Building a Big Data Machine Learning Platform

Cliff Click, CTO 0xdata
cliffc@0xdata.com
http://0xdata.com
http://cliffc.org/blog

H2O is...

- Pure Java, Open Source: 0xdata.com
  - https://github.com/0xdata/h2o/
- A Platform for doing Parallel Distributed Math
  - In-memory analytics: GLM, GBM, RF, Logistic Reg, Deep Learning, PCA, Kmeans...
  - Data munging & cleaning
- Accessible via REST & JSON, browser, Python, R, Java, Scala
  - And now *Spark*

Platform for doing Big Data Work

- "Anything" you want to do on Big 2-D Tables
  - Most any Java that reads or writes a single row
    - Plus read nearby rows, and/or computes a reduction
- Speed: data volume / memory bandwidth
  - ~50G/sec / node, varies by hardware
- Data compressed: 2x to 4x better than gzip
- Data limited to: numbers & time & strings
- Table width: <1K fast, <10K works, <100K slower
- Table length: Limit of memory

What Can I Do With It?

Simple Data-Parallel Coding

- Map/Reduce Per-Row: **Stateless**
  - Example from Linear Regression, $\Sigma\ y^2$

```
double sumY2 = new MRTask() {
  double map( double d ) { return d*d; }
  double reduce( double d1, double d2 ) {
    return d1+d2;
  }
}.doAll( vecY );
```

- Auto-parallel, auto-distributed
- Fortran speed, Java Ease

Simple Data-Parallel Coding

- Scala version in development:

```
MR {
  def map(A:Double) = A*A
  def reduce(B1, B2: Double) = B1+B2
}.doAll( vecY );
```

Simple Data-Parallel Coding

- Map/Reduce Per-Row: **Statefull**
  - Linear Regression Pass1: Σ x, Σ y, Σ y^2

```
class LRPass1 extends MRTask {
  double sumX, sumY, sumY2; // I Can Haz State?
  void map( double X, double Y ) {
    sumX += X;  sumY += Y;  sumY2 += Y*Y;
  }
  void reduce( LRPass1 that ) {
    sumX  += that.sumX ;
    sumY  += that.sumY ;
    sumY2 += that.sumY2;
  }
}
```

Simple Data-Parallel Coding

- Scala version in development:

```
MR { var X, Y, X2=0.0; var n=0L
  def map(x,y:Double) = X=x; Y=y; X2=x*x; n=1
  def reduce(@@: self) =
    { X+=@@.X; Y+=@@.Y; X2+=@@.X2; n+=@@.n }
}.doAll(vecX,vecY)
```

Simple Data-Parallel Coding

- Map/Reduce Per-Row: **Batch Statefull**

```
class LRPass1 extends MRTask {
  double sumX, sumY, sumY2;
  void map( Chunk CX, Chunk CY ) {// Whole Chunks
    for( int i=0; i<CX.len; i++ ){// Batch!
      double X = CX.at(i), Y = CY.at(i);
      sumX += X;  sumY += Y;  sumY2 += Y*Y;
    }
  }
  void reduce( LRPass1 that ) {
    sumX  += that.sumX ;
    sumY  += that.sumY ;
    sumY2 += that.sumY2;
  }
}
```

Other Simple Examples

- Filter & Count (underage males):
  - (can pass in any number of Vecs or a Frame)

```
long sumY2 = new MRTask() {
  long map( long age, long sex ) {
    return (age<=17 && sex==MALE) ? 1 : 0;
  }
  long reduce( long d1, long d2 ) {
    return d1+d2;
  }
}.doAll( vecAge, vecSex );
```

- Scala syntax

```
MR(0).map(_('age)<=17 && _('sex)==MALE )
     .reduce(add).doAll( frame );
```

Other Simple Examples

- Filter into new set (underage males):
  - Can write or append subset of rows
    - (append order is preserved)

```
class Filter extends MRTask {
  void map(Chunk CRisk, Chunk CAge, Chunk CSex){
    for( int i=0; i<CAge.len; i++ )
      if( CAge.at(i)<=17 && CSex.at(i)==MALE )
        CRisk.append(CAge.at(i)); // build a set
  }
};
Vec risk = new AppendableVec();
new Filter().doAll( risk, vecAge, vecSex );
...risk... // all the underage males
```

Other Simple Examples

- Filter into new set (underage males):
  - Can write or append subset of rows
    - (append order is preserved)

```
class Filter extends MRTask {
  void map(Chunk CRisk, Chunk CAge, Chunk CSex){
    for( int i=0; i<CAge.len; i++ )
      if( CAge.at(i)<=17 && CSex.at(i)==MALE )
        CRisk.append(CAge.at(i)); // build a set
  }
};
Vec risk = new AppendableVec();
new Filter().doAll( risk, vecAge, vecSex );
...risk... // all the underage males
```

0xdata

Other Simple Examples

- Group-by: count of car-types by age

```
class AgeHisto extends MRTask {
  long carAges[][]; // count of cars by age
  void map( Chunk CAge, Chunk CCar ) {
    carAges = new long[numAges][numCars];
    for( int i=0; i<CAge.len; i++ )
      carAges[CAge.at(i)][CCar.at(i)]++;
  }
  void reduce( AgeHisto that ) {
    for( int i=0; i<carAges.length; i++ )
      for( int j=0; i<carAges[j].length; j++ )
        carAges[i][j] += that.carAges[i][j];
  }
}
```

Other Simple Examples

- Group-by: count of car ty...

Setting carAges in **map** makes it an **output** field.
Private per-map call, single-threaded write access.
Must be rolled-up in the **reduce** call.

```
class AgeHisto
  long carAges[][]
  void map( Chunk CAge, Chunk CCar ) {
    carAges = new long[numAges][numCars];
    for( int i=0; i<CAge.len; i++ )
      carAges[CAge.at(i)][CCar.at(i)]++;
  }
  void reduce( AgeHisto that ) {
    for( int i=0; i<carAges.length; i++ )
      for( int j=0; i<carAges[j].length; j++ )
        carAges[i][j] += that.carAges[i][j];
  }
}
```

Other Simple Examples

- Uniques
  - Uses distributed hash set

```
class Uniques extends MRTask {
  DNonBlockingHashSet<Long> dnbhs = new ...;
  void map( long id ) { dnbhs.add(id); }
  void reduce( Uniques that ) {
    dnbhs.putAll(that.dnbhs);
  }
};
long uniques = new Uniques().
  doAll( vecVistors ).dnbhs.size();
```

Other Simple Examples

- Uniques
  - Uses distributed h

Setting dnbhs in <init> makes it an **input** field. Shared across all maps(). Often read-only. This one is written, so needs a **reduce**.

```
class Uniques extends MRTask {
  DNonBlockingHashSet<Long> dnbhs = new ...;
  void map( long id ) { dnbhs.add(id); }
  void reduce( Uniques that ) {
    dnbhs.putAll(that.dnbhs);
  }
};
long uniques = new Uniques().
  doAll( vecVistors ).dnbhs.size();
```

How Does It Work?

