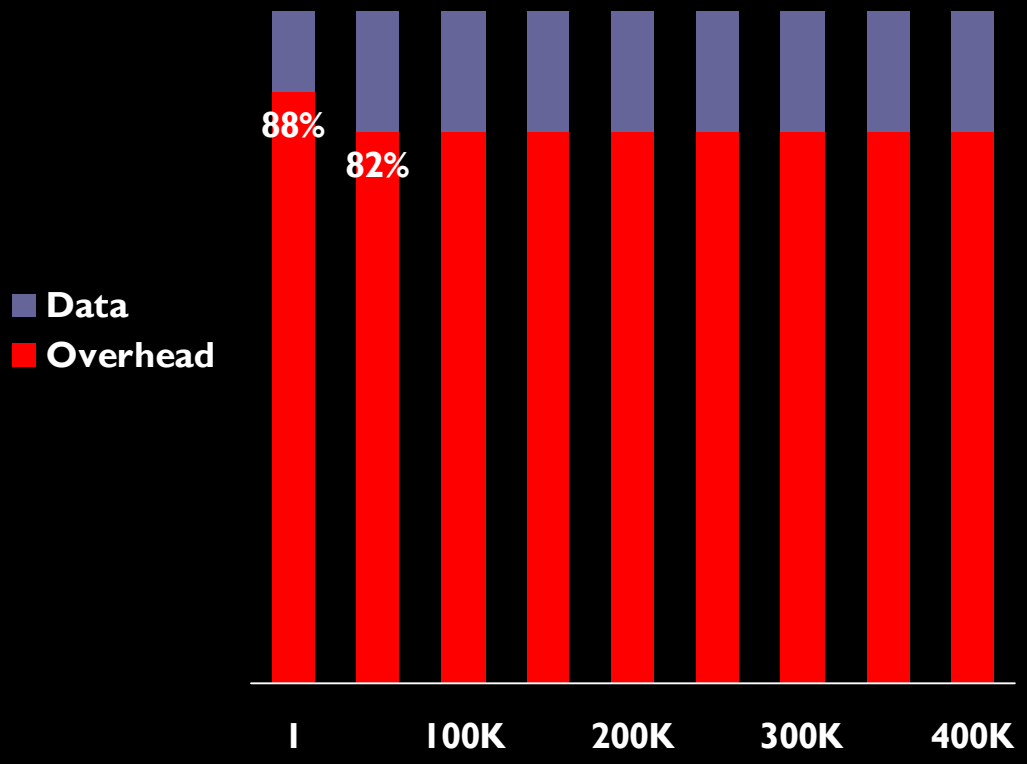
TreeMap<Double, Double> (10,000 entries)



- Overhead is still 82% of cost
- Overhead is not amortized in this design
- High constant cost per element: 88 bytes

Health as a gauge of scalability

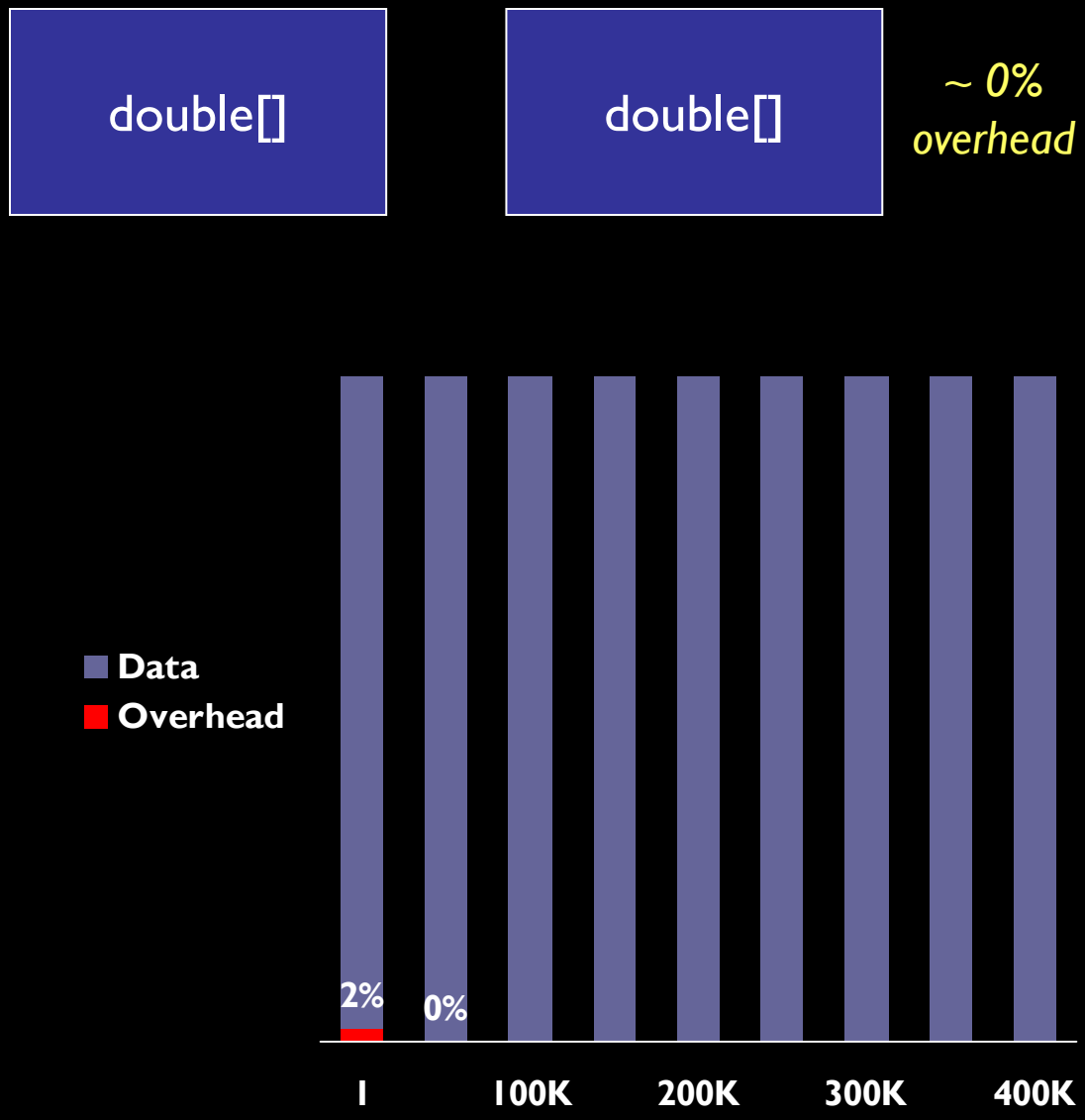
TreeMap<Double, Double>



- Overhead is still 82% of cost
- Overhead is not amortized in this design
- High constant cost per element: 88 bytes

Health as a gauge of scalability

Alternative implementation



- Overhead starts out low, quickly goes to 0
- Cost per element is 16 bytes, pure data

Summary: Health

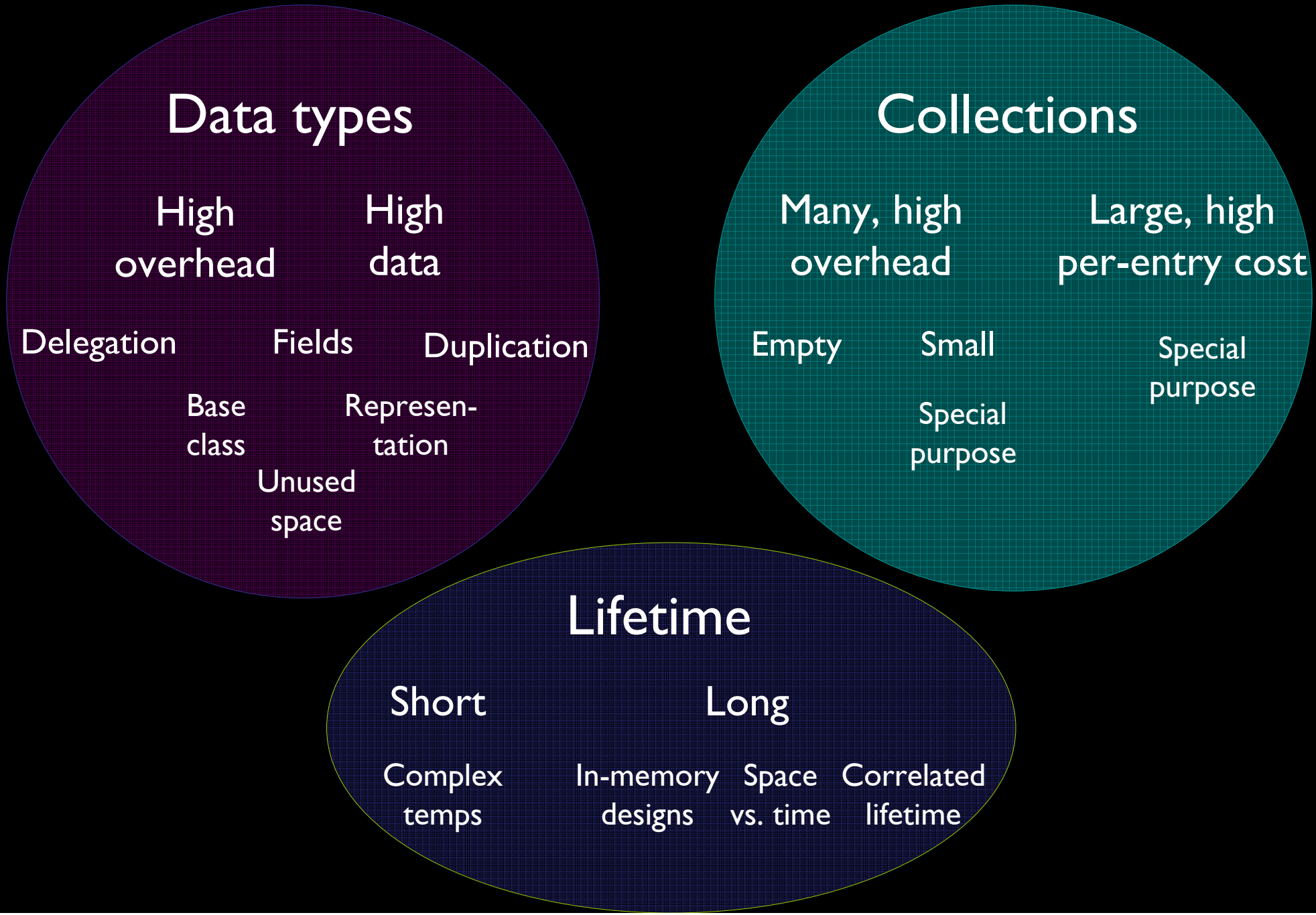
Distinguish actual data from the overhead of representation:

- Overhead from your data types
- Overhead from your collection choices, fixed vs. variable
- Many other ways to break down overhead costs
  - JVM object overhead, delegation costs, empty array slots, unused fields, duplicate data, ...

Can help answer:

- How much room for improvement?
- Is the functionality worth the price?
- Which overhead will be amortized? If constant, how large?

Patterns of memory usage



Modeling your data types

- High-overhead data types
  - Object and delegation costs

Background: the cost of objects

Boolean
16 bytes

header *12 bytes* | *boolean* *1 byte* | *alignment* *3 bytes*

Double
24 bytes

header *12 bytes* | *double* *8 bytes* | *alignment* *4 bytes*

char[2]
24 bytes

header *16 bytes* | *2 chars* *4 bytes* | *alignment* *4 bytes*

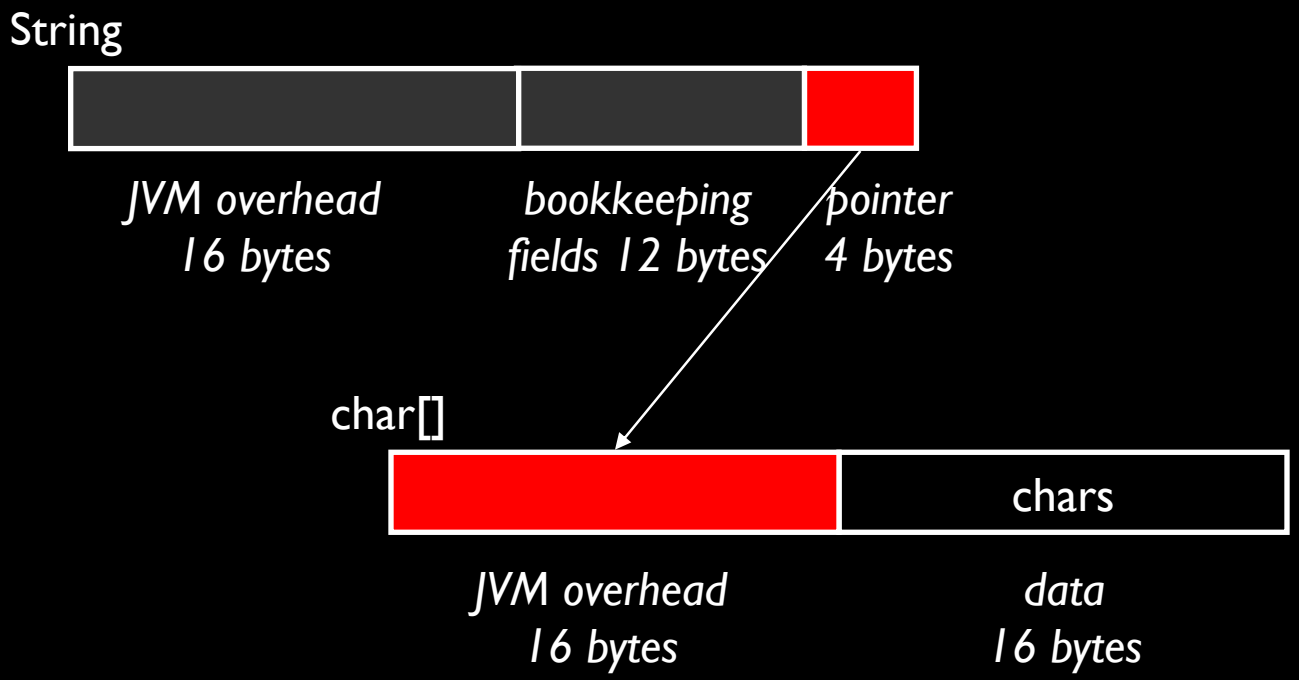
From experiment on one 32-bit JVM

- JVM & hardware impose costs on objects. Can be substantial for small objects
- Headers enable functionality and performance optimizations
- 8-byte alignment in this JVM
- Costs vary with JVM, architecture

The cost of delegation

Example: An 8-character String

- 31% is overhead due to modeling as two objects
- Effect varies with size of String



The culture of objects

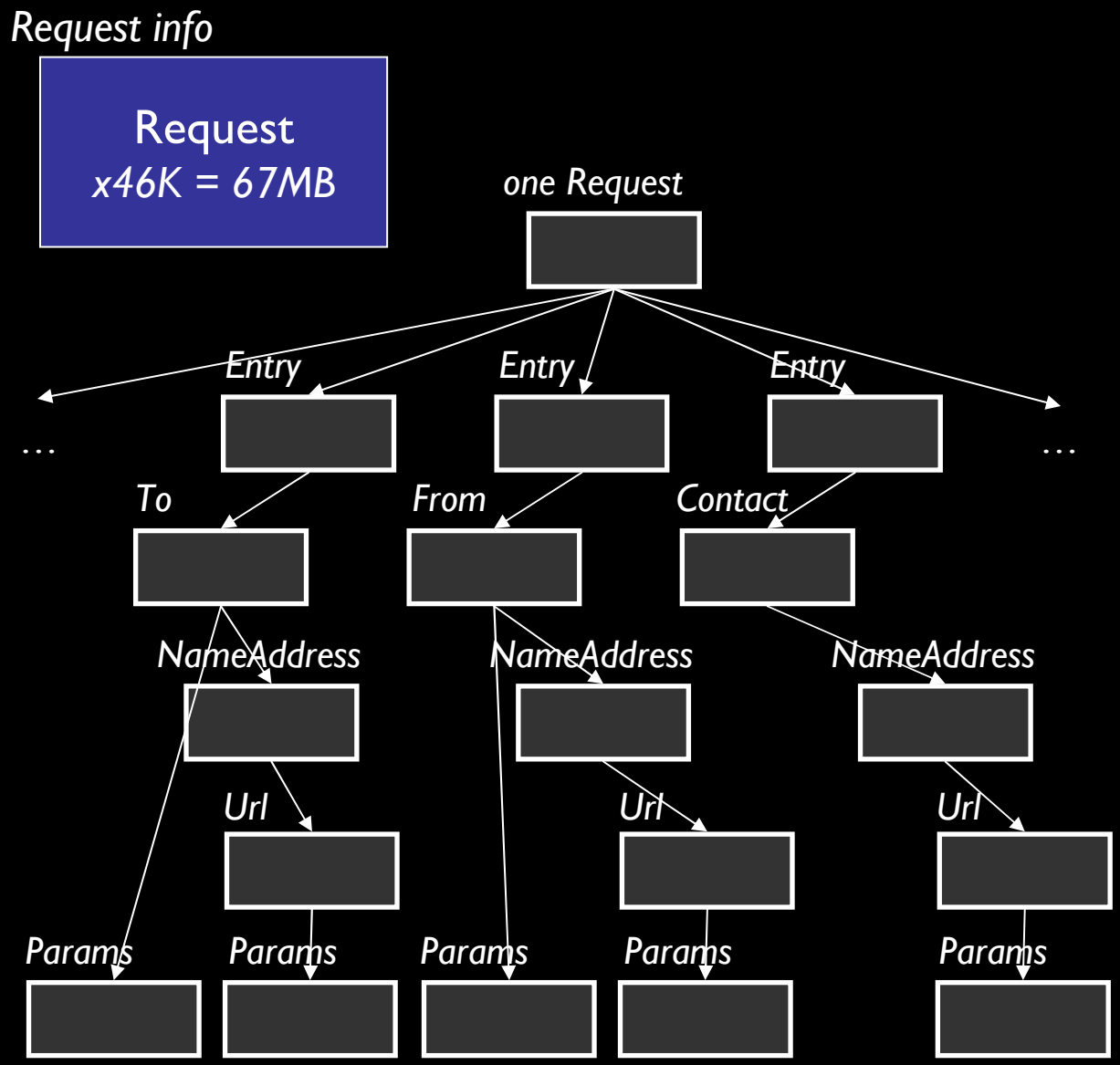
C++ has 5 ways to organize fields into data types. Java has 2.

- Delegation
- Composition
- Single inheritance
- Multiple inheritance
- Union types

Software engineering culture favors reuse and loosely coupled designs

Fine-grained modeling

Case study: server framework, part of connection



- 34 instances to represent a request. Cost: 1.5K per request. Will not scale.
- 36% of cost is delegation overhead
- Constant overhead per Request
- Can magnify the costs of other choices

Modeling your data types

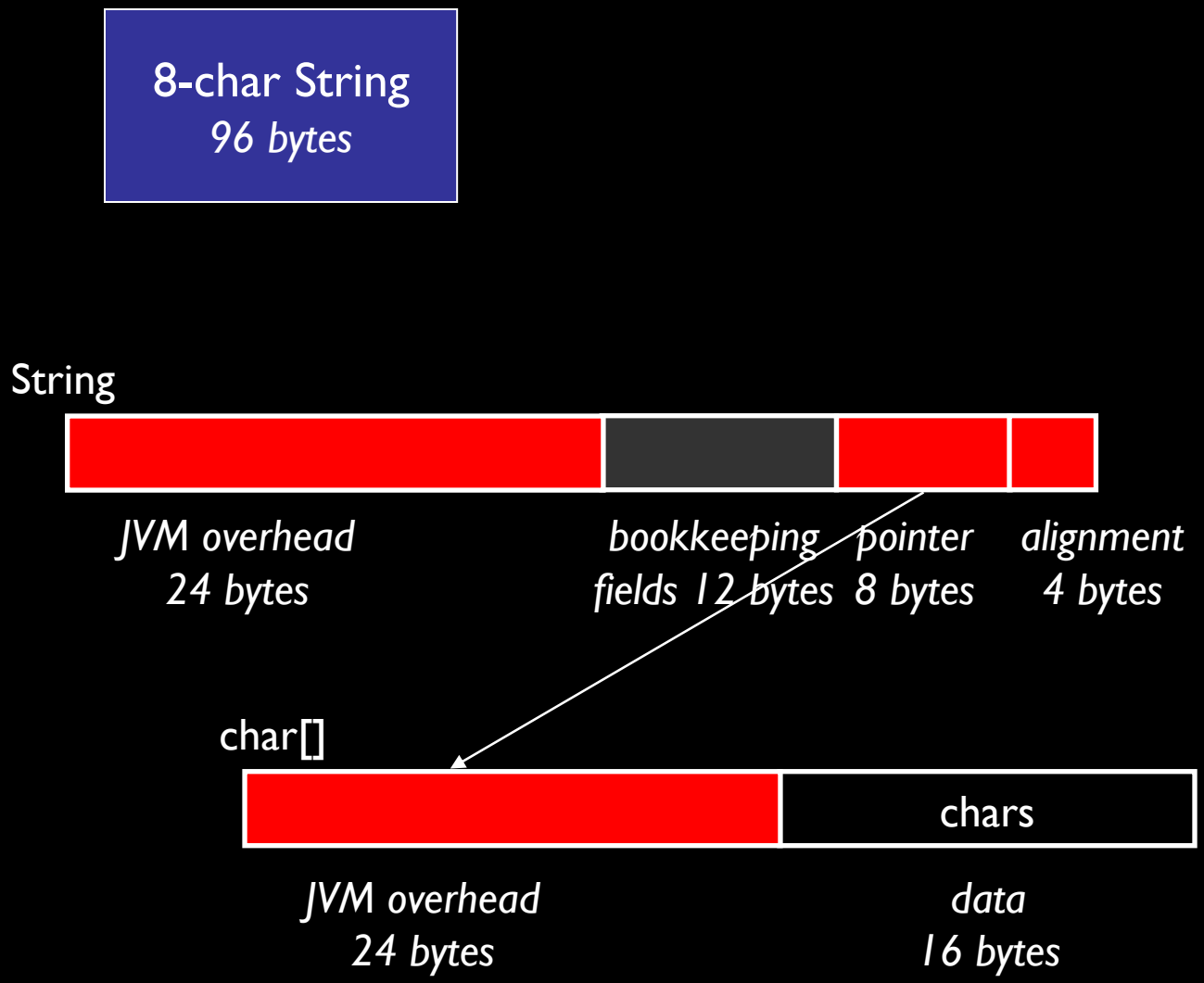
- Background: 32- vs. 64-bit JVMs

32- vs. 64-bit

- 64-bit architectures can have a big impact on memory costs. Especially in designs that have a lot of small objects and pointers
- Using 64-bit addressing to solve memory problems can cause new ones
  - Increases object header, alignment, and pointer overhead
  - One study shows 40-50% avg. increase in heap sizes for benchmarks
- Most JVMs have options for extended 32-bit addressing, allowing access to larger heaps without the footprint cost
  - e.g. IBM Java 6 sr2 compressed addressing allows ~28GB

