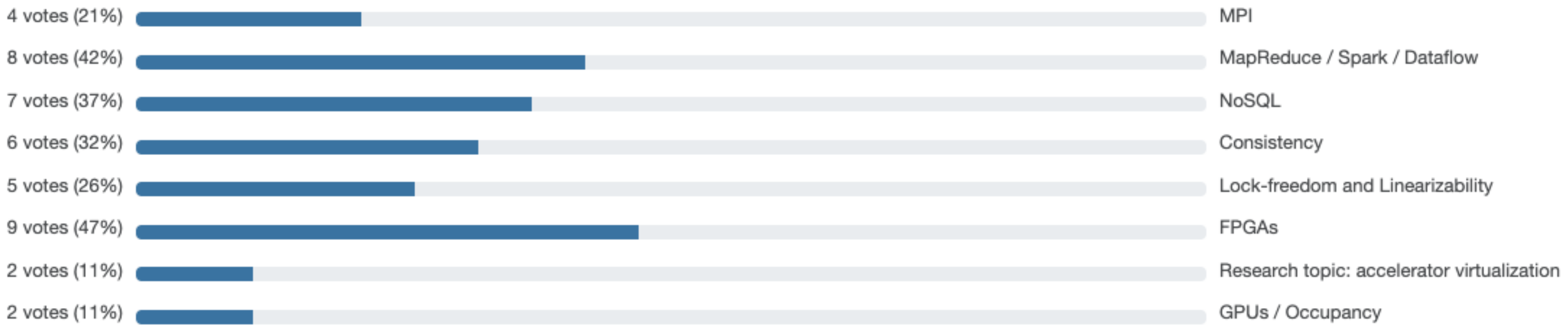
End-of-semester Review

cs378h

Outline/Administrivia

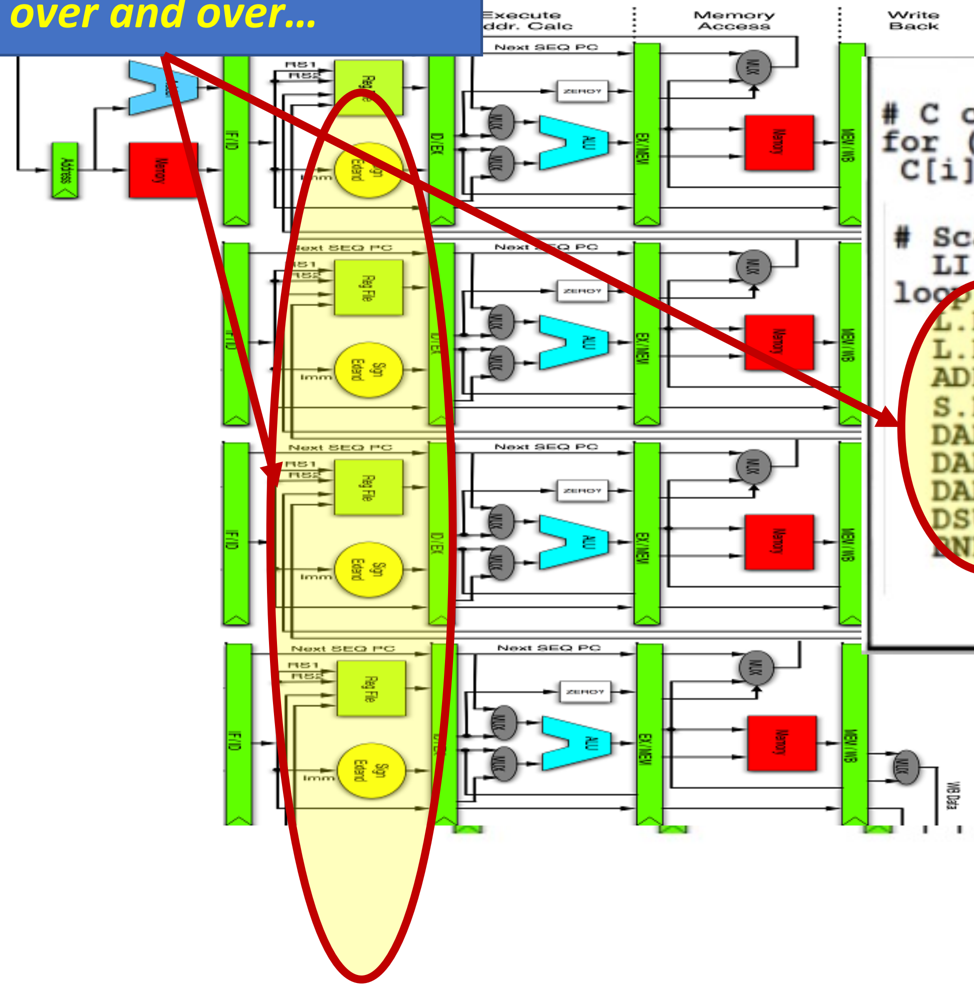
- Questions?
- Review
  - Can someone please act as scribe?
  - Requested review content:
    - GPUs: SIMT vs SIMD, schedulers, limitations on threads/blocks and num blocks, divergence, sharing global memory
    - FPGAs/Verilog: CLB, BRAM, and LUT
    - MPI, distributed systems, shared nothing architectures, PGAS
    - Distributed systems (like CAP and NoSQL)
    - Consistency guarantees?
    - Linearizability vs. Serializability

Your requests



Review: what is a vector proc

Dont decode same instruction over and over…

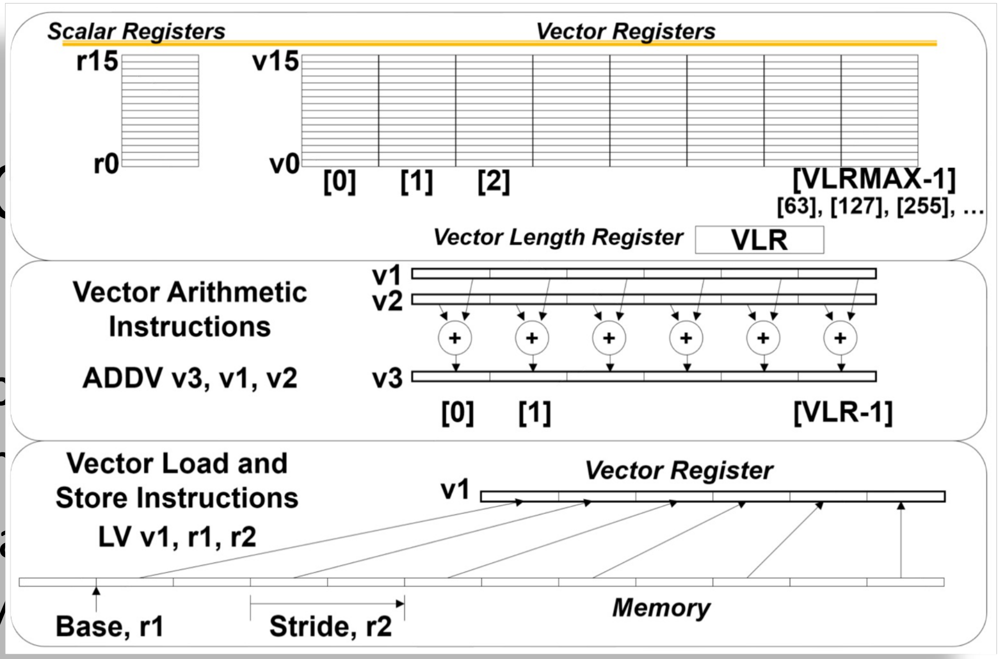



```
# C code
for (i=0; i<64; i++)
 C[i] = A[i] + B[i];

# Scalar Code
  LI     R4, 64
loop:
  L.D    F0, 0(R1)
  L.D    F2, 0(R2)
  ADD.D  F4, F2, F0
  S.D    F4, 0(R3)
  DADDIU R1, 8
  DADDIU R2, 8
  DADDIU R3, 8
  DSUBIU R4, 1
  BNEZ   R4, loop
```

Imp

- In
- Sa
- M
- Multiple different operands in parallel




```
# C code
for (i=0; i<64; i++)
 C[i] = A[i] + B[i];
```

