


DePaul University

SE480 Week 5 – Performance, Scalability

Steven Engelhardt

Add WeChat powcoder

DePaul University

Autumn 2020

Table of Contents

Table of Contents

Summary of Last Week

- *Testability* is about designing systems that give up their faults easily
- *Interoperability* is about how two or more systems usefully exchange meaningful information, which requires both syntactic and semantic interoperability
- *Performance* is about characterizing the events that can occur (and when they can occur) and the system or element's time-based response to these events.

Quiz 2 and HW 2 Review

- Quiz 2 and Homework 2 review will be performed next week

Table of Contents

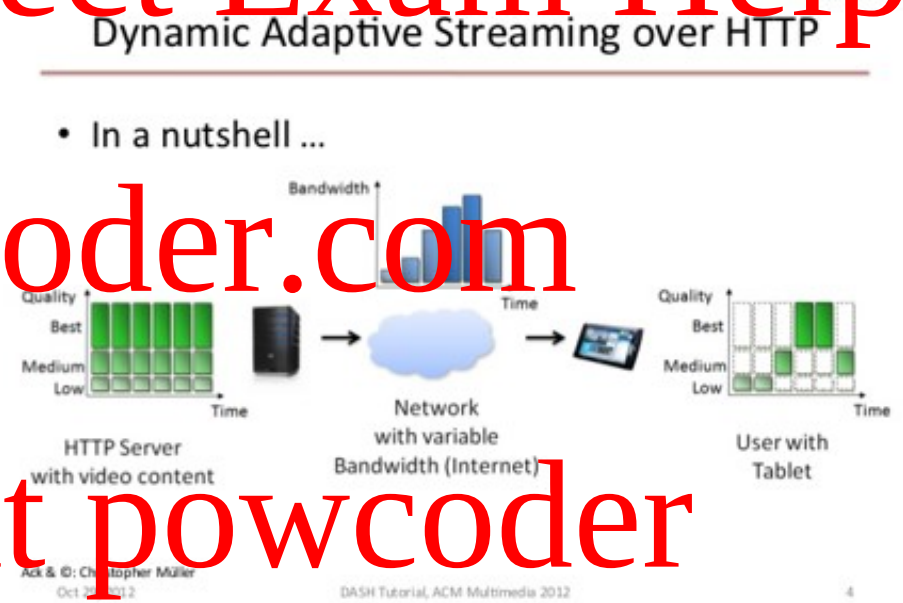
Table of Contents

Performance Tactic Categories

- *Control Resource Demand* – carefully manage the demand for resources and that the resources you have are applied judiciously
- *Manage Resources* – make resources at hand work more effectively in handling the demands put to them

Manage Sampling Rate

- *Reducing the sampling frequency* at which a stream of environmental data is captured, with a corresponding loss of fidelity
- Analogous to YouTube's adaptive bitrate streaming



Limit Event Response

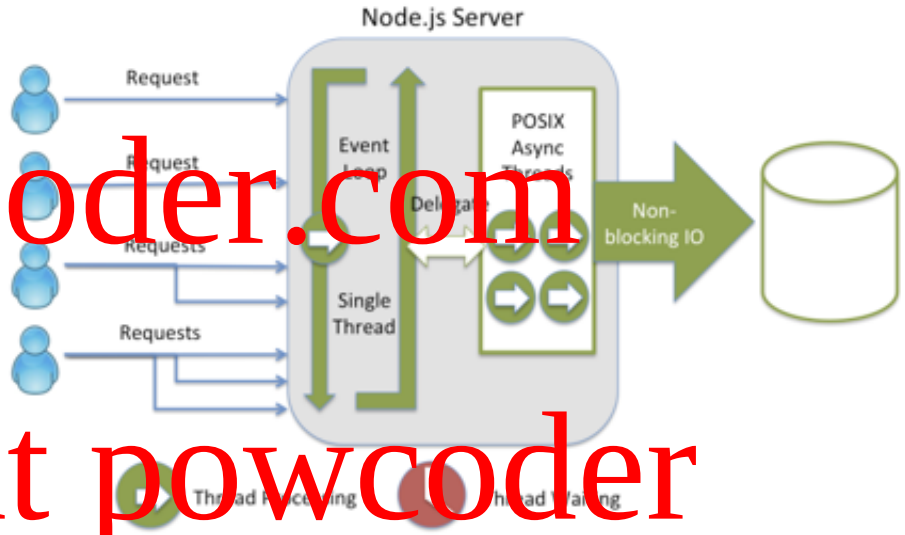
- Only process events up to a set maximum rate, ensuring more predictable processing when the events are actually processed, and queue events that you cannot process in time
- Humans often prefer slightly-higher average response time with low jitter over slightly-lower average response with high jitter
- What do you do if your queue is insufficient to handle the worst case? Drop old events? Ignore new events? (*Backpressure* is covered in *Manage Resources*)

Prioritize Events

- Rank events according to how important it is to service them
- Process events in priority order
- Consider ignoring low-priority events
- Related to scheduling (later)

Reduce Overhead

- *Remove intermediaries* to improve latency (when we talk about modifiability, we will talk about *adding intermediaries*)
- *Co-locate resources* (next slide)
- Perform *periodic cleanup* of resources that have become inefficient
- Use *single-threaded event-loop processor design* (like Node.JS) to avoid contention



Co-Locate Resources

- Avoid time delay of communication by putting processing & data in the same place
- Big Data maxim: *move computation to data, not data to computation*
  - Fundamental to Hadoop, but applies in many circumstances

- My most commonly-used performance improvement "trick"!
  - The vast majority of performance problems I've experienced have been I/O bound, not CPU bound

- In the cloud, networking has become so advanced that this adage may no longer apply

Co-Locate Resources Example 1

- SQL database with a table of 10,000,000 records, where each record is 1K, being accessed by a cluster of web servers.
- Objective: Implement a table that points to this table and supports sorting, filtering, and pagination
- *Question*: Should we perform the sorting, filtering, and pagination in the database, the web server, or the client side (i.e., in the web browser)? What are the tradeoffs involved?
- Useful figures:
  - Average connection speed in the United States as of Q3 2015: 12.6 megabits/second (1.6MB/sec)
  - AWS m4.large: 2CPU, 8GB RAM, $0.10/hr
  - AWS x1.32xlarge: 128CPU, 1,952TB RAM, $13.338/hr
  - Network: 10 gigabits/second (1.25 GB/sec)
  - SSD read throughput: 550MB/sec per drive
  - SATA 3.2 interface limit: 16 gigabits/second (2 GB/sec)
  - DDR2-800 RAM bandwidth: 12.8GB/sec

Co-Locate Resources Example 2

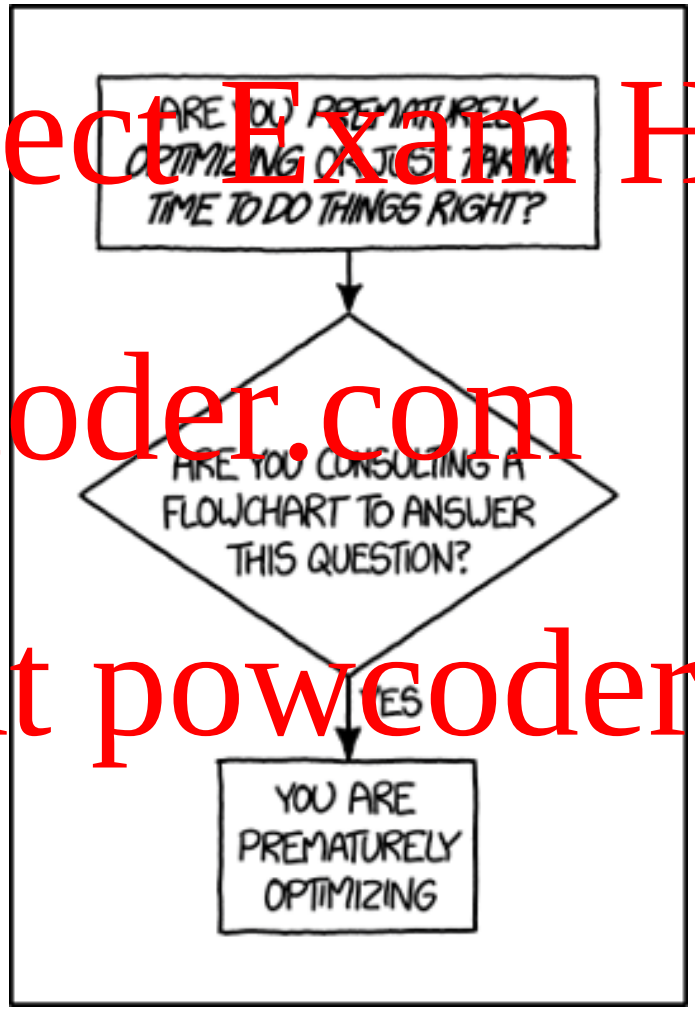
- SQL database with a table of 10,000,000 records, where each record is 1K, being accessed by a cluster of web servers.
- Objective: Calculate statistics (e.g. average, mean, median) for the numerical columns in the table
- *Question*: Should we calculate these statistics in the database, the web server, or the client side (i.e. in the web browser)? What are the tradeoffs involved?
- Useful figures:
  - Average connection speed in the United States as of Q3 2015: 12.6 megabits/second (1.6MB/sec)
  - AWS m4.large: 2CPU, 8GB RAM, $0.10/hr
  - AWS x1.32xlarge: 128CPU, 1,952TB RAM, $13.338/hr
  - Network: 10 gigabits/second (1.25 GB/sec)
  - SSD read throughput: 550MB/sec per drive
  - SATA 3.2 interface limit: 16 gigabits/second (2 GB/sec)
  - DDR2-800 RAM bandwidth: 12.8GB/sec

Bound Execution Times

- Place a limit on how much execution time is used to respond to an event
- Most useful for iterative, data-dependent algorithms, where you can tradeoff between time & accuracy, otherwise this is more of an availability tactic
  - Monte Carlo # of simulations

