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
Review: what is a vector processor?
Review: what is a vector processor?
Review: what is a vector processor?

Don't decode same instruction over and over...
Review: what is a vector processor?
Review: what is a vector processor?
Review: what is a vector processor?

Implementation:
- Instruction fetch control logic shared
- Same instruction stream executed on
- Multiple pipelines
- Multiple different operands in parallel
Review: what is a vector processor?

Implementation:
- Instruction fetch control logic shared
- Same instruction stream executed on multiple pipelines
- Multiple different operands in parallel

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

Example vector code:
```
# Scalar Code
LD 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
```

Example memory operations:
- Load scalar
- Load vector
- Add scalar
- Add vector
Hardware multi-threading
Hardware multi-threading

• Address memory bottleneck
Hardware multi-threading

- Address memory bottleneck
- Share exec unit across
  - Instruction streams
  - Switch on stalls
Hardware multi-threading

• Address memory bottleneck
• Share exec unit across
  • Instruction streams
  • Switch on stalls
Hardware multi-threading

- Address memory bottleneck
- Share exec unit across
  - Instruction streams
  - Switch on stalls
Hardware multi-threading

- Address memory bottleneck
- Share exec unit across
  - Instruction streams
  - Switch on stalls
- Looks like multiple cores to the OS
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
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

- **SISD** (Single Instruction Single Data)
- **SIMD** (Single Instruction Multiple Data)
- **MISD** (Multiple Instruction Single Data)
- **MIMD** (Multiple Instruction Multiple Data)

**Data Streams**

**Instruction Streams**

- **Single Scalar Thread**
- **Loosely synchronized threads**
- **Multiple threads**

**Register File (RF)**

- **Synchronous operation**

**SIMT** (Single Instruction Multiple Threads)

- **e.g., SSE/AVX**
- **e.g., PTX, HSA**

**e.g., pthreads**
Review
Review

• Each SM has multiple vector units (4)
  • 32 lanes wide → warp size
Review

- Each SM has multiple vector units (4)
  - 32 lanes wide → warp size
- Vector units use **hardware multi-threading**
• 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
Review

• 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
Review

- 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
Review

- 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*
GPU Performance Metric: **Occupancy**

- **Occupancy** = (#Active Warps) / (#MaximumActive Warps)
  - Measures how well concurrency/parallelism is utilized
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
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
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
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
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
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

SM – Stream Multiprocessor
SP – Stream Processor
Hardware Resources Are Finite

- Kernel Distributor
- SM Scheduler
- SM – Stream Multiprocessor
- SP – Stream Processor
- DRAM
- Warp Schedulers
- Warp Context
- Register File
- L1/Shared Memory
- Thread Block Control
- TB 0
- Limits the #thread blocks

SM – Stream Multiprocessor
SP – Stream Processor
Hardware Resources Are Finite

SM – Stream Multiprocessor
SP – Stream Processor

Kernel Distributor

SM Scheduler

SM
SM
SM
SM

DRAM

Warp Schedulers

Warp Context

Thread Block Control

TB 0

Register File

L1/Shared Memory

Limits the #thread blocks

Limits the #threads
Hardware Resources Are Finite

SM – Stream Multiprocessor
SP – Stream Processor
Hardware Resources Are Finite

SM – Stream Multiprocessor
SP – Stream Processor
Hardware Resources Are Finite

Occupancy:
- \(\frac{(\text{Active Warps})}{(\text{Maximum Active Warps})}\)
- Limits on the numerator:
  - Registers/thread
  - Shared memory/thread block
  - Number of scheduling slots: blocks, warps
- Limits on the denominator:
  - Memory bandwidth
  - Scheduler slots
Hardware Resources Are Finite

Occupancy:
- \( \frac{(\text{Active Warps})}{(\text{Maximum Active Warps})} \)
- 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
Impact of Thread Block Size

Example: v100:
Impact of Thread Block Size

Example: v100:
• max active warps/SM == 64 (limit: warp context)
Impact of Thread Block Size

Example: v100:

- max active warps/SM == 64 (limit: warp context)
- max active blocks/SM == 32 (limit: block control)
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
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
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
  - With 128 threads/block?
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
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)
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
  • 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
Impact of #Registers Per Thread

Registers/thread can limit number of active threads!
Impact of #Registers Per Thread

Registers/thread can limit number of active threads!