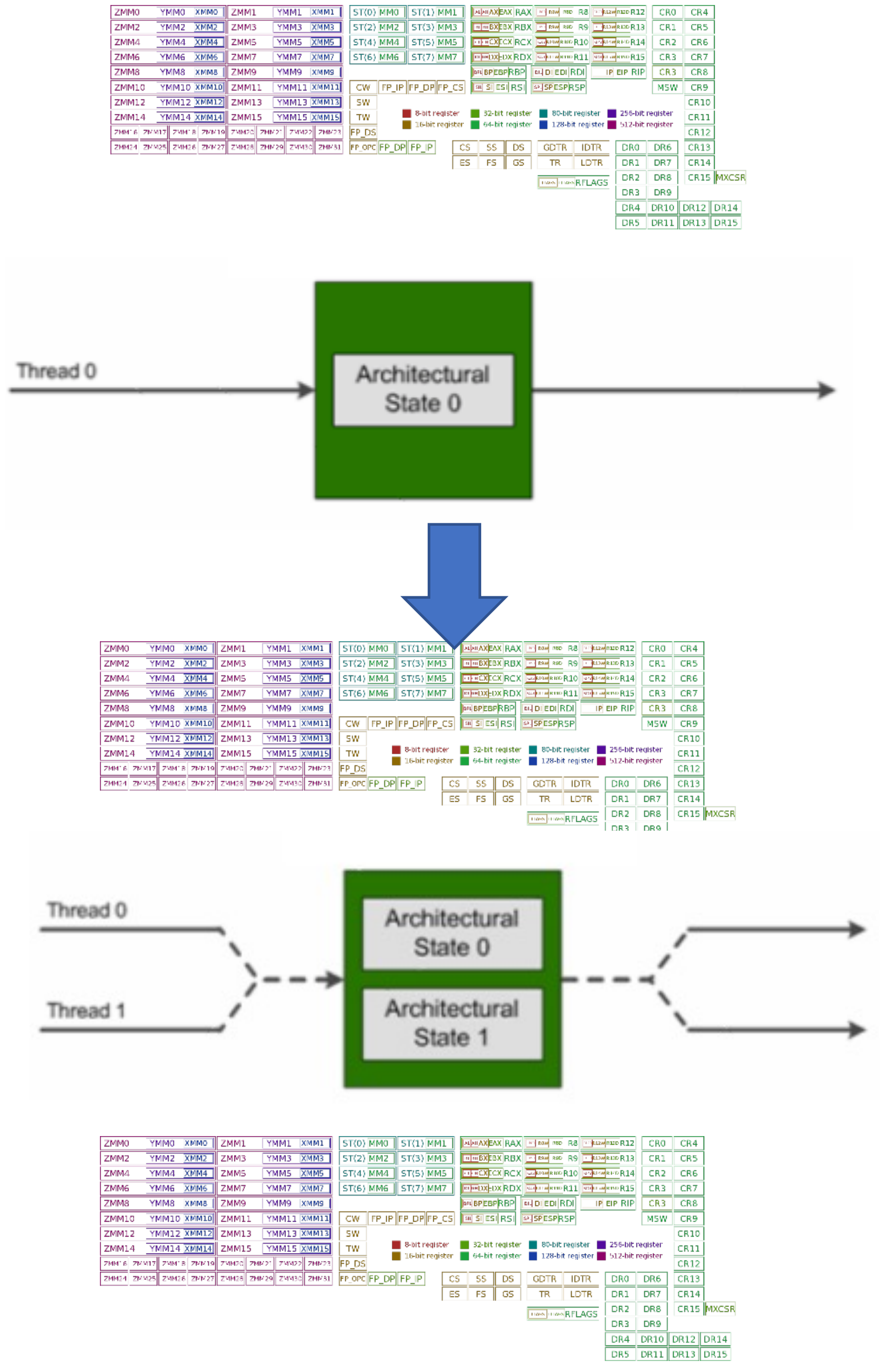
```
# Scalar Code
  LI     R4, 64
loop:
  L.D    F0, 0(R1)
  L.D    F2, 0(R2)
  ADD.D  F4, F2, F0
  S.D    F4, 0(R3)
  DADDIU R1, 8
  DADDIU R2, 8
  DADDIU R3, 8
  DSUBIU R4, 1
  BNEZ   R4, loop
```

```
# Vector Code
  LI     VLR, 64
  LV     V1, R1
  LV     V2, R2
  ADDV.D V3, V1, V2
  SV     V3, R3
```

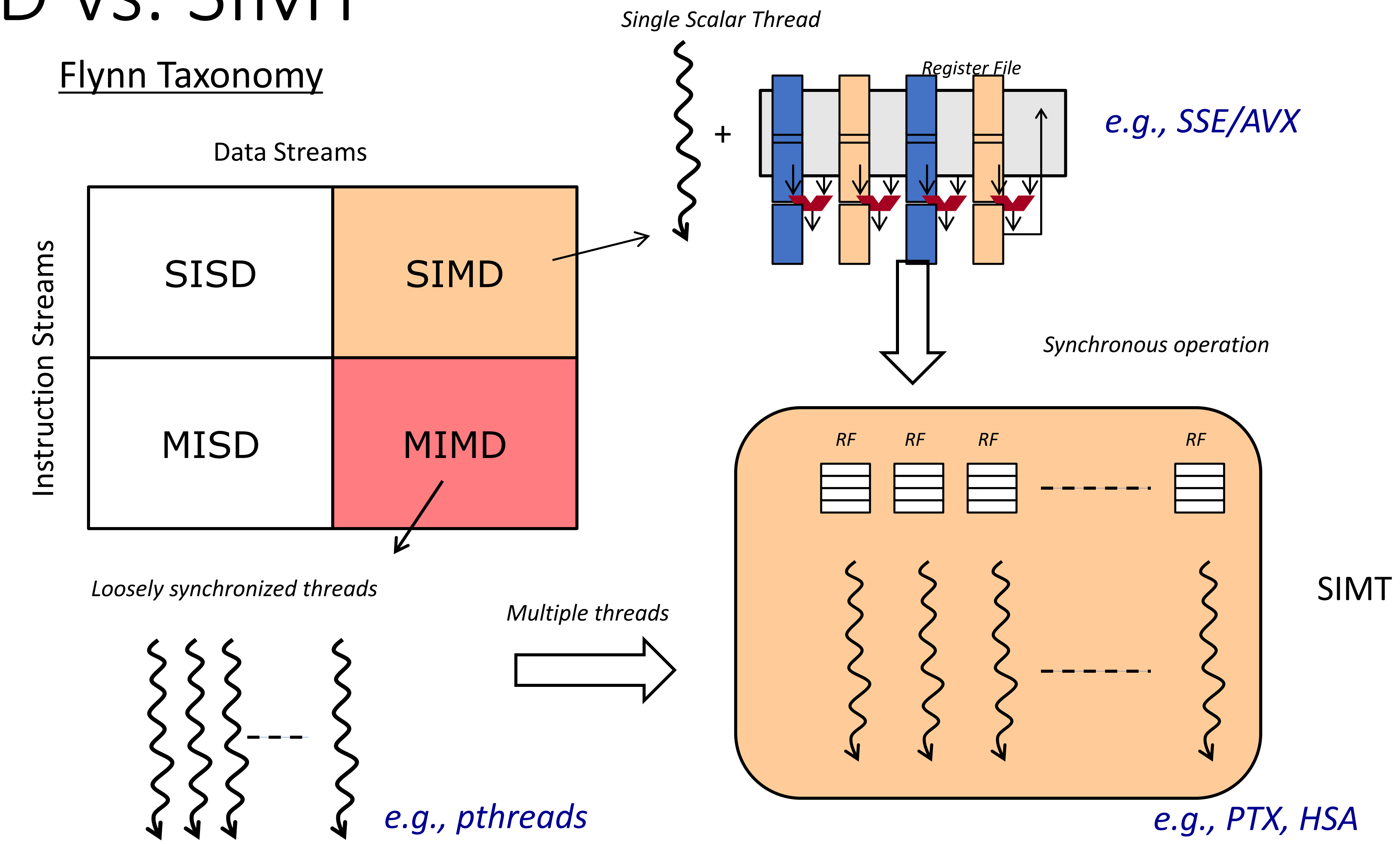
Hardware multi-threading

- Address memory bottleneck
- Share exec unit across
  - Instruction streams
  - Switch on stalls
- Looks like multiple cores to the OS
- Three variants:
  - Coarse
  - Fine-grain
  - Simultaneous

SIMT = SIMD + Hw MT



SIMD vs. SIMT


Flynn Taxonomy
Data Streams
Instruction Streams
SISD
SIMD
MISD
MIMD
Single Scalar Thread
+
Register File
e.g., SSE/AVX
Synchronous operation
Loosely synchronized threads
Multiple threads
RF
RF
RF
RF
SIMT
e.g., pthreads
e.g., PTX, HSA

Review

Thread block scheduler

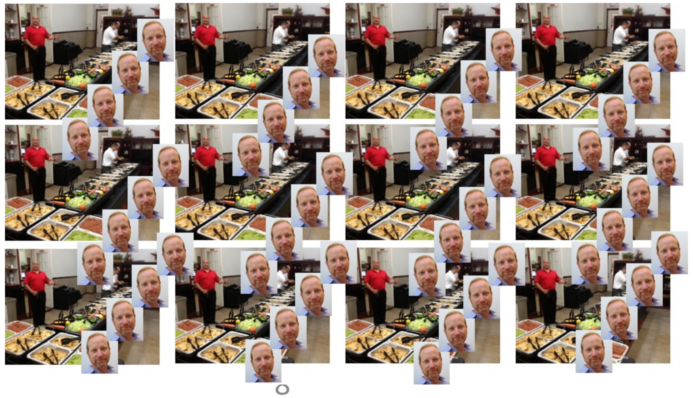
warp (thread) scheduler

- Each SM has multiple vector units (4)
  - 32 lanes wide → warp size
- Vector units use ***hardware multi-threading***
- Execution → a grid of thread blocks (TBs)
  - Each TB has some number of threads