32- vs. 64-bit

Example: An 8-character String



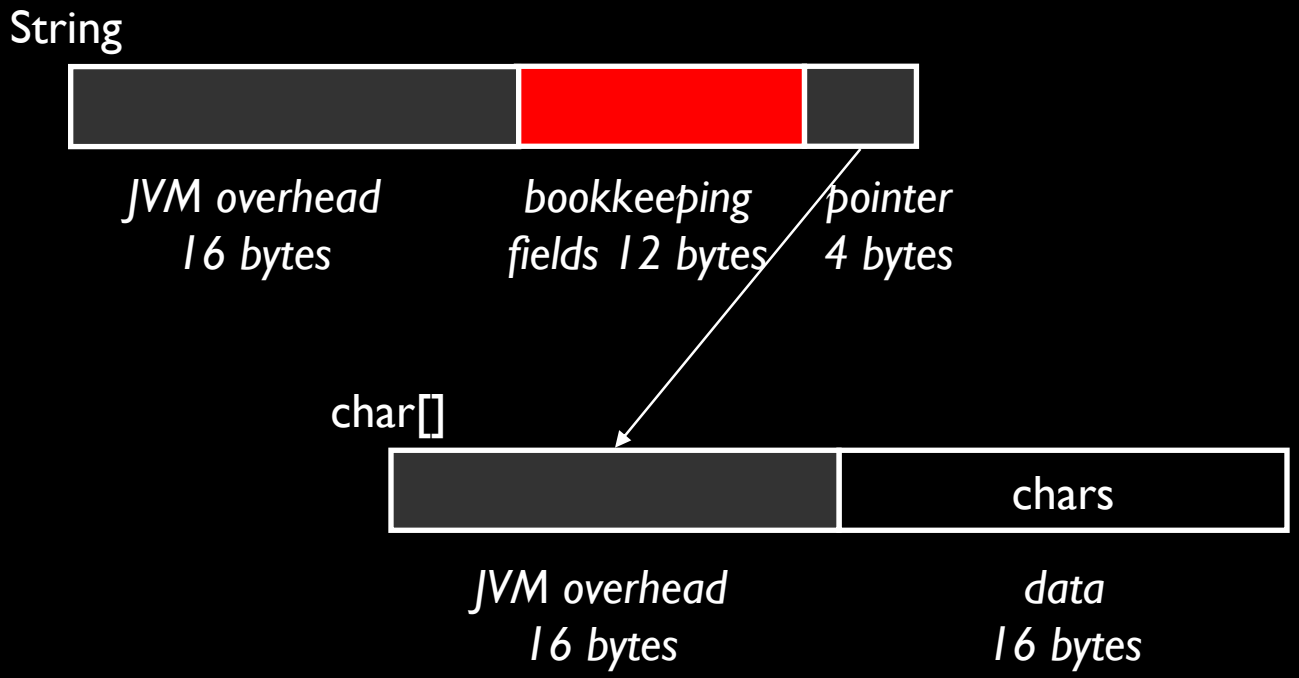
- 50% larger
- Delegated design is responsible for extra object header and pointer costs
- Fine-grained designs incur especially high costs

Modeling your data types

- High-overhead data types
  - Large instance sizes

Bookkeeping fields

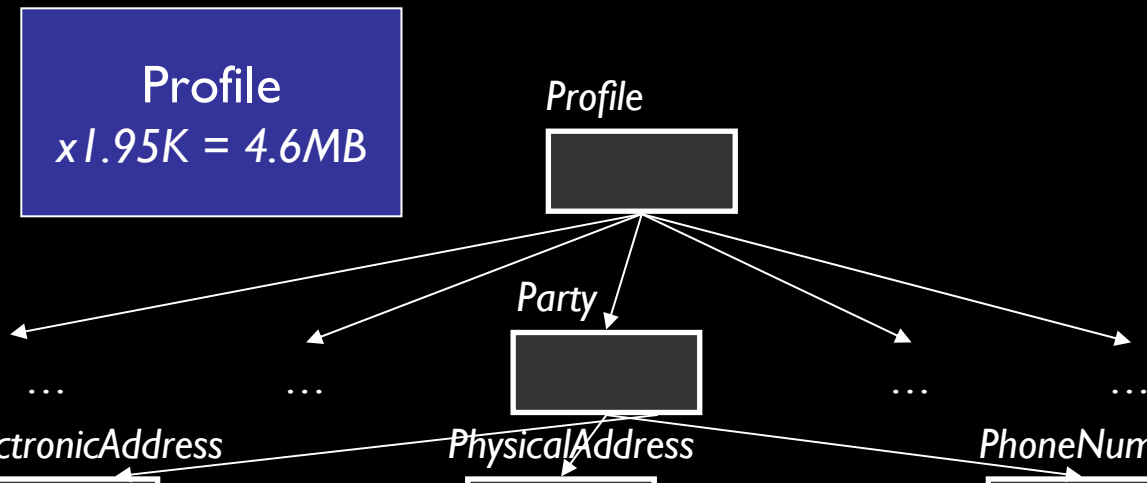
Simple example: an 8-character String



- String users pay a 12-byte tax to store offset, length, hashcode. Just one needed for common cases.
- 19% overhead for an 8-char String
- Premature optimization. Cautionary tale for library designers!

Large instance sizes II

Case study: CRM system, part of session data



Object	12
ContactInfo	40
Date createDate	
Party enteredBy	
Date updateDate	
Party updateBy	
Object primary	
int typeId	
String type	
…	
ElectronicAddress	48
…	
total:	100

Object	12
ContactInfo	40
Date createDate	
Party enteredBy	
Date updateDate	
Party updateBy	
Object primary	
int typeId	
String type	
…	
PhysicalAddress	100
…	
total:	152

Object	12
ContactInfo	40
Date createDate	
Party enteredBy	
Date updateDate	
Party updateBy	
Object primary	
int typeId	
String type	
…	
PhoneNumber	60
…	
total:	112

- Highly delegated design
  - ~40 instances each
- Large base class and subclasses, in addition to delegation costs

Problem 1:

- Functionality too fine grained
- Magnifies base class

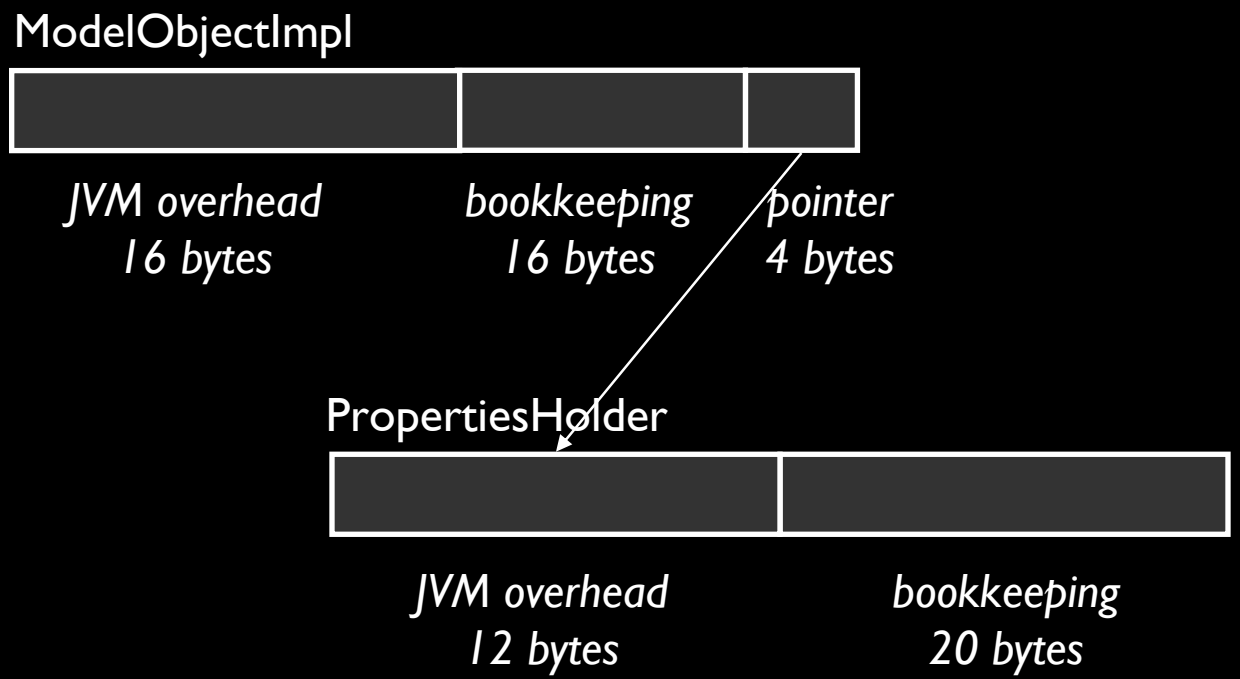
Problem 2:

- Storing computed fields

Large instance sizes III

Case study: Modeling framework

Modeled object
68 bytes +
your object cost

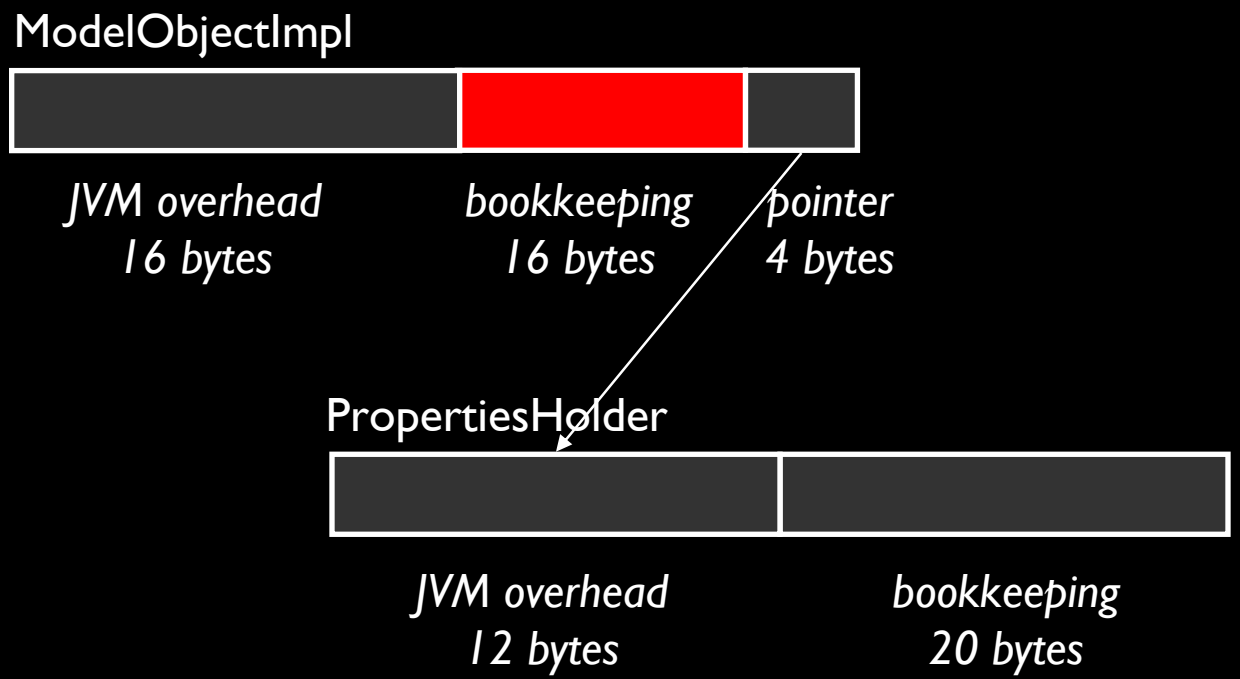


- Goal: model-based programming
- Support models with 100K objects
- Problem: high base framework costs
  - and forces modelers into inefficient choices
- Many causes
  - Some superficial, some deeper

Large instance sizes III

Case study: Modeling framework

Modeled object
68 bytes +
your object cost



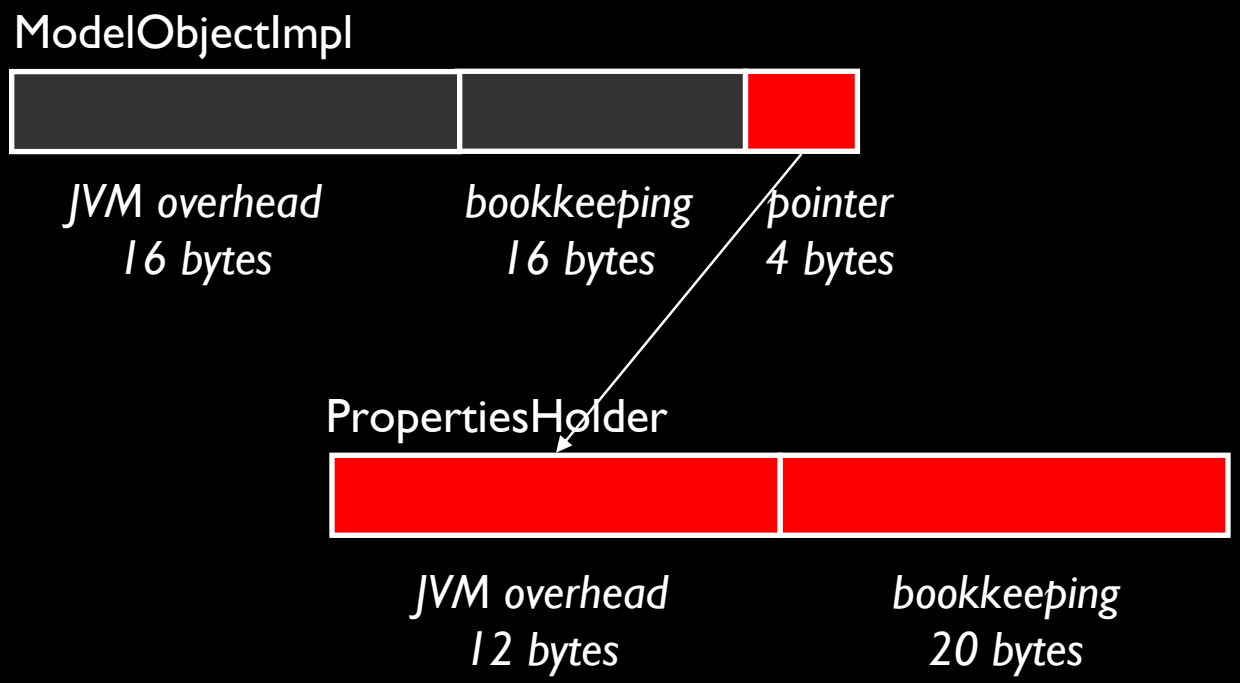
- Problem: constant field (Class) stored as instance variable
  - Replaced with static method

- Problem: fields supporting features not used in many models
  - e.g. notification, dynamic types
  - Refactored, introducing BasicObjectImpl with no storage

Large instance sizes III

Case study: Modeling framework

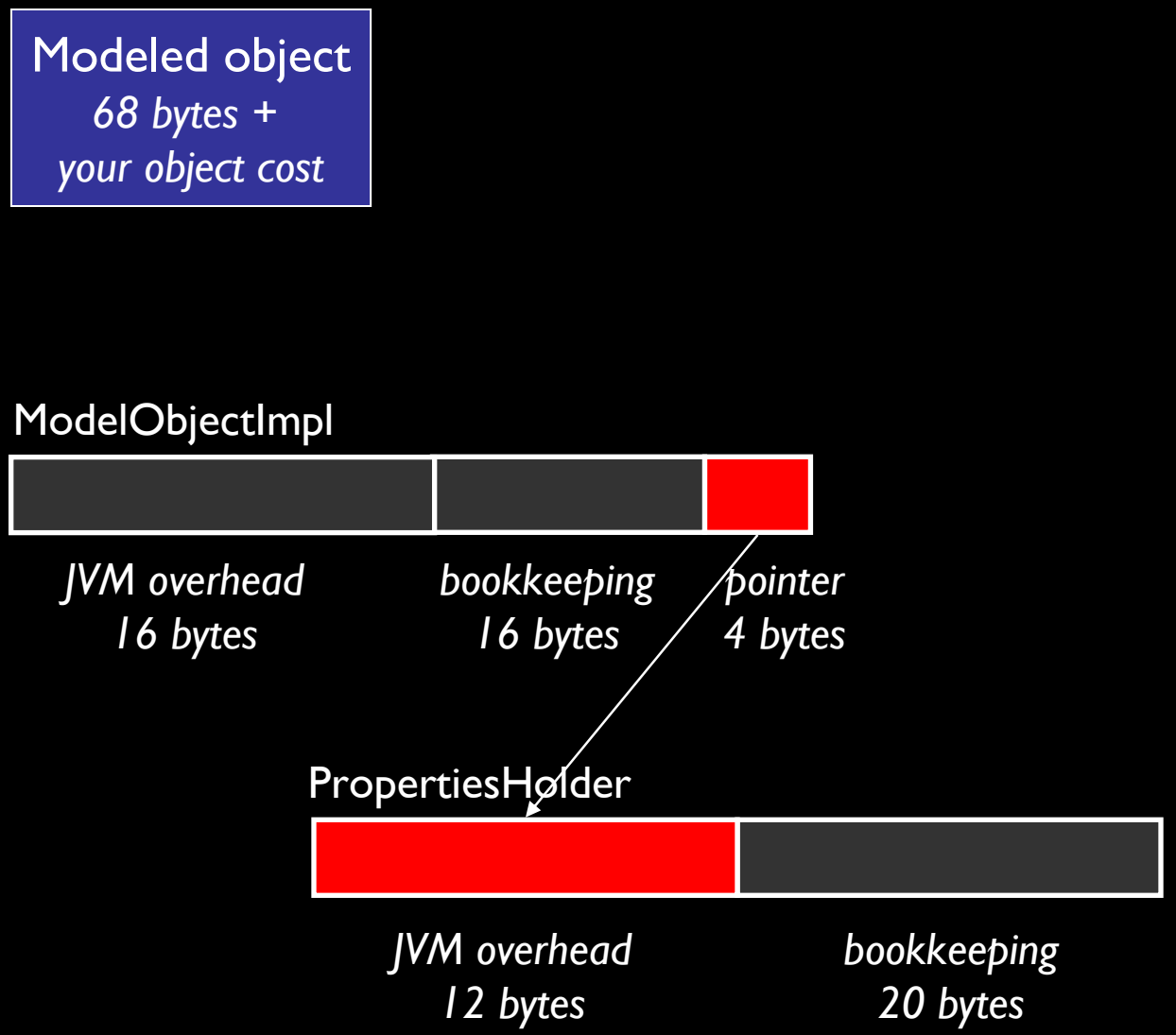
Modeled object
68 bytes +
your object cost



- Design: rarely used fields moved to side object and lazily allocated
- Problem: lazy allocation not working
  - Fixed
- Problem: 5 fields never used at the same time
  - Combined fields, using casts
- Problem: stored computations
  - Recompute

Large instance sizes III

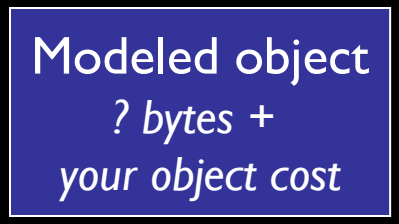
Case study: Modeling framework



- Problem: some models make heavy use of fields in side object
  - delegation costs
- Example: memory was at a premium, so model was broken into fragments. But cross-model references require these fields!
- Solution: refactoring for this case
  - FlatObjectImpl avoids delegation

Large instance sizes III

Modeling framework



Status

- Large improvements have been made
- Scalability issues still

Reflections

- Sound engineering is can make a difference, but …
- … it can only go so far. Developers are severely constrained building object frameworks in Java

Large instance sizes: patterns

- Expensive base classes
  - Some fields not needed in the general case, or are for rarely-used features
  - Fine-grained designs using a common base class multiply the cost of the base class design
- Data fields
  - Semi-constant fields
  - Sparse fields
  - Saving recomputable data unnecessarily – often the result of premature optimization. Both scalar and reference fields
- Typically, many cases occur together in the same data model

Data type modeling: challenges for developers

- Java's limited data modeling means tradeoffs require care
  - Moving rarely-used fields to side objects incurs delegation costs
  - Moving sparse fields to a map incurs high map entry costs
  - Verifying actual costs and benefits is essential
- Fixing problems of high-overhead data usually means refactoring data models
  - Not easy late in the cycle
  - Using interfaces and factories up front can help

Data type modeling: community challenges

- Many more objects and pointers than other languages
  - x high per-object cost = 35% delegation overhead avg in heaps

- Only two options for achieving variation – both are expensive
  - delegation vs. unused fields (large base classes)
  - both limit higher-level choices and magnify carelessness

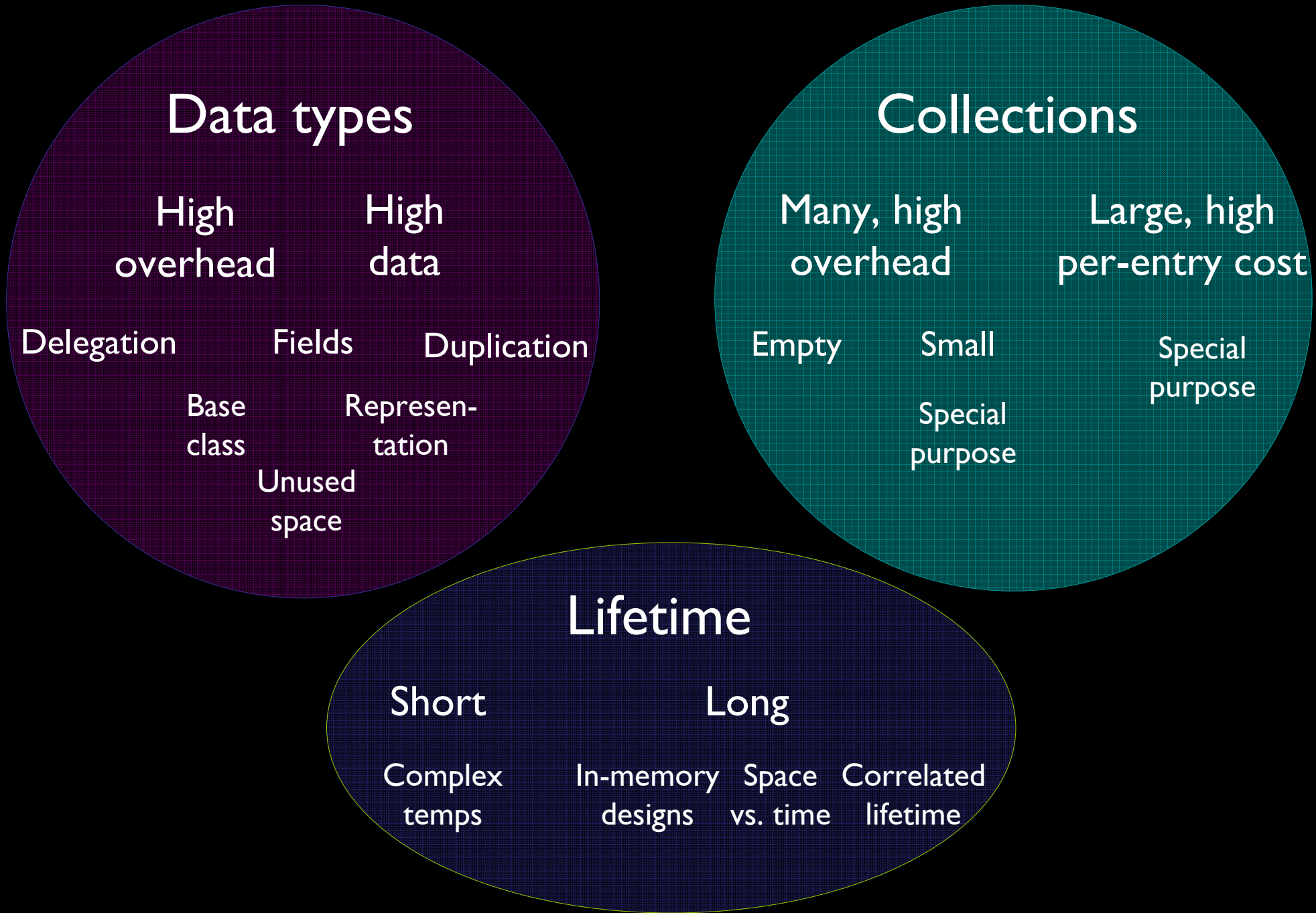
- Consequences all the way up the stack
  - Primitives as object(s): String, Date, BigDecimal, boxed scalars
  - Collections suffer these limitations
  - Many layers of frameworks implementing systems functionality

- Solutions in the language / runtime?

