Datacenter Design and Management

A Computer Architect's Perspective

Synthesis Lectures on Computer Architecture

Editor
Margaret Martonosi, *Princeton University*

Synthesis Lectures on Computer Architecture publishes 50- to 100-page publications on topics pertaining to the science and art of designing, analyzing, selecting and interconnecting hardware components to create computers that meet functional, performance and cost goals. The scope will largely follow the purview of premier computer architecture conferences, such as ISCA, HPCA, MICRO, and ASPLOS.

Datacenter Design and Management: A Computer Architect's Perspective
Benjamin C. Lee
2016

A Primer on Compression in the Memory Hierarchy
Somayeh Sardashti, Angelos Arelakis, Per Stenström, and David A. Wood
2015

Research Infrastructures for Hardware Accelerators
Yakun Sophia Shao and David Brooks
2015

Analyzing Analytics
Rajesh Bordawekar, Bob Blainey, and Ruchir Puri
2015

Customizable Computing
Yu-Ting Chen, Jason Cong, Michael Gill, Glenn Reinman, and Bingjun Xiao
2015

Die-stacking Architecture
Yuan Xie and Jishen Zhao
2015

Single-Instruction Multiple-Data Execution
Christopher J. Hughes and
2015

Power-Efficient Computer Architectures: Recent Advances
Magnus Själander, Margaret Martonosi, and Stefanos Kaxiras
2014

FPGA-Accelerated Simulation of Computer Systems
Hari Angepat, Derek Chiou, Eric S. Chung, and James C. Hoe
2014

A Primer on Hardware Prefetching
Babak Falsafi and Thomas F. Wenisch
2014

On-Chip Photonic Interconnects: A Computer Architect's Perspective
Christopher J. Nitta, Matthew K. Farrens, and Venkatesh Akella
2013

Optimization and Mathematical Modeling in Computer Architecture
Tony Nowatzki, Michael Ferris, Karthikeyan Sankaralingam, Cristian Estan, Nilay Vaish, and David Wood
2013

Security Basics for Computer Architects
Ruby B. Lee
2013

The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second edition
Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle
2013

Shared-Memory Synchronization
Michael L. Scott
2013

Resilient Architecture Design for Voltage Variation
Vijay Janapa Reddi and Meeta Sharma Gupta
2013

Multithreading Architecture
Mario Nemirovsky and Dean M. Tullsen
2013

Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU)
Hyesoon Kim, Richard Vuduc, Sara Baghsorkhi, Jee Choi, and Wen-mei Hwu
2012

Automatic Parallelization: An Overview of Fundamental Compiler Techniques
Samuel P. Midkiff
2012

Phase Change Memory: From Devices to Systems
Moinuddin K. Qureshi, Sudhanva Gurumurthi, and Bipin Rajendran
2011

Multi-Core Cache Hierarchies
Rajeev Balasubramonian, Norman P. Jouppi, and Naveen Muralimanohar
2011

A Primer on Memory Consistency and Cache Coherence
Daniel J. Sorin, Mark D. Hill, and David A. Wood
2011

Dynamic Binary Modification: Tools, Techniques, and Applications
Kim Hazelwood
2011

Quantum Computing for Computer Architects, Second Edition
Tzvetan S. Metodi, Arvin I. Faruque, and Frederic T. Chong
2011

High Performance Datacenter Networks: Architectures, Algorithms, and Opportunities
Dennis Abts and John Kim
2011

Processor Microarchitecture: An Implementation Perspective
Antonio González, Fernando Latorre, and Grigorios Magklis
2010

Transactional Memory, 2nd edition
Tim Harris, James Larus, and Ravi Rajwar
2010

Computer Architecture Performance Evaluation Methods
Lieven Eeckhout
2010

Introduction to Reconfigurable Supercomputing
Marco Lanzagorta, Stephen Bique, and Robert Rosenberg
2009

On-Chip Networks
Natalie Enright Jerger and Li-Shiuan Peh
2009

The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It
Bruce Jacob
2009

Fault Tolerant Computer Architecture
Daniel J. Sorin
2009

The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines
Luiz André Barroso and Urs Hölzle
2009

Computer Architecture Techniques for Power-Efficiency
Stefanos Kaxiras and Margaret Martonosi
2008

Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency
Kunle Olukotun, Lance Hammond, and James Laudon
2007

Transactional Memory
James R. Larus and Ravi Rajwar
2006

Quantum Computing for Computer Architects
Tzvetan S. Metodi and Frederic T. Chong
2006

Datacenter Design and Management: A Computer Architect's Perspective

Benjamin C. Lee

www.morganclaypool.com

ISBN: 9781627058476 paperback
ISBN: 9781627058483 ebook

DOI 10.2200/S00693ED1V01Y201601CAC037

A Publication in the Morgan & Claypool Publishers series
SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE

Lecture #37
Series Editor: Margaret Martonosi, *Princeton University*
Series ISSN
Print 1935-3235 Electronic 1935-3243

Datacenter Design and Management

A Computer Architect's Perspective

Benjamin C. Lee
Duke University

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE #37

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

An era of big data demands datacenters, which house the computing infrastructure that translates raw data into valuable information. This book defines datacenters broadly, as large distributed systems that perform parallel computation for diverse users. These systems exist in multiple forms—private and public—and are built at multiple scales. Datacenter design and management is multifaceted, requiring the simultaneous pursuit of multiple objectives. Performance, efficiency, and fairness are first-order design and management objectives, which can each be viewed from several perspectives. This book surveys datacenter research from a computer architect's perspective, addressing challenges in applications, design, management, server simulation, and system simulation. This perspective complements the rich bodies of work in datacenters as a warehouse-scale system, which study the implications for infrastructure that encloses computing equipment, and in datacenters as distributed systems, which employ abstract details in processor and memory subsystems. This book is written for first- or second-year graduate students in computer architecture and may be helpful for those in computer systems. The goal of this book is to prepare computer architects for datacenter-oriented research by describing prevalent perspectives and the state-of-the-art.