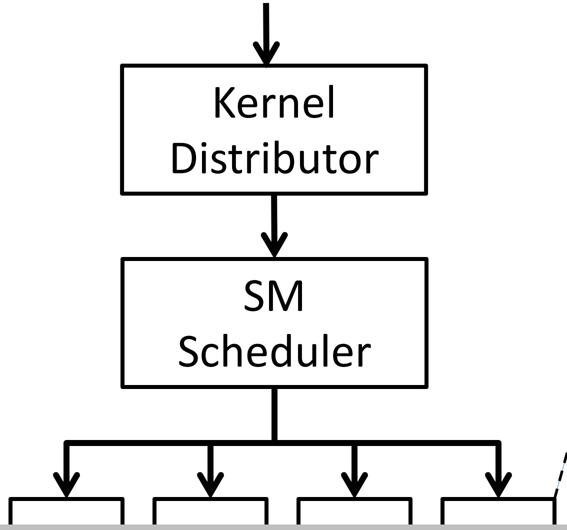
GPU Performance Metric: *Occupancy*

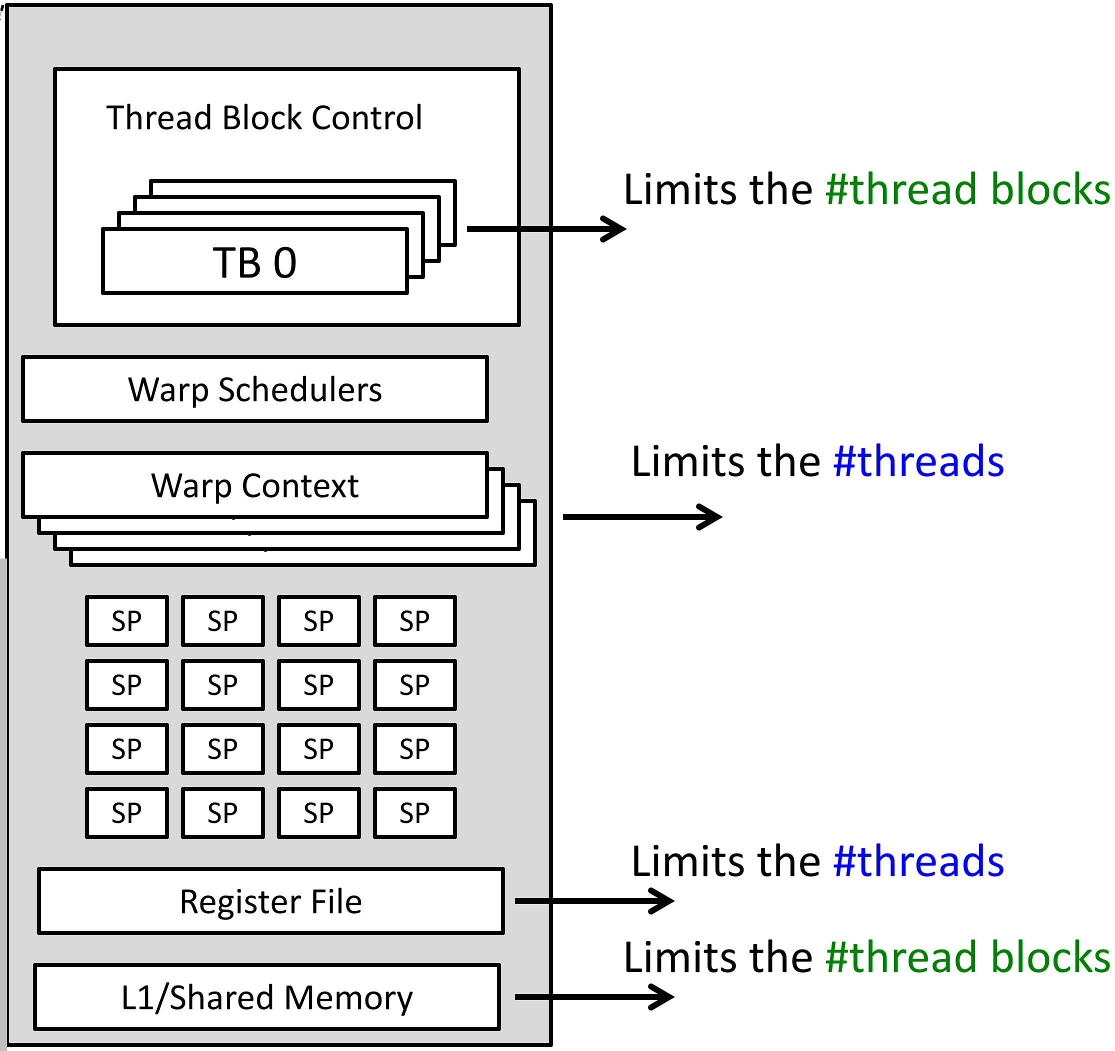
- Occupancy = (#Active Warps) /(#MaximumActive Warps)
  - Measures how well concurrency/parallelism is utilized
- Occupancy captures
  - *which resources* can be dynamically shared
  - how to reason about resource demands of a CUDA kernel
  - Enables device-specific online tuning of kernel parameters

Shouldn't we just create as many threads as possible?



Hardware Resources Are Finite



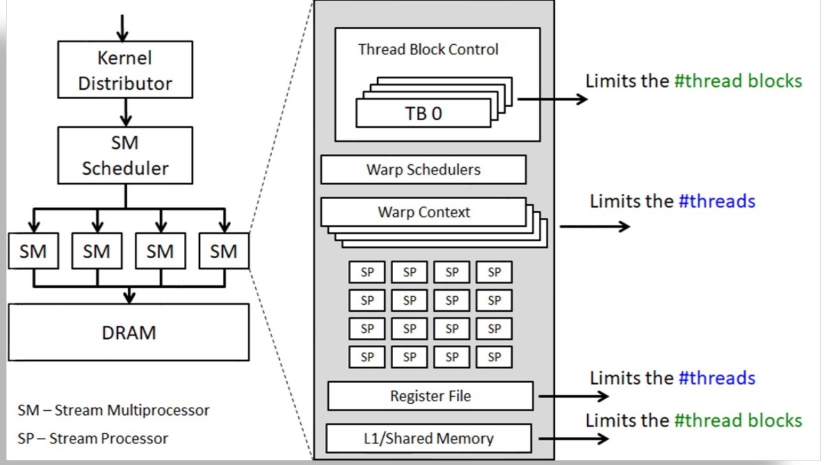


Occupancy:

- (#Active Warps) /(#MaximumActive Warps)
- Limits on the numerator:
  - Registers/thread
  - Shared memory/thread block
  - Number of scheduling slots: blocks, warps
- Limits on the denominator:
  - Memory bandwidth
  - Scheduler slots

What is the performance impact of varying kernel resource demands?

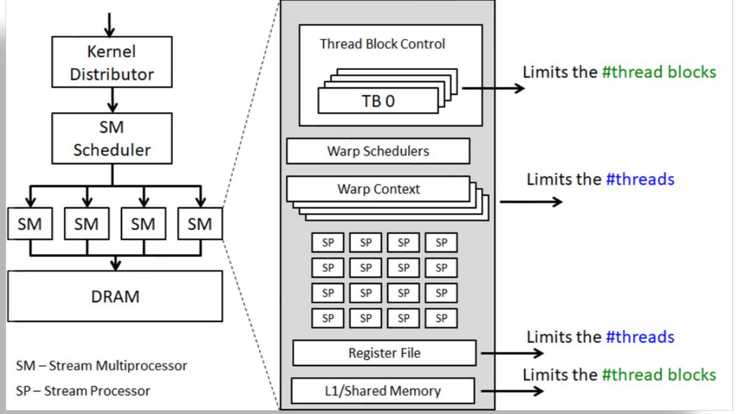
Impact of Thread Block Size



Example: v100:

- max active warps/SM == 64 (limit: warp context)
- max active blocks/SM == 32 (limit: block control)
  - With 512 threads/block how many blocks can execute (per SM) concurrently?
  - Max active warps * threads/warp = 64*32 = 2048 threads → 4
  - With 128 threads/block? → 16
- Consider HW limit of 32 thread blocks/SM @ 32 threads/block:
  - Blocks are maxed out, but max active threads = 32*32 = 1024
  - Occupancy = .5 (1024/2048)
- To maximize utilization, thread block size should balance
  - Limits on active thread blocks vs.
  - Limits on active warps

Impact of #Registers Per Thread



Registers/thread can limit number of active threads!

V100:

- Registers per thread max: 255
- 64K registers per SM

Assume a kernel uses 32 registers/thread, thread block size of 256

- Thus, A TB requires 8192 registers for a maximum of 8 thread blocks per SM
  - Uses all 2048 thread slots (8 blocks * 256 threads/block)
  - 8192 *regs/block * 8 block/SM = 64k registers*
  - *FULLY Occupied!*
- What is the impact of increasing number of registers by 2?
  - Recall: granularity of management is a thread block!
  - Loss of concurrency of 256 threads!
  - *34 regs/thread * 256 threads/block * 7 blocks/SM = 60k registers,*
  - *8 blocks would over-subscribe register file*
  - *Occupancy drops to .875!*

Control Flow Divergence

- Performance concern with branching: divergence
  - Threads within a single warp take different paths
  - Different execution paths are serialized
    - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
- Common case: branch condition is a function of thread ID
  - Example with divergence:
    - `If (threadIdx.x > 2) { }`
    - This creates two different control paths for threads in a block
    - Branch granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp
  - Example without divergence:
    - `If (threadIdx.x / WARP_SIZE > 2) { }`
    - Also creates two different control paths for threads in a block
    - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path

FPGAs/Verilog

- CLB, BRAM, and LUT?
- CLB: combinational logic block
- BRAM: block random access memory
- LUT: lookup table
- Other questions?