More data type patterns later

Representing relationships

Patterns of memory usage

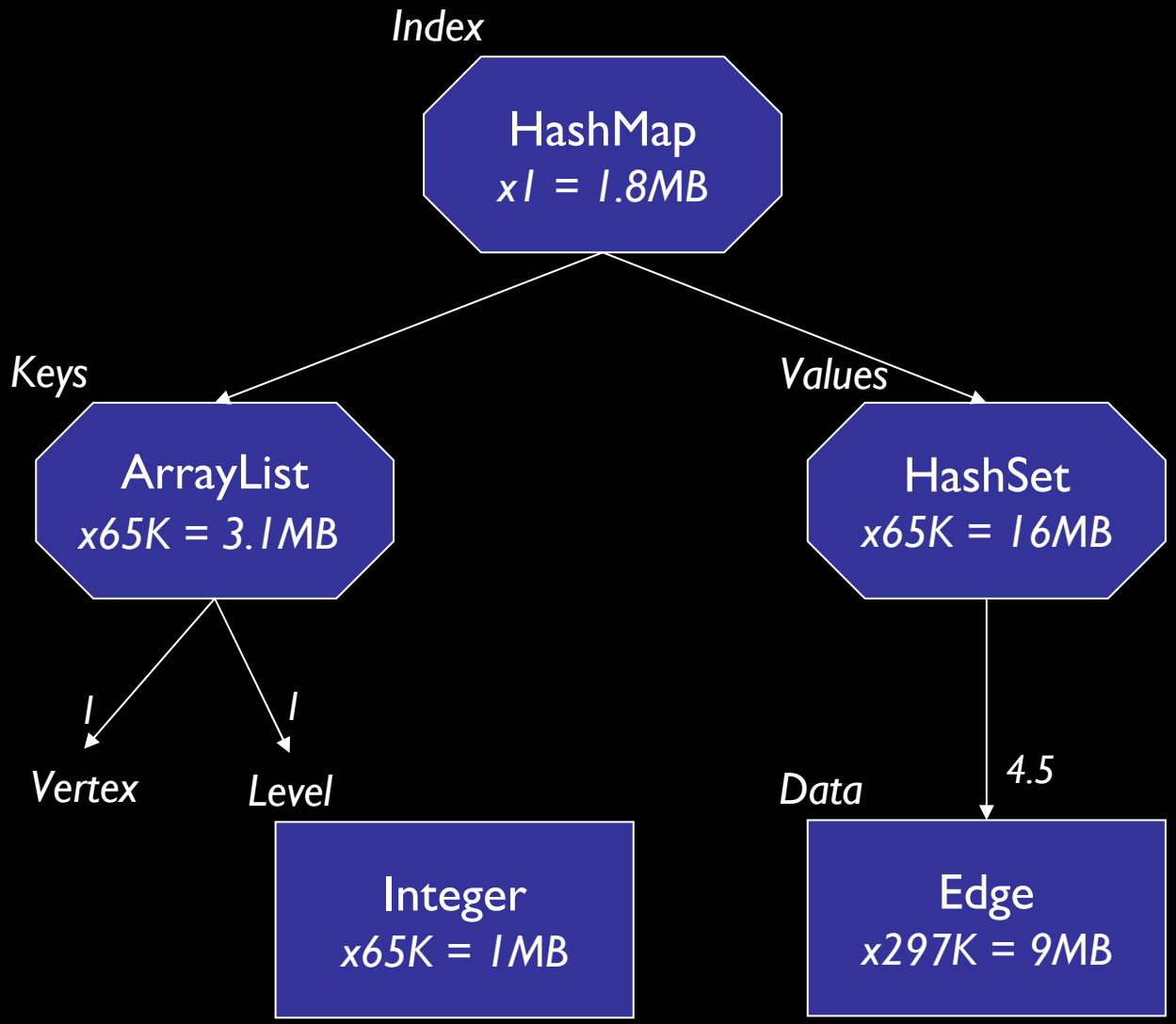


Representing relationships

- many, high-overhead collections
  - small collections

Small collections in context

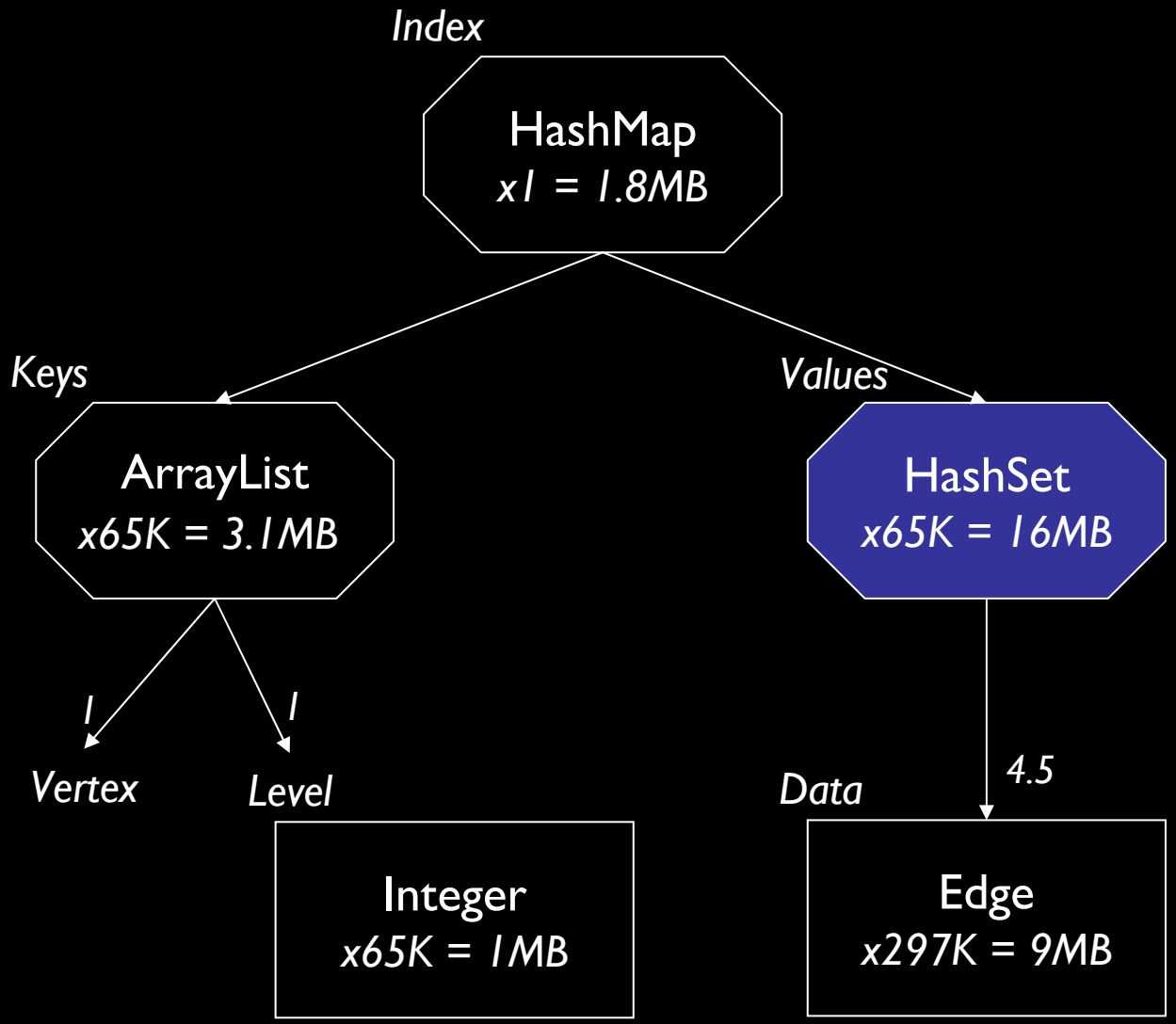
Case study: Planning system, level graph edges



- Two examples of small high-overhead collections
- 297K edges cost 31MB
- Overhead of representation: 83%
- Overhead will not improve with more vertices

Small collections in context

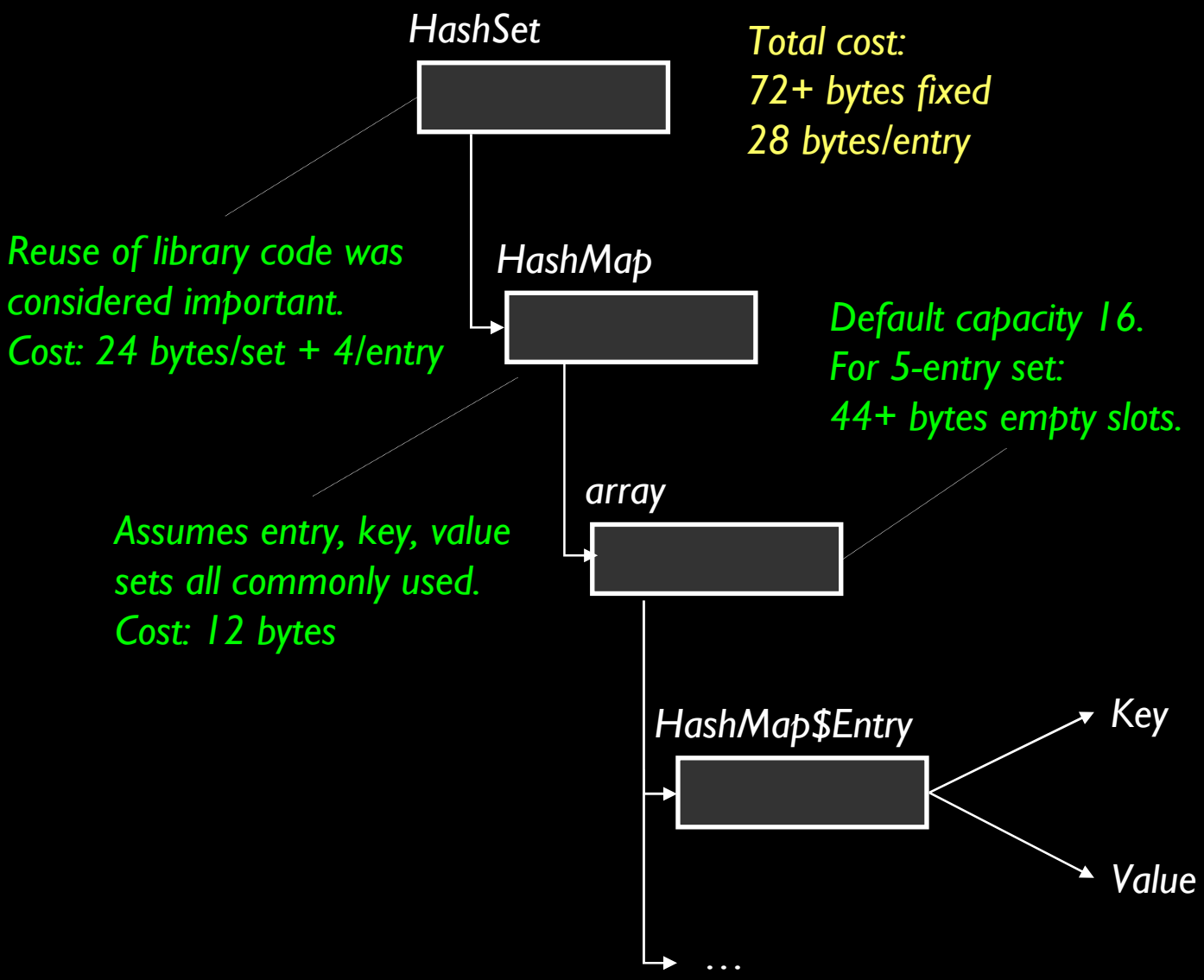
Map with multiple values per entry



- Only 5% of sets had more than a few elements each

Inside the Java collections

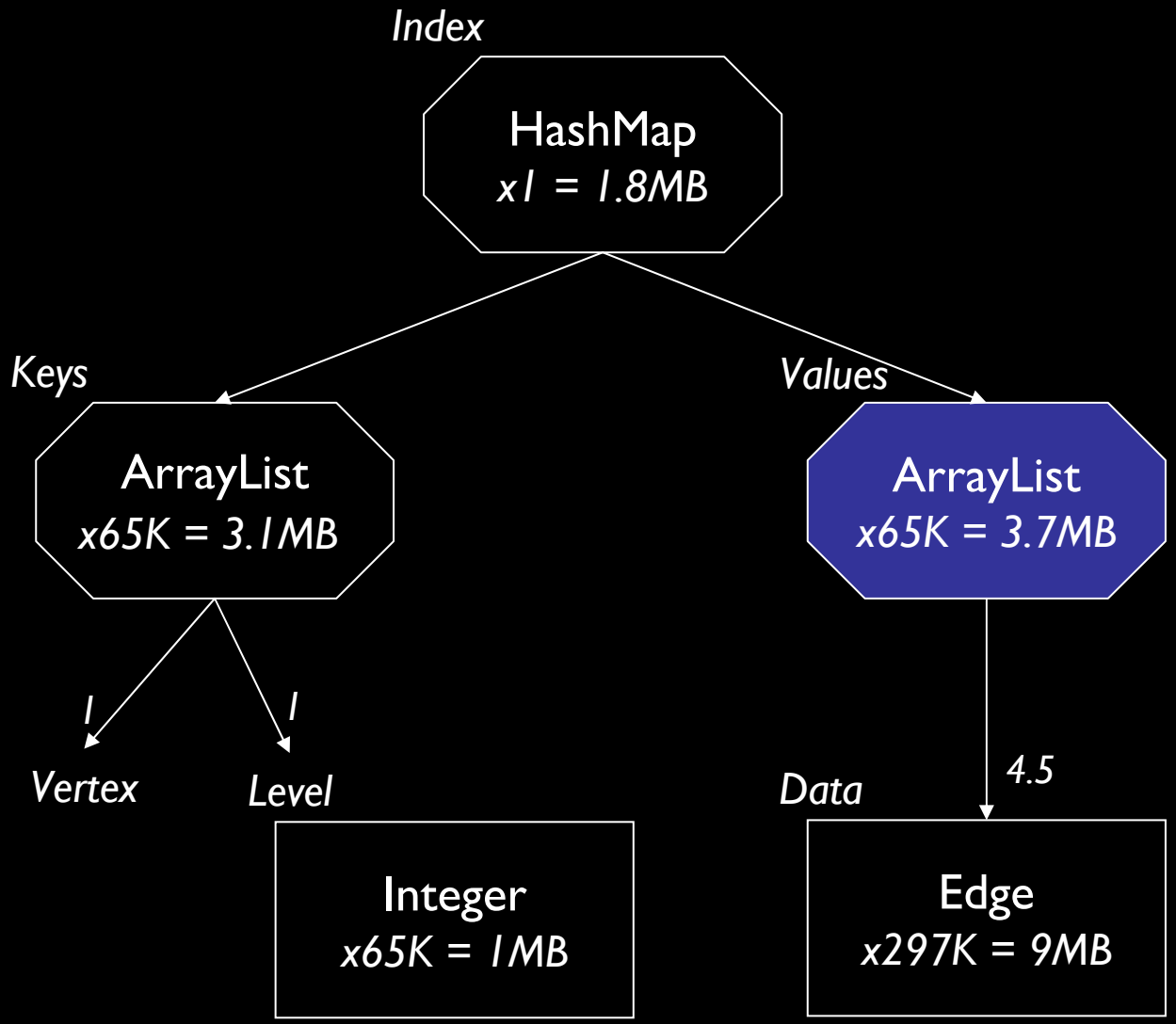
HashSet: many embedded usage assumptions



- Not a good choice for small collections
- Users, look before you leap – always measure
- Framework designers, beware making usage assumptions

Small collections in context

Map with multiple values per entry



Remedy

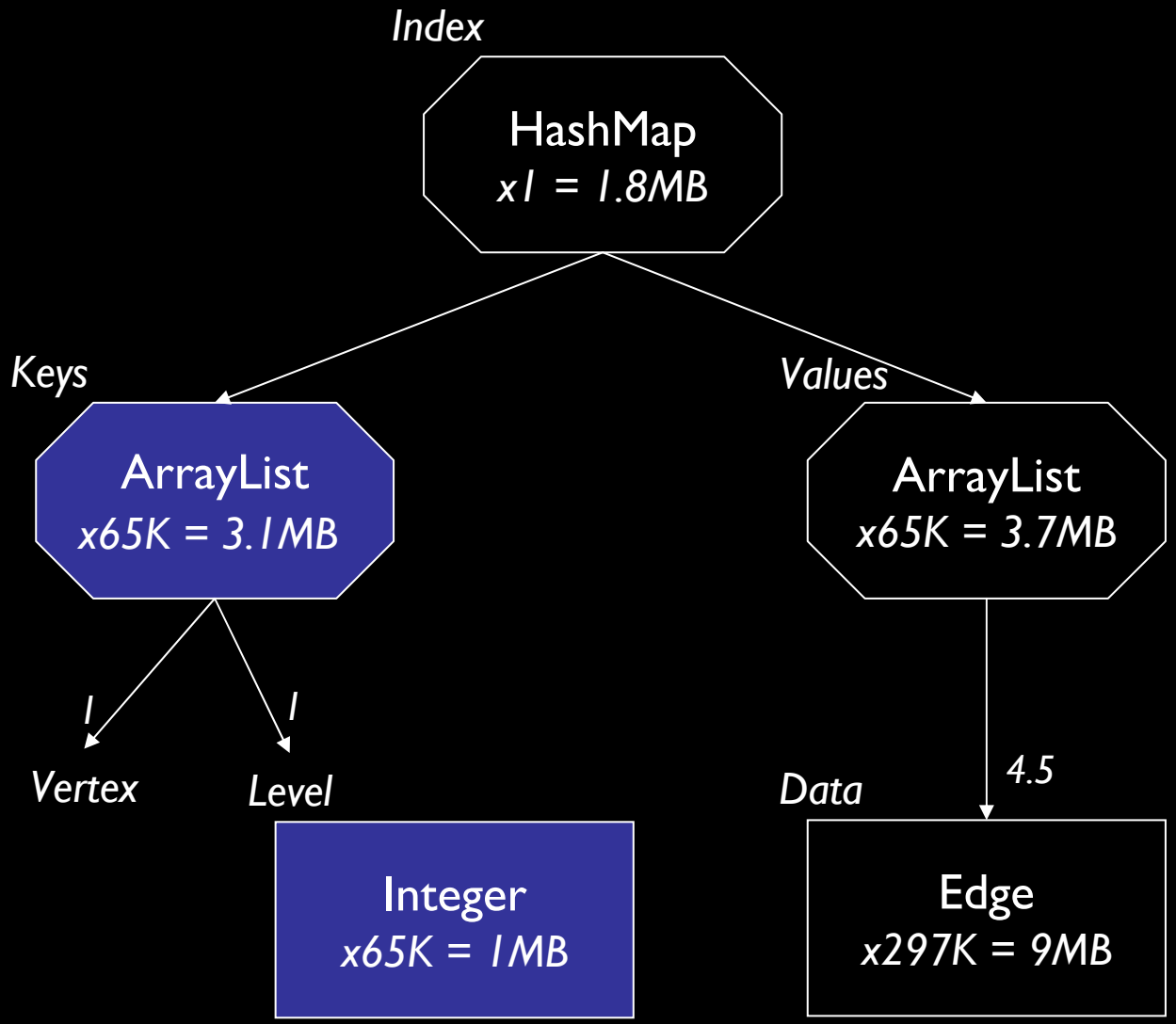
- Switched to ArrayList. Saved 77% of that region.
- HashSet functionality was not worth the cost. Uniqueness already guaranteed elsewhere

Wish list

- Gracefully-growing collections

Small collections in context

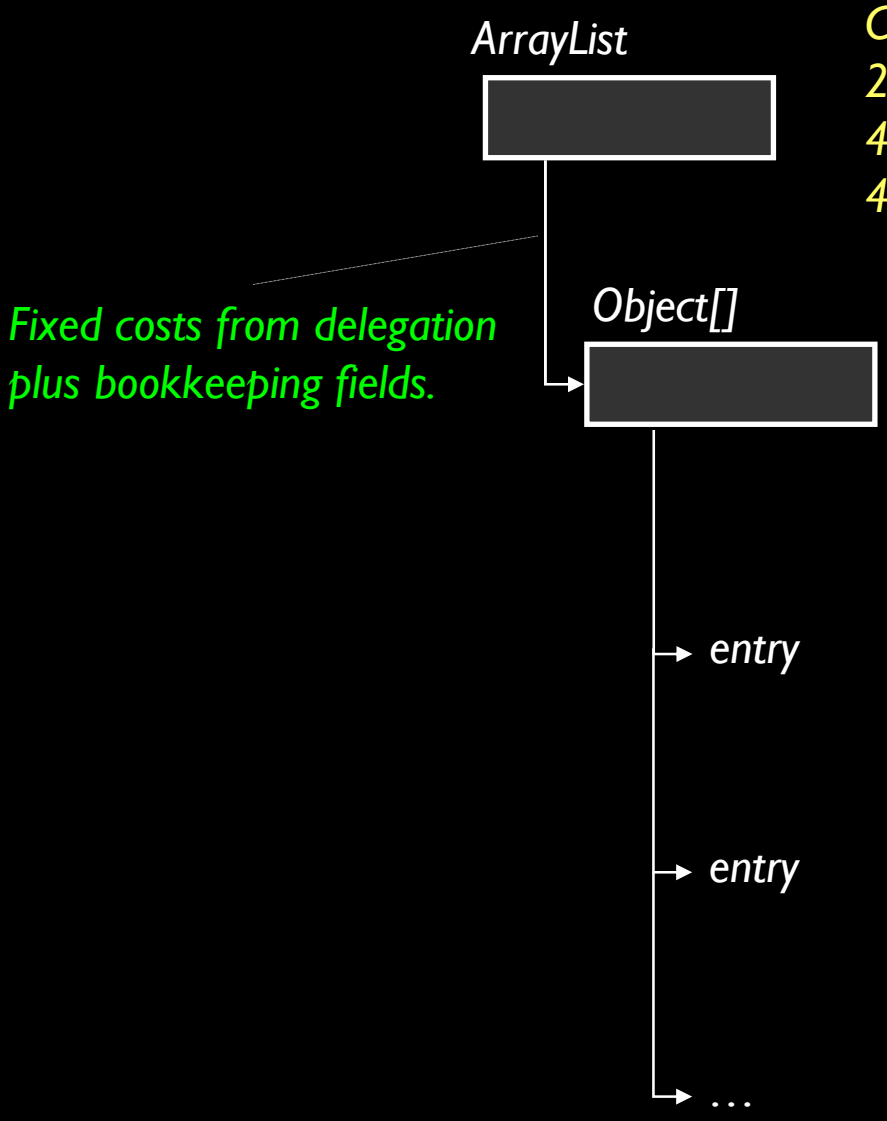
Multipart key as 2-element ArrayList



- ArrayList has a high fixed cost. Also required boxing of integers.