Increase Resource Efficiency

- Improve the algorithms used in critical areas
- Donald Knuth: "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%" [Knu74]
- *Always* use tools (e.g. a *profiler*) and a data-driven approach to find that 3%
  - Many performance problems are due to non-obvious causes
  - Your intuition, and mine, for performance hotspots is terrible



Increase Resource Efficiency Non-Obvious Example

```
#include ...

void do_work()
{
    // Generate data
    const unsigned arraySize = 32768;
    int data[arraySize];
    for (unsigned c = 0; c < arraySize; ++c)
        data[c] = std::rand() % 256;

    // !!! With this, the next loop runs 6 times faster !!!
    std::sort(data, data + arraySize);

    // Test
    clock_t start = clock();
    long long sum = 0;
    for (unsigned i = 0; i < 100000; ++i)
    {
        // Primary loop
        for (unsigned c = 0; c < arraySize; ++c)
        {
            if (data[c] >= 128)
                sum += data[c];
        }
    }

    double elapsedTime =
      static_cast<double>(clock() - start) / CLOCKS_PER_SEC;
    std::cout << elapsedTime << std::endl;
    std::cout << "sum = " << sum << std::endl;
}
```

Increase Resource Efficiency Non-Obvious Example

```
#include ...

void do_work()
{
    // Generate data
    const unsigned arraySize = 32768;
    int data[arraySize];
    for (unsigned c = 0; c < arraySize; ++c)
        data[c] = std::rand() % 256;

    // !!! With this, the next loop runs 6 times faster !!!
    std::sort(data, data + arraySize);

    // Test
    clock_t start = clock();
    long long sum = 0;
    for (unsigned i = 0; i < 100000; ++i)
    {
        // Primary loop
        for (unsigned c = 0; c < arraySize; ++c)
        {
            if (data[c] >= 128)
                sum += data[c];
        }
    }

    double elapsedTime =
      static_cast<double>(clock() - start) / CLOCKS_PER_SEC;
    std::cout << elapsedTime << std::endl;
    std::cout << "sum = " << sum << std::endl;
}
```


Increase Resources

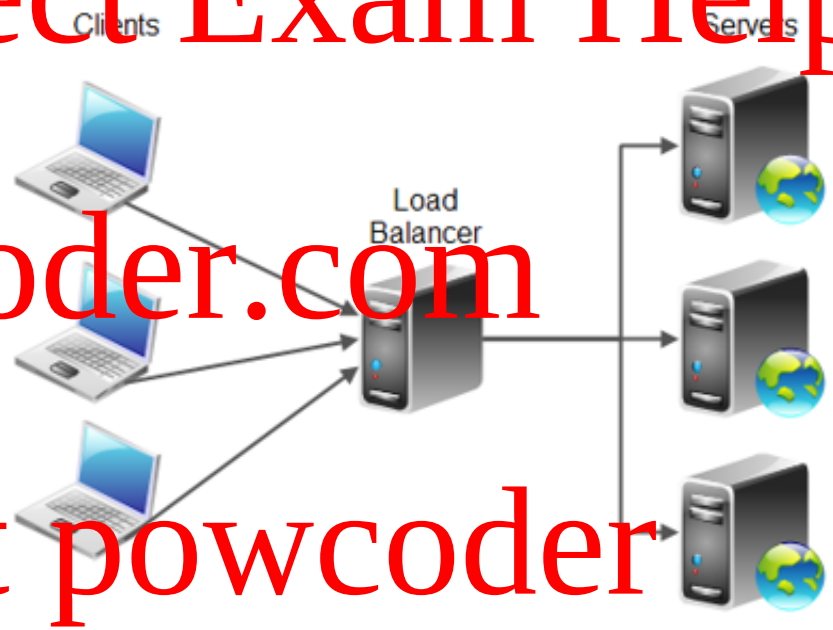
- Buy your way out of the problem!
- Use faster processors, additional processors, additional memory, etc.
- In many cases, the *cheapest and fastest way* to make an improvement

	SSD	HDD
Capacity	1TB	1TB
IOPS (4K read)	9,488	45
Throughput (sequential read)	500MB/s	143MB/s
Seek time	0.0342ms	8.88ms
Cost (2017)	$179.99	$64.99

- Process more requests in parallel to reduce blocking time
- Use *scheduling policies* to achieve goals you find desirable (see later)

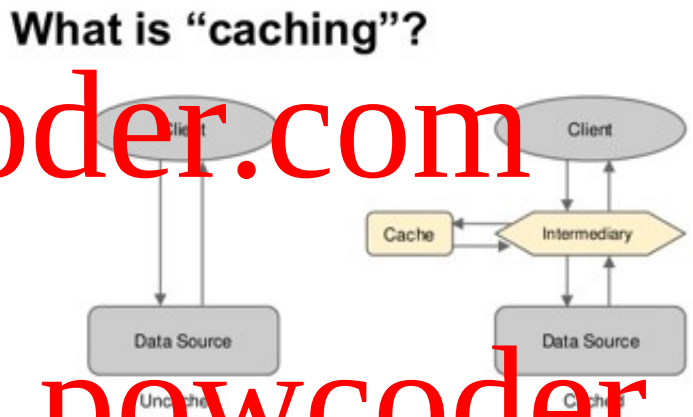
Maintain Multiple Copies of Computations

- Introduce replicas to reduce the contention that would occur if all computations took place on a single server
- Often paired with a *load balancer* to assign new work to one of the available duplicate servers



Maintain Multiple Copies of Data

- *Caching* is about keeping copies of data (possibly one a subset of the other) on storage with different access speeds (more on next slide)
- *Data replication* involves keeping separate copies of the data to reduce the contention from multiple simultaneous accesses
- Don't underestimate the challenge required to keep data copies consistent and synchronized



Caching Design Considerations

- How many? What sizes?
- Distributed? Shared?
- Replacement policy
  - FIFO, LIFO, LRU, etc.
- Expiration policy
  - Absolute, sliding, etc.
- Write policy
  - *Write-through caching* directs write I/O onto cache and through to underlying permanent storage before confirming I/O completion to the host.
  - *Write-back caching* directs write I/O cache and completion is immediately confirmed to the host. The cache asynchronously copies the data to the permanent storage at a later time.
  - *Write-around caching* directs write I/O directly to permanent storage, bypassing the cache.

Bound Queue Sizes

- Limit the number of queued arrivals and consequently the resources used to process the arrivals
- Can you prevent clients from enqueueing new data (i.e. implement *backpressure*)? If not, how will you handle clients exceeding your queue limits?
  - Remember Limit Event Response?

Schedule Resources

- When there is contention for a resource, the resource must be *scheduled*.
- To schedule you must choose a *scheduling policy*.

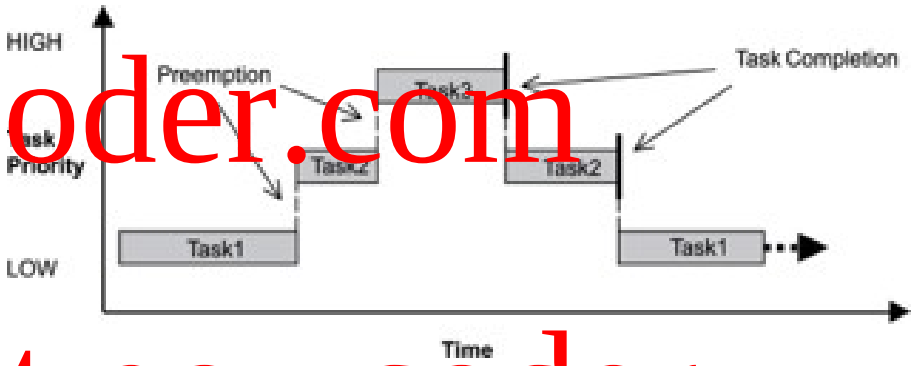
Table of Contents

Scheduling Overview

- A *scheduling policy* is the strategy by which a *scheduler* decides which work should be executed next
- A scheduling policy conceptually has two parts:
  - Priority assignment
  - Dispatching
- *Competing criteria* for scheduling
  - Optimal resource usage
  - Request importance
  - Minimizing the number of resources used
  - Minimizing latency
  - Maximizing throughput
  - Preventing starvation to ensure fairness

Scheduling Concepts

- *Preemption* – can a task be temporarily interrupted with the intention of resuming the task at a later time?
- *Priority inversion* – a high priority task is indirectly preempted by a lower priority task
- *Priority boosting* – can lower priority tasks have their temporarily priority boosted to avoid deadlock?