Blocking vs Non-blocking Behavior

- A sequence of nonblocking assignments don’t communicate

```
a = 1;
b = a;
c = b;
```

Blocking assignment:
a = b = c = 1

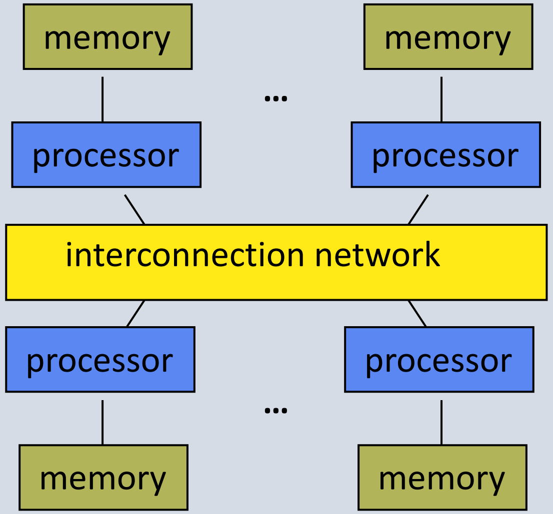
```
a <= 1;
b <= a;
c <= b;
```

Nonblocking assignment:
a = 1
b = old value of a
c = old value of b

MPI

Distributed Memory Multiprocessor

Messaging between nodes

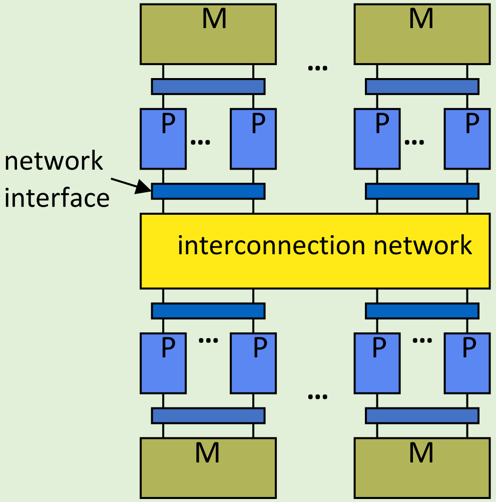


Massively Parallel Processor (MPP)

Many, many processors

Cluster of SMPs

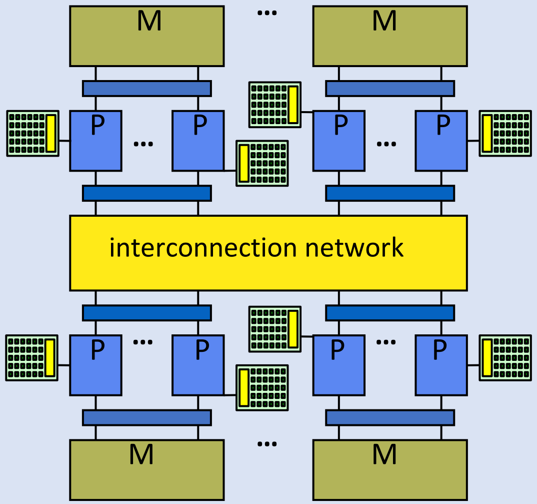
- Shared memory in SMP node
- Messaging ←→ SMP nodes



- also regarded as MPP if processor # is large

Multicore SMP+GPU Cluster

- Shared mem in SMP n
- Messaging between n



- GPU accelerators attached

PGAS = partitioned global address space

How is that different from shared nothing?

What is NoSQL?

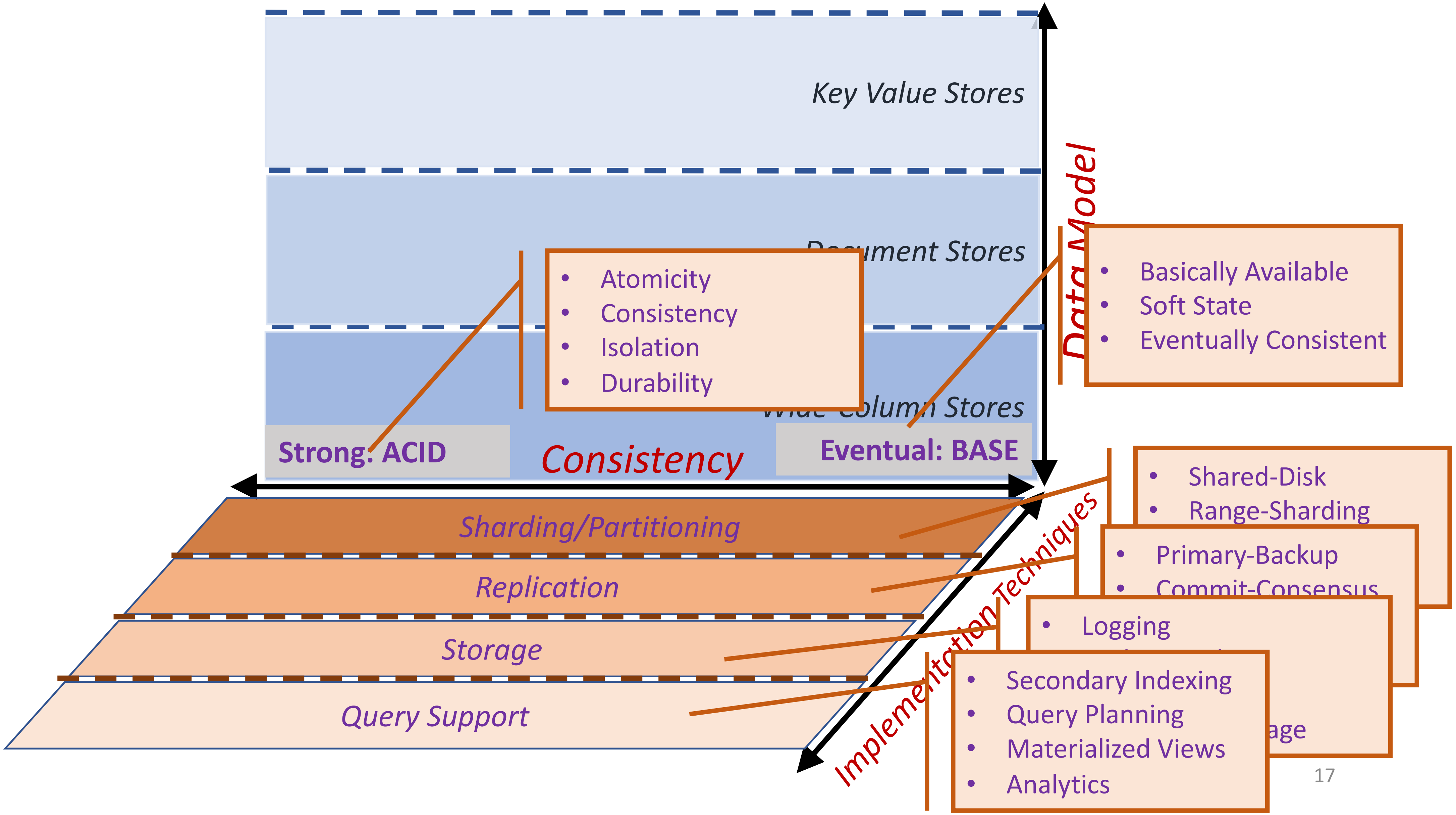
- Next Generation Compute/Storage engines (databases)
  - **non-relational**
  - **distributed**
  - **open-source**
  - **horizontally scalable**
- One view: “no” → elide SQL/da
- Another view: “NoSQL” is actua
  - more appropriate term is actually “No

What NoSQL gives up in exchange for scale:

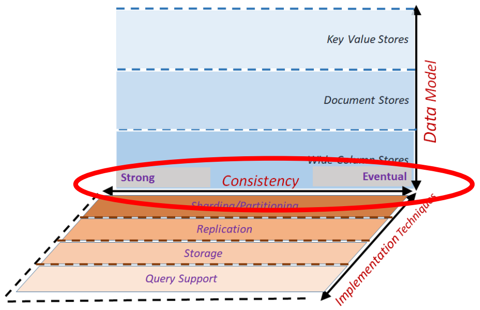
Why talk about NoSQL in concurrency class?