A Collection of Distributed Vectors

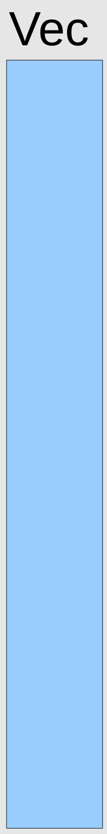
```
// A Distributed Vector
//   much more than 2billion elements
class Vec {
  long length(); // more than an int's worth

  // fast random access
  double at(long idx); // Get the idx'th elem
  boolean isNA(long idx);

  void set(long idx, double d); // writable
  void append(double d); // variable sized
}
```

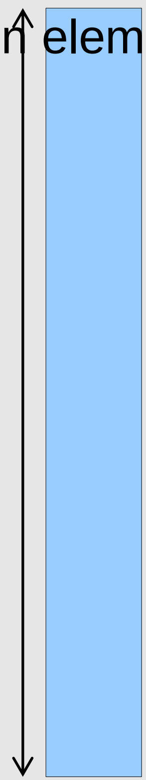
Distributed Data Taxonomy

A Single Vector



Distributed Data Taxonomy

A Very Large Single Vec



- Java primitive
- Usually **`double`**

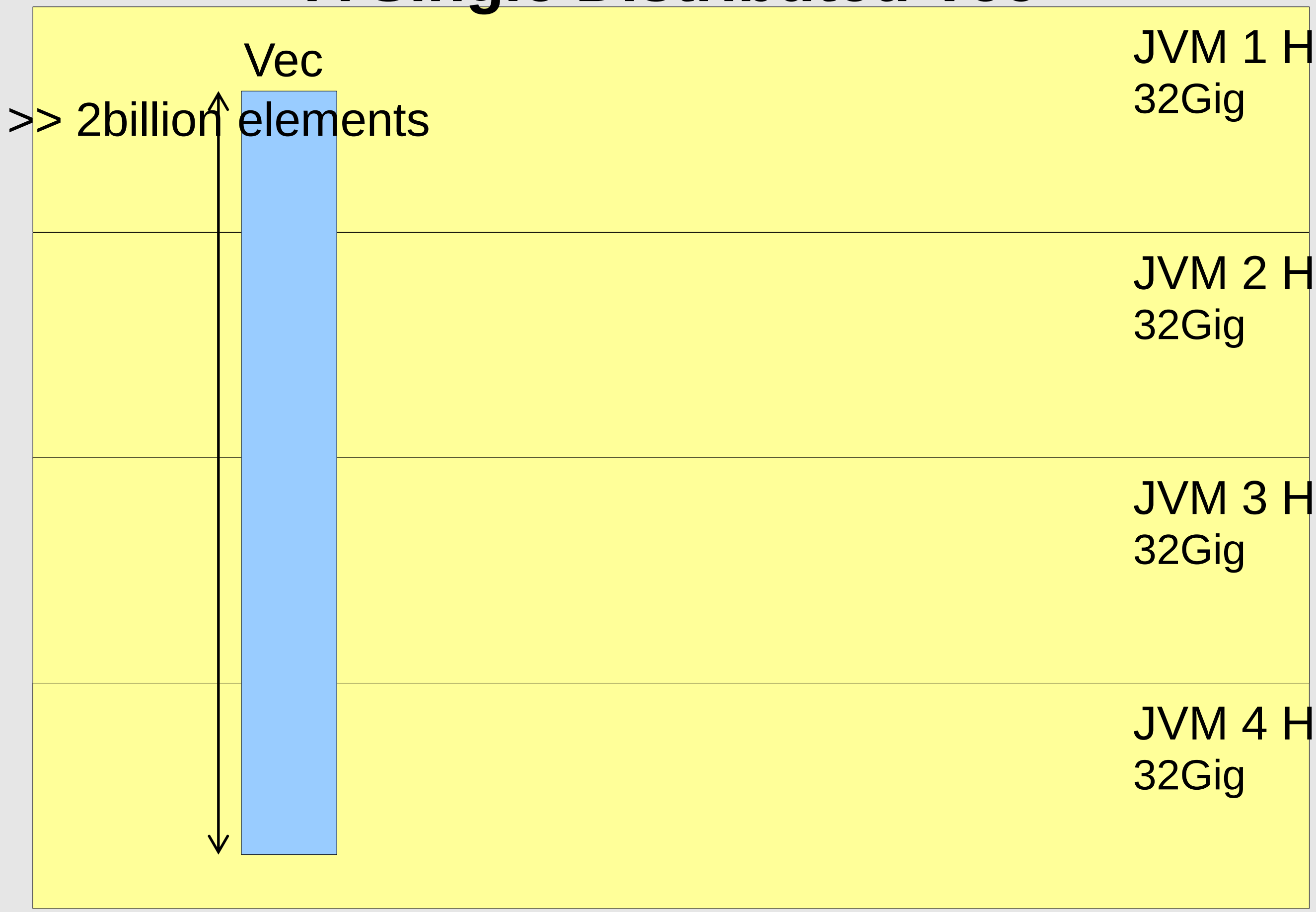
- Length is a **`long`**
- >> 2^31 elements