Priority Inversion Example

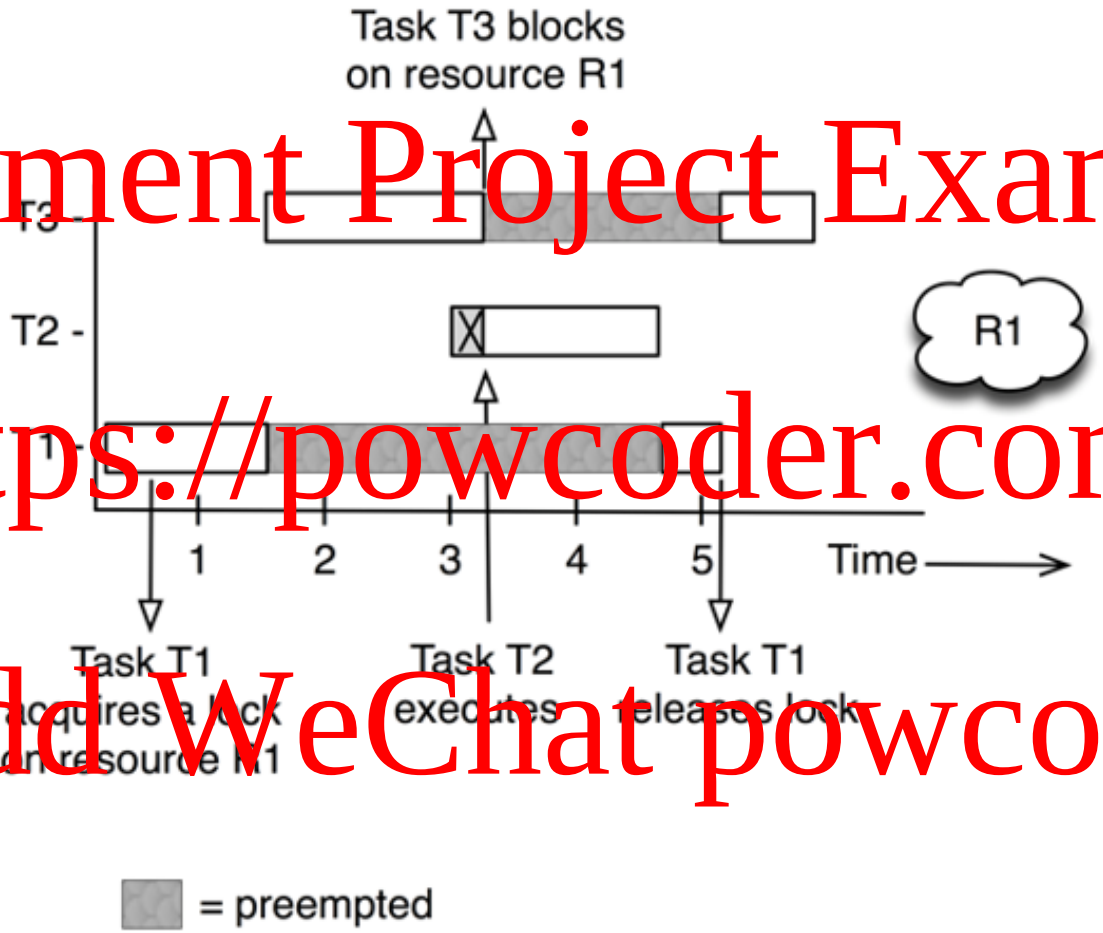


Figure: Priority Inversion

Common Scheduling Policies

- *First-in first-out (FIFO)* – process jobs in the order that they arrive in the ready queue
- *Fixed-priority scheduling* – assign each source of resource requests a particular priority and assign the resources in that priority order. Prioritization strategies include:
  - *Semantic importance* – Assign priority statically according to some domain characteristic of the task
  - *Deadline monotonic* – Assign higher priority to streams with shorter deadlines
  - *Rate monotonic* – Assign higher priority to streams with shorter periods
- *Dynamic priority scheduling*
  - *Round-robin* – Assign jobs to resources in a rotating fashion. *Cyclic executive* – Assign a fixed time unit per process, and cycle through them (requires preemption)
  - *Earliest-deadline-first* – Assign priorities based on the pending requests with the earliest deadline (optimal)
  - *Least-slack-first* – Assign the highest priority to the job with the least *slack time*, which is the difference between the execution time remaining and the time to the job's deadline (optimal)
- *Static scheduling* – the preemption points and sequence of assignment to the resource are determined offline (thus there is no runtime overhead for scheduling)

Table of Contents

Other Performance Topics

- Be data-driven. Religiously collect and analyze performance metrics and use these to drive your decisions.

Other Performance Topics

- Be data-driven. Religiously collect and analyze performance metrics and use these to drive your decisions.
- Understand and work with (not against) your CPU's branch predictor and memory prefetcher

Other Performance Topics

- Be data-driven. Religiously collect and analyze performance metrics and use these to drive your decisions.
- Understand and work with (not against) your CPU's branch predictor and memory prefetcher
  - Linear search is faster than binary search for small and medium sized arrays [Ano17]

Other Performance Topics

- Be data-driven. Religiously collect and analyze performance metrics and use these to drive your decisions.
- Understand and work with (not against) your CPU's branch predictor and memory prefetcher
  - Linear search is faster than binary search for small and medium sized arrays [Ano17]
  - Contiguous data structures (e.g. vectors) are often faster than non-contiguous ones (e.g. lists) regardless of what the theoretical big-O differences are.

Other Performance Topics

- Be data-driven. Religiously collect and analyze performance metrics and use these to drive your decisions.
- Understand and work with (not against) your CPU's branch predictor and memory prefetcher
  - Linear search is faster than binary search for small and medium sized arrays [Ano17]
  - Contiguous data structures (e.g. vectors) are often faster than non-contiguous ones (e.g. lists) regardless of what the theoretical big-O differences are.
- Use specialized compute resources when available. GPUs are far faster than CPUs on many different computational tasks.

Other Performance Topics

- Be data-driven. Religiously collect and analyze performance metrics and use these to drive your decisions.
- Understand and work with (not against) your CPU's branch predictor and memory prefetcher
  - Linear search is faster than binary search for small and medium sized arrays [Ano17]
  - Contiguous data structures (e.g. vectors) are often faster than non-contiguous ones (e.g. lists) regardless of what the theoretical big-O differences are.
- Use specialized compute resources when available. GPUs are far faster than CPUs on many different computational tasks.
- Use tools like profilers. Consider embracing *causal profilers* like Coz.

Other Performance Topics

- Be data-driven. Religiously collect and analyze performance metrics and use these to drive your decisions.
- Understand and work with (not against) your CPU's branch predictor and memory prefetcher
  - Linear search is faster than binary search for small and medium sized arrays [Ano17]
  - Contiguous data structures (e.g. vectors) are often faster than non-contiguous ones (e.g. lists) regardless of what the theoretical big-O differences are.
- Use specialized compute resources when available. GPUs are far faster than CPUs on many different computational tasks.
- Use tools like profilers. Consider embracing *causal profilers* like Coz.
- Computer systems have a lot of noise which can make measuring the impact of individual performance improvements very difficult. A strong background in statistics is extremely helpful.

Optional Readings

- Don't Compare Averages
- How Shopify Reduced Storefront Response Times with a Rewrite
- Asynchronous computing @Facebook: Driving efficiency and developer productivity at Facebook scale
- Impact of Intel vs. ARM CPU Performance for Object Storage
- How we reduced latency and cost-to-serve by merging two systems
- Why are services slow sometimes?
- Introduction to Profiling and Optimizing SQL Queries for Software Engineers
- Understanding CPU Microarchitecture for Performance
- Reflections on software performance
- Memory Bandwidth Napkin Math
- Why Kafka Is so Fast

Table of Contents

Table of Contents

What is Scalability?

- *Performance* is related to the resource usage associated with a single unit of work.
- *Scalability* is about how resource usage changes as units of work grow in number or size.
- Ideally, resource should increase *linearly* (or better) with *load*, and without bound. (Load is measured by user traffic or data volume etc.).

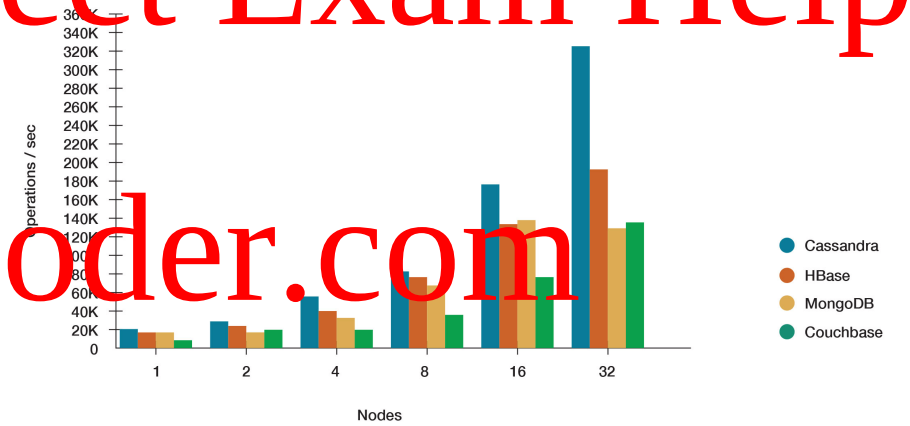


Figure: https://www.datastax.com/nosql-databases/benchmarks-cassandra-vs-mongodb-vs-hbase

Why is Scalability Important?

- A scalable system means that it is prepared to handle an increasing number of customers, clients, and/or users, largely via adding resources
- If your system cannot scale to meet your *peak load*, you will have availability issues, which we know can cost an incredible amount of money

Types of Scalability

- *Horizontal scalability* (*scaling out*) – add more resources to logical units, such as adding another server to a cluster of servers
- *Vertical scalability* (*scaling up*) – add more resources to a physical unit, such as adding more memory to a single computer
- *Elasticity* – a form of horizontal scalability with the ability to dynamically add or remove resources in an autonomic manner
- Horizontal scalability is more desirable than vertical scalability, but generally harder to design & implement

Table of Contents

Partition by Function

- Put related pieces of functionality together and separate unrelated pieces
- Example: eBay
  - Segments different functions into separate application pools (e.g. selling functionality, bidding functionality, search)
  - 16,000 application servers organized into 220 different pools, and each pool can be scaled independently
  - 1,000 logical databases across 400 physical hosts, separated into sets by function (e.g. user data, item data, purchase data), allowing independent scaling

Figure: http://www.infoq.com/interviews/shoup-ebay-architecture

Split Horizontally

- Within a functional area, break the workload into manageable units.
- eBay:
  - At the application tier (where eBay's interactions are stateless), splitting horizontally is trivial.
  - As all application servers are equal and none retain transactional state, any one will do.
  - If more processing power is needed, simply add more application servers.
  - At the database tier (which is stateful by definition) split/shard the data horizontally along its primary access path.
  - User data divided over 20 hosts (each host containing 1/20 of the user)
  - As user numbers grow and/or data per users grows, then more hosts are added and data is further subdivided.