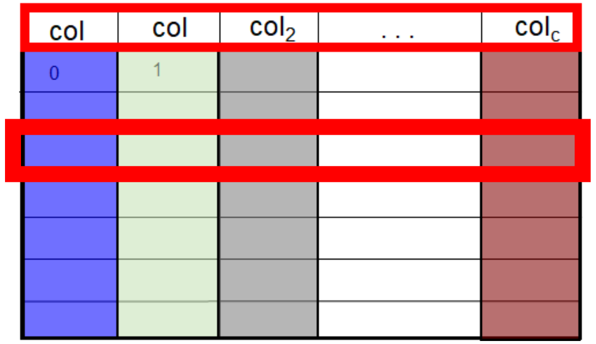
- Principle
  - Most tradeoffs are a ***direct result*** of concurrency
- Practice
  - NoSQL systems are ubiquitous
- Relevant aspects
  - scale/performance tradeoff space
  - Correctness/programmability tradeoff space

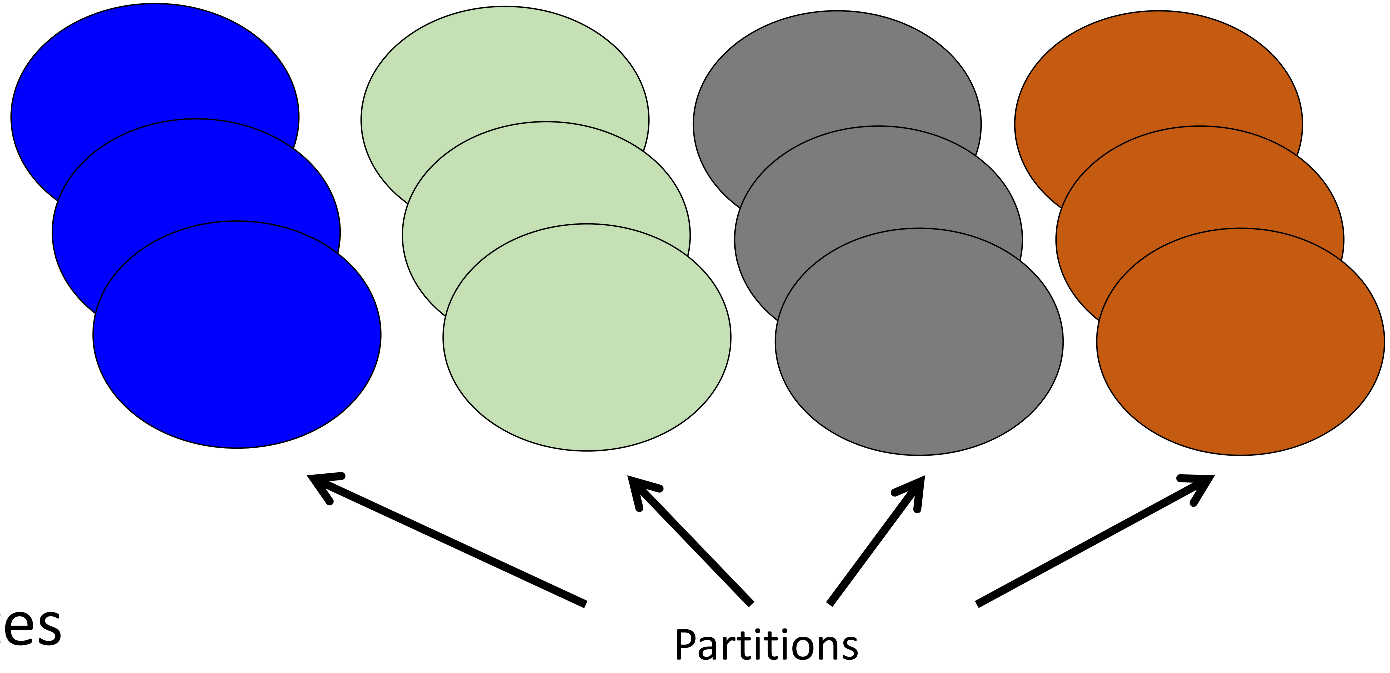
Review: noSQL Taxonomy



Consistency



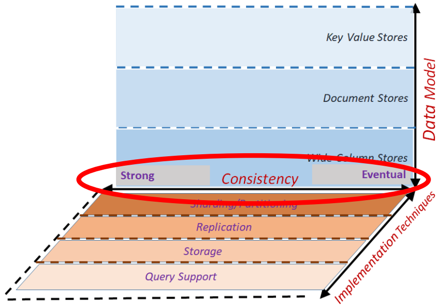




- Clients perform reads and writes
- Data is replicated among a set of servers
- Writes must be performed at all servers
- Reads return the result of one or more past writ
- How to keep data in sync?

Consistency != Correctess

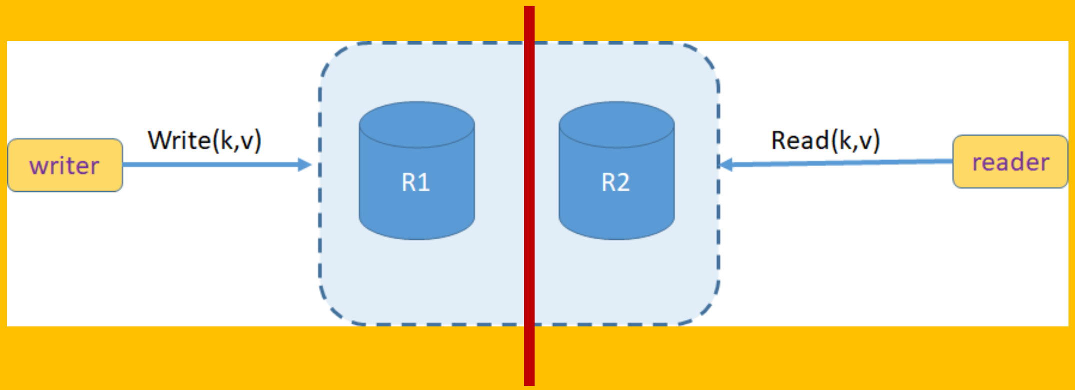
- consistency: no internal contradictions
- Correct: higher-level property
- Inconsistency → code does wrong things



Consistency: CAP Theorem

- A distributed system can satisfy at most 2/3 guarantees of:
  1. **Consistency**:
     - all nodes see same data at any time
     - or reads return latest written value by any client
  2. **Availability**:
     - system allows operations all the time,
     - and operations return quickly
  3. **Partition-tolerance**:
     - system continues to work in spite of netwo

Why is this "theorem" true?

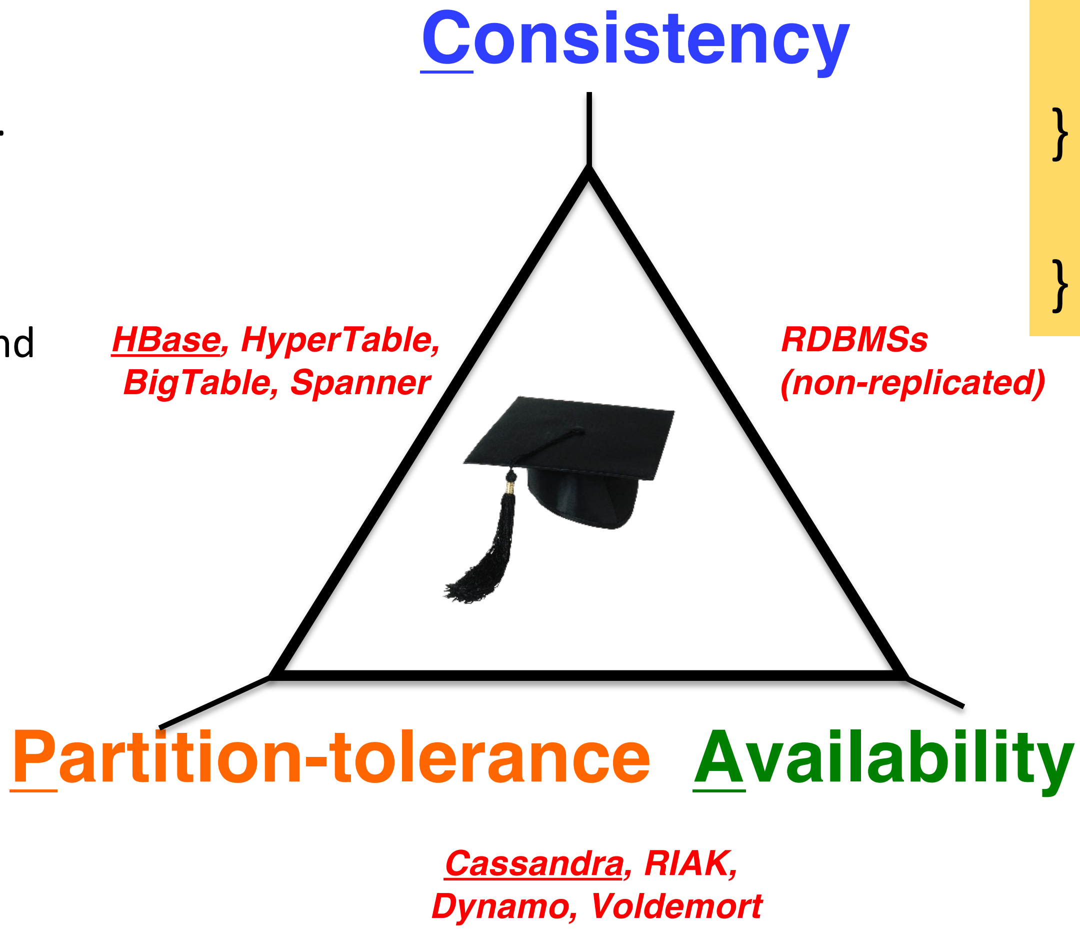


if(partition) { keep going } → !consistent && available

if(partition) { stop } → consistent && !available

CAP Implications

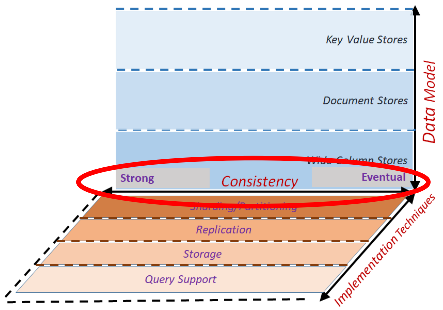
- A distributed storage system can achieve at most two of C, A, and P.
- When partition-tolerance is important, you have to choose between consistency and availability



PACELC:

```
if(partition) {
   choose A or C
} else {
   choose latency or consistency
}
```

Consistency Spectrum



- Eventual Consistency
  - If writes to a key stop, all replicas of key will converge
  - Originally from Amazon's Dynamo and LinkedIn's Voldemort systems

