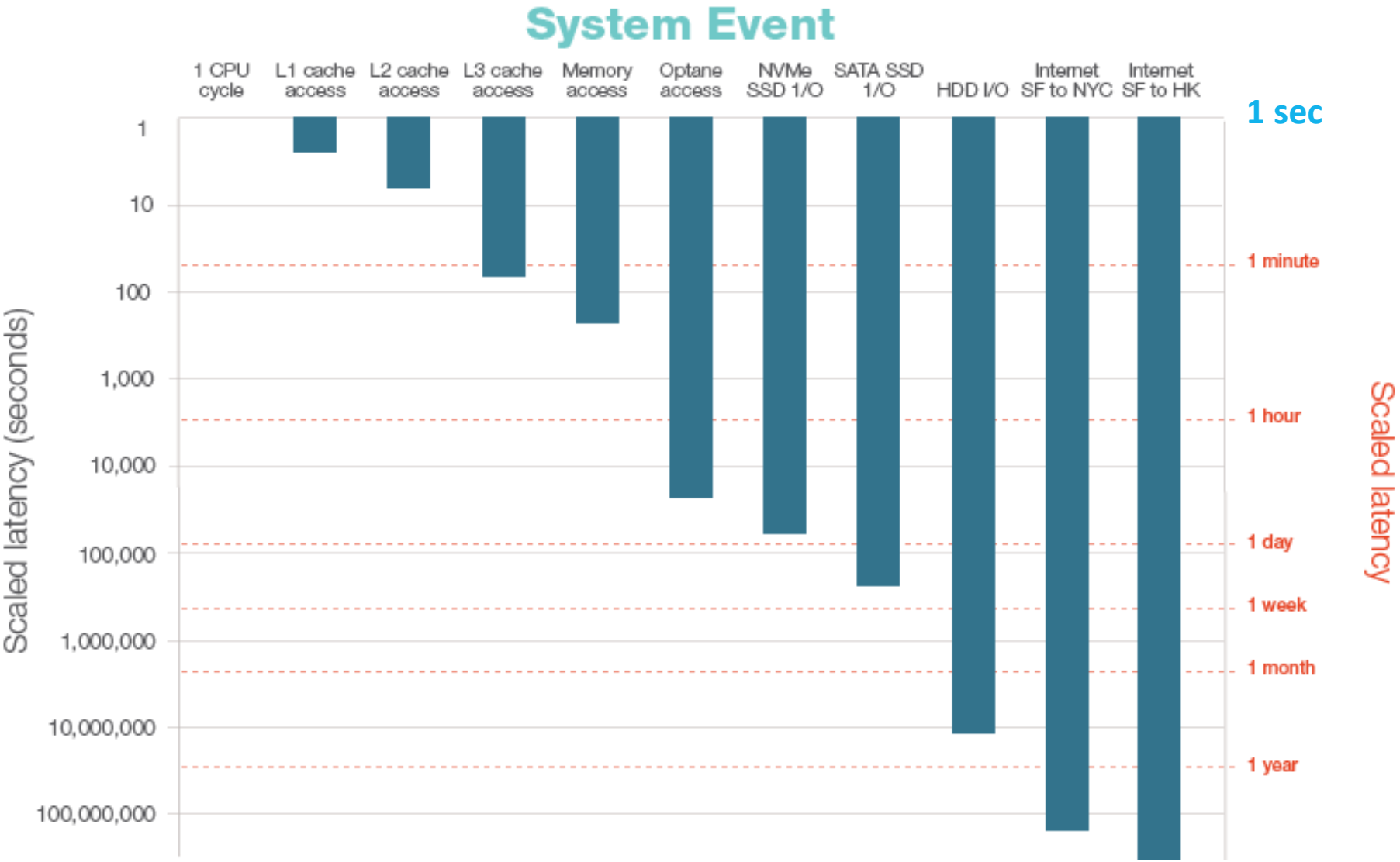
L3: Spark & RDD

Latency & Bandwidth

- L1 cache reference 0.5 ns
- L2 cache reference 7 ns
- Main memory reference
- Send 1K bytes over 1 Gbps network
- Read 4K randomly from SSD*
- Round trip within same datacenter
- Disk seek
- Send packet CA->NL->CA