Inside the Java collections

ArrayList



Cost of minimally sized 2-element ArrayList: 40 bytes fixed + 4 bytes/entry

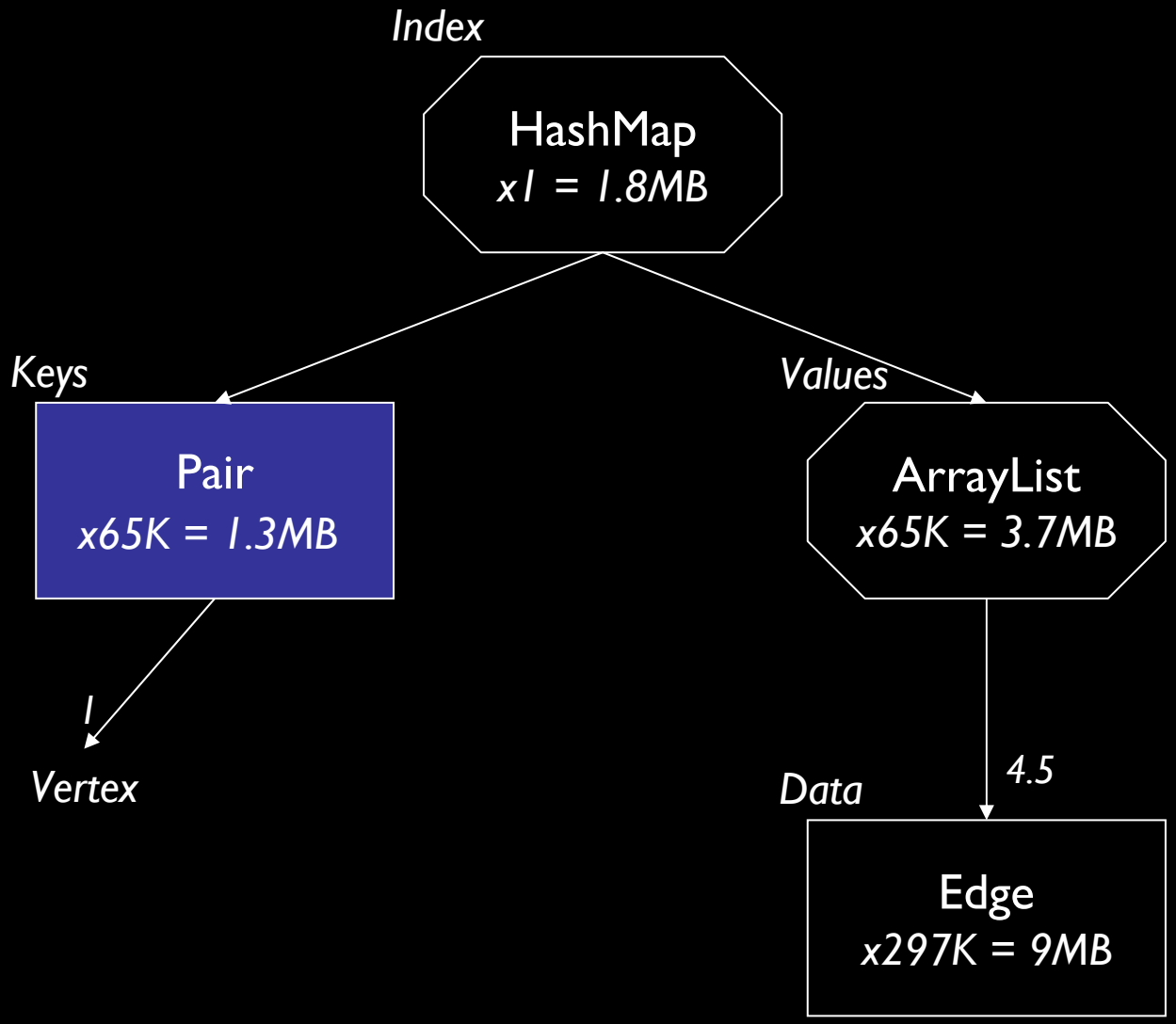
Fixed costs from delegation plus bookkeeping fields.

Default size and growth policy can mean overhead from empty slots

- Much lower fixed and variable costs than HashMap or HashSet
- Fixed costs can still add up for small collections

Small collections in context

Multipart key class

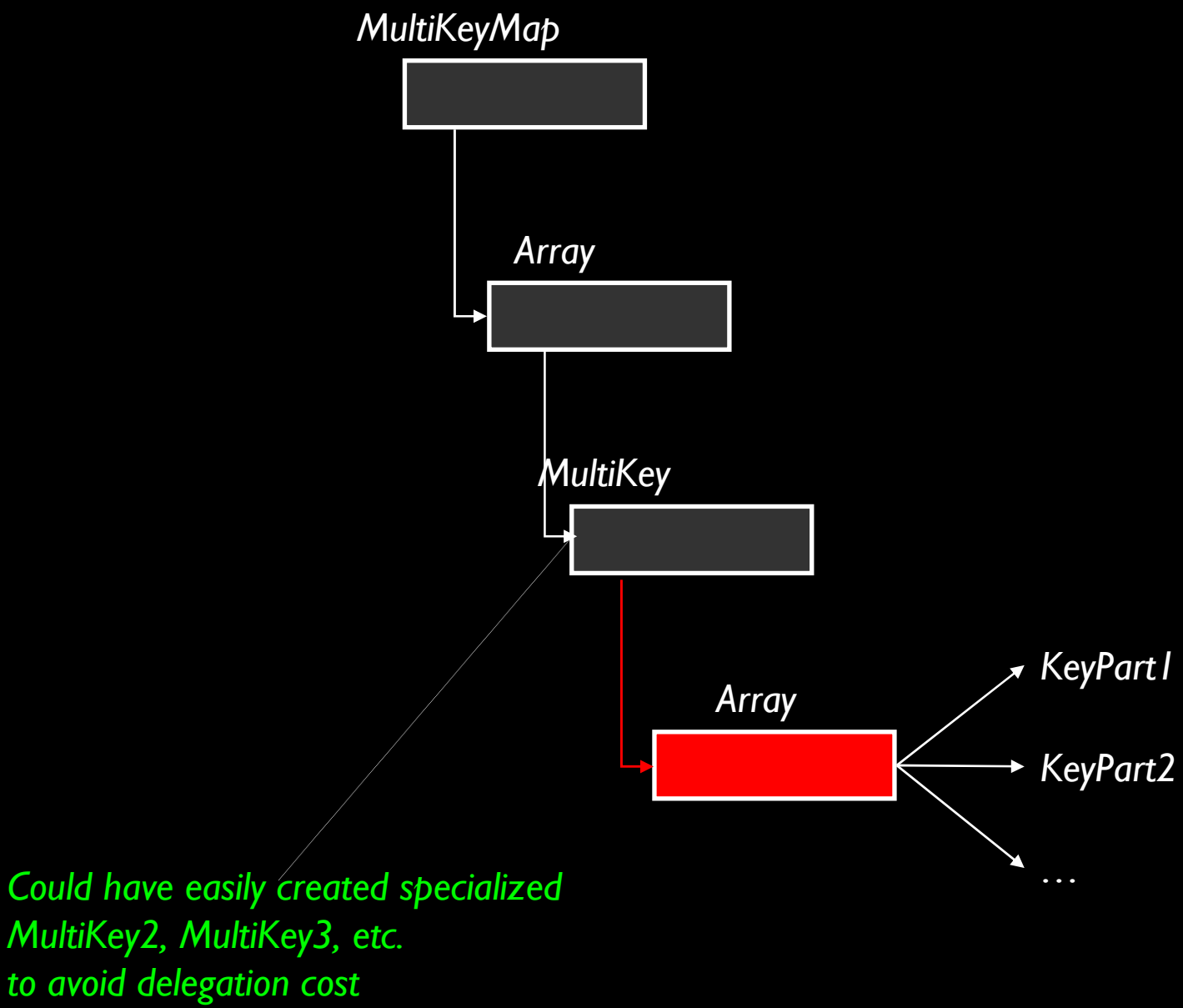


Remedy:

- Introduced Pair class (Vertex, int level)
- Again, functionality of original design was not worth the cost
- Reduced key overhead by 68%

Multipart key

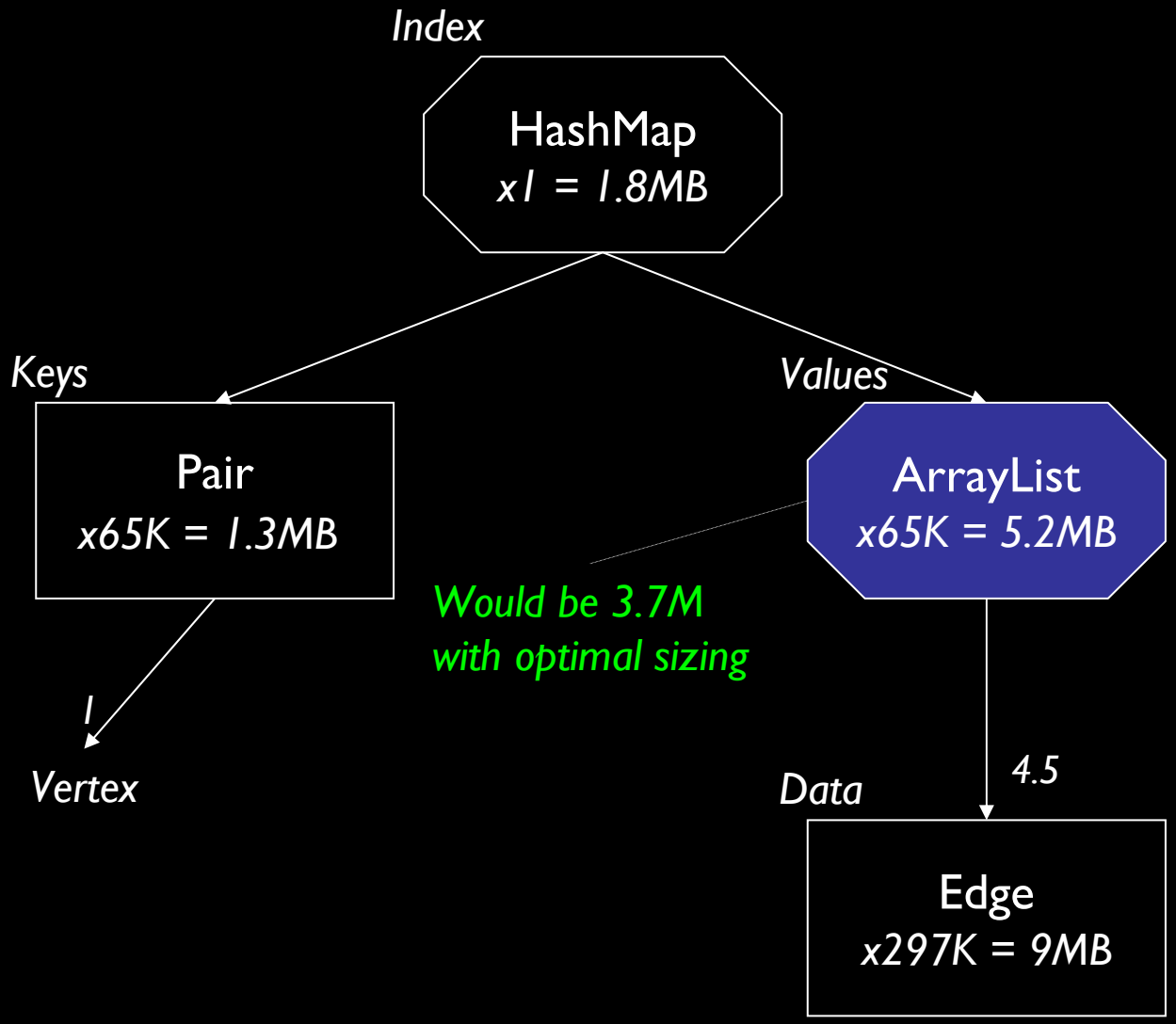
Case study: Apache Commons MultiKeyMap



- Apache Commons collections frameworks has the same pattern
- Paying for flexibility that's not needed
- Cost: additional 20 bytes per entry

Growth policies

Example: creating default-size ArrayLists



- 28% overhead in ArrayLists just from empty slots
- collections optimized for growth
- large defaults and jumps – doubling
- 10% tax on some copies

Remedies:

- Set initial capacity
- trimToSize() after load

Inside the Java Collections

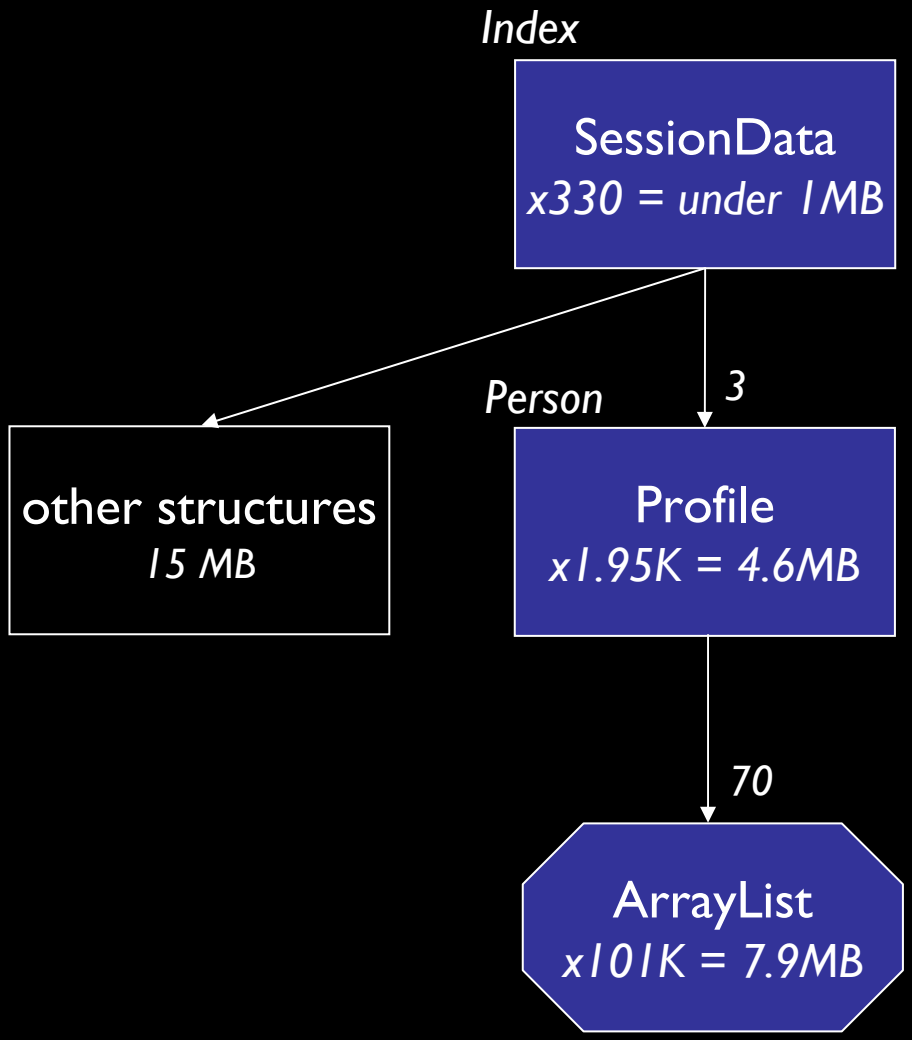
Cost of a 2-element collection

	Minimal size (bytes)	Default size (bytes)	# of slots for 2 elements using default size
LinkedList	96	96	3
ArrayList	48 or 56	80 or 88	10
HashMap	116 or 168	168	16
HashSet	132 or 184	184	16

From experiments with a few different JVMs, all 32-bit.

The cost of empty collections

Case study: CRM system, part of session data

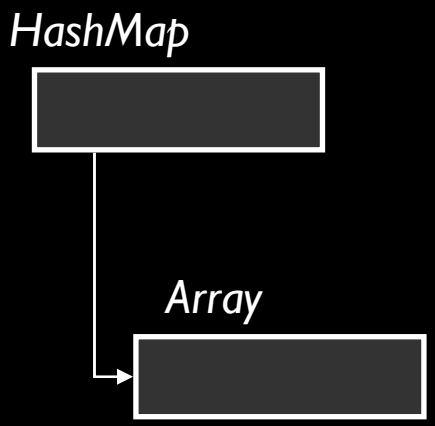


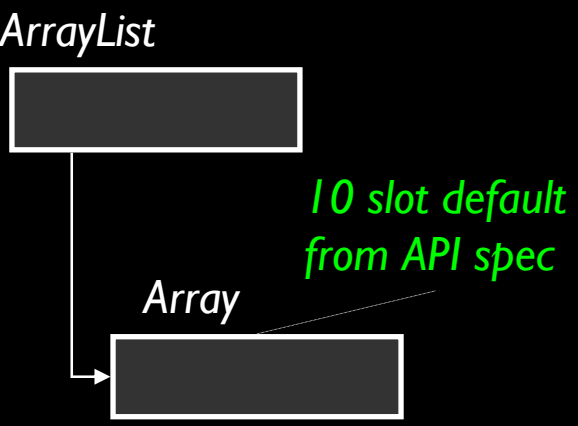
- Small run had 26M of session data. Will not scale.
- 210 empty collections per session = 28% of session cost

Remedies:

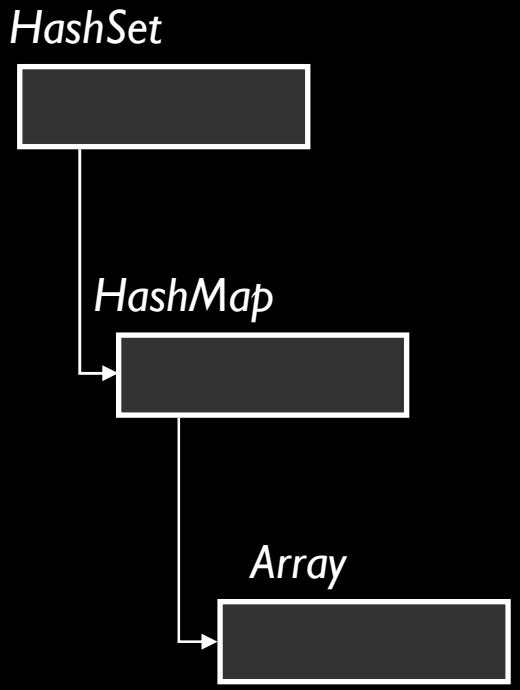
- Lazily allocate
- Collections.emptySet()
- Avoid giving out references

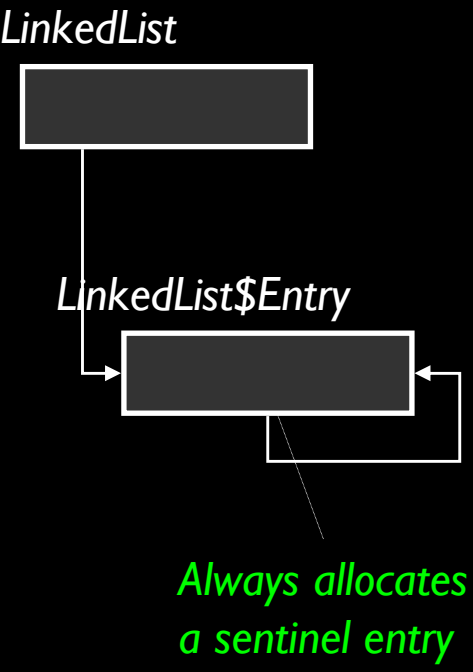
The Not-so-empty Collections





- Minimum of 2 objects each – component parts are always allocated





- Default sizing increases cost (e.g. 16 elements for HashMap/HashSet)

Inside the Java Collections

Cost of an empty collection

	Minimal size (bytes)	Default size (bytes)	Default # of slots
LinkedList	48	48	1 sentinel entry
ArrayList	40 or 48	80 or 88	10
HashMap	56 or 120	120	16
HashSet	72 or 136	136	16

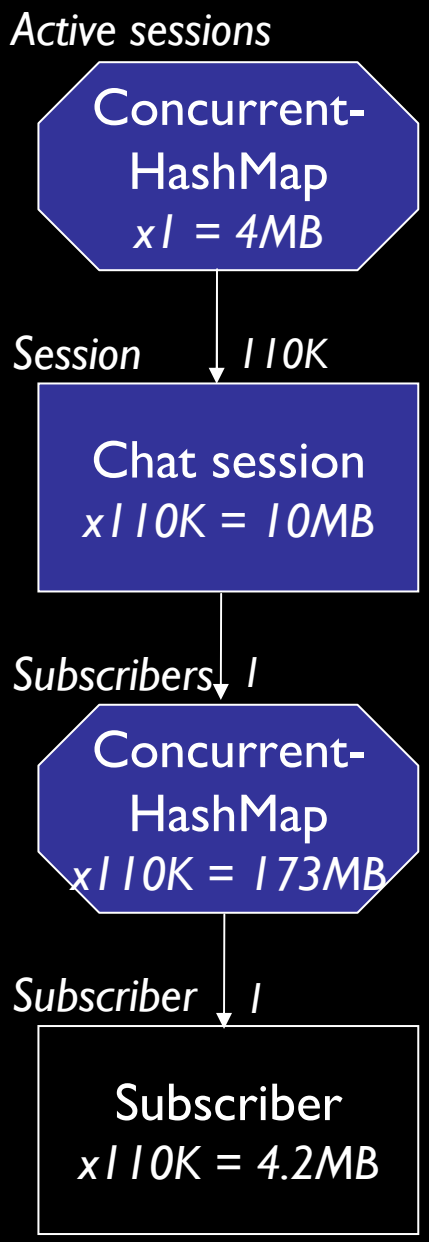
From experiments with a few different JVMs, all 32-bit.