- Compressed
- Often 2x to 4x
- Random access

- Linear access is
  FORTRAN speed

Distributed Data Taxonomy

A Single Distributed Vec



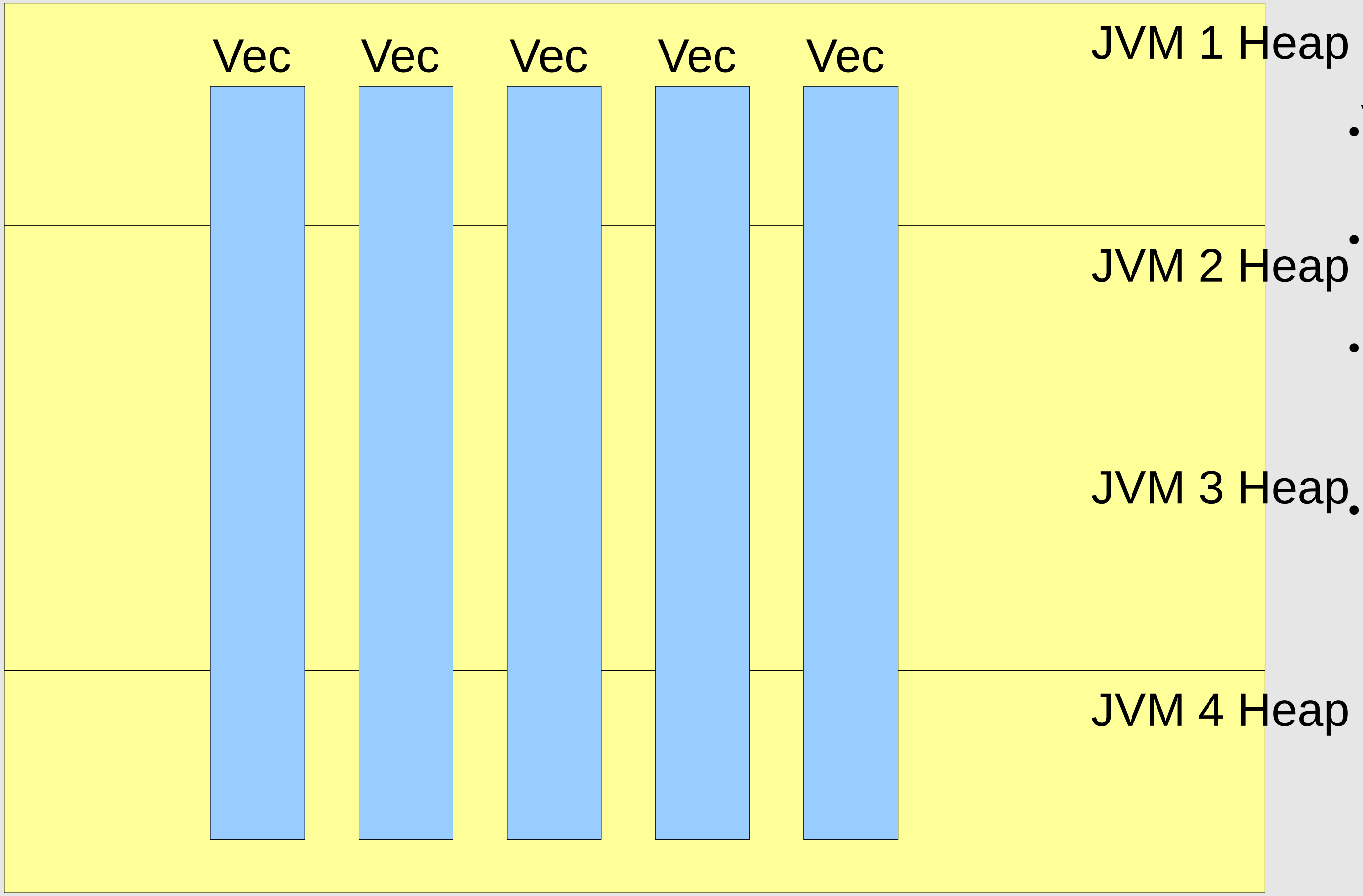
- Java Heap
- Data In-Heap
- Not off heap
- Split Across Heaps

- GC management
- Watch FullGC
- Spill-to-disk
- GC very cheap
- Default GC

- To-the-metal speed
- Java ease

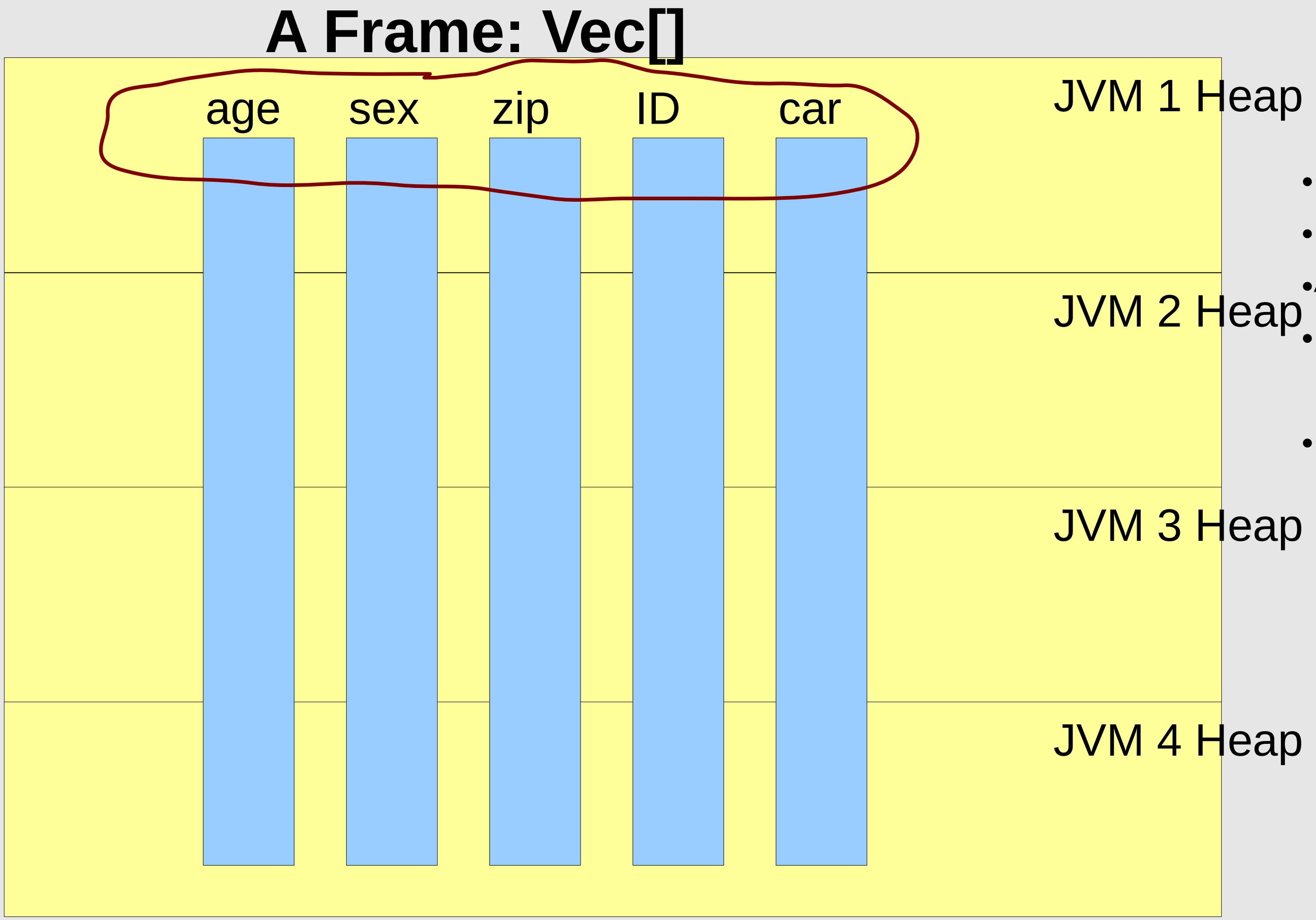
Distributed Data Taxonomy

A Collection of Distributed Vecs



- Vecs aligned in heaps
- Optimized for concurrent access
- Random access any row, any JVM
- But faster if local... more on that later