Data Management

- The basis for much of scalability is the principle of divide-and-conquer. *If you can't split it, you can't scale it.*
- The vast majority of your scalability challenges will be related to *data management*
- Data management approaches
  - *Replication* – create additional copies of your data so you can read from more locations (useful for scaling reads)
  - *Sharding (split horizontally)* – partition your data into multiple, disjoint sets, each stored on a separate server instance (useful for scaling writes)

Sharding Strategies

- Simple *modulo* or *hash* of the key (i.e. item IDs or hashes ending in 1 go to one host, those ending in 2 go to another host, etc.)
- Range of IDs (0-1M, 1-2M, etc)
- *Lookup table* (e.g. Hadoop's master node)
- *Consistent hashing* – a special kind of hashing such that when a hash table is resized, only K/n keys need to be remapped on average, where K is the number of keys, and n is the number of slots
- Regardless of strategy, an infrastructure which supports partitioning *and repartitioning* of data will be more scalable than one which does not.

Distributed Transactions Problems

- The orthodox approach for handling transactional guarantees is to create a distributed transaction across various resources, using two-phase commit.
- Pessimistic approach has substantial costs
- Scaling, performance and latency are all impacted by the *costs of coordination*
- Costs worsen *geometrically* as the number of dependent resources and incoming clients increases.

Distributed Transactions Solutions

- Relax transactional guarantees across unrelated systems
- Techniques you can use to help the system reach eventual consistency:
  - Careful ordering of database operations
  - Asynchronous recovery events
  - Reconciliation or settlement batches
- Consistency is not an all or nothing proposition
- Most real-world use cases don't require immediate consistency.

Virtualize at All Levels

- You can use the concept of *virtualization* within your software architecture!
- Judicious use of virtualization makes higher levels of infrastructure unaware of changes to components, machines, and partitions.
- eBay
  - Virtualize the database so that applications interact with a logical representation of the database which is then mapped onto a particular physical machine.
  - Applications abstracted from split routing logic (which assigns a particular record i.e. "User XYZ" to a particular partition)
  - Allows operations team to rebalance logical hosts between physical hosts by separating them, consolidating them, or moving them.

Table of Contents

Overview

- As a general rule, the textbook's treatment on concurrency and parallelism is terrible, and its treatment on asynchrony is basically nonexistent
- Understanding and exploiting concurrency, parallelism, and asynchrony is key to making your application performant and scalable
- Modern systems must be designed to be concurrent at all levels (i.e. intraprocess and interprocess)

The Free Lunch Has Been Over Since 2005

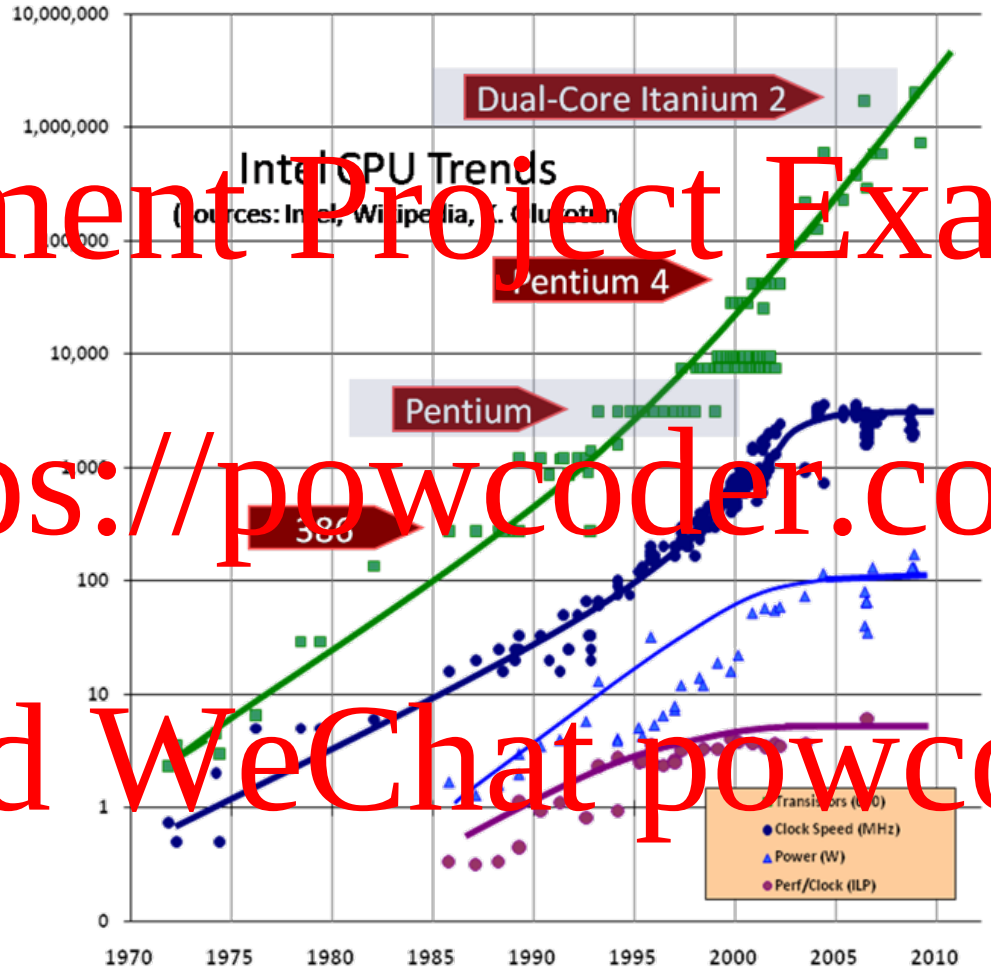


Figure: Required Reading: Sutter, Herb. "The Free Lunch is Over – A Fundamental Turn Toward Concurrency in Software" *Dr. Dobb's Journal*, 30(3), Mar 2005

Concurrency vs. Parallelism

- *Concurrency* is when two or more tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant. For example, multitasking on a single-core machine.
- *Parallelism* is when tasks literally run at the same time, e.g., on a multicore processor.
- If I bring my laptop to the DMV and work on this week's lecture while waiting in line to renew my driver's license, I am working on the two tasks *concurrently*. However, I cannot perform both in *parallel* in that I cannot work on the lecture *while* speaking with the DMV employee to renew my driver's license; I am only able to work on the lecture while waiting in line.

Concurrent Programming Models

- The traditional, single-threaded programming model does not express allowable concurrency. This limits the scalability of programs written in this paradigm
- Many alternative programming models exist which help express concurrency and/or parallelism. They include:
  - Traditional threads and locks
  - Staged Event-Driven Architecture (SEDA)
  - Data parallelism (OpenMP, CUDA, PLINQ)
  - Actor models (Akka, Erlang, Orleans, Scala)
  - Task parallelism (Java's Executor framework, C#'s `Task`)
  - Futures and promises (C++, Java, etc.)
  - Go's goroutines
- We will explore a few of these programming models
- If you want your program to use the full power of your computer, you must use concurrent programming techniques

Threads and Locks

- A *thread* is a concurrent sequence of programming that is managed independently by a scheduler, which is typically part of the operating system
- Threads have their own *thread context*, which is their privately-owned resources (e.g. their own stack, registers, thread-local storage, etc.)
- Within each thread, execution proceeds sequentially, but multiple threads may have their execution interleaved in unpredictable ways

Threads and Locks: Example

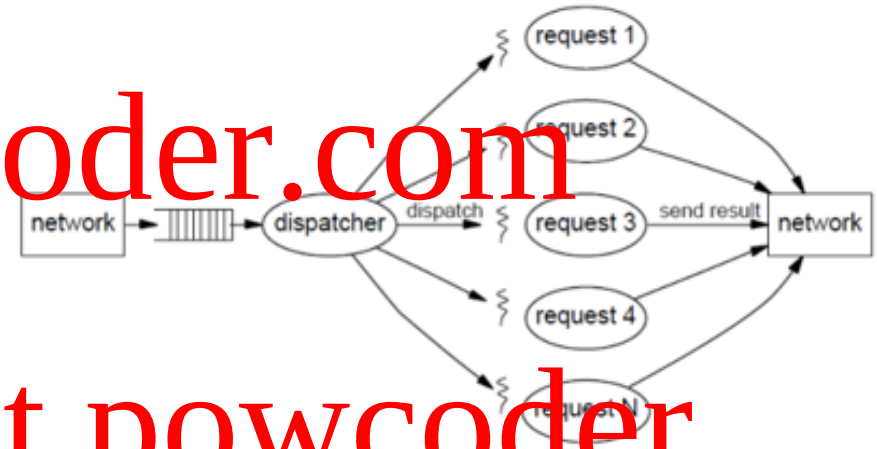
```
public class MyClass implements Runnable {
    public void run() {
        while (true) {
            System.out.println("MyClass running");
        }
    }
}

public class Program {
    public static void main(String args[]) {
        Thread t1 = new Thread(new MyClass());
        Thread t2 = new Thread(new MyClass());
        t1.start();
        t2.start();
        // Do more work
    }
}
```

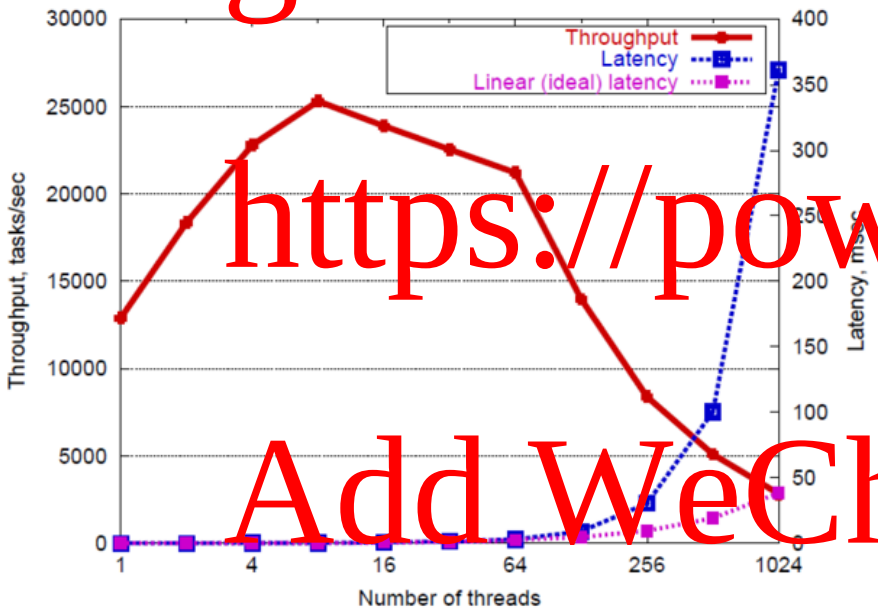
Listing 1: Simple Java threading example

Threads and Locks: Thread Per Request Model

- Traditional *threaded server design*. Each incoming request is dispatched to a separate operating system thread, which processes the request and returns a result to the client.
- Edges represent control flow between components. Note that other I/O operations, such as disk access, are not shown here, but would be incorporated into each threads' request processing.