BASE:

- Basically Available
- Soft State
- Eventually Consistent

- **Strict:**
  - Absolute time ordering of all shared accesses, reads always return last write
- **Linearizability**:
  - Each operation is visible (or available) to all other clients in real-time order
- **Sequential Consistency** [Lamport]:
  - *"... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*
  - After the fact, find a "reasonable" ordering of the operations (can re-order operations) that obeys sanity (consistency) at all clients, and across clients.
- **ACID** properties

Eventual: BASE

Faster reads and writes ←

Strong: ACID

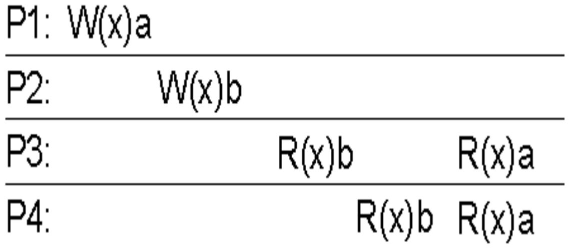
More consistency →

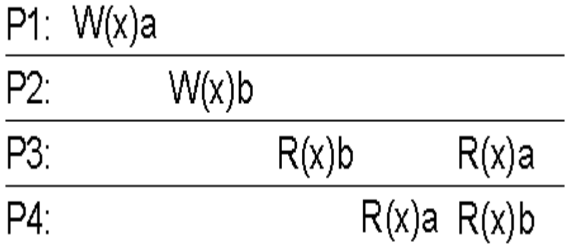
Eventual

Strong
(e.g., Sequential)

Sequential Consistency

- weaker than strict/strong consistency
  - All operations are executed in *some* sequential order
  - each process issues operations in program order
    - Any valid interleaving is allowed
    - All agree on the same interleaving
    - Each process preserves its program order





(b)

- ***Why is this weaker than strict/strong?***
- ***Nothing is said about "most recent write"***

Causal consistency

- Causally related writes seer
    - *Causally?*
    - *Concurrent* writes may be se machines

Causal:
If a write produces a value that
causes another write, they are causally related

X = 1
if(X > 0) {
    Y = 1
}
Causal consistency → all see X=1, Y=1 in same order

P1:	W(x)a			
P2:	R(x)a	W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

(a)

Not permitted

P3:	R(x)b	R(x)a
P4:	R(x)a	R(x)b

(b)

Permitted

Linearizability vs. Serializability

- Linearizability assumes sequential consistency *and*
  - If TS(x) < TS(y) then OP(x) should precede OP(y) in the sequence
  - Stronger than sequential consistency
- Difference between linearizability and serializability?
  - Granularity: reads/writes versus transactions

Linearizability:

- Single-operation, single-object, real-time order
- Talks about order of ops on single object (e.g. atomic register)
- Ops should appear instantaneous, reflect real time order

Serializability:

- Talks about groups of 1 or more ops on one or more objects
- Txns over multiple items equivalent to serial order of txns
- Only requires *some* equivalent serial order

Serializability + Linearizability == "Strict Serializability"

- Txn order equivalent to some serial order ***that respects real time order***
- Linearizability: degenerate case of Strict Ser: txns are single op single object

http://www.bailis.org/blog/linearizability-versus-serializability/

Some Consistency Guarantees

Strong Consistency	See all previous writes.
Eventual Consistency	See subset of previous writes.
Consistent Prefix	See initial sequence of writes.
Bounded Staleness	See all “old” writes.
Monotonic Reads	See increasing subset of writes.
Read My Writes	See all writes performed by reader.

NoSQL faux quiz:

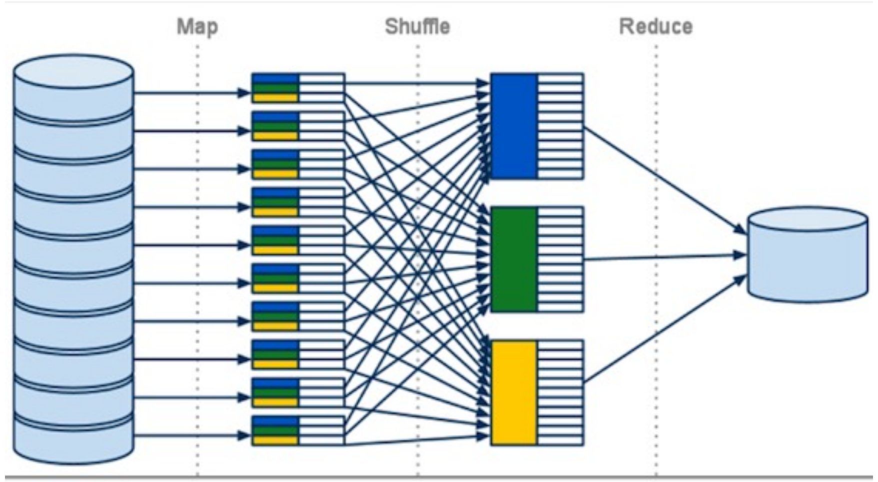
- What is the CAP theorem? What does “PACELC” stand for and how does it relate to CAP?
- What is the difference between ACID and BASE?
- Why do NoSQL systems claim to be more horizontally scalable than RDMBSes? List some features NoSQL systems give up toward this goal?
- What is eventual consistency? Give a concrete example of how of why it causes a complex programming model (relative to a strongly consistent model).
- Compare and contrast Key-Value, Document, and Wide-column Stores
- Define and contrast the following consistency properties:
  - strong consistency, eventual consistency, consistent prefix, monotonic reads, read-my-writes, bounded staleness

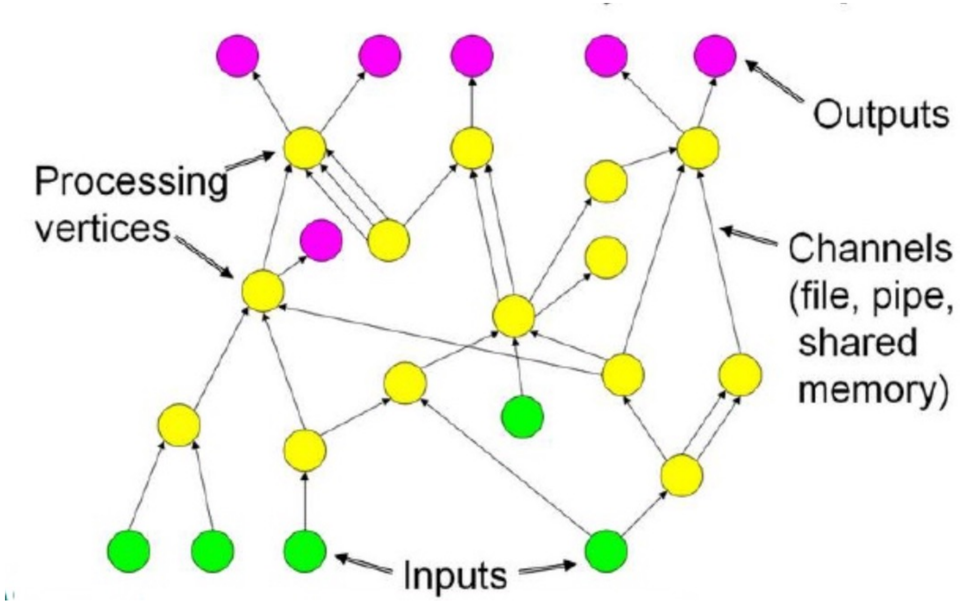
NoSQL faux quiz:

- What is the CAP theorem? What does "PACELC" stand for and how does it relate to CAP?
- What is the difference between ACID and BASE?
- Why do NoSQL systems claim to be more horizontally scalable than RDMBSes? ~~List some features NoSQL systems give up toward this goal?~~
- What is eventual consistency? Give a concrete example of how of why it causes a complex programming model (relative to a strongly consistent model).
- ~~Compare and contrast Key-Value, Document, and Wide-column Stores~~
- Define and contrast the following consistency properties:
  - strong consistency, eventual consistency, consistent prefix, monotonic reads, read-my-writes, bounded staleness

Dataflow

- MR is a ***dataflow*** engine
- So are Lots of others
    - Dryad
    - DryadLINQ
    - Dandelion
    - CIEL
    - GraphChi/PowerGraph/Pregel
    - Spark



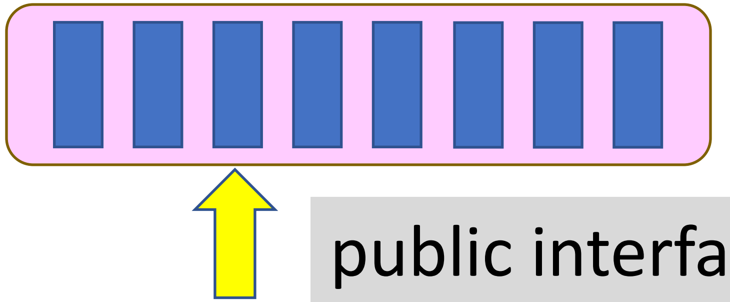


Spark faux quiz (5 min, any 2):

- What is the difference between *transformations* and *actions* in Spark?
- Spark supports a persist API. When should a programmer want to use it? When should she [not] use use the "*RELIABLE*" flag?
- Compare and contrast fault tolerance guarantees of Spark to those of MapReduce. How are[n't] the mechanisms different?
- Is Spark a good system for indexing the web? For computing page rank over a web index? Why [not]?
- List aspects of Spark's design that help/hinder multi-core parallelism relative to MapReduce. If the issue is orthogonal, explain why.