https://gist.github.com/jboner/2841832

Latency in Human Scales



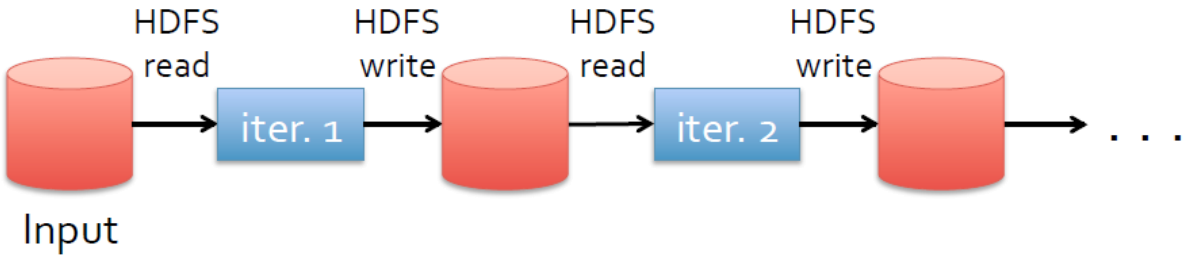
http://www.prowesscorp.com/computer-latency-at-a-human-scale/

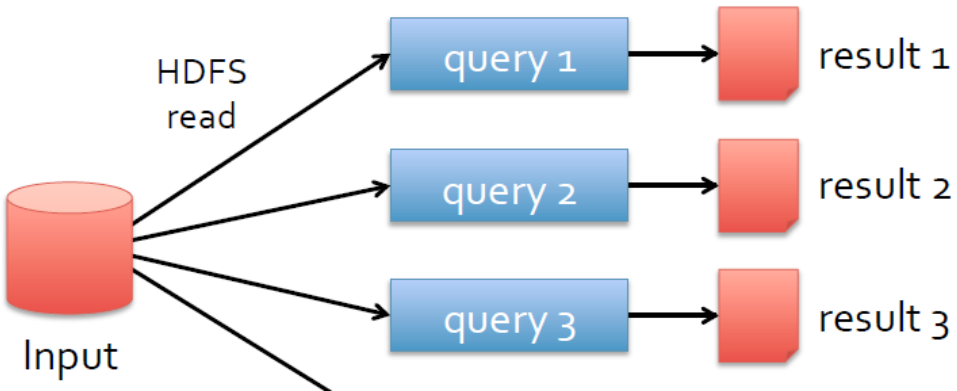
Map Reduce

- Data-parallel execution
- Move program to data, rather than data to program
- Cost for moving data: network, disk, memory, cache
- But moves data from disk to memory often