Distributed Data Taxonomy

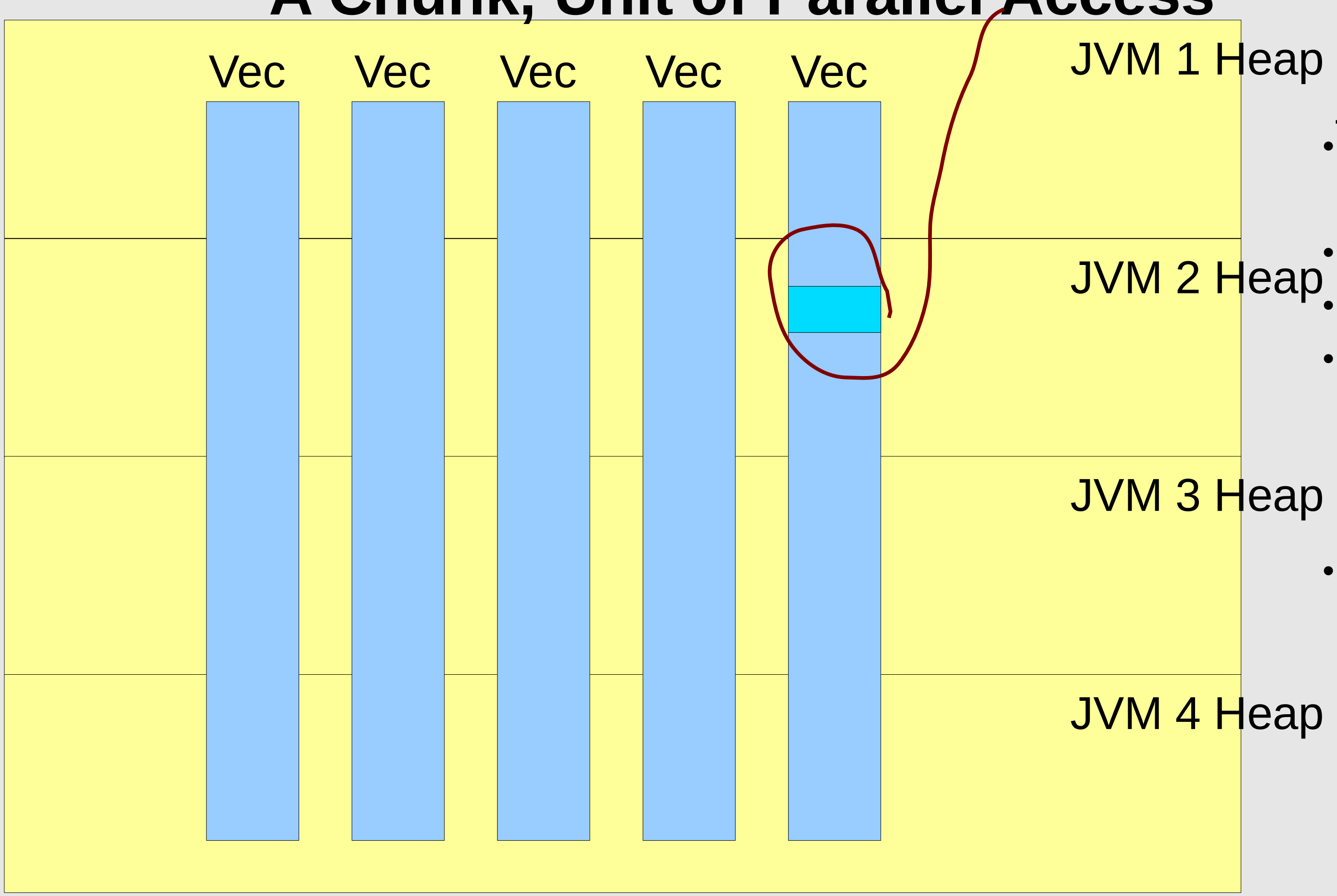


- Similar to R frame
- Change Vecs freely
- Add, remove Vecs
- Describes a row of user data
- Struct-of-Arrays (vs ary-of-structs)