Threads and Locks: Thread Per Request Scalability



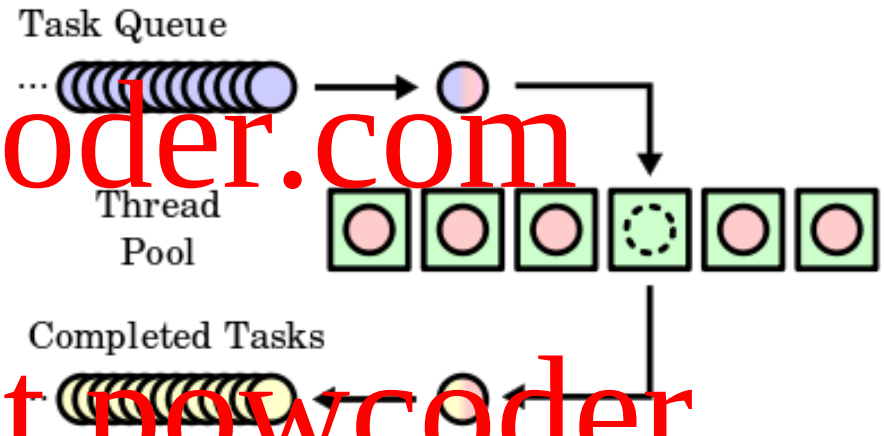
- The *cost of switching threads* can be nontrivial
- To avoid the overuse of threads, a number of systems adopt a coarse form of *load conditioning* that serves to bound the size of the thread pool associated with a service.
- When the number of requests in the server exceeds some fixed limit, additional connections are not accepted.
- However, there is often no differentiation between "expensive" and "cheap" requests – even though shedding a couple of expensive requests might be very effective.

Threads and Locks: Summary

- Often the fundamental way programming languages express *parallelism* – ultimately, in order to use multiple cores, *something* is going to create threads
- Writing correct code is exceedingly difficult
- Replicating errors and debugging is challenging
- Managing concurrency among threads is difficult
- No data isolation among threads. Avoiding data races is very tricky.
- I do not recommend using this model unless you are doing extremely low-level programming

Thread Pooling

- A *thread pool* is a software design pattern for achieving concurrency
- The thread pool maintains multiple threads waiting for tasks to be allocated for concurrent execution by the supervising program
- Provides a form of *managed concurrency* where the number of threads is designed to balance between efficiency and avoiding excessive thread switching overhead



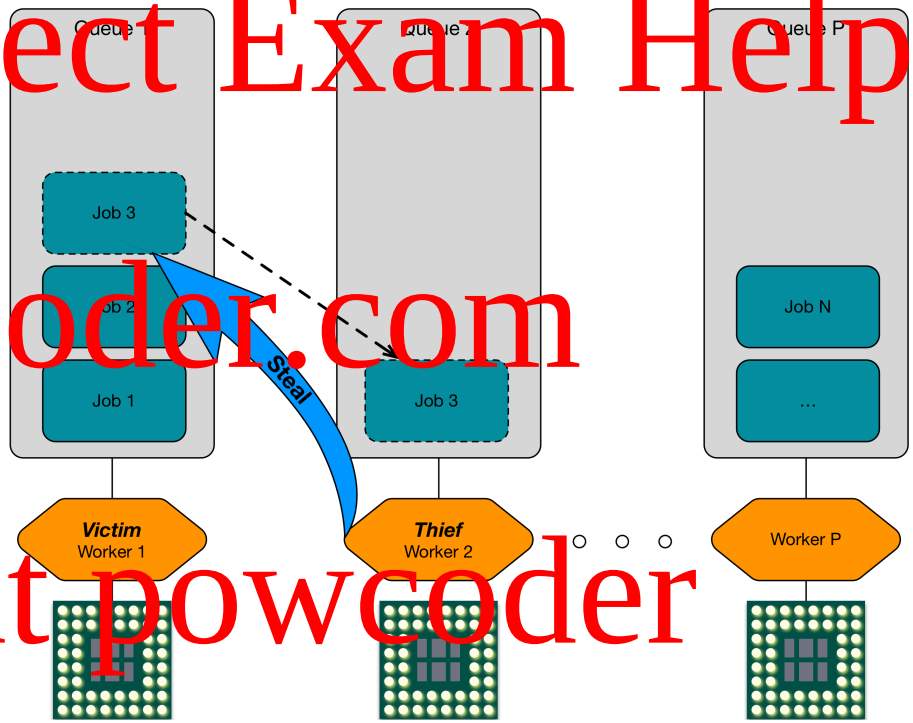
Thread Pooling: Example

```
using System;
using System.Threading;
namespace ThreadPoolApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 0; i < 10; i++)
            {
                ThreadPool.QueueUserWorkItem(new WaitCallback(MyMethod));
            }
            Console.Read();
        }
        public static void MyMethod(object obj)
        {
            Thread thread = Thread.CurrentThread;
            string message = $"Background: {thread.IsBackground}"
              + $", Thread Pool: {thread.IsThreadPoolThread}"
              + $", Thread ID: {thread.ManagedThreadId}";
            Console.WriteLine(message);
        }
    }
}
```


Listing 2: Simple C# ThreadPool example

Thread Pooling: Work Stealing

- Some advanced thread pools implement a queue-per-thread with *work stealing* model
- This approach can improve cache locality and avoid locks in certain cases

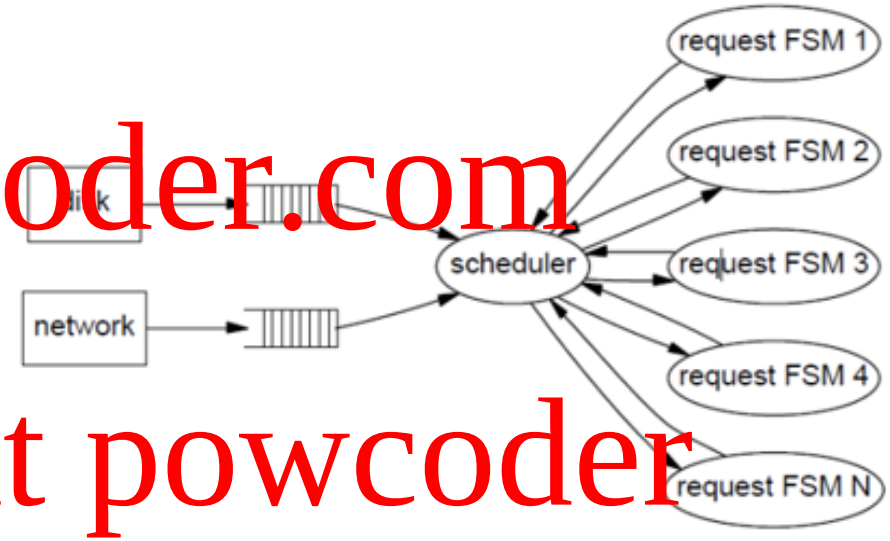


Thread Pooling: Summary

- Thread pool sizing can be dynamic based on number of waiting tasks
- Avoids the degenerate bad performance of thread-per-request
- No control over the state and priority of the thread
- Your thread does not have a stable identity
- Blocking operations (e.g. long-lived I/O) can cause thread pool starvation
- Does not easily model a producer-consumer pattern, where one worker produces data and another worker consumes it

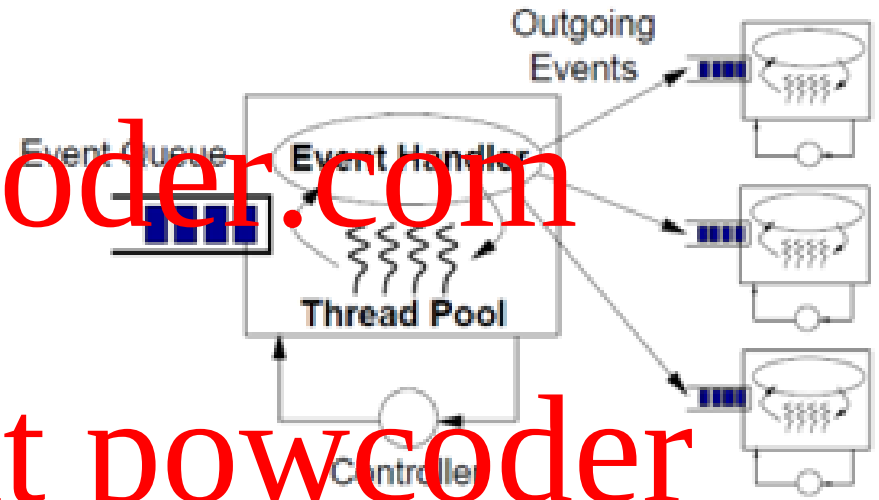
Staged Event-Driven Architecture (SEDA)

- *Staged event-driven architecture (SEDA)* decomposes a complex, event-driven application into a set of stages connected by queues
- The processing of each task is implemented as a *finite state machine*, where transitions between states in the FSM are triggered by events.
- In this way the server maintains its own *continuation state* for each task rather than relying upon a *thread context*.