V100:
Impact of #Registers Per Thread

Registers/thread can limit number of active threads!

V100:
• Registers per thread max: 255
Impact of #Registers Per Thread

Registers/thread can limit number of active threads!

V100:
- Registers per thread max: 255
- 64K registers per SM
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
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)
  - $8192 \text{ regs/block} \times 8 \text{ block/SM} = 64k \text{ registers}$
  - FULLY Occupied!
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?
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!
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:
    • \( \text{If (threadIdx.x > 2) \{ \}} \)
    • 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:
    • \( \text{If (threadIdx.x / WARP_SIZE > 2) \{ \}} \)
    • 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
FPGAs/Verilog

- CLB, BRAM, and LUT?
FPGAs/Verilog

- CLB, BRAM, and LUT?
- CLB: combinational logic block
FPGAs/Verilog

• CLB, BRAM, and LUT?
• CLB: combinational logic block
• BRAM: block random access memory
FPGAs/Verilog

• CLB, BRAM, and LUT?
• CLB: combinational logic block
• BRAM: block random access memory
• LUT: lookup table
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

<table>
<thead>
<tr>
<th>Blocking assignment:</th>
<th>Nonblocking assignment:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = b = c = 1</code></td>
<td><code>a = 1</code></td>
</tr>
<tr>
<td><code>b = old value of a</code></td>
<td><code>b = old value of a</code></td>
</tr>
<tr>
<td><code>c = old value of b</code></td>
<td><code>c = old value of b</code></td>
</tr>
</tbody>
</table>

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

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

Distributed Memory
Multiprocessor
Messaging between nodes

processor
memory
interconnection network
processor
memory
...
MPI

Distributed Memory Multiprocessor
Messaging between nodes

Massively Parallel Processor (MPP)
Many, many processors
Massively Parallel Processor (MPP)
Many, many processors

Distributed Memory Multiprocessor
Messaging between nodes

MPI

Interconnection network
MPI

Distributed Memory Multiprocessor
Messaging between nodes

Massively Parallel Processor (MPP)
Many, many processors

Cluster of SMPS
• Shared memory in SMP node
• Messaging $\leftrightarrow$ SMP nodes

• also regarded as MPP if processor # is large
Massively Parallel Processor (MPP)
Many, many processors

Distributed Memory Multiprocessor
Messaging between nodes

Cluster of SMPs
- Shared memory in SMP node
- Messaging ↔ SMP nodes

Multicore SMP+GPU Cluster
- Shared mem in SMP node
- Messaging between nodes

- also regarded as MPP if processor # is large

- GPU accelerators attached
**MPI**

Distributed Memory Multiprocessor
- Messaging between nodes

Massively Parallel Processor (MPP)
- Many, many processors

Cluster of SMPs
- Shared memory in SMP node
- Messaging $\leftrightarrow$ SMP nodes

Multicore SMP+GPU Cluster
- Shared mem in SMP node
- Messaging between nodes
- 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/database functionality to achieve scale
- Another view: “NoSQL” is actually misleading.
  - more appropriate term is actually “Not Only SQL”
What is NoSQL?

- Next Generation Compute/Storage engines (databases)
  - non-relational
  - distributed
  - open-source
  - horizontally scalable
- One view: “no” → elide SQL/database functionality to achieve scale
- Another view: “NoSQL” is actually misleading.
  - more appropriate term is actually “Not Only SQL”

What NoSQL gives up in exchange for scale:
- Relationships between entities are non-existent
- Limited or no ACID transactions
- No standard language for queries (SQL)
- Less structured
What is NoSQL?