Collections and Iterators

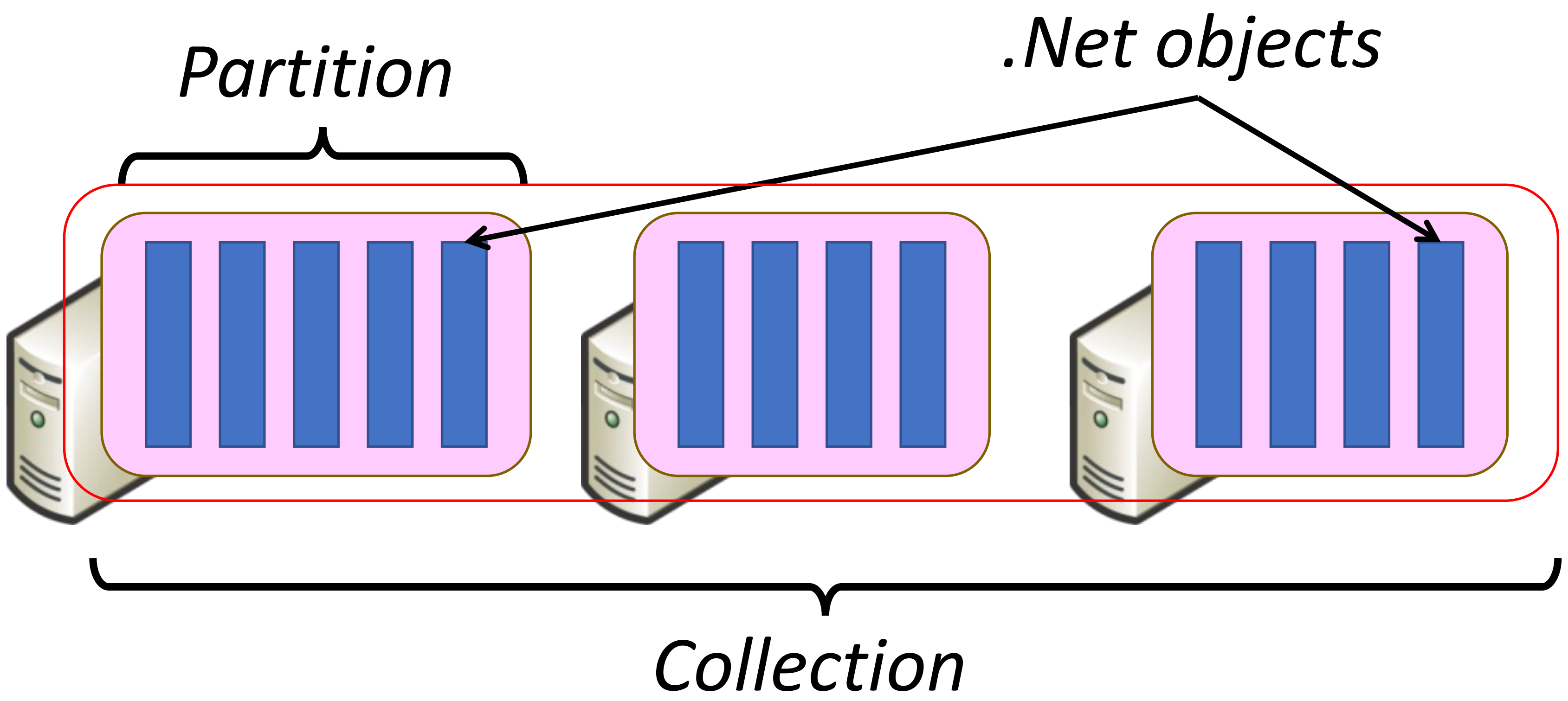
```
class Collection<T> : IEnumerable<T>;
```



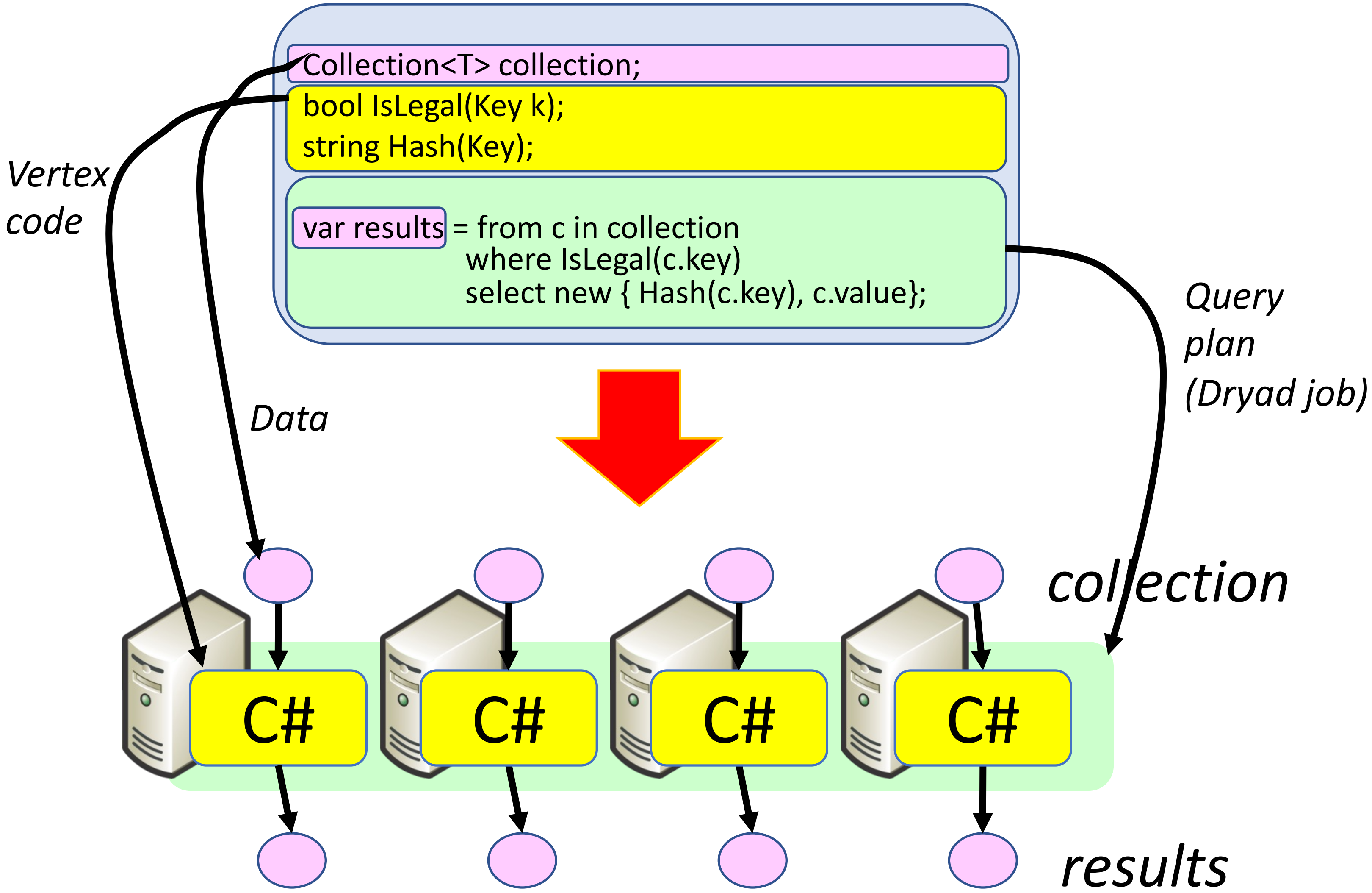
```
public interface IEnumerable<T>  {
        IEnumerator<T> GetEnumerator();
}
```

```
public interface IEnumerator <T> {
        T Current { get; }
        bool MoveNext();
        void Reset();
}
```