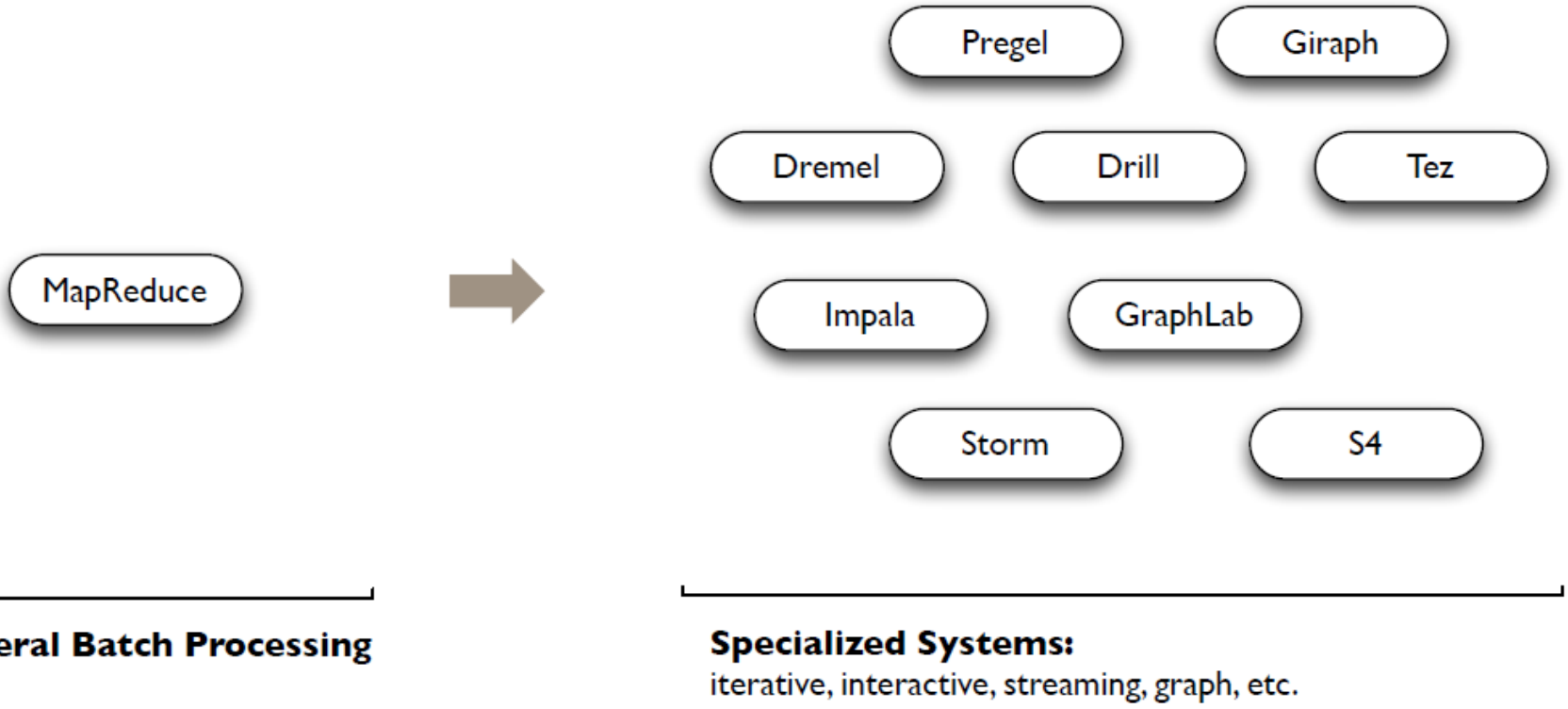
Downsides of MR

- NOT Designed for data sharing
  - the only way to share data across jobs is stable storage ➔ slow!
- NOT suited for
  - Complex, multi--stage applications (e.g. iterative machine learning & graph processing)
  - Interactive *ad hoc* queries