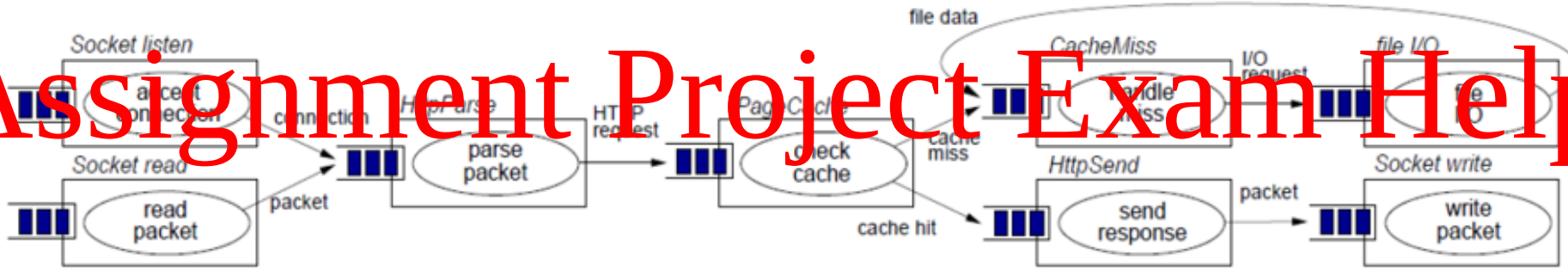


SEDA Stage

- The fundamental unit of processing within SEDA is the *stage*.
- A stage is a self-contained application component consisting of an *event handler*, an *incoming event queue*, and a *thread pool*
- Each stage is managed by a *controller* that affects scheduling and thread allocation.
- Stage threads operate by *pulling a batch of events* off of the incoming event queue and invoking the application-supplied event handler.
- The event handler processes each batch of events, and dispatches zero or more events by *enqueuing them on the event queues of other stages*.



SEDA Application



- The application is composed as a set of stages separated by queues.
- Edges represent the flow of events between stages. Each stage can be independently managed, and stages can be run in sequence or in parallel, or a combination of the two.
- The use of event queues allows each stage to be individually load-conditioned, for example, by thresholding its event queue.

SEDA: Summary

- SEDA Advantages
  - Supports massive concurrency – Requires the application to provide efficient and scalable I/O primitives
  - Simplifies construction of well-conditioned services – Shields developer from details of scheduling and resource management.
  - Enables introspection – Applications able to analyze request stream and adapt behavior according to changing load conditions, e.g. prioritize and filter requests to support degraded service under heavy loads.
  - Supports self-tuning resource management – Dynamic resource allocation according to load, e.g. determine # of threads per stage dynamically.
- However, SEDA comes with a number of disadvantages, such as extensive context switching which leads to instruction & data cache thrashing
- Some SEDA-based software, like Cassandra, are looking into a message queue-based thread per core architecture

Task Parallelism

- *Task parallelism* is a concurrent programming model which allows a programmer to represents work as fine-grained tasks
- Basically identical to futures and promises
- Provides features to allow tasks to communicate & coordinate (e.g. wait for one another, pass data to one another, start new tasks, task cancellation)
- Provides excellent asynchrony primitives (e.g. closures, completable futures, async/await)
- Provides a runtime to efficiently execute tasks which handles scheduling & parallelism automatically

Task Parallelism: Example

```
static void Main(string[] args)
{
    List<Task> tasks = new List<Task>();
    CancellationTokenSource cancelToken = new CancellationTokenSource();
    string result = String.Empty;
    Task t_MethodA = Task.Factory.StartNew(() =>
        {
            result = Services.GetResultsMethodA();
            cancelToken.Token.ThrowIfCancellationRequested();
        }, cancelToken.Token);
    Task t_MethodB = Task.Factory.StartNew(() =>
        {
            result = Services.GetResultsMethodB();
            cancelToken.Token.ThrowIfCancellationRequested();
        }, cancelToken.Token);
    tasks.Add(t_MethodA);
    tasks.Add(t_MethodB);
    Task.WaitAny(tasks.ToArray());
    cancelToken.Cancel();
    Console.WriteLine(result);
    Console.ReadKey();
}
```

Listing 3: Simple C# Task parallelism example

Task Parallelism: Difference from Threads

- Tasks include a rich runtime which handles scheduling concerns
- Unlike (most) threads, tasks are explicitly designed to be extremely cheap to create and use
- Tasks express potential concurrency but do not require it. In certain cases tasks may be executed synchronously on the current thread.
- Includes a rich feature set such as cancellation and task chaining

Async/Await: Backgroud

- Tasks model asynchronous operations, but the way in which they are chained together can be awkward
- This gets especially complicated when you consider things like proper exception handling


```
fetch('coffee.jpg')
    .then(response => {
        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        } else {
            return response.blob();
        }
    })
    .then(myBlob => {
        let objectURL = URL.createObjectURL(myBlob);
        let image = document.createElement('img');
        image.src = objectURL;
        document.body.appendChild(image);
    })
    .catch(e => {
        console.log('There has been a problem with your fetch operation: ' + e.message);
    });
```


Listing 4: Chaining tasks example in JavaScript

Async/Await: C# Example

- Async/await lets you write asynchonous code in a style that's nearly identical to traditional, synchronous code


```
async function myFetch() {
    let response = await fetch('coffee.jpg');

    if (!response.ok) {
        throw new Error(`HTTP error: status: ${response.status}`);
    } else {
        let myBlob = await response.blob();
        let objectURL = URL.createObjectURL(myBlob);
        let image = document.createElement('img');
        image.src = objectURL;
        document.body.appendChild(image);
    }
}
```


Listing 5: JavaScript async/await example

Concurrency Summary

- Concurrent and asynchronous programming is unavoidable if you want your system to scale
- As architect, you will have to choose what concurrency programming model your application uses
- Async/await is a great programming model which allows you to express concurrency in a model that is almost identical to the traditional, synchronous one

Table of Contents

Overview

- This section will talk about design considerations around designing a highly-scalable database architecture
- The content and images are largely from this Red Gate article: https://www.red-gate.com/simple-talk/cloud/cloud-data/designing-highly-scalable-database-architectures/ [Beh10]

Vertical or Horizontal Scaling?