Representing relationships

- many, high-overhead collections
  - small collections
    - special-purpose collections

Small concurrent maps

Case study: Chat server framework



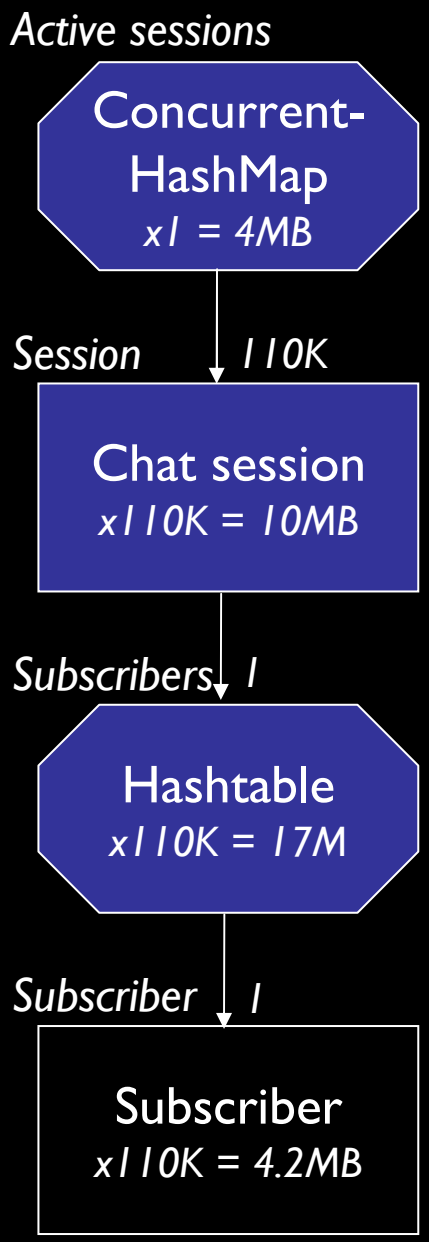
- Nested CHMs: > 1600 bytes each!
- Cost was 90% of this structure; 10-20% of total heap

What went wrong:

- Library not intended for use at this scale
- Concurrency requirements were different at fine vs. coarse grain

Small concurrent maps

Case study: Chat server framework



Remedies:

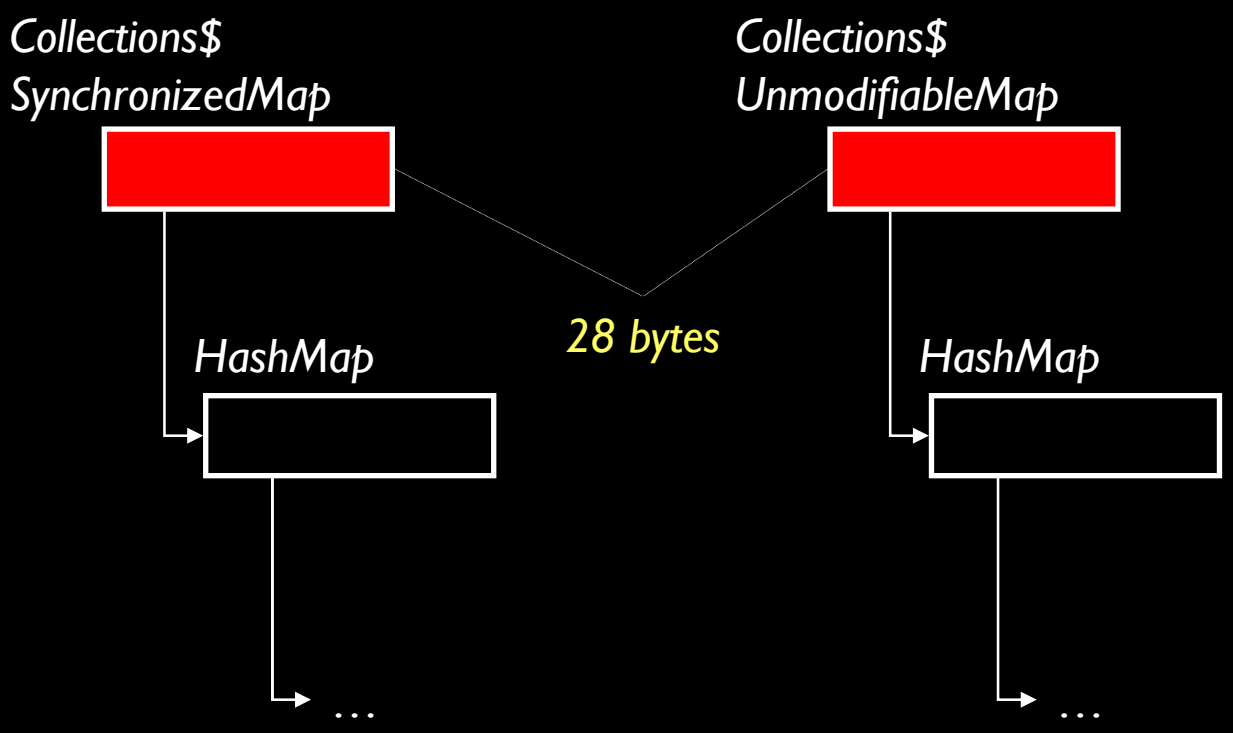
- First considered reducing width of inner ConcurrentHashMap from 16 to 3. Savings: 67%
- Used Hashtable, since high level of concurrency not needed. Savings: 90+%

Note:

- Hashtable less expensive than similar Collections$ SynchronizedMap

Inside the Java Collections

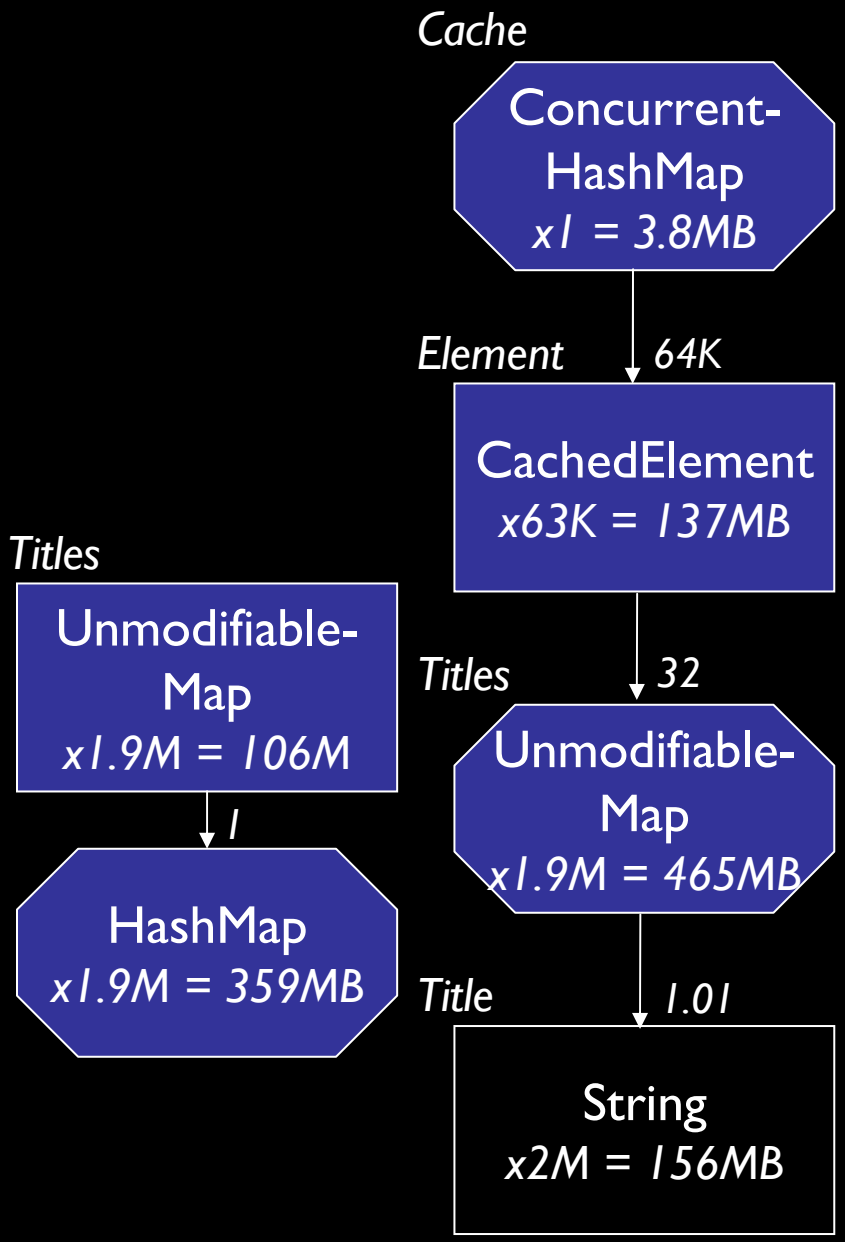
Wrapped collections



- Design is based on delegation
- Costs are significant when collections are small
- Fine for larger collections

Small wrapped collections

Case study: media e-commerce site (64-bit)



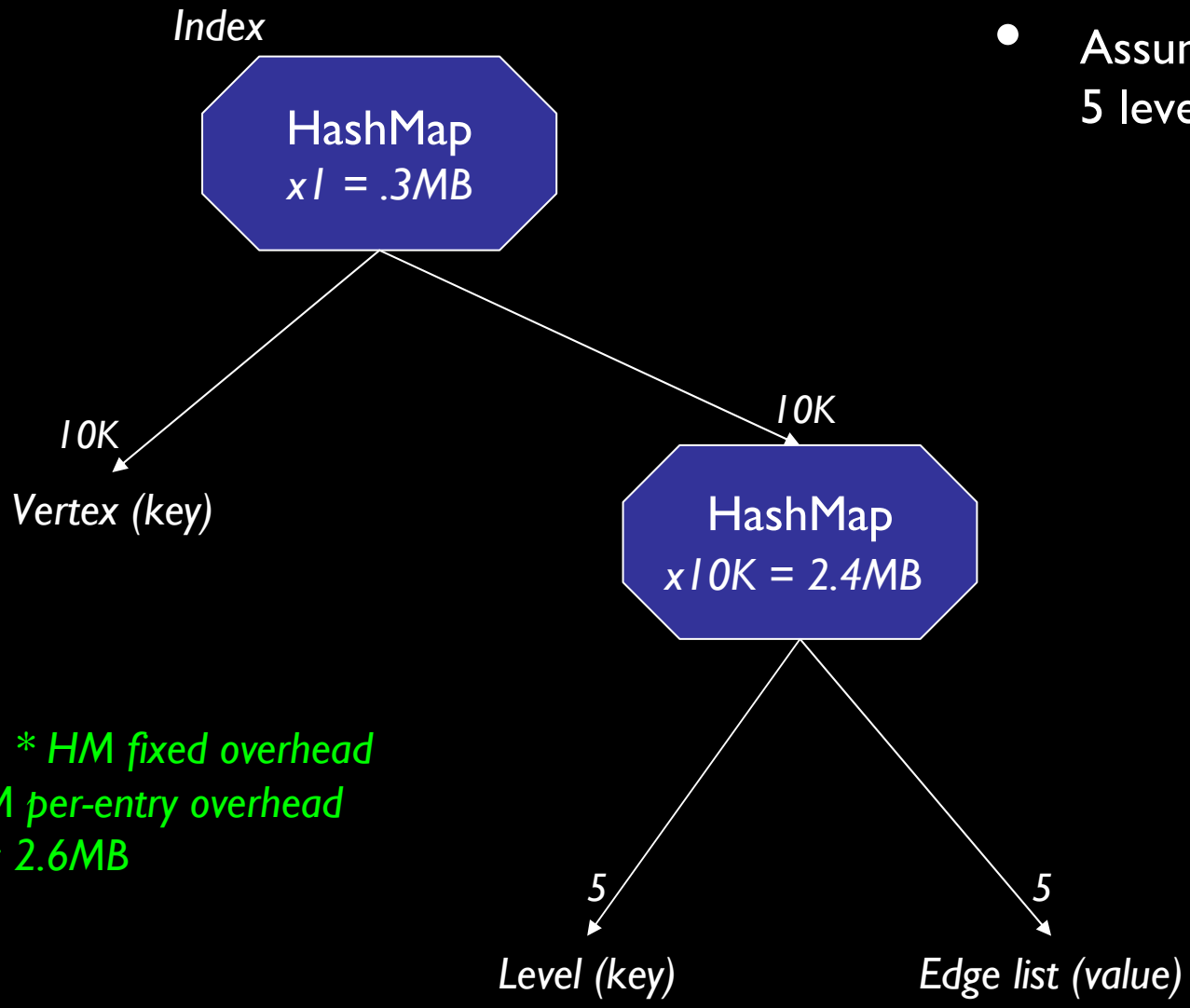
- 108MB for UnmodifiableMap wrappers. 56 bytes each
- Twice the cost as on a 32-bit JVM

What went wrong:

- Functionality not worth the cost at this scale. *Unmodifiable* serves a development-time purpose

Multikey map: design 1

Level graph edge index: as nested map

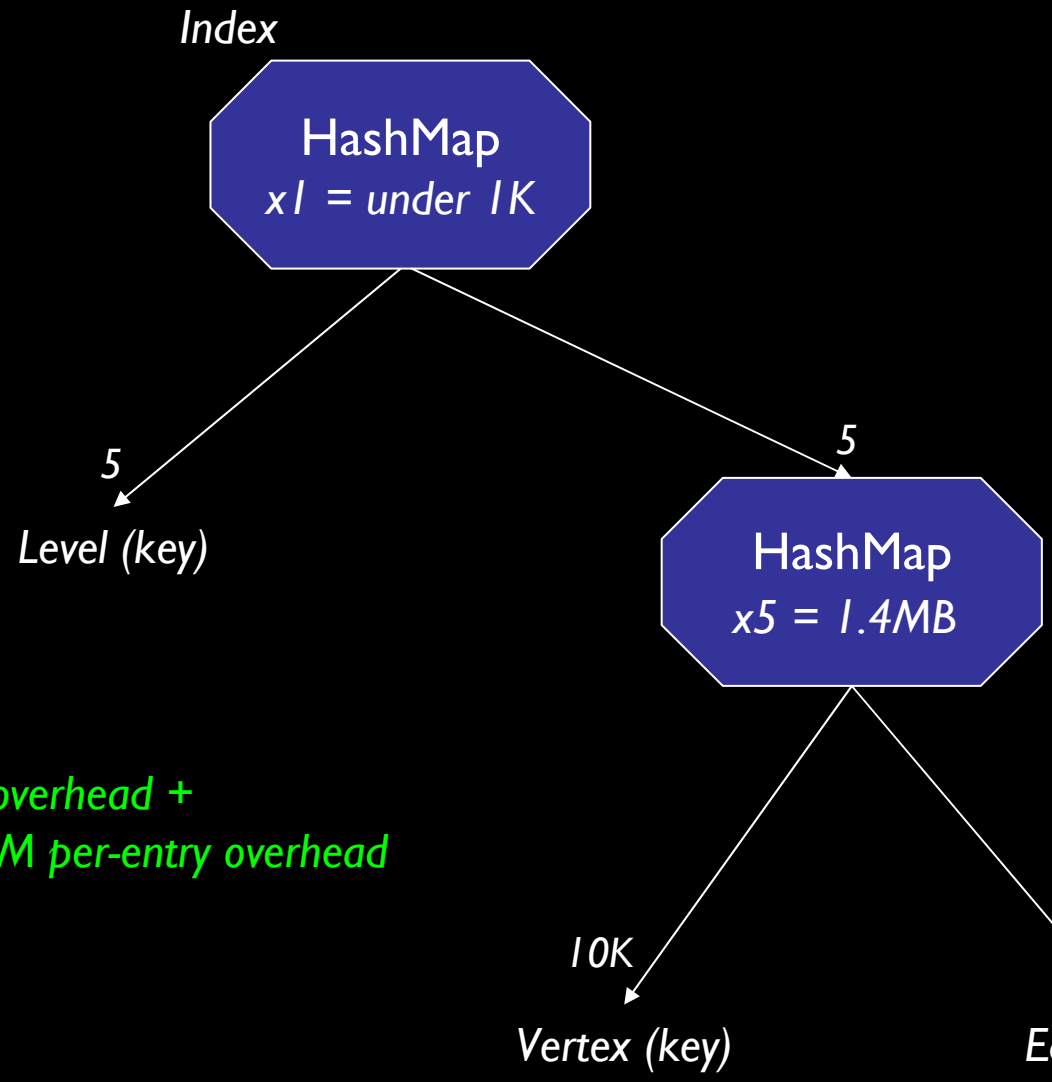


- Assume 10K vertices, 5 levels

*(10K + 1) * HM fixed overhead*
*60K * HM per-entry overhead*
Total cost: 2.6MB

Multikey map: design II

Level graph edge index: nested map, reordered

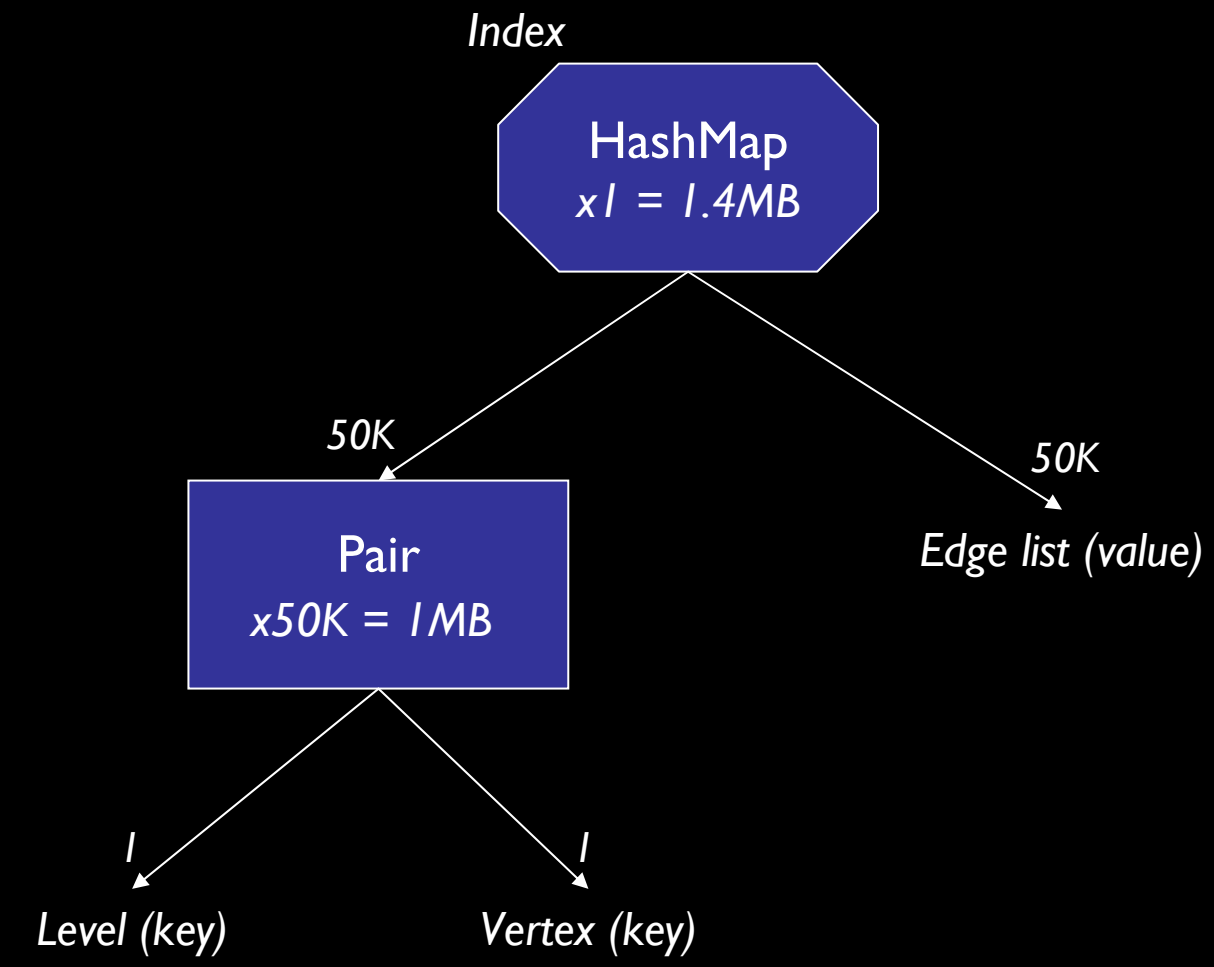


*6 * HM fixed overhead +*
*(50K + 5) * HM per-entry overhead*
Total: 1.4MB

- Switching order eliminated nested collection fixed costs
- Savings: 46%. Consistent savings as vertices increase
- Good approach if you know the distribution