- Next Generation Compute/Storage engines (databases)
  - non-relational
  - distributed
  - open-source
  - horizontally scalable
- One view: “no” $\rightarrow$ elide SQL/database functionality to achieve scale
- Another view: “NoSQL” is actually misleading.
  - more appropriate term is actually “Not Only SQL”

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
Review: noSQL Taxonomy

Consistency
Review: noSQL Taxonomy

Data Model

Consistency
Review: noSQL Taxonomy

Consistency

Data Model

Implementation Techniques
Review: noSQL Taxonomy

Strong: ACID  

Eventual: BASE  

Consistency
Review: noSQL Taxonomy

Strong: ACID

- Atomicity
- Consistency
- Isolation
- Durability

Eventual: BASE

Consistency

Implementation Techniques

Data Model
Review: noSQL Taxonomy

ACID vs. BASE

Consistency:
- Atomicity
- Consistency
- Isolation
- Durability

Data Model:
- Basically Available
- Soft State
- Eventually Consistent

Implementation Techniques:

16
Review: noSQL Taxonomy

- Strong: ACID
- Eventual: BASE

Consistency

Data Model

Implementation Techniques
Review: noSQL Taxonomy

Consistency

Strong: ACID

Eventual: BASE
Review: noSQL Taxonomy

Key Value Stores

Strong: ACID
Eventual: BASE

Consistency
Review: noSQL Taxonomy

- **Key Value Stores**
- **Document Stores**

**Consistency**
- **Strong: ACID**
- **Eventual: BASE**

**Data Model**

**Implementation Techniques**
Review: noSQL Taxonomy

- Key Value Stores
- Document Stores
- Wide-Column Stores

Consistency:
- Strong: ACID
- Eventual: BASE
Review: noSQL Taxonomy

- Key Value Stores
- Document Stores
- Wide-Column Stores

Consistency:
- Strong: ACID
- Eventual: BASE

Implementation Techniques:
- Sharding/Partitioning
Review: noSQL Taxonomy

- Key Value Stores
- Document Stores
- Wide-Column Stores

**Consistency**
- Strong: ACID
- Eventual: BASE

**Implementation Techniques**
- Sharding/Partitioning
- Replication
Review: noSQL Taxonomy

- Key Value Stores
- Document Stores
- Wide-Column Stores

Consistency

- Strong: ACID
- Eventual: BASE

Implementation Techniques

- Sharding/Partitioning
- Replication
- Storage
Review: noSQL Taxonomy

- **Key Value Stores**
- **Document Stores**
- **Wide-Column Stores**

**Consistency**
- Strong: ACID
- Eventual: BASE

**Implementation Techniques**
- Sharding/Partitioning
- Replication
- Storage
- Query Support
Review: noSQL Taxonomy

- **Key Value Stores**
- **Document Stores**
- **Wide-Column Stores**

**Consistency**
- Strong: ACID
- Eventual: BASE

**Sharding/Partitioning**
- Shared-Disk
- Range-Sharding
- Hash-Sharding
- Consistent Hashing

**Replication**

**Storage**

**Query Support**

**Implementation Techniques**

[Diagram showing data model with various categories and techniques]
Review: noSQL Taxonomy