0xdata

Distributed Data Taxonomy

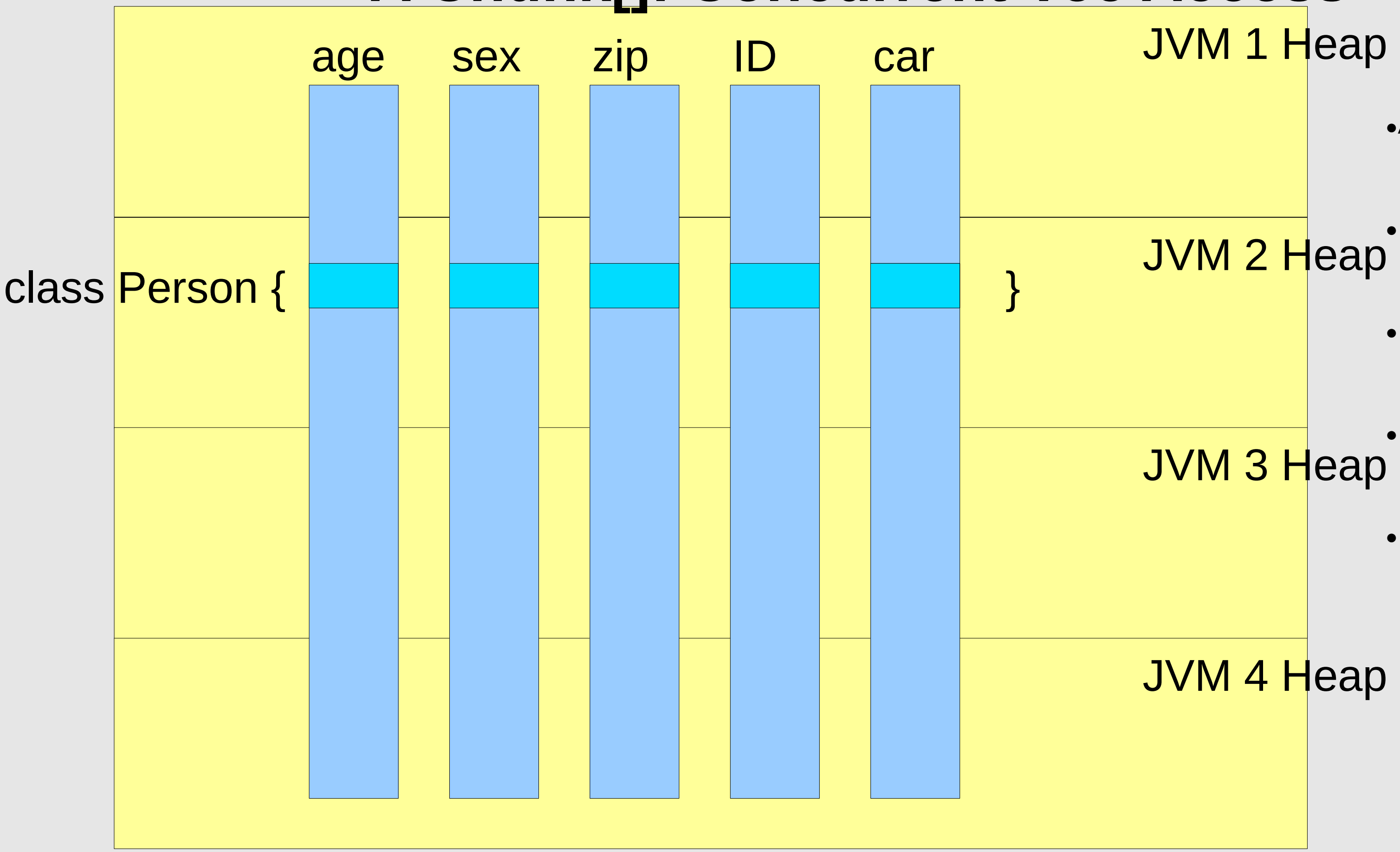
A Chunk, Unit of Parallel Access



- Typically 1e3 to 1e6 elements
- Stored compressed
- In byte arrays
- Get/put is a few clock cycles **including** compression
- Compression is Good: more data per cache-miss

Distributed Data Taxonomy

A Chunk[]: Concurrent Vec Access

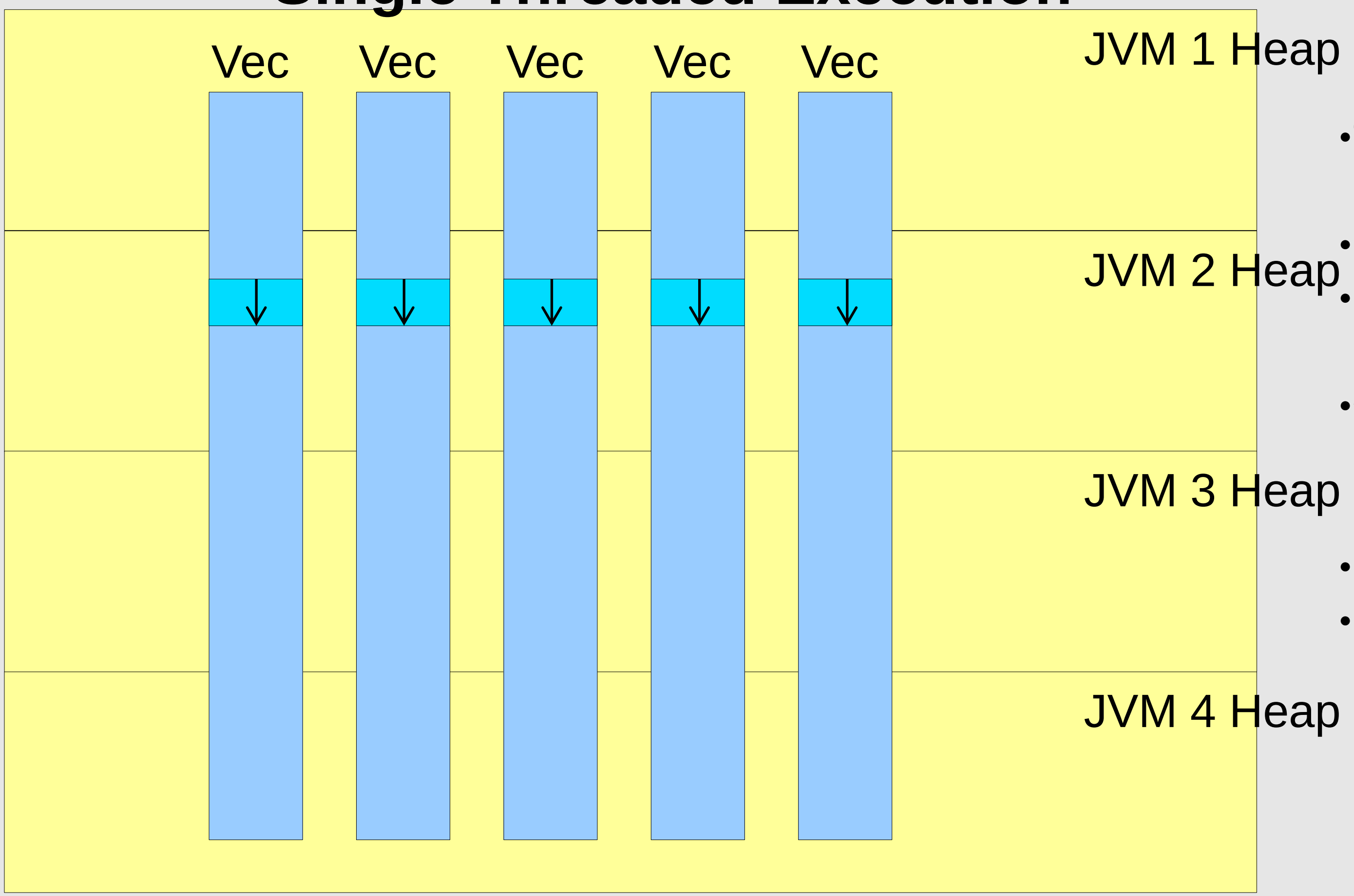


- Access Row in a single thread
- Like a Java object

- Can read & write: Mutable Vectors
- Both are full Java speed
- Conflicting writes: use JMM rules