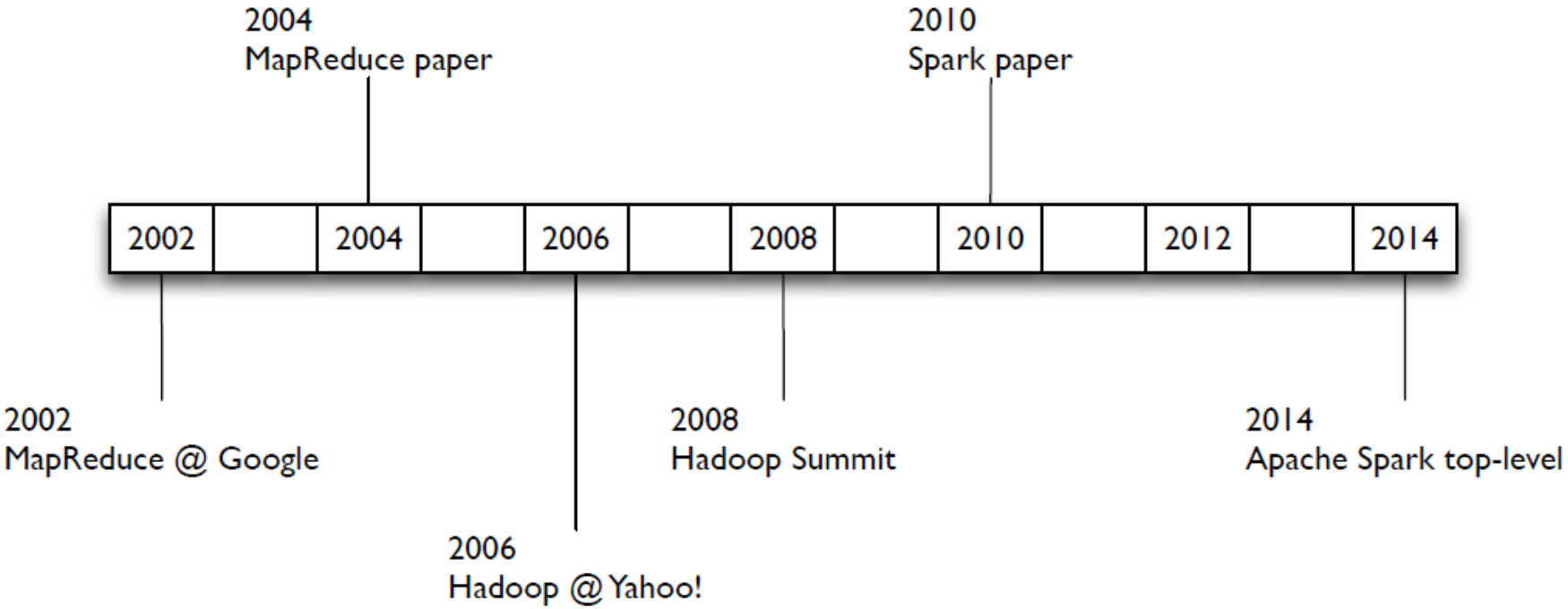




A Brief History: *MapReduce*



https://stanford.edu/~rezab/sparkclass/slides/itas_workshop.pdf



https://stanford.edu/~rezab/sparkclass/slides/itas_workshop.pdf

Spark: Cluster Computing with Working Sets
Zaharia, Chowdhury, Franklin, Shenker, Stoica
USENIX HotCloud 2010

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing
Zaharia, Chowdhury, Das, Dave, Ma, McCauley, Franklin, Shenker, Stoica
USENIX NSDI 2012

Challenge

- How to design a distributed memory abstraction that is both fault-tolerant and efficient?

DAG Model

- More expressive
  - Iterative Map Reduce
  - Dryad/DryadLINQ
  - Pig, Hive

Working Sets

- Reuse same data multiple times
  - Pregel/Giraph
  - Parameter sweep in ML
- Use resulting data immediately as a pipeline
  - Iterative Map Reduce
- Analogous to virtual memory

RDD

- Distributed
- Read-only
- Can be rebuilt
- Can be cached
- MR like data-parallel operations

- Sweet-spot: Expressivity, Scalability, Reliability
- *Efficient* fault tolerance
  - Avoid replication, transaction logging
  - Retain in-memory for performance
  - Coarse-grained logging of *lineage*

RDD

- Handle to the RDD
  - Virtual collection of partitions
  - Recipe to rebuild those partitions from disk
- Immutable
- Construct
  - From disk
  - From memory collection (constant)
  - By transformation
  - By loading from persistent RDD

Cache

- Hint of retaining in memory. Trade-off:
- cost of in-memory storage,
- speed of access,
- probability of loss,
- cost of recompute