- Key Value Stores
- Document Stores
- Wide-Column Stores

**Consistency**
- Strong: ACID
- Eventual: BASE

**Implementation Techniques**
- Storage
- Replication
- Sharding/Partitioning
- Query Support
Review: noSQL Taxonomy

- **Key Value Stores**
- **Document Stores**
- **Wide-Column Stores**

**Consistency**
- Strong: ACID
- Eventual: BASE

**Implementation Techniques**
- Primary-Backup
- Commit-Consensus Protocol
- Sync/Async

**Data Model**
Review: noSQL Taxonomy

- **Strong: ACID**
- **Eventual: BASE**

**Data Model**
- **Key Value Stores**
- **Document Stores**
- **Wide-Column Stores**

**Implementation Techniques**
- **Sharding/Partitioning**
- **Replication**
- **Storage**
- **Query Support**

Consistency
Review: noSQL Taxonomy

Key Value Stores

Document Stores

Wide-Column Stores

Data Model

Consistency

Strong: ACID

Eventual: BASE

Implementation Techniques

- Logging
- Update In Place
- Caching
- In-Memory Storage

Query Support

Storage

Replication

Sharding/Partitioning
Review: noSQL Taxonomy

- Key Value Stores
- Document Stores
- Wide-Column Stores

Consistency:
- Strong: ACID
- Eventual: BASE

Implementation Techniques:
- Sharding/Partitioning
- Replication
- Storage
- Query Support

Data Model
Review: noSQL Taxonomy

Key Value Stores

Document Stores

Wide-Column Stores

Consistency

Strong: ACID

Eventual: BASE

Sharding/Partitioning

Replication

Storage

Query Support

Implementation Techniques

• Secondary Indexing
• Query Planning
• Materialized Views
• Analytics
Review: noSQL Taxonomy

- **Key Value Stores**
- **Document Stores**
- **Wide-Column Stores**

**Consistency**
- Strong: ACID
- Eventual: BASE

**Implementation Techniques**
- Sharding/Partitioning
- Replication
- Storage
- Query Support
Consistency

<table>
<thead>
<tr>
<th>col1</th>
<th>col2</th>
<th>...</th>
<th>coln</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Consistency

<table>
<thead>
<tr>
<th>col1</th>
<th>col2</th>
<th>col3</th>
<th>...</th>
<th>coln</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Consistency

<table>
<thead>
<tr>
<th>col</th>
<th>col</th>
<th>col₂</th>
<th>...</th>
<th>colₙ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Partitions

- Blue
- Green
- Brown
- Grey

**Partitions**

- Consistency
- System
- Internal
- External
- Key-Value Store
- Document Store
- Query
## Consistency

<table>
<thead>
<tr>
<th>col</th>
<th>col</th>
<th>col₂</th>
<th>...</th>
<th>colₙ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Partitions**
Consistency

<table>
<thead>
<tr>
<th>col</th>
<th>col</th>
<th>col2</th>
<th>...</th>
<th>coln</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Partitions
Consistency

- Clients perform reads and writes
Consistency

- Clients perform reads and writes
- Data is replicated among a set of servers
Consistency

- Clients perform reads and writes
- Data is replicated among a set of servers
- Writes must be performed at all servers
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 writes
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 writes
- How to keep data in sync?
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 writes
- How to keep data in sync?

Consistency ≠ Correctness

- Consistency: no internal contradictions
- Correct: higher-level property
- Inconsistency → code does wrong things
Consistency: CAP Theorem
Consistency: CAP Theorem

• A distributed system can satisfy at most 2/3 guarantees of:
Consistency: CAP Theorem

• A distributed system can satisfy at most 2/3 guarantees of:
  1. Consistency:
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
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:
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
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:
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 network partitions
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 network
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 network partitions

---

Why care about CAP Properties?

**Availability**
- Reads/writes complete reliably and quickly.
- E.g. Amazon, each ms latency → $6M yearly loss.

**Partitions**
- Internet router outages
- Under-sea cables cut
- rack switch outage
- system should continue functioning normally!

**Consistency**
- all nodes see same data at any time, or reads return latest written value by any client.
- *This basically means correctness!*
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 network partitions

---

Why care about CAP Properties?

- **Availability**
  - Reads/writes complete reliably and quickly.
  - E.g. Amazon, each ms latency $\rightarrow$ $6M$ yearly loss.

- **Partitions**
  - Internet router outages
  - Under-sea cables cut
  - Rack switch outage
  - System should continue functioning normally!