Distributed Data Taxonomy

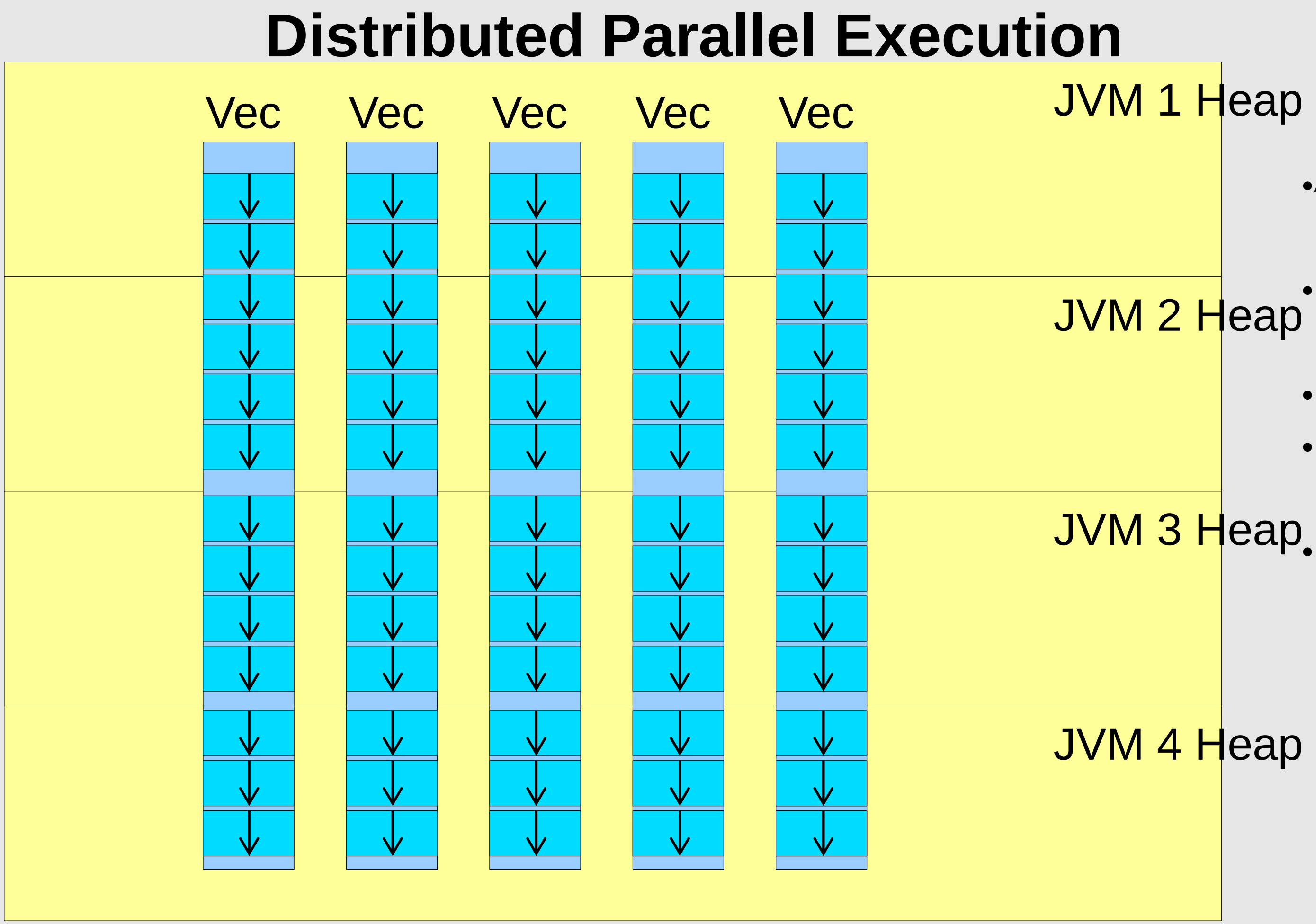
Single Threaded Execution



- One CPU works a Chunk of rows
- Fork/Join work unit
- Big enough to cover control overheads
- Small enough to get fine-grained par

- Map/Reduce
- Code written in a simple single-threaded style

Distributed Data Taxonomy



- All CPUs grab Chunks in parallel
- F/J load balances

- Code moves to Data
- Map/Reduce & F/J handles all sync
- H2O handles all comm, data manage

0xdata

Distributed Data Taxonomy

```
Frame – a collection of Vecs
  Vec – a collection of Chunks
    Chunk – a collection of 1e3 to 1e6 elems
      elem – a java double

Row i – i'th elements of all the Vecs in a Frame
```

Sparkling Water

- Bleeding edge*: Spark* & H2ORDDs
  - Move data back & forth, model & munge
  - Same process, same JVM
- H2O Data as a:
  - Spark RDD or
  - Scala Collection

```
Frame.toRDD.runJob(...)

Frame.foreach{...}
```

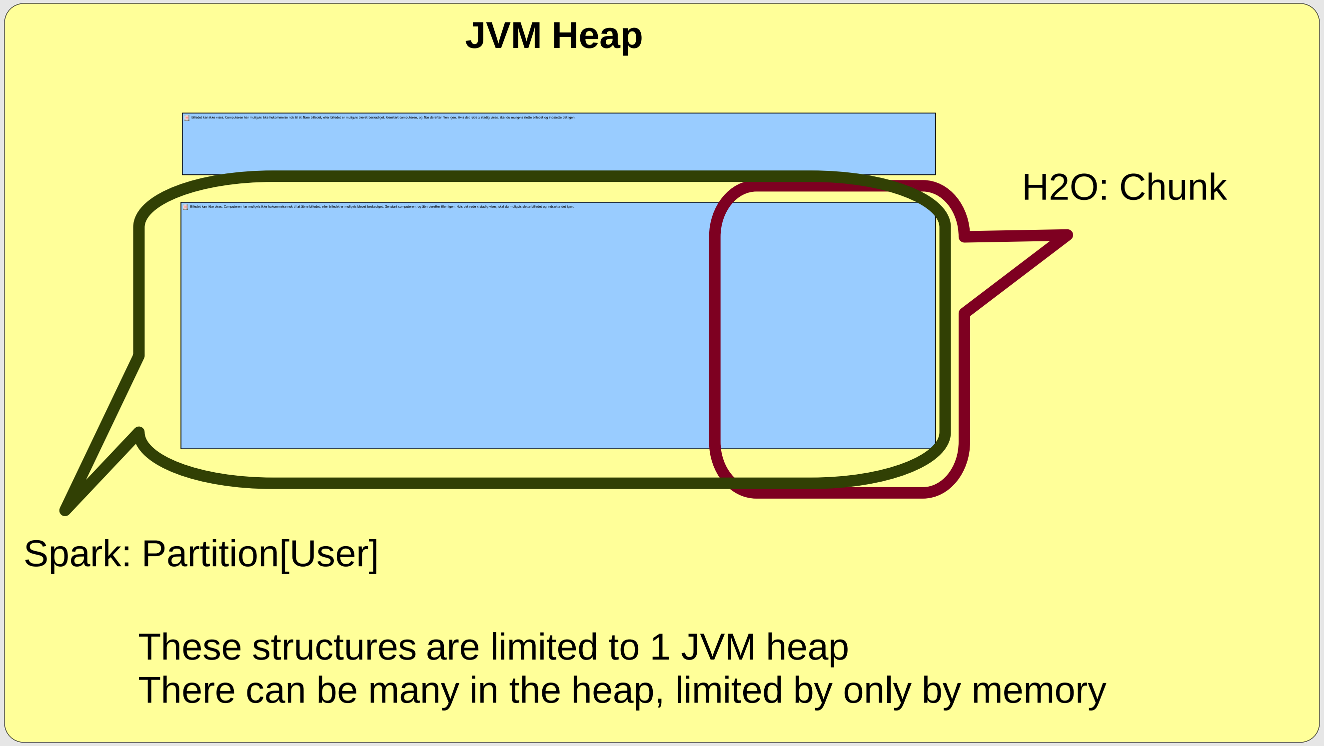
- Code in:
  - https://github.com/0xdata/h2o-dev
  - https://github.com/0xdata/perrier