Parallel Ops

- Similar to DryadLINQ

- Foreach
- Reduce, associative function
- Collect
- No GroupBy yet

- Lazy evaluation, pipelining of transformations

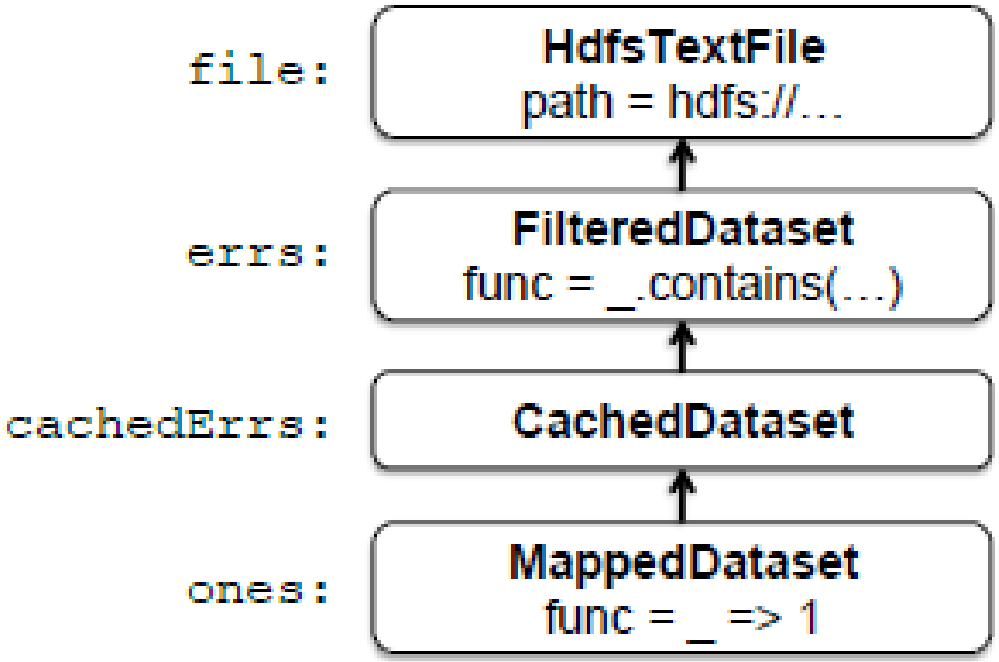
Shared Variables

- *Broadcast* read-only piece of data
  - Initially used HDFS to broadcast variables
- *Accumulator*, using zero & associative function
  - Only driver can read
  - On the workers, a copy of zero'ed accumulator is created per thread
  - After task run, the worker sends a message to the driver program containing the updates
  - Driver folds these

Examples

```
val file = spark.textFile("hdfs://...")
val errs = file.filter(_.contains("ERROR"))
val cachedErrs = errs.cache()
val ones = cachedErrs.map(_ => 1)
val count = ones.reduce(_+_)
```

Lineage


file:
HdfsTextFile
path = hdfs://…
errs:
FilteredDataset
func = _.contains(…)
cachedErrs:
CachedDataset
ones:
MappedDataset
func = _ => 1

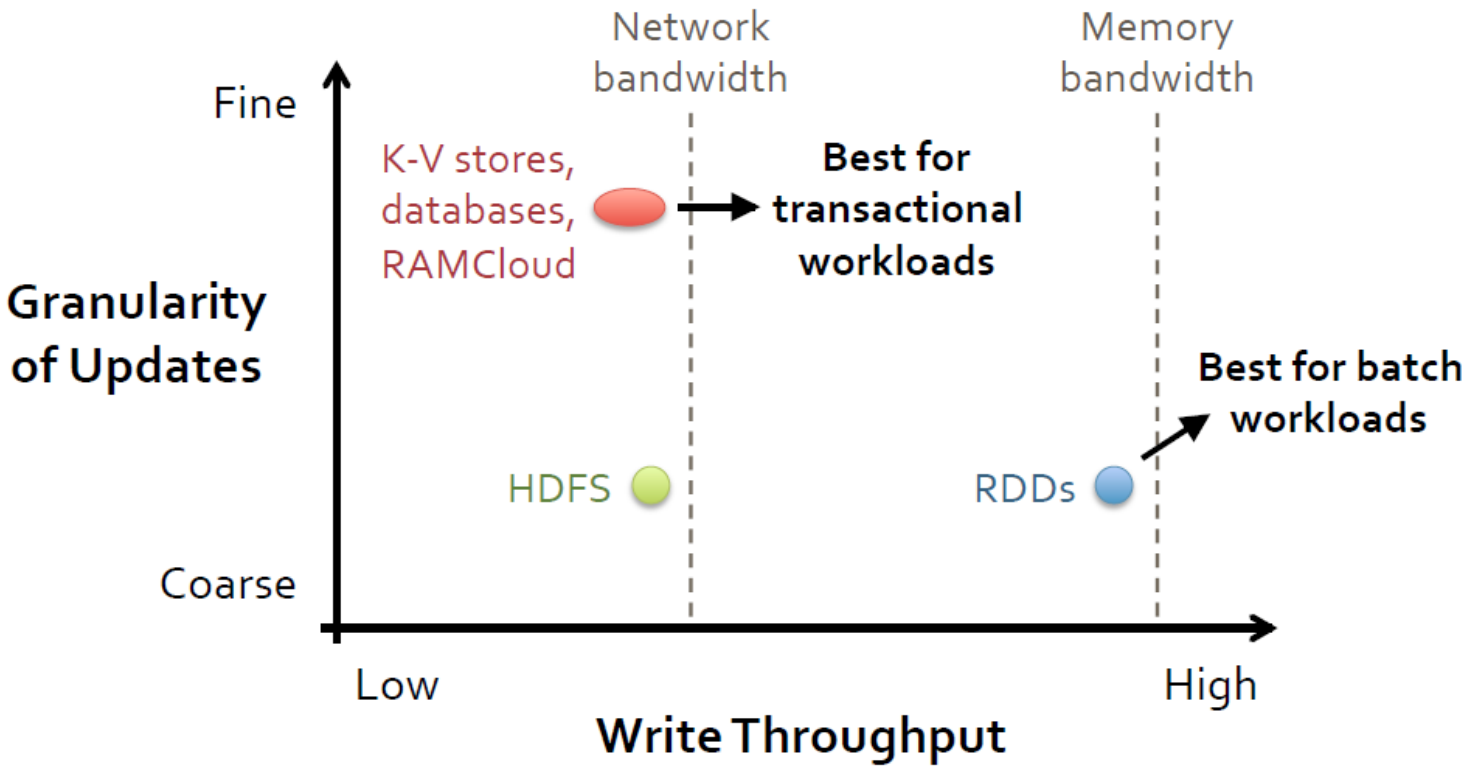
Persistence

- Explicit persistence by developer
- Spill to disk if memory full
  - Priority among RDD's on memory usage
- Persist on disk, replicate