Multikey map: design III

Level graph edge index: single map, multikey

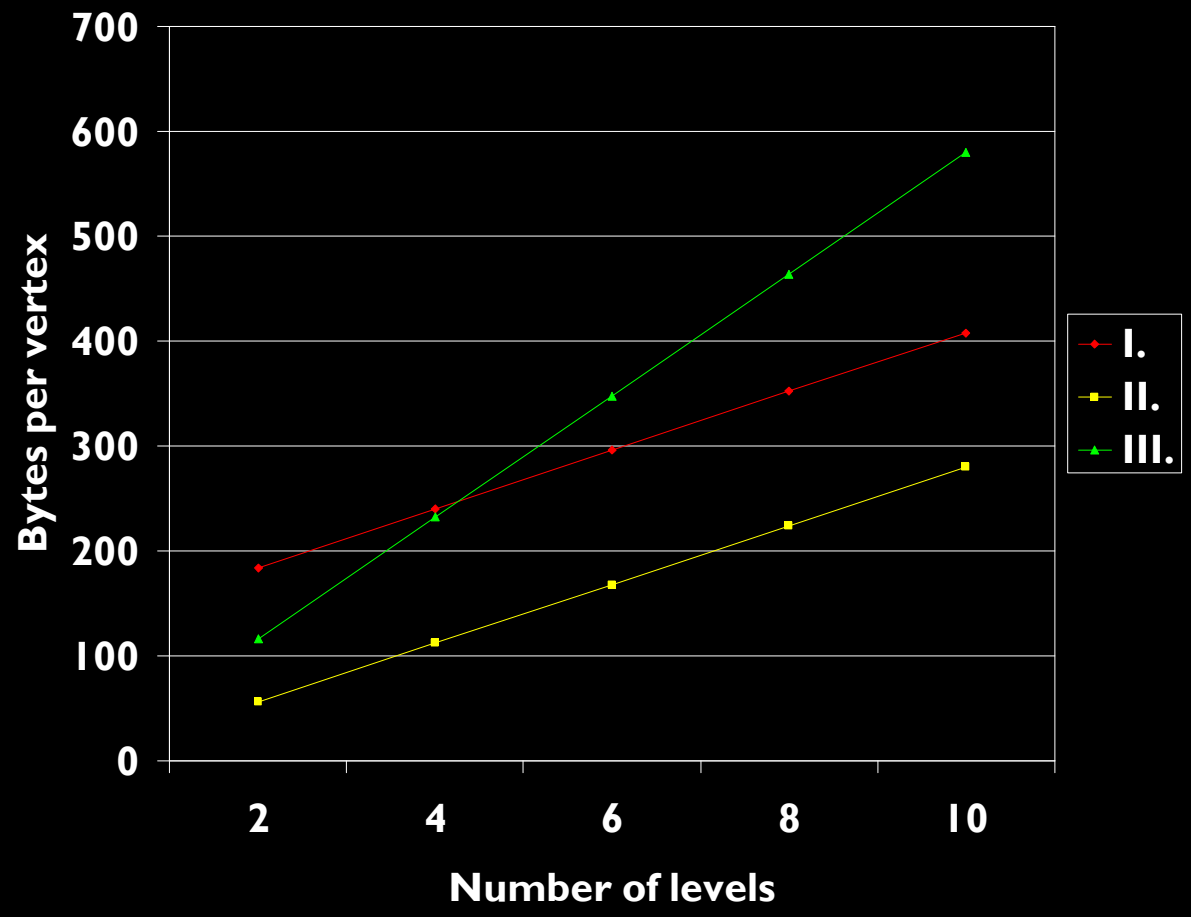


*1 * HM fixed overhead +*
*50K * HM per-entry overhead +*
*50K * Pair overhead*
Total: 2.4 MB

- 11% better than I, 70% worse than II.
- Trading fixed costs of small collections for per-element cost in a large collection: 28-byte HM entry + 20-byte Pair
- Results were surprising
- Wish list: be able to extend entry classes

Multikey map: comparison

Incremental cost per vertex



- Assume num levels is much smaller than num vertices
- Then II is consistently better than I
  - delta per vertex is constant 128 bytes
- Difference of III vs. others is sensitive to the number of levels, even within a small range

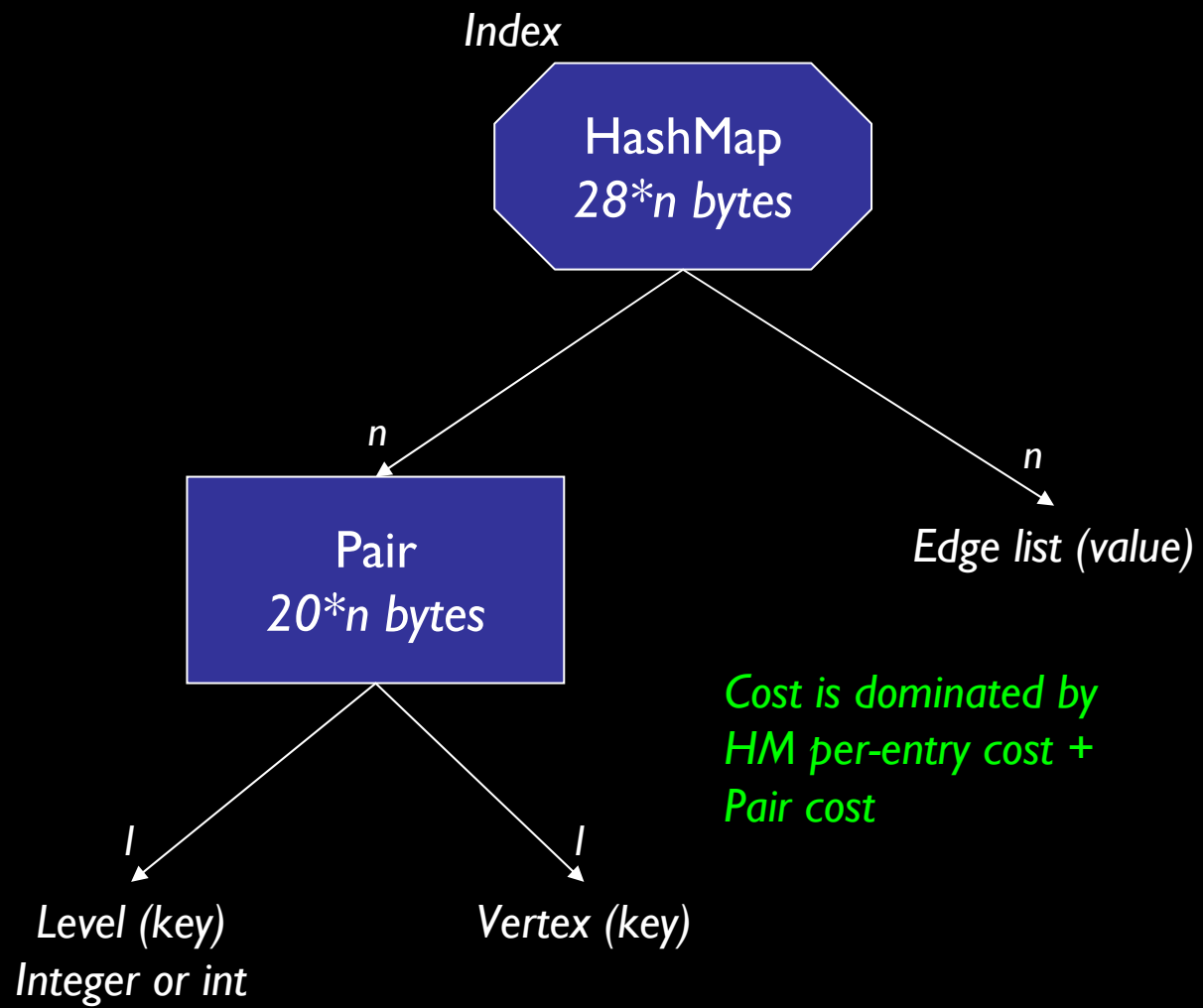
Break

Representing relationships

- large collections, high per-entry overhead relative to data

Large collections and scaling

Level graph edge index: single map, multikey



Cost is dominated by HM per-entry cost + Pair cost

- Per-element cost is constant. Constant is large relative to actual data.
- Cost: 48 bytes per element
  Overhead: 83%

What went wrong:

- high collection per-entry + delegation costs

Inside the Java Collections

Standard collections: per-entry costs.

	Per-entry cost (bytes)
LinkedList	24
ArrayList	4
HashMap	28 or 36
HashSet	28 or 36

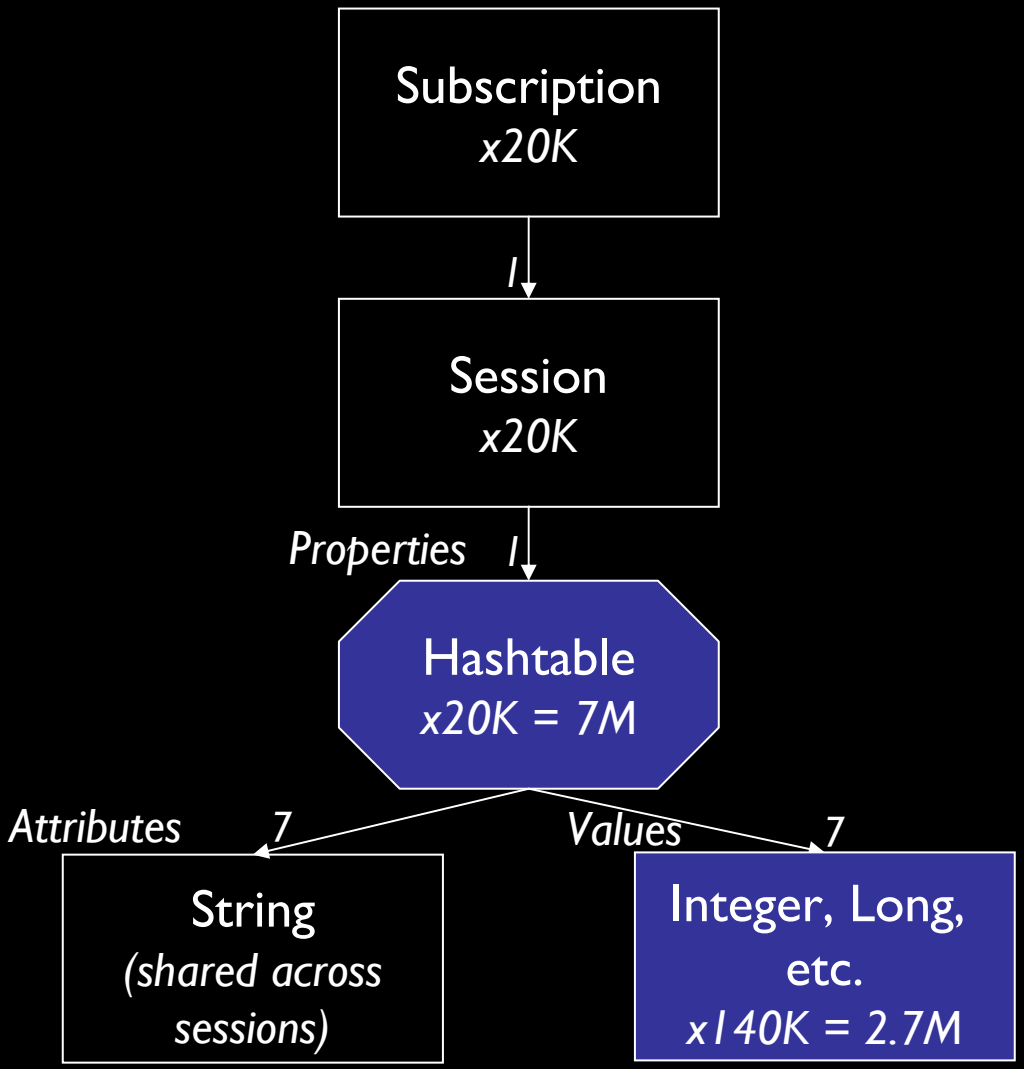
- Plus any overhead of introducing a key or value object

From experiments with a few different JVMs, all 32-bit.

Excludes amortized per-collection costs such as empty array slots. Includes pointer to entry.

Nested collections, high per-element costs

Collaboration service: storing properties



- Stores 7 properties per subscription, via session API
- HT per-entry, boxing costs add 350 bytes overhead per session, impeding scalability

What went wrong:

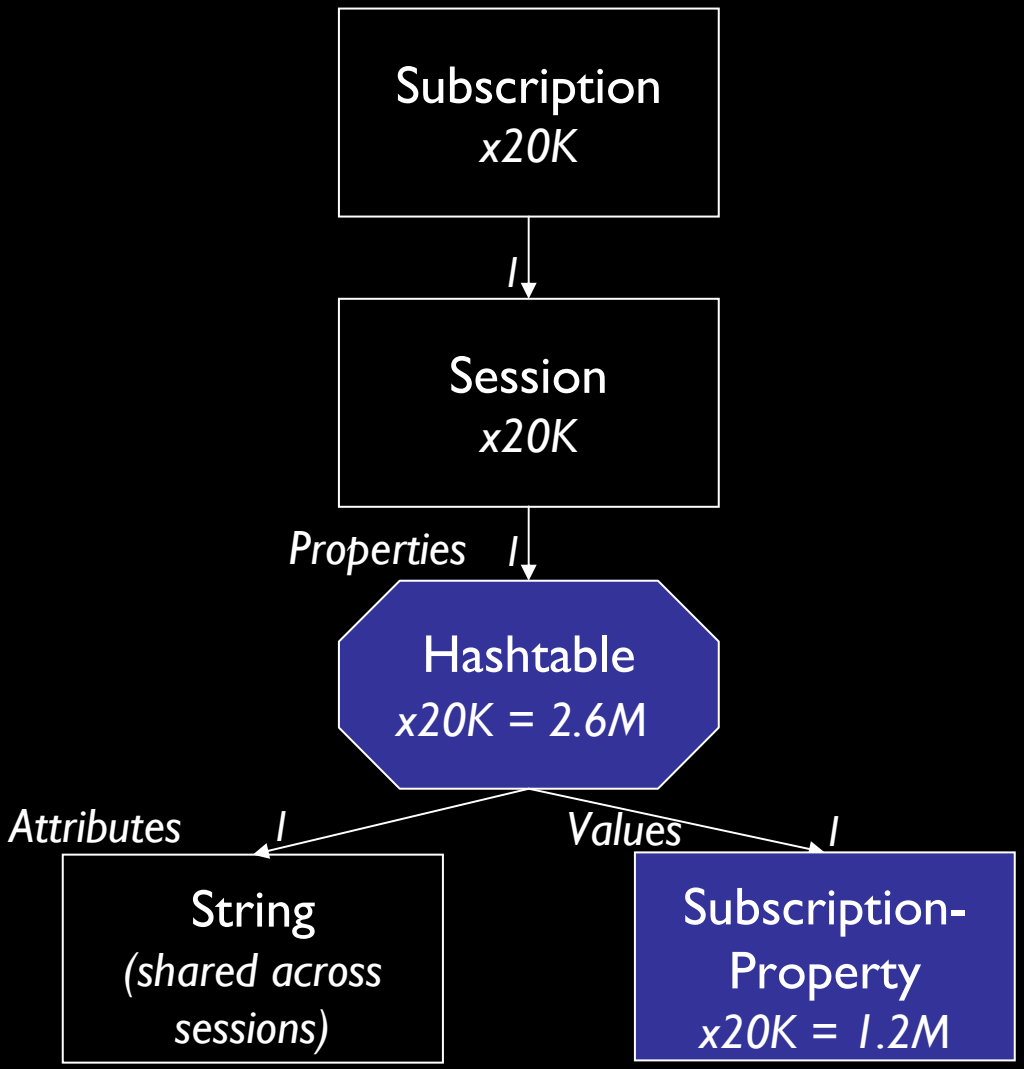
- Cost obscured by multiple layers, fanouts

What went right:

- Shared attribute names across sessions

Nested collections, high per-element costs

Collaboration service: storing properties



Remedy:

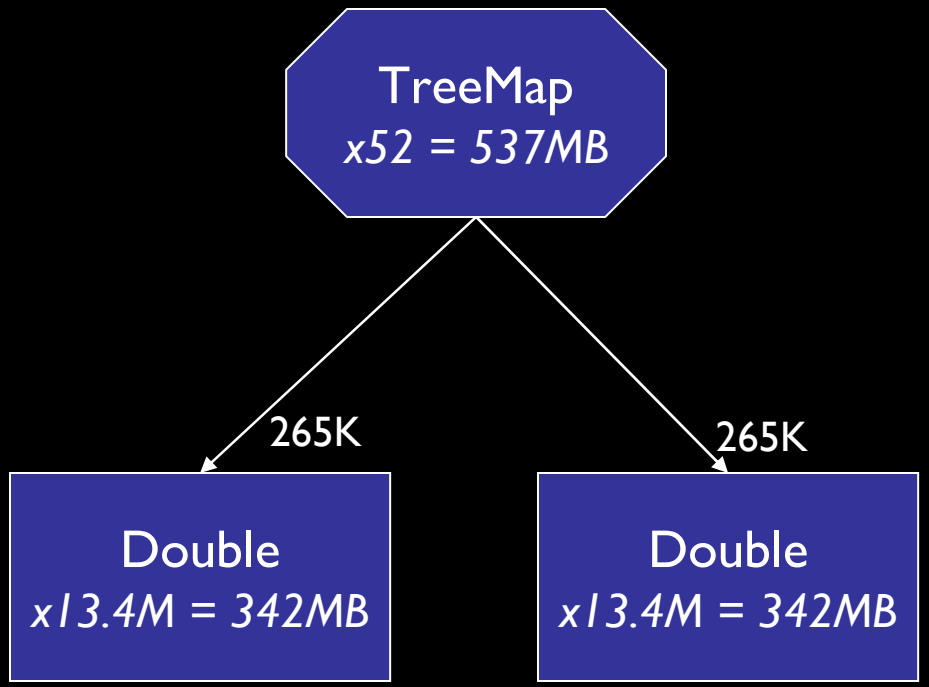
- Combined properties into a single high-level property, inlining scalar values
- 7 : 1 reduction in collection entry costs, plus reduced boxing costs
- Note: still paying for HT fixed cost

Representing relationships

- large collections, high per-entry overhead relative to data
  - special-purpose collections

Collections involving scalars

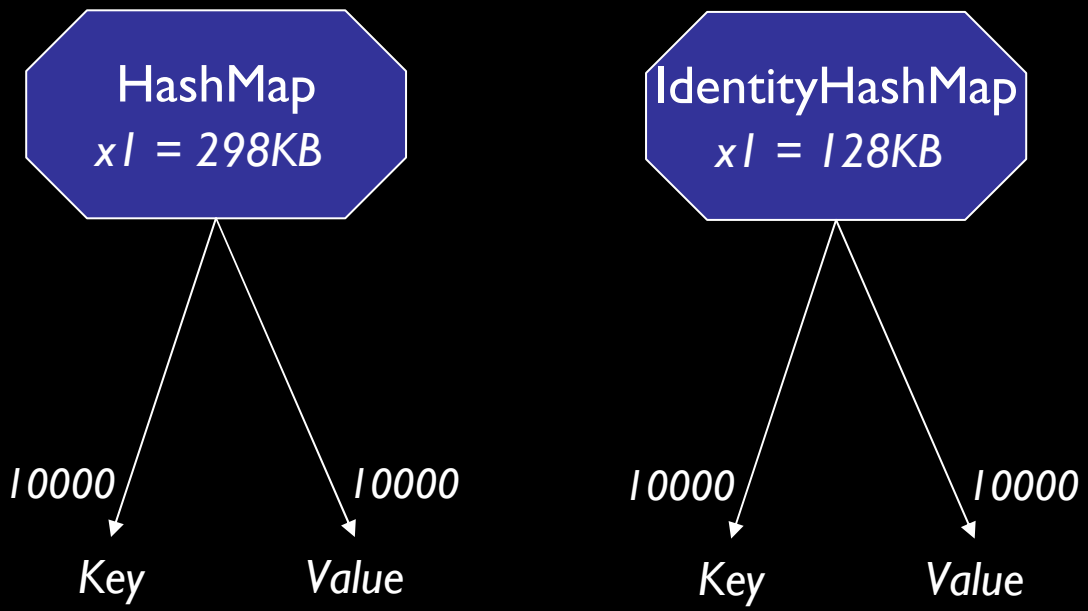
Case study: monitoring infrastructure



- Data structure took 1.2GB
- Overhead is still 82% at this giant scale
- Some alternative scalar maps/collections available, with much lower overhead

Identity maps

Comparison: HashMap vs. IdentityHashMap



- For maintaining a map of unique objects, where the reference is the key
- Equality based on ==
- Open addressing implementation avoids the cost of Entry objects
- Cost reduced by 59% in this experiment

Collections & Scaling

The health near the leaves will limit a design's scalability

- Fixed costs of nested collections
- Constant costs for elements
- Collection per-entry costs
- Delegation overhead of contained data

The standard collections

JDK Standard Collections

- Speed has been the focus, not footprint

IBM (Harmony) and Sun implementations not that different in footprint

Hard-wired assumptions, few policy knobs (e.g. growth policies)

Specialized collections are worth learning about:

- IdentityHashMap, WeakHashMap, ConcurrentHashMap, etc.

Collections alternatives

Apache Commons

- Many useful collections:
  - Flat3Map, MultiMap, MultiKeyMap
- Focus is mostly on functionality. Maps allow some extension.
- Footprint similar to standard, with a few exceptions

GNU Trove

- Many space-efficient implementations
- e.g. scalar collections
- e.g. list entries without delegation cost

Cliff Click nonblocking; Javolution; Amino

Specialized collections within frameworks you use

Important: check your corporate policy re: specific open source frameworks

Collections: summary (for developers)

- Choosing and configuring carefully can make a big difference (within limits)
  - consider context of collections as well
- Avoid writing your own if possible

Collections: community challenges

- Efficiency improvements to the standard collections that match the common use cases
  - Implementations and APIs
- Benchmarks that consider both space and time

Collections: deeper challenges

- Better libraries will only go so far
  - e.g. a few million objects to represent a relationship
- Java / runtime features to enable much more optimal representations. Some possibilities:
  - Richer data modeling features (e.g. to reduce delegation)
  - Adaptive representations
  - Bulk storage
  - Specification or inference of intent (e.g. nested maps, relationships, caches, dynamic types)
- Tools that help developers make choices

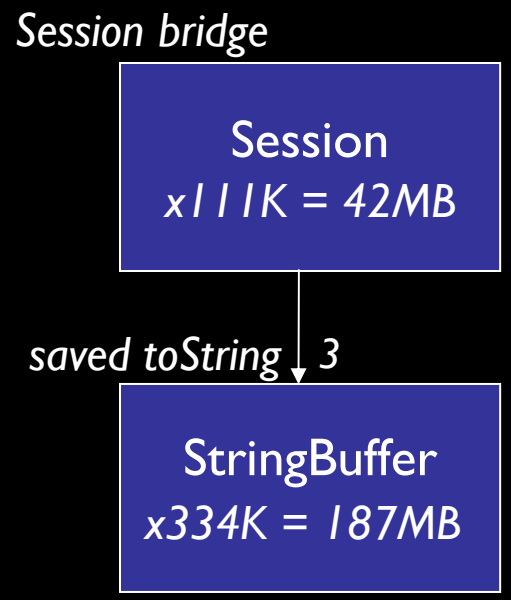
Modeling your data types

- Too much data

Saving formatted data I

Case study: one layer of chat framework

Session data:



- 82% of cost of this layer, due to saving computation of toString()

What went wrong?