Sparkling Water: Spark and H2O

- Convert RDDs <==> Frames
  - In memory, simple fast call
  - In process, no external tooling needed
  - Distributed – data does not move*
  - Eager, not Lazy
- Makes a data copy!
  - H2O data is **highly** compressed
  - Often 1/4 to 1/10th original size

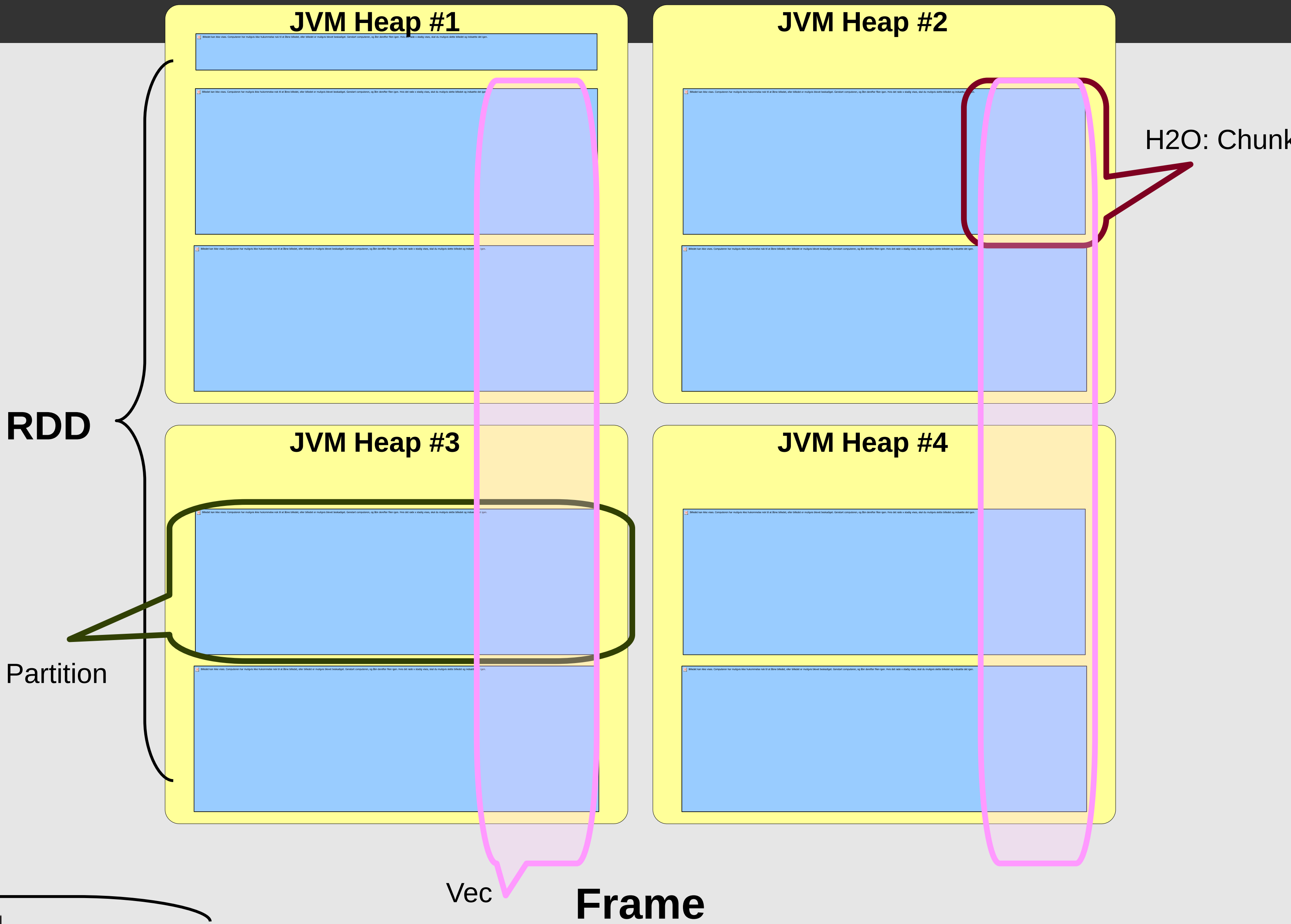
*See fine print

Spark Partitions and H2O Chunks



*Only data correspondance is shown; a real data copy is required!

Spark RDDs and H2O Frames



Sparkling Water

- Convert to H2O Frame
  - Eager, executes RDDs immediately
  - Makes a compressed H2O copy

```
val fr = toDataFrame(sparkCx,rdd)
```

- Convert to Spark RDD
  - Lazy, defines a normal RDD
  - When executed, acts as a checkpoint

```
val rdd = toRDD(sparkCx,fr)
```

Distributed Coding Taxonomy

- No Distribution Coding:
  - Whole Algorithms, Whole Vector-Math
  - REST + JSON: e.g. load data, GLM, get results
  - R, Python, Web, bash/curl
- Simple Data-Parallel Coding:
  - Map/Reduce-style: e.g. Any dense linear algebra
  - Java/Scala foreach* style
- Complex Data-Parallel Coding
  - K/V Store, Graph Algo's, e.g. PageRank

Summary: Write (parallel) Java

- Most simple Java “just works”
  - Scala API is experimental, but will also "just work"
- **Fast**: parallel distributed reads, writes, appends
  - Reads same speed as plain Java array loads
  - Writes, appends: slightly slower (compression)
  - Typically memory bandwidth limited
    - (may be CPU limited in a few cases)
- **Slower**: conflicting writes (but follows strict JMM)
  - Also supports transactional updates

Summary: Writing Analytics

- We're writing Big Data Distributed Analytics
  - Deep Learning
  - Generalized Linear Modeling (ADMM, GLMNET)
    - Logistic Regression, Poisson, Gamma
  - Random Forest, GBM, KMeans, PCA, ...
- Solidly working on 100G datasets
  - Testing Tera Scale Now
- Paying customers (in production!)
- Come write your own (distributed) algorithm!!!

Q & A

0xdata.com

https://github.com/0xdata/h2o

https://github.com/0xdata/h2o-dev
https://github.com/0xdata/perrier

Cool Systems Stuff...