RDD vs. Distributed Shared Mem

Aspect	RDDs	Distr. Shared Mem.
Reads	Coarse- or fine-grained	Fine-grained
Writes	Coarse-grained	Fine-grained
Consistency	Trivial (immutable)	Up to app / runtime
Fault recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using backup tasks	Difficult
Work placement	Automatic based on data locality	Up to app (runtimes aim for transparency)
Behavior if not enough RAM	Similar to existing data flow systems	Poor performance (swapping?)

Trade-off Space



Non-goals

- Does not work for apps with async, fine-grained updates to shared state
- Suited for batch, data-parallel apps, coarse transformations, concise lineage

Creating RDD

- Load external data from distributed storage
- Create logical RDD on which you can operate
- Support for different input formats
  - HDFS files, Cassandra, Java serialized, directory, gzipped
- Can control the number of partitions in loaded RDD
  - Default depends on external DFS, e.g. 128MB on HDFS

```
JavaRDD<String> distFile = sc.textFile("data.txt");
```

RDD Operations

- Transformations
  - From one RDD to one or more RDDs
  - Lazy evaluation…*use with care*
  - Executed in a distributed manner
- Actions
  - Perform aggregations on RDD items
  - Return single (or distributed) results to “driver” code
- RDD.collect() brings RDD partitions to single driver machine

Anonymous Classes

- Data-centric model allows functions to be passed
  - Functions applied to items in the RDD
  - Typically, on individual partitions in data-parallel
- Anonymous class implements interface

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(new Function<String, Integer>() {
  public Integer call(String s) { return s.length(); }
});
int totalLength = lineLengths.reduce(new Function2<Integer, Integer, Integer>() {
  public Integer call(Integer a, Integer b) { return a + b; }
});
```

```
class GetLength implements Function<String, Integer> {
  public Integer call(String s) { return s.length(); }
}
class Sum implements Function2<Integer, Integer, Integer> {
  public Integer call(Integer a, Integer b) { return a + b; }
}

JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(new GetLength());
int totalLength = lineLengths.reduce(new Sum());
```

Anonymous Classes & Lambda Expressions

- Or Java 8 functions are short-forms for simple code fragments to iterate over collections

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(s -> s.length());
int totalLength = lineLengths.reduce((a, b) -> a + b);
```

- Caution: Cannot pass “local” driver variables to lambda expressions/anonymous classes….only **final**
  - Will fail when distributed

```
int counter = 0;
JavaRDD<Integer> rdd = sc.parallelize(data);

// Wrong: Don't do this!!
rdd.foreach(x -> counter += x);

println("Counter value: " + counter);
```

RDD and PairRDD

- RDD is logically a collection of items with a generic type
- PairRDD is like a “Map”, where each item in collection is a <key,value> pair, each a generic type
- Transformation functions use RDD or PairRDD as input/output
- E.g. Map-Reduce