- Empty space overhead in StringBuffer
- Space cost not worth the time savings

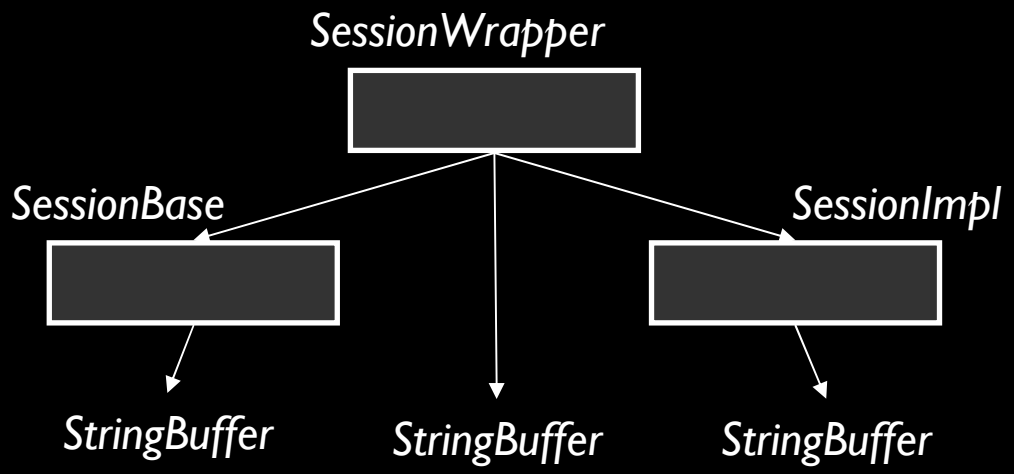
Remedies:

- String, not StringBuffer
- Recompute as needed

Saving formatted data I: delegation effects

Case study: one layer of chat framework

Inside each Session:



- Data type had been split in three
- Same coding pattern copied to each part

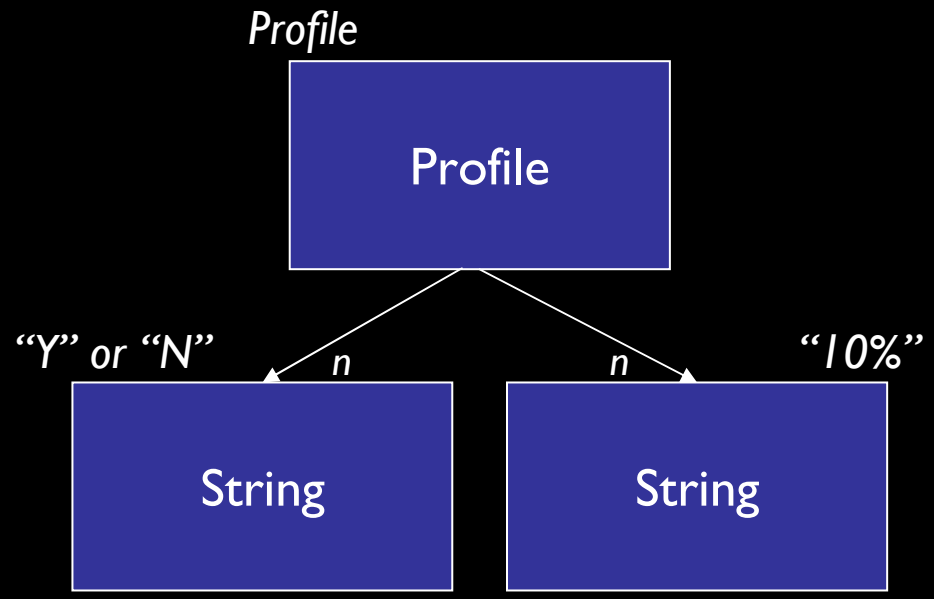
What went wrong?

- Delegated design magnified other costs

Saving formatted data II

Case study: CRM system

Session state fragment:



- Saving formatted data
- Some were constants ("10%"). Some had few values ("Y", "N")
- Storing a boolean as a String. Health ratio is 48 : 1

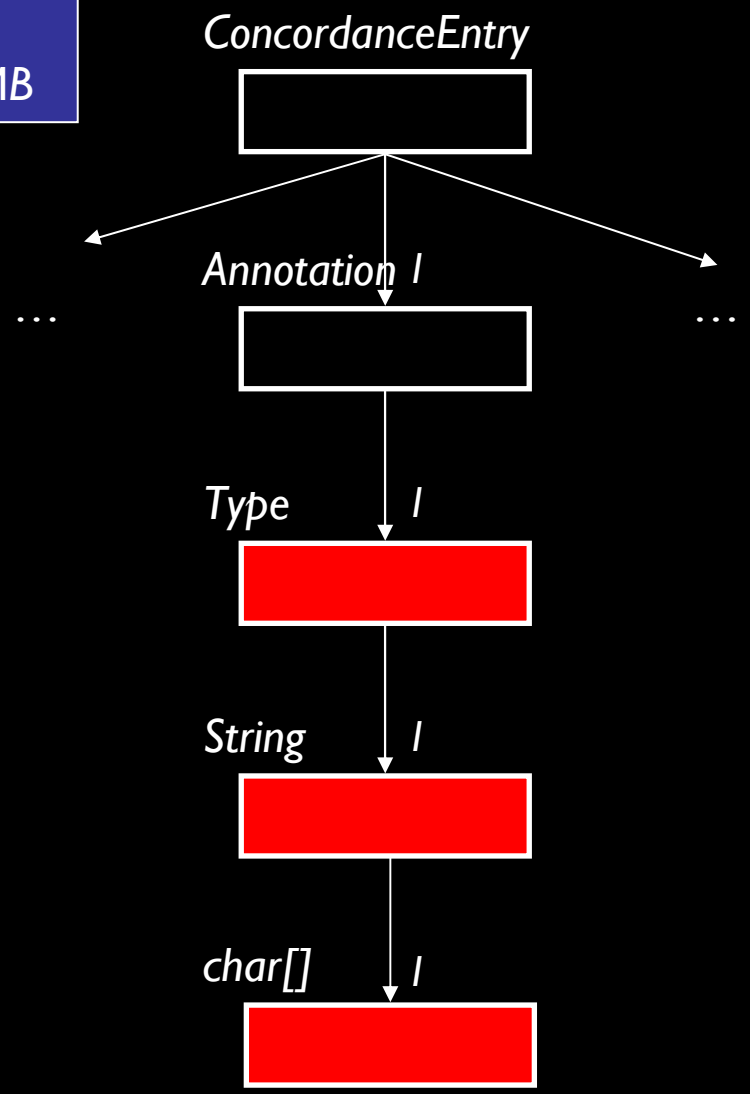
What went wrong?

- Duplicating data with high-overhead representation
- Space cost not worth the time savings

Duplicate, immutable data

Case study: Text analysis system, concordance

Concordance-Entry
x131K = 41MB



- 17% of cost due to duplication of Type and its String data
- Only a small number of immutable Types

What went wrong?

- Interface design did not provide for sharing
- Full cost of duplication was hidden

Remedy

- Use shared immutable factory pattern

Background: sharing low-level data

String.intern()

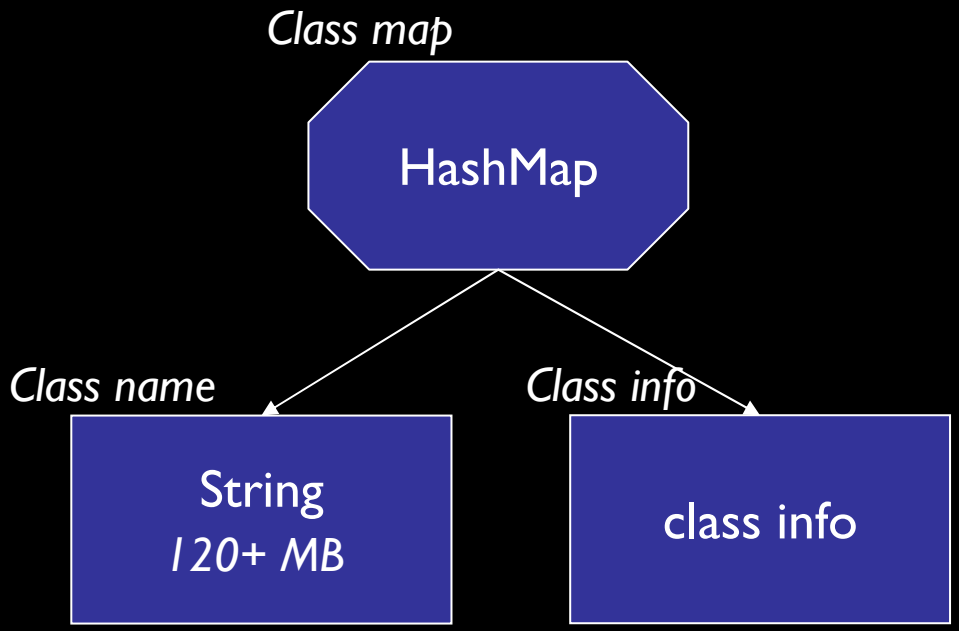
- You specify which Strings to share
- Shares the String object and the character array
- Make sure it's worth it, since there is a space cost
- Myth that is causes memory leaks
  - Though can hit permspace limits

Boxed scalars

- Integer.valueOf(), Boolean.valueOf(), etc.
- Shares some common values (not all)
- Make sure you don't rely on ==

Common-prefix data

Case study: application server, class cache



- Class loader map of class names to jar files
- > 120M of Strings, mostly duplicate prefix information

What went wrong?

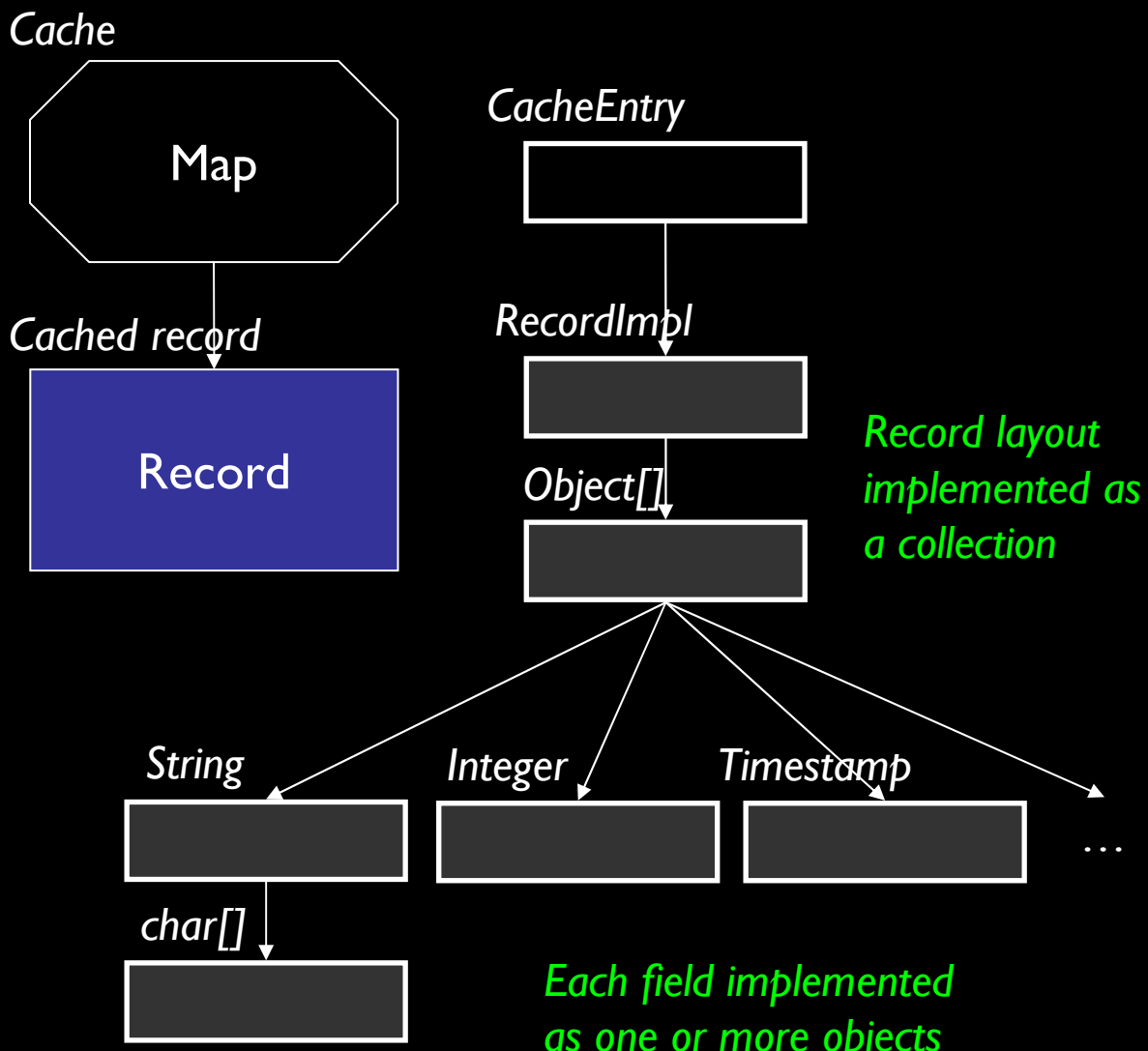
- Duplication cost
- Deeper problem: misplaced optimization

Remedy

- Implemented trie
- Simpler, 2-part factoring can also work

Dynamic types

Case study: caching mechanism



Granularity inflation

- Object → Collection
- Field → Object(s)

Costs

- 92% overhead
- Barrier for JIT

Many cases, variants

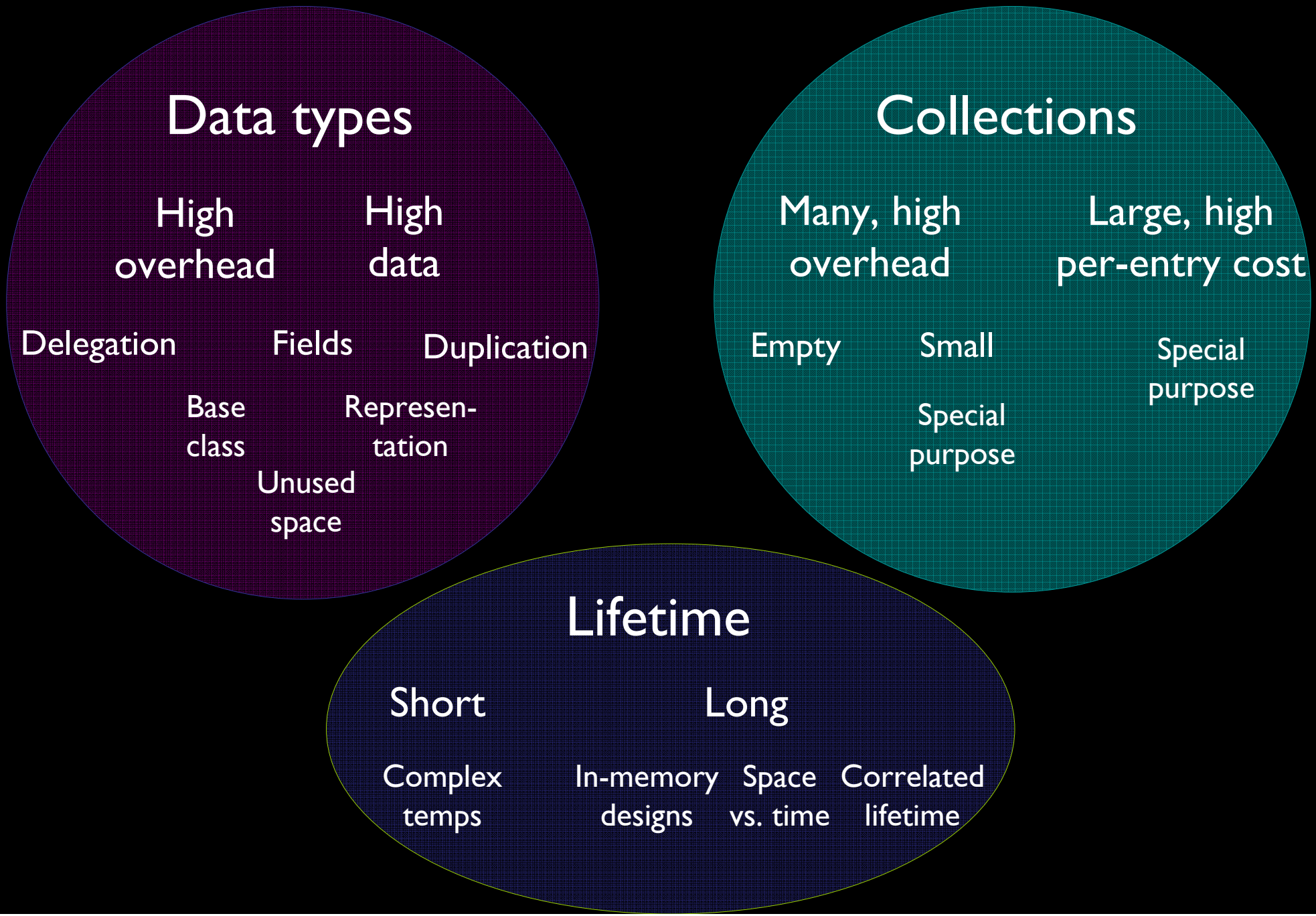
- e.g. XML DOM, mediators, temporaries
- Map version, with field names on each record

Data modeling: more challenges for the community

- Duplicate, unchanging data is a major source of footprint bloat
  - Strings as well as whole structures
  - Layers of encapsulation and large-scale boundaries (e.g. plugins, ear files) make duplication beyond the scope of developers to address
- Strings are 30-50% of the heap, collections 10-15% or more
  - Why are they so prevalent? What are they representing?
- Dynamic types without inflation
  - Both long- and short-lived

Managing object lifetime

Patterns of memory usage



Managing object lifetime

- short-lived data

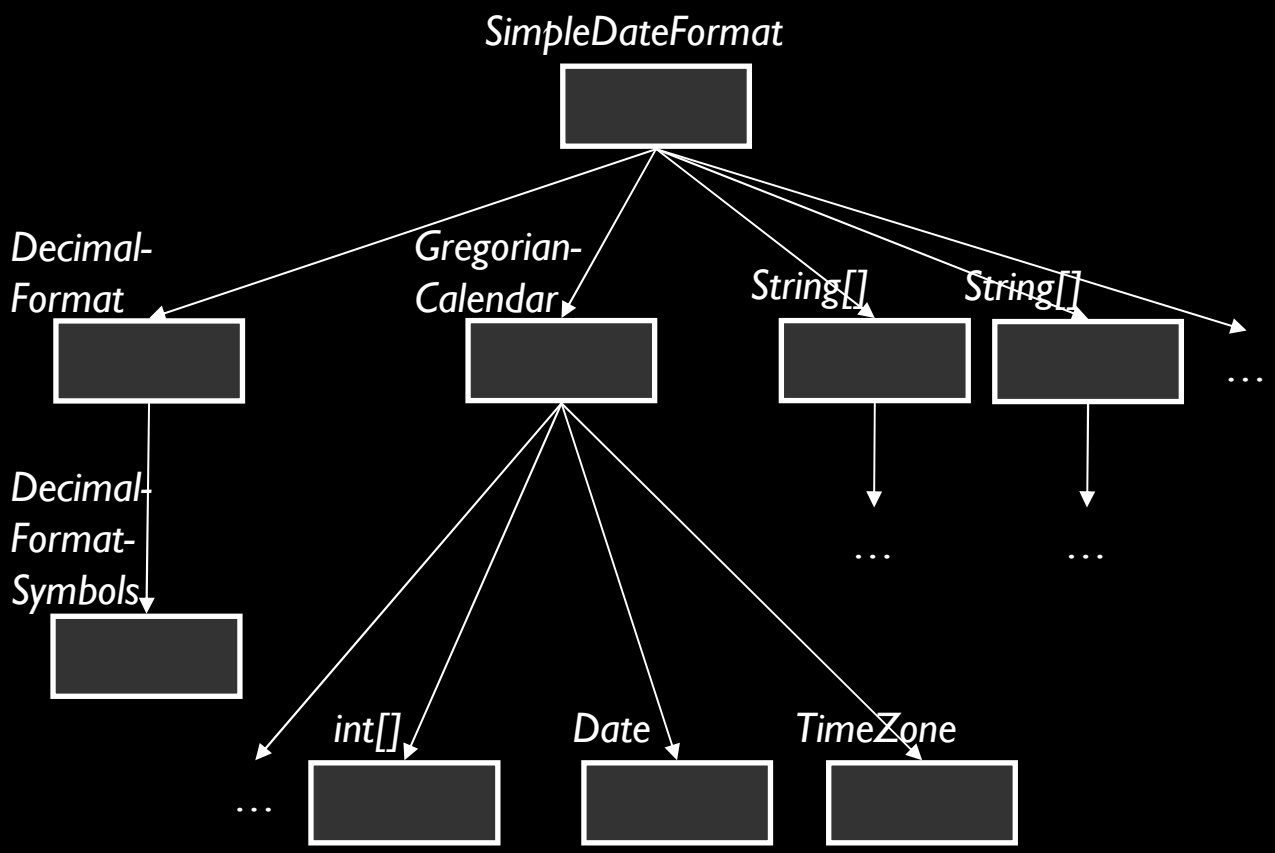
Temporaries

Aren't temporary objects free these days?

- Some are, and some definitely aren't

Expensive temporaries

Example: SimpleDateFormat



- Costly construction process. Each call to the default constructor results in:
  - 123 calls to 55 distinct methods
  - 44 new instances
- Designed for costs to be amortized over many uses
- Remedy: reuse via a local variable or thread-local storage

Background: ThreadLocal storage

- ThreadLocal: JDK-supplied per-thread variable
- An application can create many of these for different purposes
- Enables reuse without introducing concurrency problems

Tradeoffs

- *Converter, formatter, factory, schema, connection, etc.* may be good candidates for reuse. They can be expensive to create, and are often designed for reuse

- Use ThreadLocal or specialized resource pools, depending on requirements
  - Sometimes local variables are good enough
  - Avoid writing your own resource pools

- Not worth caching simple temporaries
  - Some temporaries are inexpensive to create (e.g. Integer, many iterators)
  - ThreadLocal access is usually a hash lookup

Managing object lifetime

- long-lived data

Managing lifetime: understanding requirements

Three very different reasons for long-lived data

1. In-memory design. Data is in memory forever
2. Space vs. time. Data may be discarded and recomputed
3. Correlated lifetime. Data alive only during the lifetime of other objects or during specific phases

Each has its own best practices and pitfalls

Many problems stem from misunderstanding requirements

Managing Object Lifetime

- *If not careful, extending the lifetime of objects can introduce concurrency problems, leaks, and additional memory overhead from structures that manage lifetime*

Managing object lifetime

- long-lived data
  - in-memory designs

The limits of objects

Case study: memory analysis tool

Requirement: analyze 80-million object heap

Design: one object per target application object

Hypothetical minimum: if each object needed just 4 fields (type, id, ptr to references, flags):

80M x 32 bytes =
2.5G just to model application objects!