- **Consistency**
  - all nodes see same data at any time, or reads return latest written value by any client
  - This basically means correctness!
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 network partitions

Why is this “theorem” true?
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 network partitions
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 network partitions

Why is this “theorem” true?
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 network partitions

Why is this “theorem” true?

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

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.

Consistency

Partition-tolerance

Availability

HBase, HyperTable, BigTable, Spanner

RDBMSs (non-replicated)

Cassandra, RIAK, Dynamo, Voldemort
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.

**Consistency**

- HBase, HyperTable, BigTable, Spanner

**Partition-tolerance**

- Cassandra, RIAK, Dynamo, Voldemort

**Availability**

RDBMSs (non-replicated)

CAP is flawed
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:**

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

CAP is flawed.
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**

---

**Faster reads and writes**

**More consistency**

Eventual → Strong (e.g., Sequential)
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

**Strong:**
- **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 Consistency**
- Faster reads and writes
- More consistency

**Strong Consistency**
- 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

<table>
<thead>
<tr>
<th>P1: W(x)a</th>
<th>P1: W(x)a</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2: W(x)b</td>
<td>P2: W(x)b</td>
</tr>
<tr>
<td>P3: R(x)b R(x)a</td>
<td>P3: R(x)b R(x)a</td>
</tr>
<tr>
<td>P4: R(x)b R(x)a</td>
<td>P4: R(x)a R(x)b</td>
</tr>
</tbody>
</table>

(a) (b)
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

<table>
<thead>
<tr>
<th>P1: W(x)a</th>
<th>P2: W(x)b</th>
<th>P3: R(x)b R(x)a</th>
<th>P4: R(x)b R(x)a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Why is this weaker than strict/strong?
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

Why is this weaker than strict/strong?

Nothing is said about “most recent write”
Causal consistency
Causal consistency

• Causally related writes seen by all processes in same order.
Causal consistency

• Causally related writes seen by all processes in same order.
  • *Causally?*
Causal consistency

- Causally related writes seen by all processes in same order.
  - *Causally?*

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

```
X = 1
if(X > 0) {
  Y = 1
}
```

Causal consistency $\rightarrow$ all see $X=1, Y=1$ in same order
Causal consistency

• Causally related writes seen by all processes in same order.
  • *Causally?*
Causal consistency

• Causally related writes seen by all processes in same order.
  • Causally?
  • Concurrent writes may be seen in different orders on different machines
Causal consistency

• Causally related writes seen by all processes in same order.
  • *Causally?*
  • *Concurrent* writes may be seen in different orders on different machines

```
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
```
Causal consistency

• Causally related writes seen by all processes in same order.
  • Causally?
  • Concurrent writes may be seen in different orders on different machines

<table>
<thead>
<tr>
<th>P1: W(x)a</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2: R(x)a W(x)b</td>
</tr>
<tr>
<td>P3: R(x)b R(x)a</td>
</tr>
<tr>
<td>P4: R(x)a R(x)b</td>
</tr>
</tbody>
</table>

(a)

Not permitted
Causal consistency

- Causally related writes seen by all processes in same order.
  - *Causally?*
  - *Concurrent* writes may be seen in different orders on different machines

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

Not permitted
Causal consistency

- Causally related writes seen by all processes in same order.
  - *Causally*?
  - *Concurrent* writes may be seen in different orders on different machines

<table>
<thead>
<tr>
<th>P1: W(x)a</th>
<th>P2: R(x)a W(x)b</th>
<th>P3: R(x)b R(x)a</th>
<th>P4: R(x)a R(x)b</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Not permitted

<table>
<thead>
<tr>
<th>P1: W(x)a</th>
<th>P2: W(x)b</th>
<th>P3: R(x)b R(x)a</th>
<th>P4: R(x)a R(x)b</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Permitted
Linearizability vs. Serializability

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

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

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

• Linearizability assumes sequential consistency \textit{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:

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

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

• Linearizability assumes sequential consistency \textit{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

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

- Linearizability assumes sequential consistency and
  - If $\text{TS}(x) < \text{TS}(y)$ then $\text{OP}(x)$ should precede $\text{OP}(y)$ 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:
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

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

• Linearizability assumes sequential consistency \textit{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

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

Serializable:
• 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

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

- Linearizability assumes sequential consistency *and*
  - If $\text{TS}(x) < \text{TS}(y)$ then $\text{OP}(x)$ should precede $\text{OP}(y)$ 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

<table>
<thead>
<tr>
<th>Consistency Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong Consistency</td>
<td>See all previous writes.</td>
</tr>
<tr>
<td>Eventual Consistency</td>
<td>See subset of previous writes.</td>
</tr>
<tr>
<td>Consistent Prefix</td>
<td>See initial sequence of writes.</td>
</tr>
<tr>
<td>Bounded Staleness</td>
<td>See all “old” writes.</td>
</tr>
<tr>
<td>Monotonic Reads</td>
<td>See increasing subset of writes.</td>
</tr>
<tr>
<td>Read My Writes</td>
<td>See all writes performed by reader.</td>
</tr>
</tbody>
</table>
## Some Consistency Guarantees

<table>
<thead>
<tr>
<th>Consistency Guarantees</th>
<th>See all previous writes.</th>
<th>See subset of previous writes.</th>
<th>See initial sequence of writes.</th>
<th>See all “old” writes.</th>
<th>See increasing subset of writes.</th>
<th>See all writes performed by reader.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong Consistency</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eventual Consistency</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consistent Prefix</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bounded Staleness</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monotonic Reads</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read My Writes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Consistency</th>
<th>Performance</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>
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 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 RDBMSes? 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
Dataflow

• MR is a *dataflow* engine
Dataflow

• MR is a *dataflow* engine
Dataflow

• MR is a *dataflow* engine
• So are Lots of others
  • Dryad
  • DryadLINQ
  • Dandelion
  • CIEL
  • GraphChi/PowerGraph/Pregel
  • Spark
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[not] 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>;
Collections and Iterators

class Collection<T> : IEnumerable<T>;

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

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();
}
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

Partition

.Net objects

Collection
DryadLINQ = LINQ + Dryad

Collection<T> collection;
bool IsLegal(Key k);
string Hash(Key);

var results = from c in collection
where IsLegal(c.key)
select new { Hash(c.key), c.value};
DryadLINQ = LINQ + Dryad

```csharp
Collection<T> collection;
bool IsLegal(Key k);
string Hash(Key);

var results = from c in collection
    where IsLegal(c.key)
    select new { Hash(c.key), c.value};
```
```csharp
Collection<T> collection;
bool IsLegal(Key k);
string Hash(Key);

var results = from c in collection
              where IsLegal(c.key)
              select new { Hash(c.key), c.value};
```

DryadLINQ = LINQ + Dryad
DryadLINQ = LINQ + Dryad

```csharp
Collection<T> collection;
bool IsLegal(Key k);
string Hash(Key);

var results = from c in collection
              where IsLegal(c.key)
              select new { Hash(c.key), c.value};
```

Vertex code

Data

Query plan (Dryad job)

Collection

results
Language Summary
Language Summary

Where
Language Summary

Where
Language Summary

Where
Select
Language Summary

Where

Select
Language Summary

Where
Select
GroupBy
Language Summary

Where
Select
GroupBy
Language Summary

Where
Select
GroupBy
OrderBy
Language Summary

Where
Select
GroupBy
OrderBy
Language Summary

Where
Select
GroupBy
OrderBy
Aggregate
Language Summary

Where
Select
GroupBy
OrderBy
Aggregate
Language Summary

Where
Select
GroupBy
OrderBy
Aggregate
Join
Language Summary

Where  
Select  
GroupBy  
OrderBy  
Aggregate  
Join
Language Summary

Where
Select
GroupBy
OrderBy
Aggregate
Join
Language Summary

Where
Select
GroupBy
OrderBy
Aggregate
Join
Apply
Language Summary

Where
Select
GroupBy
OrderBy
Aggregate
Join
Apply
Language Summary

Where
Select
GroupBy
OrderBy
Aggregate
Join
Apply
Materialize
Language Summary

Where
Select
GroupBy
OrderBy
Aggregate
Join
Apply
Materialize
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