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaPairRDD<String, Integer> pairs = lines.mapToPair(s -> new Tuple2(s, 1));
JavaPairRDD<String, Integer> counts = pairs.reduceByKey((a, b) -> a + b);
```

Transformations

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).

- **JavaRDD<R> map(Function<T,R> f)** : 1:1 mapping from input to output. Can be different types.
- **JavaRDD<T> filter(Function<T,Boolean> f)** : 1:0/1 from input to output, same type.
- **JavaRDD<U> flatMap(FlatMapFunction<T,U> f)** : 1:N mapping from input to output, different types.

Transformations

mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.

- Earlier Map and Filter operate on one item at a time. No state across calls!
- **JavaRDD<U> mapPartitions(FlatMapFunc<Iterator<T>,U> f)**
- mapPartitions has access to iterator of values in entire partition, jot just a single item at a time.

Transformations

sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.

- **JavaRDD<T> sample(boolean withReplacement, double fraction)**: fraction between [0,1] without replacement, >0 with replacement
- **JavaRDD<T> union(JavaRDD<T> other)**: Items in other RDD added to this RDD. Same type. Can have duplicate items (i.e. not a 'set' union).

Transformations

intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numTasks*]))	Return a new dataset that contains the distinct elements of the source dataset.

- **JavaRDD<T> intersection(JavaRDD<T> other)**: Does a set intersection of the RDDs. Output *will not* have duplicates, even if inputs did.
- **JavaRDD<T> distinct()**: Returns a new RDD with unique elements, eliminating duplicates.

Transformations: **PairRDD**

groupByKey([*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numTasks` argument to set a different number of tasks.
reduceByKey(*func*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.

- **JavaPairRDD<K,Iterable<V>> groupByKey()**: Groups values for each key into a single iterable.
- **JavaPairRDD<K,V> reduceByKey(Function2<V,V,V> func)** : Merge the values for each key into a single value using an *associative* and *commutative* reduce function. Output value is of same type as input.
- For aggregate that returns a different type?
- **numPartitions** can be used to generate output RDD with different number of partitions than input RDD.

Transformations

aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numTasks*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean `ascending` argument.

- **JavaPairRDD<K,U> aggregateByKey(U zeroValue, Function2<U,V,U> seqFunc, Function2<U,U,U> combFunc)** : Aggregate the values of each key, using given combine functions and a neutral "zero value".
  - **SeqOp** for merging a V into a U within a partition
  - **CombOp** for merging two U's, within/across partitions
- **JavaPairRDD<K,V> sortByKey(Comparator<K> comp)**: Global sort of the RDD by key
  - *Each partition* contains a sorted range, i.e., output RDD is range-partitioned.
  - Calling collect will return an ordered list of records

Transformations

join(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through `leftOuterJoin`, `rightOuterJoin`, and `fullOuterJoin`.
cartesian(*otherDataset*)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).

- **JavaPairRDD<K, Tuple2<V,W>> join(JavaPairRDD<K,W> other, int numParts):** Matches keys in *this* and *other*. Each output pair is (k, (v1, v2)). Performs a hash join across the cluster.
- **JavaPairRDD<T,U> cartesian(JavaRDDLike<U,?> other)**: Cross product of values in each RDD as a pair

Actions

reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset.

RDD Persistence & Caching

- RDDs can be reused in a dataflow
  - Branch, iteration
- But it will be re-evaluated each time it is reused!
- *Explicitly persist* RDD to reuse output of a dataflow path multiple times
- Multiple storage levels for persistence
  - Disk or memory
  - Serialized or object form in memory
  - Partial spill-to-disk possible
  - *Cache* indicates "persist" to memory

RePartitioning

repartition

```
public JavaRDD<T> repartition(int numPartitions)
```

Return a new RDD that has exactly numPartitions partitions.

Can increase or decrease the level of parallelism in this RDD. Internally, this uses a shuffle to redistribute data.

If you are decreasing the number of partitions in this RDD, consider using `coalesce`, which can avoid performing a shuffle.

coalesce

```
public JavaRDD<T> coalesce(int numPartitions,
                           boolean shuffle)
```

Return a new RDD that is reduced into `numPartitions` partitions.

Job Scheduling: Static

- Apps get excusive set of executors
- Standalone Mode: Apps execute in FIFO, try and use all cores available. Can bound cores & memory per app.
- YARN: Can decide executors per app, cores/memory per executor

https://spark.apache.org/docs/latest/job-scheduling.html

Job Scheduling: Dynamic

- Allows in-flight apps to return resources to the cluster
  - Set a flag
  - Use an *external shuffle service*
- Heuristic to decide executor request & remove policy
  - Request if pending tasks waiting beyond timeout. Multiple rounds, exponential increase in executors requested
  - Remove if executor idle for longer than timeout
- Remove will delete memory/disk contents of executor
  - In-flight tasks may rely on shuffle output from it!
  - External shuffle service copies in the shuffle output
  - If RDD is cached in an executor, executor will NOT be removed!

https://spark.apache.org/docs/latest/job-scheduling.html

Jobs within an App

- FIFO, first job gets all resources for its stages. Then next job, etc.
  - Heavy jobs can delay later jobs
- Fair scheduling of tasks across jobs is possible
  - Round robin assignment of tasks from jobs to resources