To model references (2-3 per object), and leave scratch space for algorithms, design would require at least 10G

- Some object-oriented designs will never fit in memory
- Estimate and measure early

Note

- An earlier design used a modeling framework with high overhead costs. Just optimizing those costs would not have been sufficient.

The limits of objects

Case study: memory analysis tool

Solution:

- Backing store using memory-mapped files (java.nio)
- Built a column-based storage infrastructure with scalar arrays, to reduce working set and avoid object header costs
  - Specialized for this application's access patterns
  - Don't try this at home!
- Result is highly scalable – runs in a 256M heap

Recommendations:

- java.nio is one way to implement a backing store
- Column-based approach is a last resort. For optimization of highly specialized and protected components
  - some XML DOM implementations use this approach

Managing object lifetime

- long-lived data
  - space vs. time designs

Space vs. time designs: mechanisms

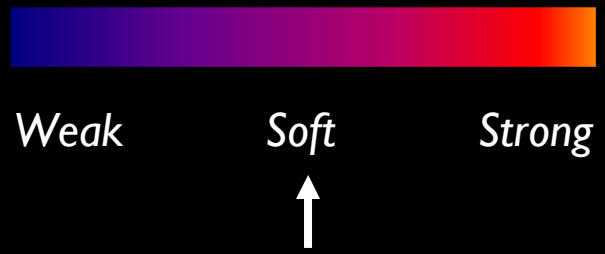
Mechanisms for saving time by maintaining data in memory:

- caches
- resource pools

Also:

- thread-local storage
- adding computed fields to data types

Background: Soft References



Soft References:

- Tells GC to reclaim these objects *only when the space is really needed*
- Will keep an object alive after it is no longer strongly referenced, just in case it is needed again
- Used mostly to avoid recomputation
  - e.g. for caches and resource pools
  - e.g. for side objects (cached fields) which can be recreated if lost

Caches & pools: a sampling

Case study: class loader "cache"

- > 100M of classname strings
- Implemented an in-memory design. Purpose was for performance - should have been a small, *bounded* cache
- Cache itself was only needed during startup

Case study: high-volume web application

- Unbounded growth (leak). An object pool framework was used for 20 different purposes, to improve performance. Unbounded size; strong references.
- Solution: soft references

Case study: financial web application

- Cache sized too large, aiming for 95% hit rate
- Result: performance problems due to excessive GC

- Caches & pools should always be bounded
- Larger caches aren't necessarily better

Caches & resource pools: best practices

Soft references are useful for implementing simple caches/pools, but …

- Relying solely on soft references gives up control over policy
- May not leave enough headroom for temporary objects, causing the GC to run more often

Caches / pools should in general be bounded in size

Soft references can be used as an additional failsafe mechanism

Many implementations of caches and resource pools are available

Avoid writing your own if possible

Managing object lifetime

- long-lived data
  - correlated lifetime designs

Correlated lifetime

Objects needed …

… only while other objects are alive

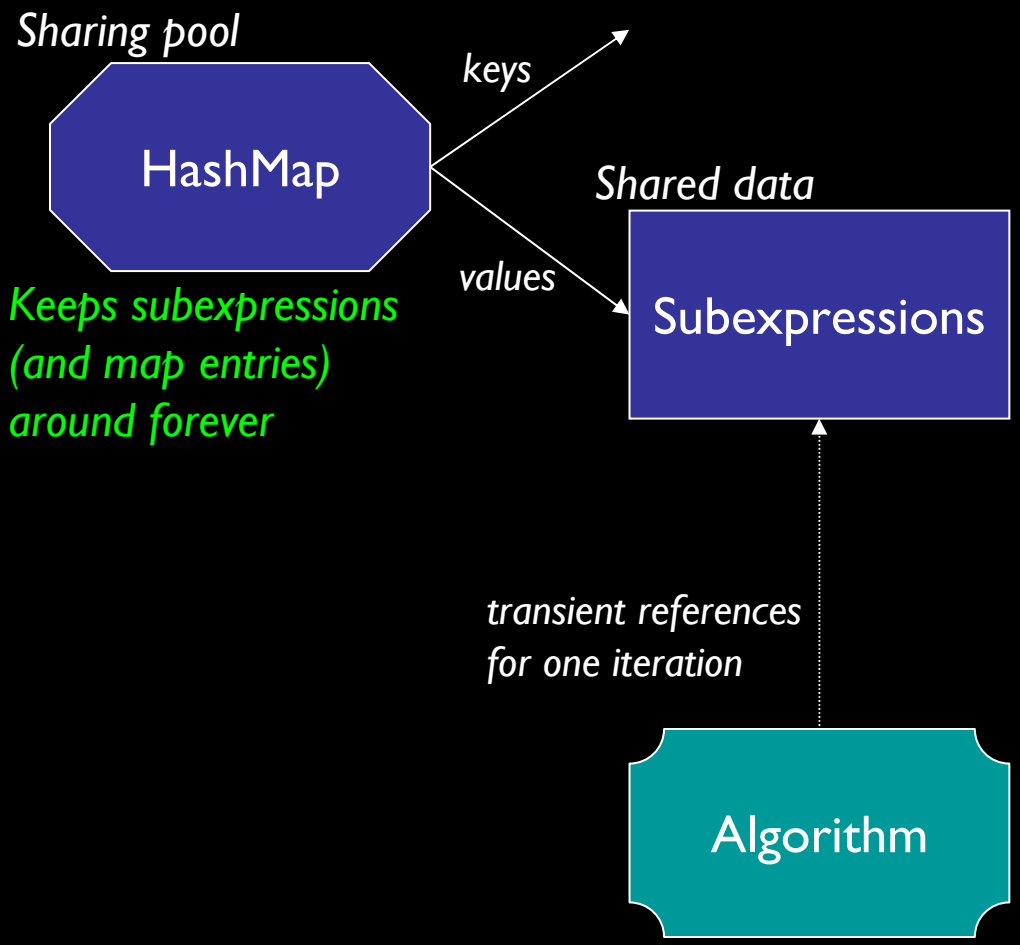
- e.g. annotations on existing objects
- e.g. sharing pools
- e.g. listeners

… or during specific phases or time intervals

- e.g. loading
- e.g. session state, for a bounded length of time

Sharing and growing

Case study: Planning system, sharing pool



- Each iteration of the algorithm creates hundreds of thousands of new expressions
- Used shared immutable factory pattern to save space on common subexpressions
- Result: unbounded growth due to pool

Background: Weak References

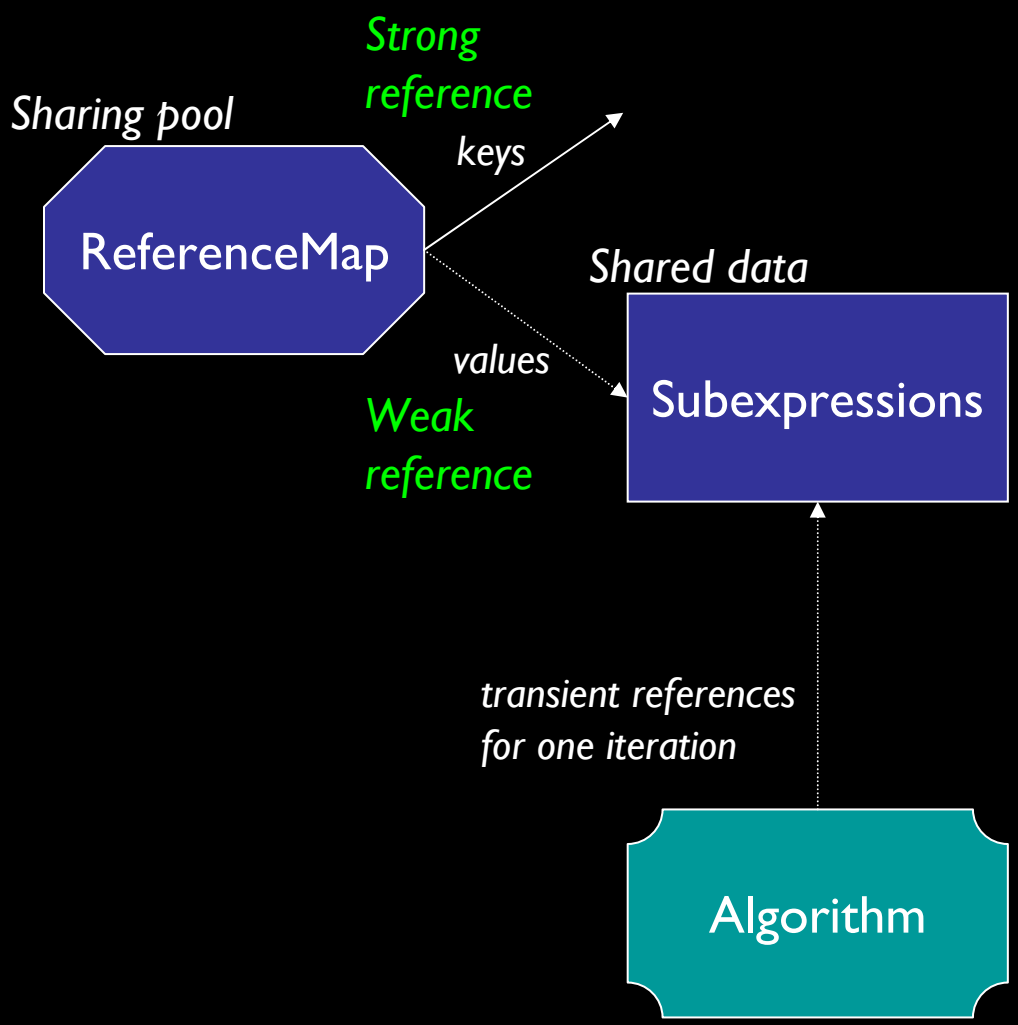
Weak Reference:

- Tells GC it may reclaim an object *as soon as it is no longer needed*
  - as long as there are no stronger references to the object

- Useful for preventing leaks – ties the lifetime of objects to other objects
  - e.g. for annotations, sharing pools, listener lists

Sharing and not growing

Case study: Planning system, sharing pool



Remedy:

- Apache Commons ReferenceMap (Strong, Weak)
- Pool entry will be removed when value is no longer needed

Note:

- Also considered soft references. But each iteration used different expressions, so no need to prolong lifetime. Goal was space, not time.

Using Weak References

A few common usage patterns

Weak key, strong value

- The standard Java WeakHashMap.
- Example usage: key = object to be annotated, value = annotation
- Caution if key is the same as or strongly reachable from value

Strong key, weak value

- As in previous example, for sharing pool

Background: weak and soft reference costs

- Weak and soft references are Objects, and so incur footprint costs
  - e.g. 24 bytes for each WeakReference on one 32-bit JVM
- Some weak/soft maps entries extend Weak/SoftReference; others add yet another level of delegation
  - e.g. Apache Commons ReferenceMap: at least 2 objects per entry

Leaks & drag: a sampling

Case study: CRM application

- Leak: bug in end-of-request processing failed to remove an object from a listener queue
- Immediate fix: fixed bug in request
- For robustness: have listener queue use weak references

- Failure to unregister listeners is a common cause of leaks

Case study: development tool

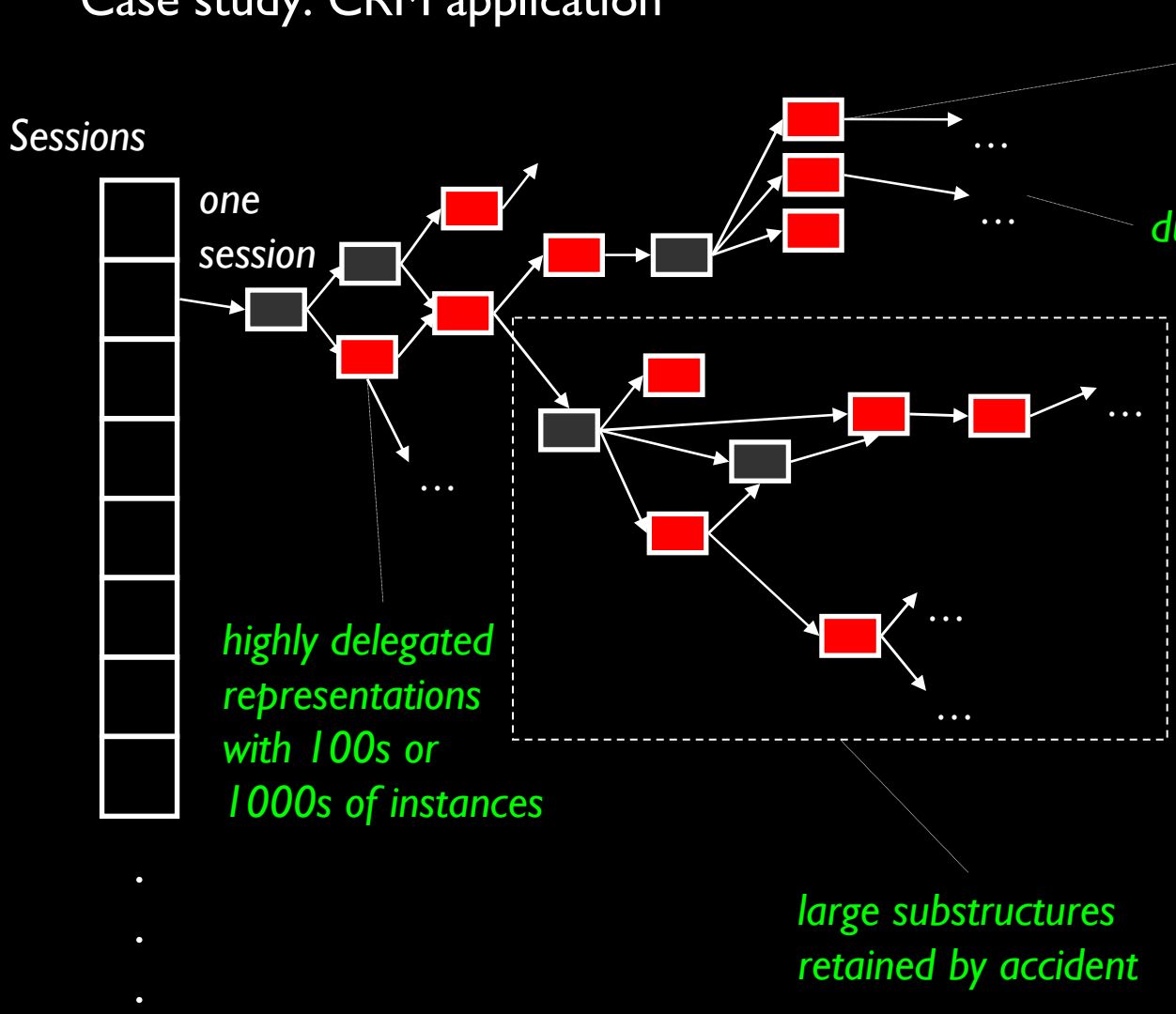
- Large index needed only during load time
- Easy solution: nulled out pointer

Case study: CRM application

- Session state retained for 8 hours
  - Made worse by costly session state (200K / user)
- Easy solution: fixed configuration

Entries too large

Case study: CRM application



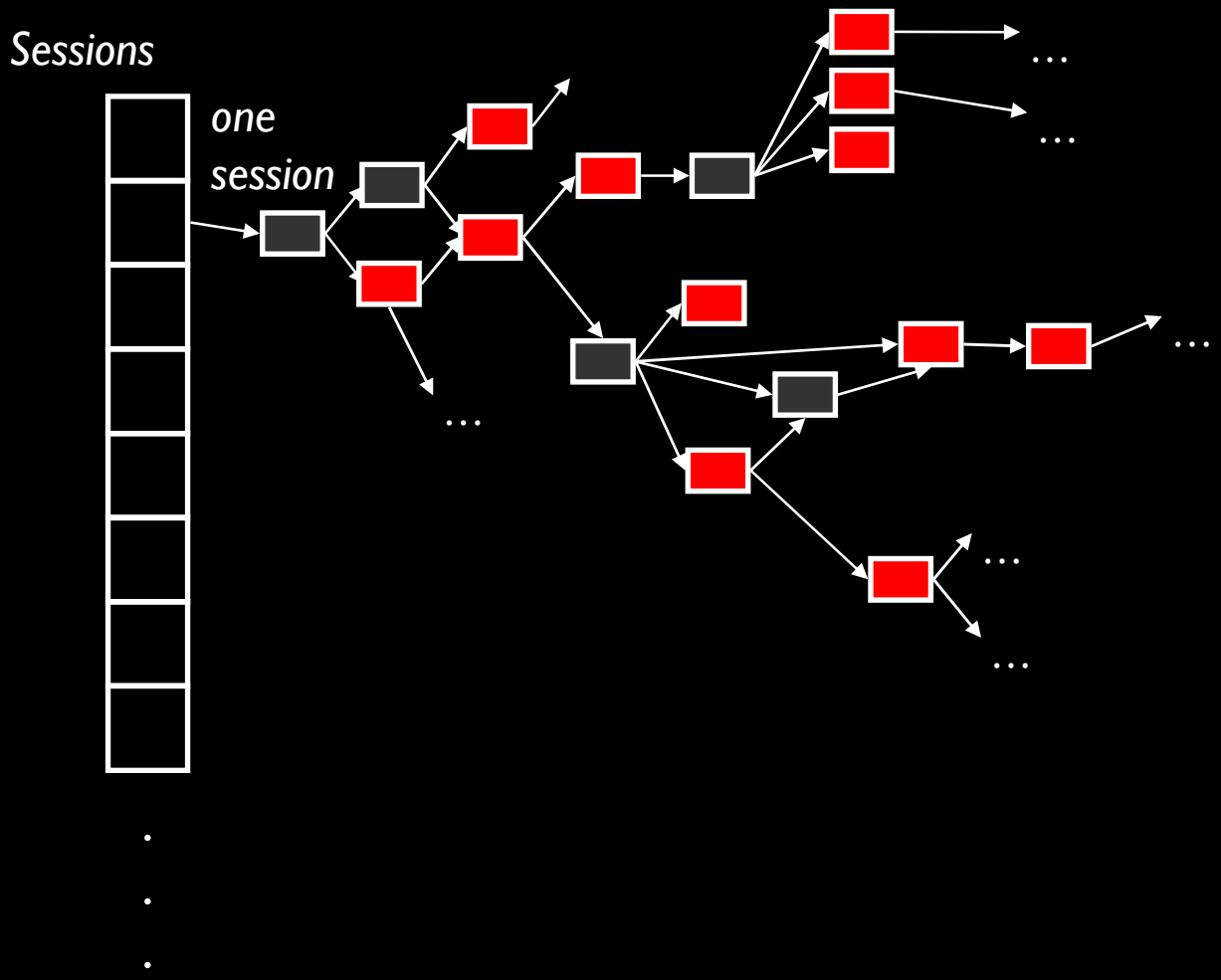
- 200K session state per user!
- Often a pile-up of multiple problems

Process

- simple techniques, tools, and resources

Surprises are everywhere

Case study: CRM application



- Developers expected 2K and found 200K session state per user
- Unit costs can be very difficult to predict at every level

Measurement

- Many surprises. It is essential to verify assumptions empirically throughout the lifecyle.

- Not so easy with framework layers and current tools

What and when

A few techniques among many:

- Small, synthetic experiments are extremely valuable, to test out frameworks and design patterns *before* they are adopted
- Of course, incorporate measurement into unit and system tests
  - Use detailed diagnostic tools to periodically check for scale, and look for surprises
  - Be mindful of normalization units when designing tests: how many concurrent users? active sessions?
  - Understand costs of major units used in lower layers
  - Run experiments at different scales early on. Are costs amortized as expected?
- Cardinality of relationships: state as part of design; verify periodically; then use in combination with measurement as the basis for estimation
- Caches and pools: verify that they are working and they are worth it

Managing long-lived data: challenges for the community

Tools that make costs visible *early*

- expected use of framework vs. granularity of use
- unit costs, e.g. per session, per user, per data structure
- predict/evaluate scalability

Mechanisms for balancing competing needs for memory

- current mechanisms are low level: weak and soft references
- subheaps? specifying lifetime intent?
- tools to validate and tune caches and pools

Mechanisms to enable larger designs with backing stores

- reduce transformation costs

Tools for heap analysis

- For analyzing the sources of memory bloat and for verifying assumptions, tools that rely on heap snapshots are the most valuable
- Some free tools from IBM and Sun
  - IBM DeveloperWorks & alphaWorks; Sun Developers Network
  - Tech preview (beta) of MDD4J in IBM Support Assistant – based on Yeti
- Commercial and open source tools
  - Eclipse MAT open source
  - YourKit, JProfiler, ….

Gathering heap data

- IBM and Sun diagnostic guides have information on gathering and analyzing heap snapshots, and pointers to free tools
  - IBM: http://www.ibm.com/developerworks/java/jdk/diagnosis/
  - Sun: http://java.sun.com/javase/, info with specific JDKs
- Formats
  - hprof: Sun & IBM
  - phd, javadump/DTFJ: IBM only
- The choice of when to take snapshots is key
  - For footprint: at steady state with known load
  - For footprint of a single feature or for suspected growth: before/after fixed number of operations, starting after system is warmed up

Additional resources

JDK library source code is freely available, and can be very worthwhile to consult

Many valuable articles on the web

- IBM DeveloperWorks, Sun Developer Network are good starting points
- Some misinformation occasionally found on reputable sites
- Best practices and tuning guides for specific frameworks

Garbage collection and overall heap usage

- IBM and Sun diagnosis sites have GC tuning guides, free tools
  - IBM Pattern Modeling and Analysis Tool for GC (PMAT) on alphaWorks, Health Center on developerWorks
- Some performance analysis tools have heap monitoring features

Object allocation

- Most Java performance profilers can show allocation information with calling context. e.g. hprof (free)

Conclusions

- Distributed development, layers of frameworks, and Java's modeling limitations make it easy to create bloated data designs.
- Awareness to costs can enable large gains without sacrificing speed or design. At the same time, there are limits to what developers can achieve.
- There are many research opportunities to make the Java language, runtime, and tools better address current programming practice.
- The concept of data structure health – the ratio of actual data to its representation – can illuminate where there is room for improvement, and highlight aspects of a design that will not scale.

Acknowledgments

Thanks to:

- Matthew Arnold
- Dave Grove
- Tim Klinger
- Trevor Parsons
- Peter Santhanam
- Edith Schonberg
- Yeti

See also:

- N. Mitchell, G. Sevitsky, “The Causes of Bloat, the Limits of Health”, OOPSLA 2007, Montreal, Canada.
- N. Mitchell, E. Schonberg, G. Sevitsky, “Making Sense of Large Heaps”, ECOOP 2009, Genoa, Italy.