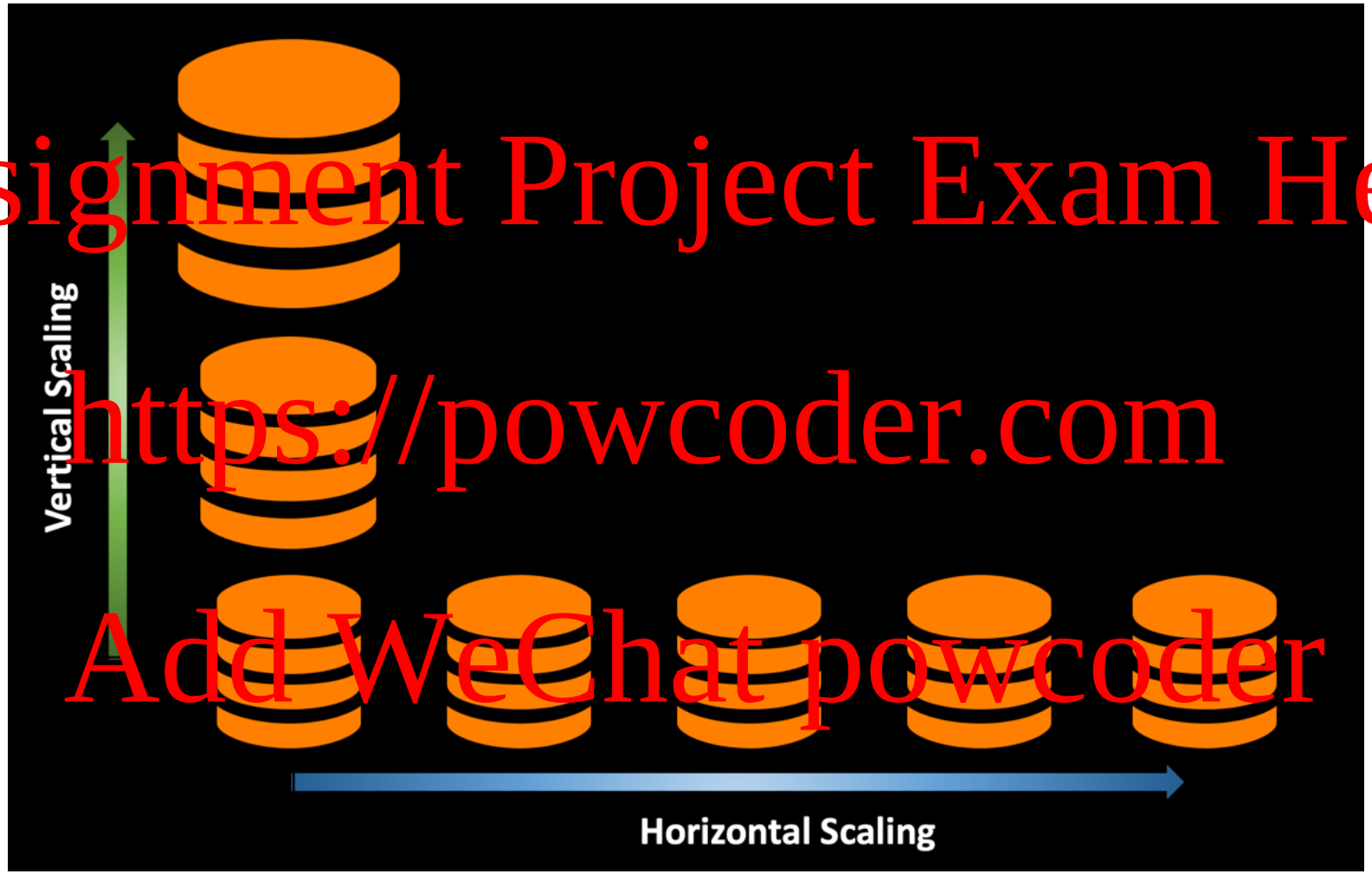


Figure: Vertical vs. Horizontal Scaling

Database Read Replicas

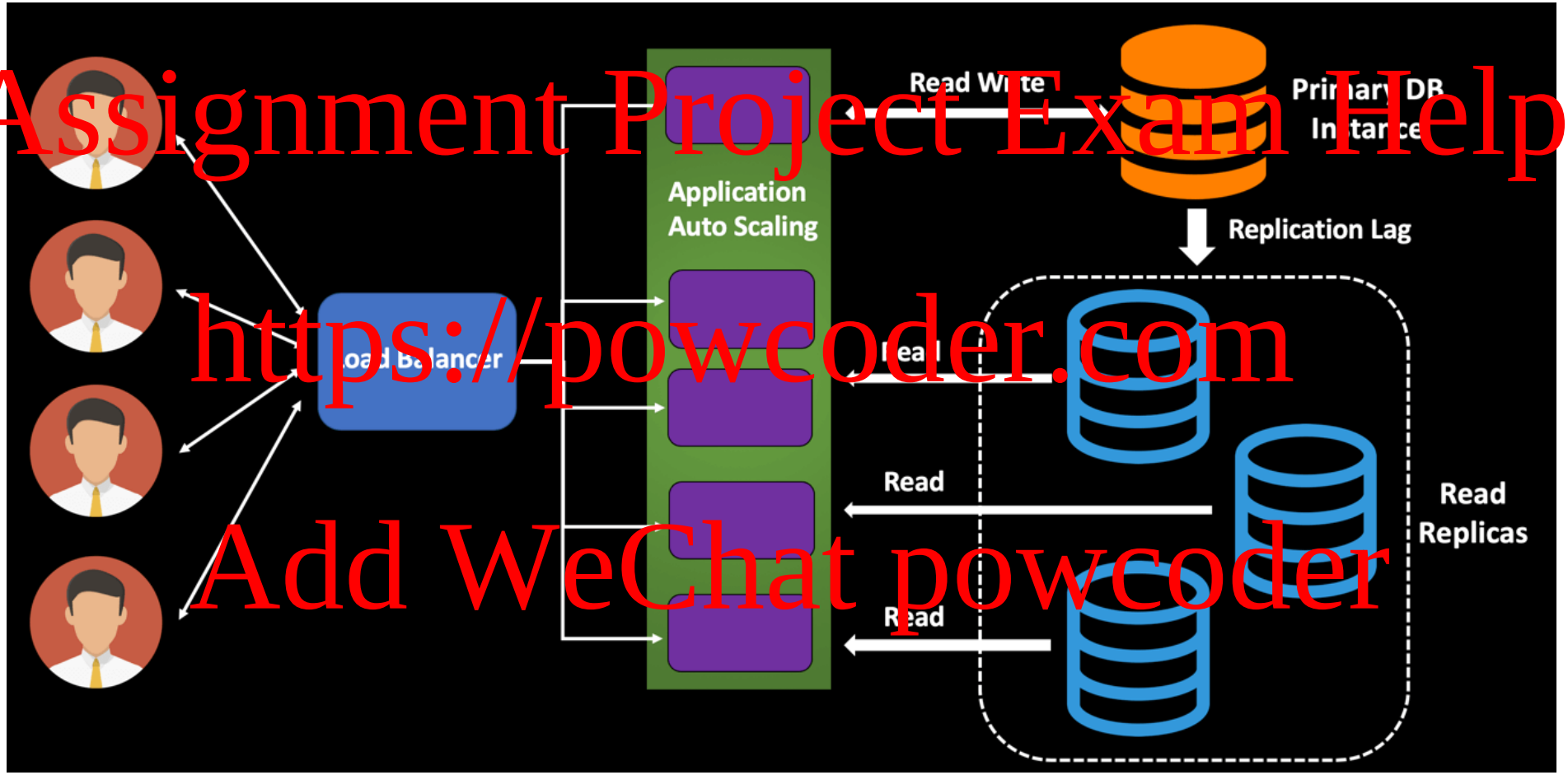


Figure: Database Read Replicas

Database Caching

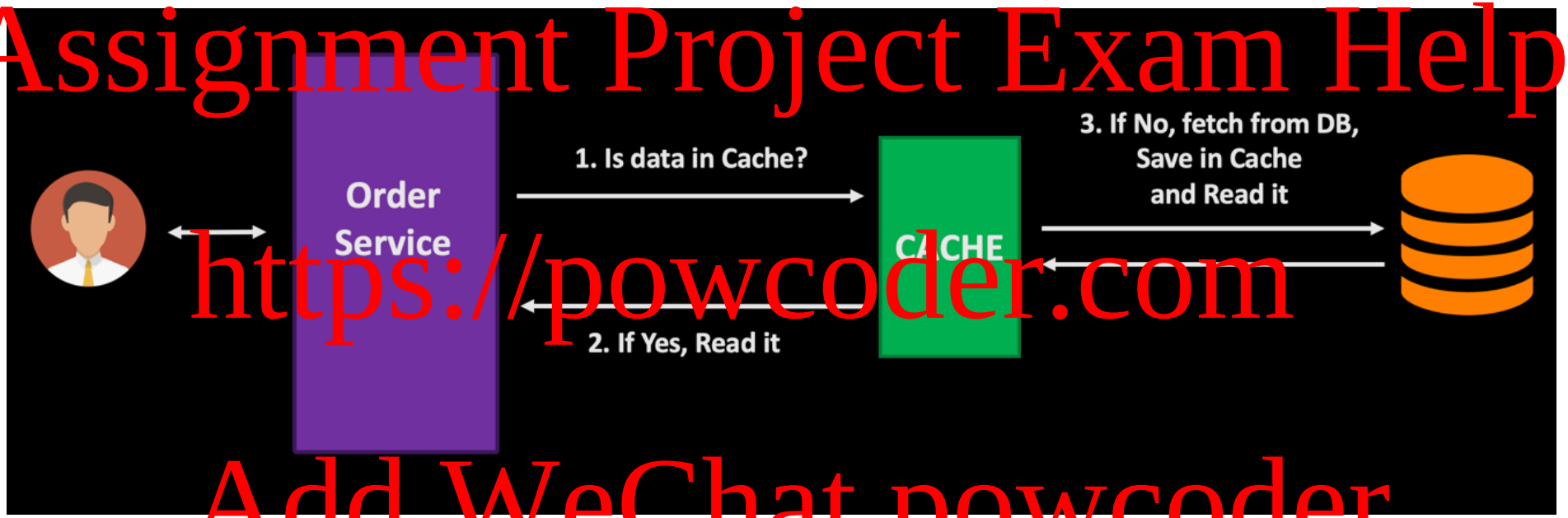


Figure: Database Caching

Database Sharding

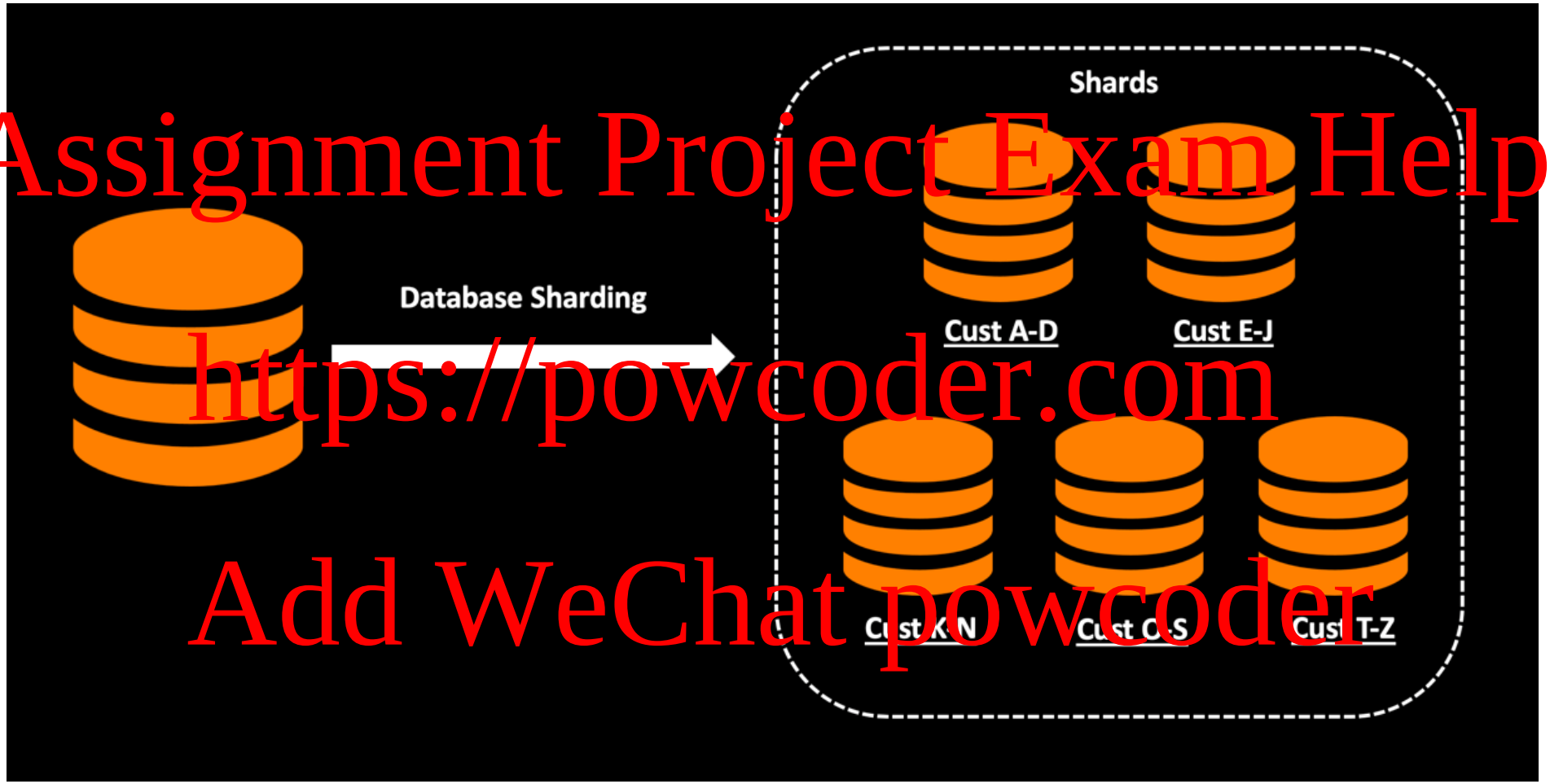


Figure: Database Sharding

Sharding with Hashing

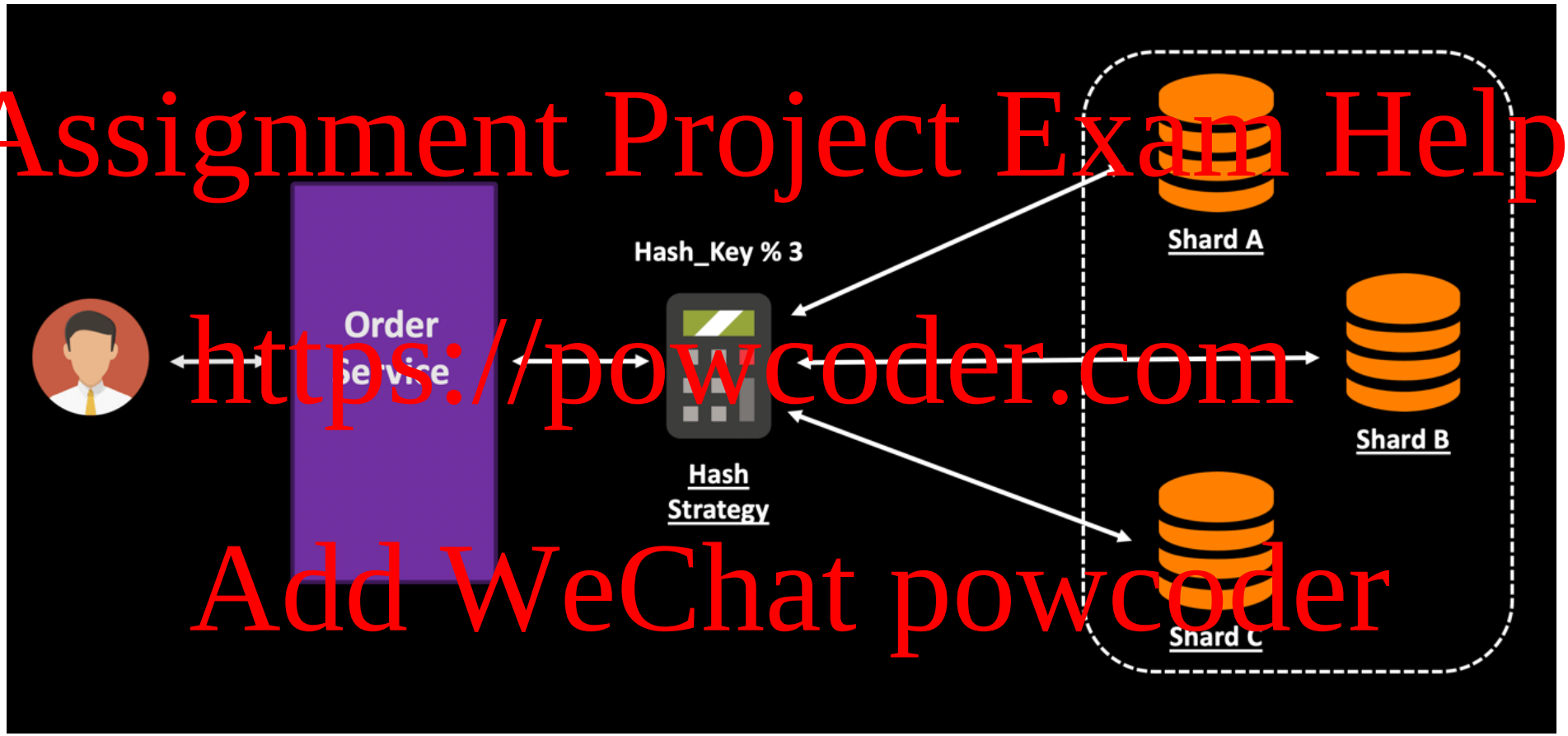


Figure: Sharding With Hashing

Microservices and Databases

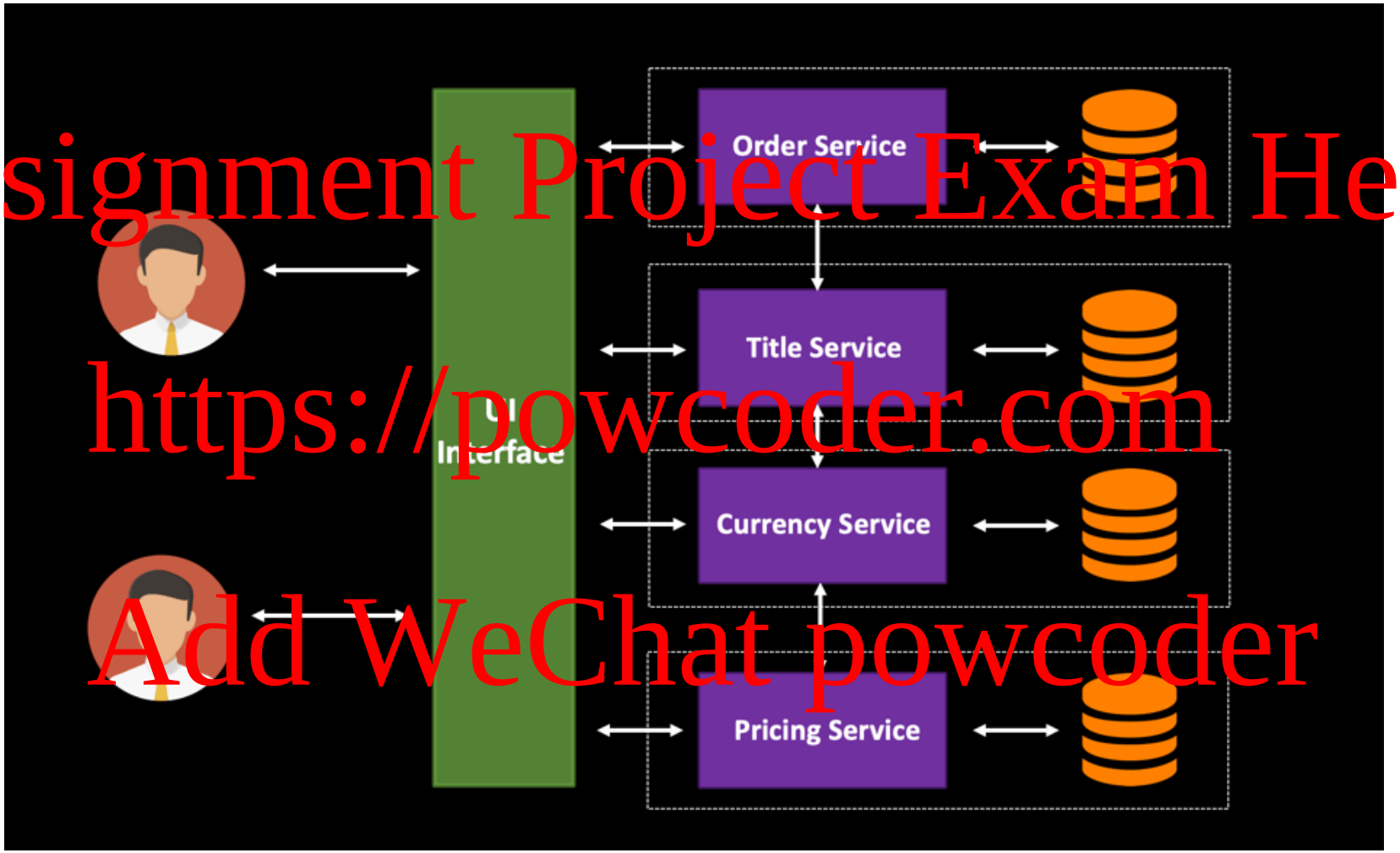


Figure: Microservices and Databases

Optional Readings

- Throughput autoscaling: Dynamic sizing for Facebook.com
- Scaling services with Shard Manager
- Asynchronous computing at Facebook: Driving efficiency and developer productivity at Facebook scale
- Here's How Zoom Provides Industry-Leading Video Capacity
- State of Loom (Java virtual threads)
- Building an elastic query engine on disaggregated storage
- Handling Huge Traffic Spikes with Azure Functions and Cloudflare
- How Amazon is solving big-data challenges with data lakes
- Coroutine Theory
- Here's What Makes Apache Flink scale
- io_uring By Example: An Article Series
- I was wrong. CRDTs are the future

Table of Contents

Wrap-Up

- Read *The Free Lunch is Over – A Fundamental Turn Toward Concurrency in Software*
- Read chapters 7, 10, 11, 12 from SAIP
- Homework 3 is due Thursday, October 22 at 5:30PM
- Quiz 3 will be made available next week. It will be due Thursday, October 22 at 5:30PM

References

[Ano17] Anonymous.
Performance comparison: linear search vs binary search.
Dirty hands coding, 2017.

[Beh19] Samir Behara.
Designing highly scalable database architectures.
Red Gate Simple Talk, 2019.

[Knu74] Donald E. Knuth.
Structured programming with go to statements.
ACM Comput. Surv., 6(4):261–301, December 1974.