\[ \text{rank}_i = \frac{\sum_{\text{neighbors} \in \text{neighbors}} \text{rank}_j}{|\text{neighbors}|} \]

```
lights = // RDD of (url, neighbors) pairs
ranks = // RDD of (url, rank) pairs
for (i <= 1 to ITERATIONS) {
    ranks = lights.join(ranks).flatMap {
        (url, (links, rank)) =>
        .map(dest => (dest, rank(links.size)))
    ).reduceByKey(_ + _)
}
```
## RDD Operations

<table>
<thead>
<tr>
<th>Transformations (define a new RDD)</th>
<th>Parallel operations (return a result to driver)</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>reduce</td>
</tr>
<tr>
<td>filter</td>
<td>collect</td>
</tr>
<tr>
<td>sample</td>
<td>count</td>
</tr>
<tr>
<td>union</td>
<td>save</td>
</tr>
<tr>
<td>groupByKey</td>
<td>lookupKey</td>
</tr>
<tr>
<td>reduceByKey</td>
<td>...</td>
</tr>
<tr>
<td>join</td>
<td>persist / cache</td>
</tr>
</tbody>
</table>
# RDD Operations

<table>
<thead>
<tr>
<th>Transformations (define a new RDD)</th>
<th>Parallel operations (return a result to driver)</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>reduce</td>
</tr>
<tr>
<td>filter</td>
<td>collect</td>
</tr>
<tr>
<td>sample</td>
<td>count</td>
</tr>
<tr>
<td>union</td>
<td>save</td>
</tr>
<tr>
<td>groupByKey</td>
<td>lookupKey</td>
</tr>
<tr>
<td>reduceByKey</td>
<td>...</td>
</tr>
<tr>
<td>join</td>
<td>...</td>
</tr>
<tr>
<td>persist/cache</td>
<td>Where</td>
</tr>
<tr>
<td></td>
<td>Select</td>
</tr>
<tr>
<td></td>
<td>GroupBy</td>
</tr>
<tr>
<td></td>
<td>OrderBy</td>
</tr>
<tr>
<td></td>
<td>Aggregate</td>
</tr>
<tr>
<td></td>
<td>Join</td>
</tr>
<tr>
<td></td>
<td>Apply</td>
</tr>
<tr>
<td></td>
<td>Materialize</td>
</tr>
</tbody>
</table>

...
RDD Fault Tolerance

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

• Ex:
  ```scala
cachedMsgs = textFile(...).filter(_.contains("error"))
    .map(_.split('t')(2))
    .persist()
  ```
## RDDs vs Distributed Shared Memory

<table>
<thead>
<tr>
<th>Concern</th>
<th>RDDs</th>
<th>Distr. Shared Mem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reads</td>
<td>Fine-grained</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Writes</td>
<td>Bulk transformations</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Consistency</td>
<td>Trivial (immutable)</td>
<td>Up to app / runtime</td>
</tr>
<tr>
<td>Fault recovery</td>
<td>Fine-grained and low-overhead using lineage</td>
<td>Requires checkpoints and program rollback</td>
</tr>
<tr>
<td>Straggler mitigation</td>
<td>Possible using speculative execution</td>
<td>Difficult</td>
</tr>
<tr>
<td>Work placement</td>
<td>Automatic based on data locality</td>
<td>Up to app (but runtime aims for transparency)</td>
</tr>
</tbody>
</table>