DryadLINQ Data Model

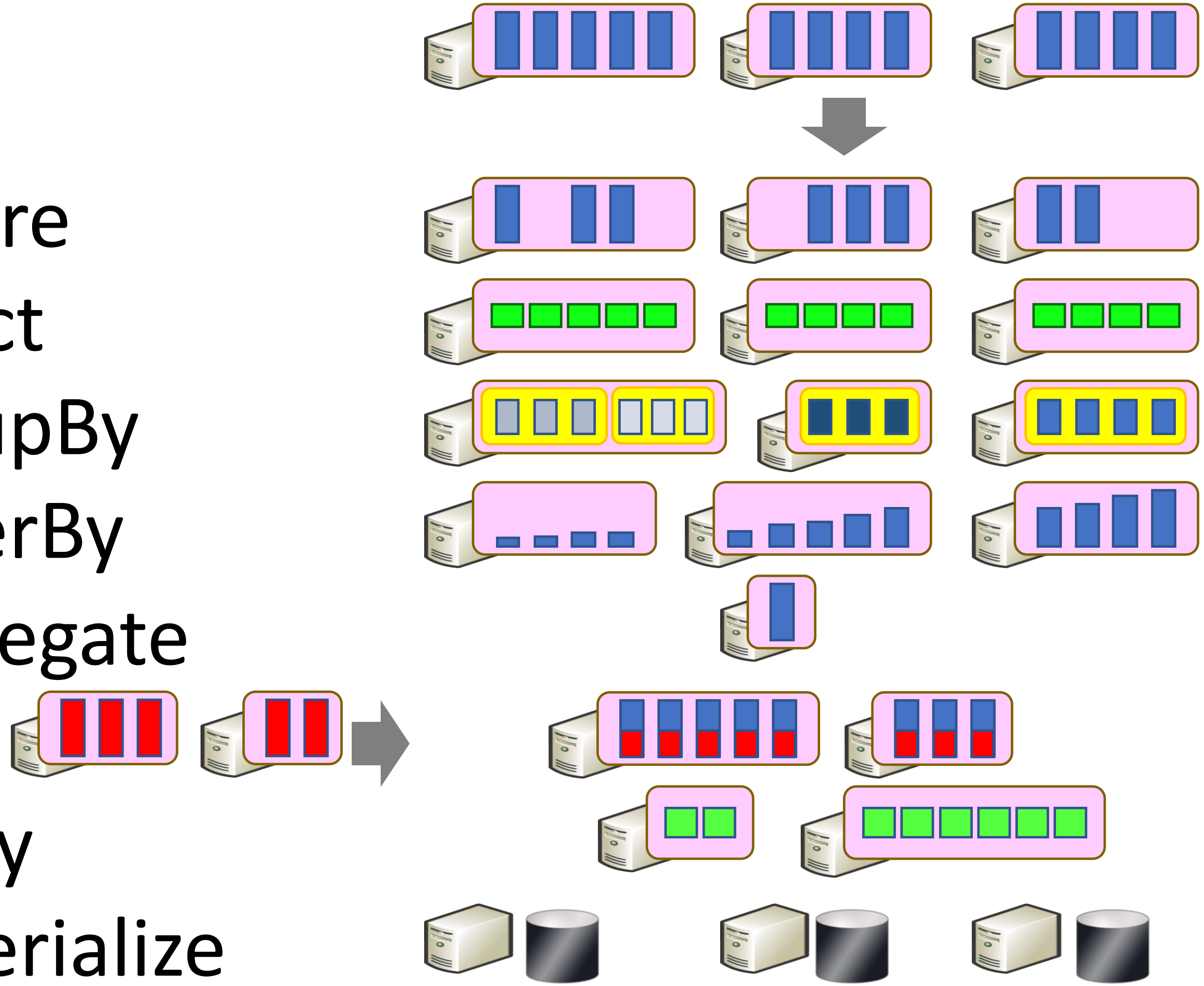


DryadLINQ = LINQ + Dryad



Language Summary

Where
Select
GroupBy
OrderBy
Aggregate
Join
Apply
Materialize



Example: Histogram

```
public static IQueryable<Pair> Histogram(
      IQueryable<LineRecord> input, int k)
{
    var words = input.SelectMany(x => x.line.Split(' '));
    var groups = words.GroupBy(x => x);
    var counts = groups.Select(x => new Pair(x.Key, x.Count()));
    var ordered = counts.OrderByDescending(x => x.count);
    var top = ordered.Take(k);
    return top;
}
```

“A line of words of wisdom”
[“A”, “line”, “of”, “words”, “of”, “wisdom”]
[[“A”], [“line”], [“of”, “of”], [“words”], [“wisdom”]]
[{“A”, 1}, {“line”, 1}, {“of”, 2}, {“words”, 1}, {“wisdom”, 1}]
[{“of”, 2}, {“A”, 1}, {“line”, 1}, {“words”, 1}, {“wisdom”, 1}]
[{“of”, 2}, {“A”, 1}, {“line”, 1}]

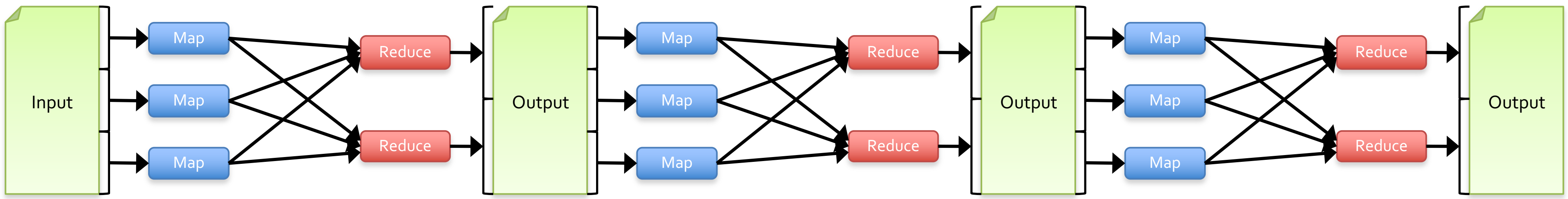
Iterative Computations: PageRank

1. Start each page with a rank of 1
2. On each iteration, update each page's rank to

$$\Sigma_{i \in neighbors} rank_i / |neighbors_i|$$

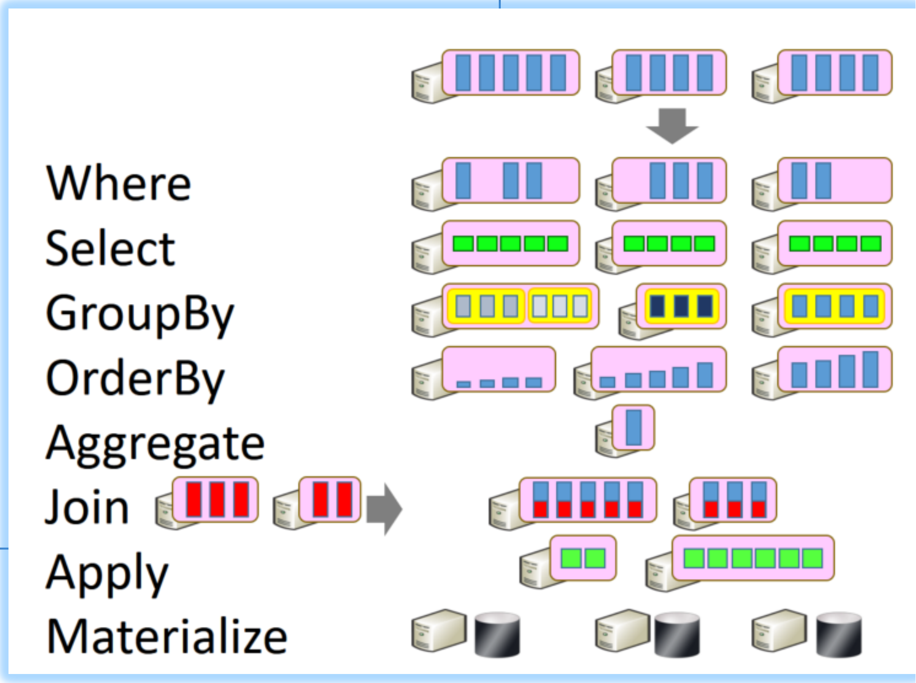
```
links = // RDD of (url, neighbors) pairs
ranks = // RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
  ranks = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }.reduceByKey(_ + _)
}
```



RDD Operations

Transformations (define a new RDD)	**Parallel operations** (return a result to driver)
map	reduce
filter	collect
sample	count
union	save
groupByKey	lookupKey
reduceByKey	…
join	
persist/*cache*	
…	

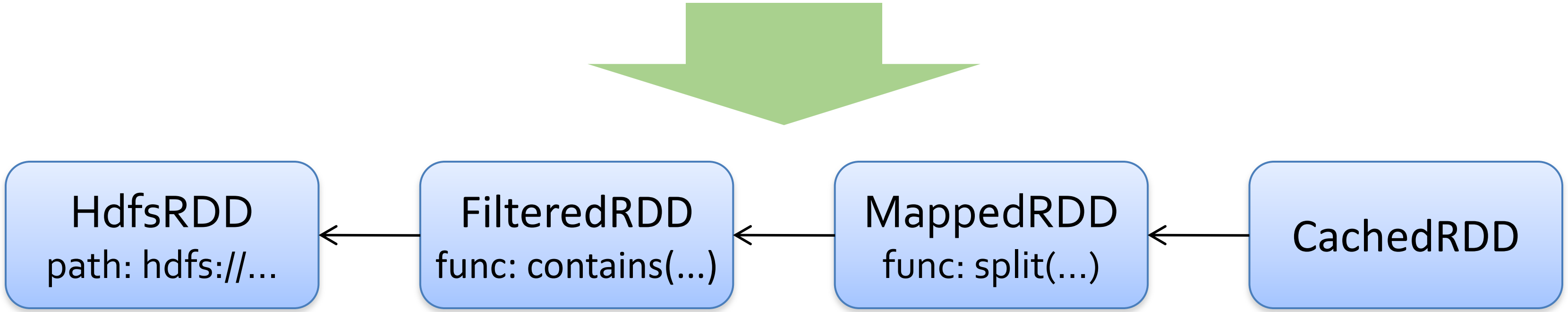


RDD Fault Tolerance

- RDDs maintain *lineage* information that can be used to reconstruct lost partitions

- Ex:

```
cachedMsgs = textFile(...).filter(_.contains(“error”))
                          .map(_.split(‘\t’)(2))
                          .persist()
```



RDDs vs Distributed Shared Memory

Concern	RDDs	Distr. Shared Mem.
Reads	Fine-grained	Fine-grained
Writes	Bulk transformations	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 speculative execution	Difficult
Work placement	Automatic based on data locality	Up to app (but runtime aims for transparency)