- … that I ran out of space for
- Reliable UDP, integrated w/RPC
- TCP is reliably UNReliable
  - Already have a reliable UDP framework, so no prob
- Fork/Join Goodies:
  - Priority Queues
  - Distributed F/J
  - Surviving fork bombs & lost threads
- K/V does JMM via hardware-like MESI protocol

Speed Concerns

- How fast is fast?
- Data is Big (by definition!) & must see it all
- Typically: less math than memory bandwidth
  - So decompression happens while waiting for mem
  - More (de)compression is better
  - Currently 15 compression schemes
  - Picked per-chunk, so can (does) vary across dataset
  - All decompression schemes take 2-10 cycles max
- Time leftover for plenty of math

Speed Concerns

- Serialization:
  - Rarely send Big Data around (too much of that! Must be normally node-local access)
  - Instead it's POJO's doing the math (Histograms, Gram Matrix, sums & variances, etc)
- Bytecode weaver on class load
  - Write fields via Unsafe into DirectByteBuffers
  - 2-byte unique token defines type (and nested types)
  - Compression on that too! (more CPU than network)

Serialization

- Write fields via Unsafe into DirectByteBuffers
  - All from simple JIT'd code -
    - Just the loads & stores, nothing else
  - 2-byte token once per top-level RPC
    - (more tokens if subclassed objects used)
- Streaming async NIO
  - Multiple shared TCP & UDP channels
    - Small stuff via UDP & big via TCP
  - Full app-level retry & error recovery
    - (can pull cable & re-insert & all will recover)

Map / Reduce

- Map: Once-per-chunk; typically 1000's per-node
  - Using Fork/Join for fine-grained parallelism
- Reduce: reduce-early-often – after every 2 maps
  - Deterministic, same Maps, same rows every time
  - Until all the Maps & Reduces on a Node are done
- Then ship results over-the-wire
  - And Reduce globally in a log-tree rollup
- Network latency is 2 log-tree traversals

Fork/Join Experience

- **Really Good** (except when it's not)
- Good Stuff: easy to write...
  - (after a steep learning curve)
- Works! Fine to have many many small jobs, load balances across CPUs, keeps 'em all busy, etc.
- Full-featured, flexible
  - We've got 100's of uses of it scattered throughout

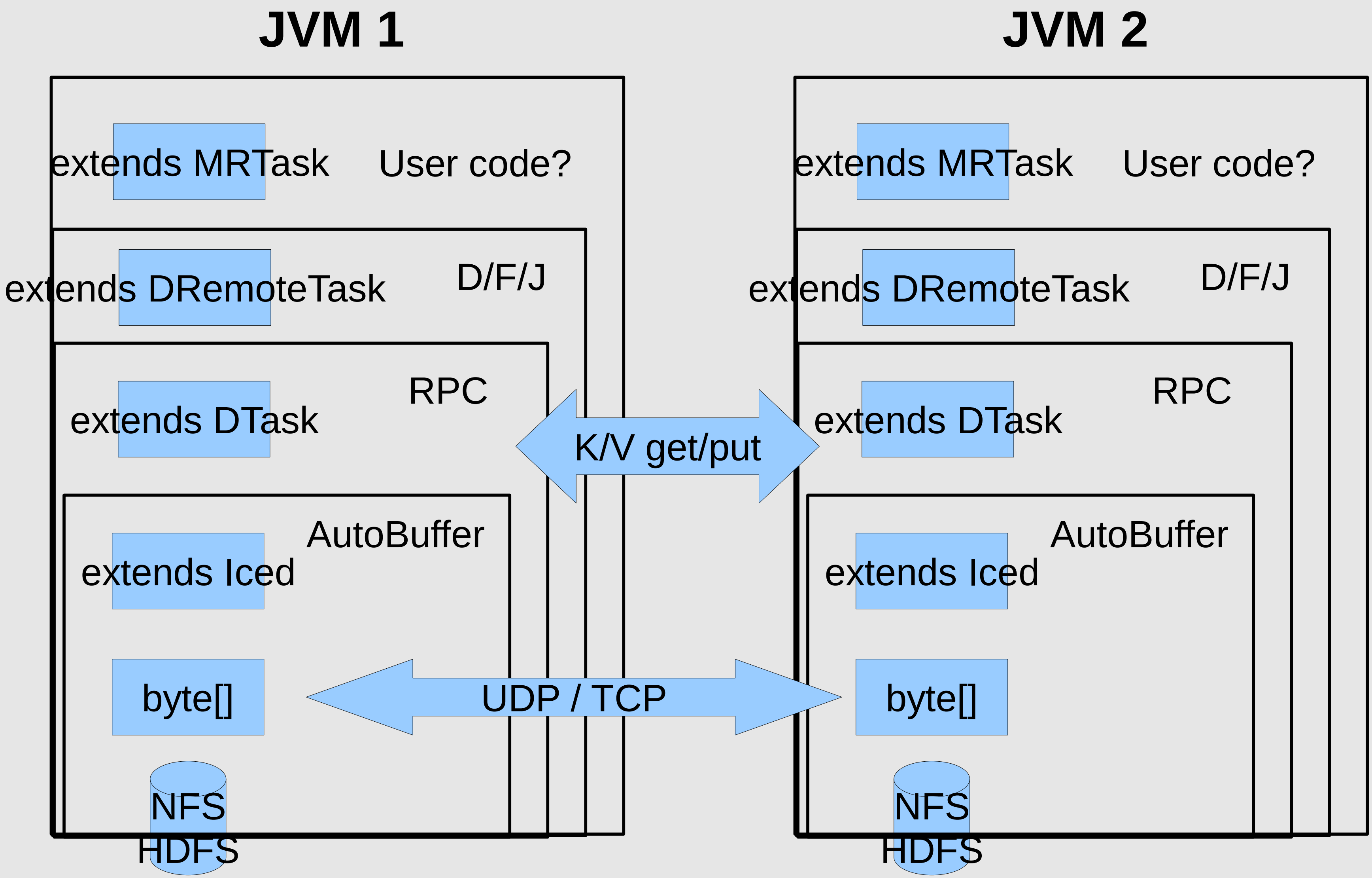
Fork / Join Experience

- Really Good (**except when it's not**)
  - Blocking threads is hard on F/J
    - (ManagedBlocker.block API is painful)
    - Still get thread starvation sometimes
  - "CountedCompleters" – CPS by any other name
    - Painful to write explicit-CPS in Java
- No priority queues – a Must Have
  - And no Java thread priorities
  - So built up priority queues over F/J & JVM

Fork/Join Experience

- Default exception is silently dropped
  - Usual symptom: all threads idle, but job not done
  - Complete maintenance disaster – must catch & track & log all exceptions
    - (and even pass around cluster distributed)
- Forgotten “tryComplete()” not too hard to track
- Fork-Bomb – must cap all thread pools
  - Which can lead to deadlock
  - Which leads to using CPS-style occasionally
- *Despite issues, I'd use it again*

The Platform



0xdata

TCP Fails

- In <5mins, I can force a TCP fail on Linux
- "Fail": means Server opens+writes+closes
  - NO ERRORS
  - Client gets no data, no errors
  - In my lab (no virtualization) or EC2
- Basically, H2O can mimic a DDOS attack
  - And Linux will "cheat" on the TCP protocol
  - And cancel valid, in-progress, TCP handshakes
  - Verified w/wireshark

TCP Fails

- Any formal verification? (yes lots)
- Of **recent** Linux kernals?
  - Ones with DDOS-defense built-in?