KEYWORDS

computer organization and design, energy efficiency, cluster computing, data centers, distributed systems, cloud computing, performance evaluation methodologies, resource allocation, software scheduling

Contents

Preface

This synthesis lecture is written for first- or second-year graduate students in computer architecture. The reader is expected to have completed graduate coursework in computer architecture; additionally, a course in distributed systems would be helpful. Moreover, the reader is expected to have some basic knowledge and prior experience in using the tools of the trade: cycle-level simulators. This background provides the requisite perspective on benchmarking and simulation for conventional workloads to help the reader appreciate challenges that are new and unique to datacenter workloads.

Moreover, this lecture may be helpful for graduate students in computer systems. Because the determinants of datacenter performance and efficiency increasingly lie at the hardware-software interface, architecture and systems perspectives on datacenters could be integrated to reveal new research directions. Because datacenter operators are rightfully wary of introducing new hardware into well-tuned systems, architects must anticipate system management challenges during architectural design. Furthermore, emerging hardware technologies and architectures require new system organizations and management.

Related Lectures. This synthesis lecture complements two existing synthesis lectures, one on datacenters and another on performance evaluation methods. Both lectures are highly recommended for their breadth and complementary perspective. Barroso et al. present a lecture on datacenters that focuses on the design of warehouse-scale machines, which is often taken to mean the datacenter itself [16]. For example, the lecture describes the facility, the peripheral infrastructure that supports the computing equipment, and figures of merit for evaluating datacenter efficiency and costs (e.g., total cost of ownership). This lecture is highly recommended for its breadth, its focus on warehouse-scale systems and facilities, and its industry-strength perspective. In contrast, our lecture focuses on processor and memory design, and emphasizes experimental methodologies that draw on a rich body of widely deployed open-source applications.

Eeckhout's lecture on performance evaluation methods focuses on performance evaluation methodologies, with a specific emphasis on strategies that accelerate the evaluation process [42]. The lecture describes analytical performance models that concisely represent processor performance. It also describes varied statistical strategies that reveal application performance while reducing the number of instructions simulated with cycle-level timing models. These research methodologies are best suited for understanding broad, general-purpose benchmark suites. In contrast, we focus on datacenter workloads and full system simulation. Previously proposed strategies for rapid design space exploration may apply to datacenter workloads as well, but we would need to adapt them to full system simulation.

We organize this lecture on datacenter research methodologies in several chapters. Chapter 2 describes several representative datacenter applications, surveys their implications for hardware architectures, and proposes benchmarking strategies. Chapters 3–4 survey recent research in server design and management. Chapter 5 details strategies for simulating datacenter servers. Specifically, we present approaches to processor and memory simulation, and demonstrate methodologies for precise simulations that target application regions of interest. Finally, Chapter 6 describes strategies for simulating datacenter dynamics at scale. We present analytical and empirical approaches to understanding task behaviors and queueing dynamics.

Collectively, the goal of this book is to prepare a computer architect for datacenter-oriented research. It describes prevalent perspectives and the state-of-the-art. Yet, for all the research that is surveyed in this book, many challenges remain and the required advances in datacenter design and management would very much benefit from a computer architect's perspective.

Benjamin C. Lee
January 2016

Acknowledgments

This synthesis lecture draws from varied research projects that have spanned the last eight years. During this time, I have been fortunate to work with extraordinary collaborators and students. My perspectives on computer architecture and datacenter systems have been enriched by these collaborations. I am thankful for my collaborators during projects at Microsoft Research—Vijay Janapa Reddi, Trishul Chilimbi, and Kushagra Vaid—an institution where fundamental research is leavened with applied perspectives. I am also grateful for my collaborators during projects at Stanford University—Mark Horowitz, Christos Kozyrakis, and Krishna Malladi—a place where interdisciplinary work is pervasive. Finally, I am incredibly fortunate to work with a group of outstanding Ph.D. students at Duke University—Songchun Fan, Marisabel Guevara, Ziqiang Huang, Tamara Lehman, Bryan Prosser, Qiuyun Wang, Seyed Majid Zahedi, and Pengfei Zheng.

For this project, I am thankful for Margaret Martonosi and Michael Morgan's invitation to write a synthesis lecture on datacenter research. The lecture has benefited substantially from reviews by Christina Delimitrou, Stijn Eyerman, Boris Grot, and Lingjia Tang. In advance, I thank the faculty and students who use and share this lecture to advance their own research projects. I would very much appreciate any feedback from researchers and practitioners.

Lastly, I would like to thank our research sponsors for their support of my projects and the students who contribute to them. Our projects have been supported by the National Science Foundation, Duke University, and the Google Faculty Research Award. They have also been supported by STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA. Any opinions, findings, or recommendations expressed in this material are those of the author and do not necessarily reflect the views of these sponsors.

Benjamin C. Lee
January 2016

CHAPTER 1

Introduction

An era of big data demands datacenters, which house the computing infrastructure that translates raw data into valuable information. Data flows are increasingly diverse and multi-directional. Early Internet services supplied data, which users translated into information. In contrast, today's services supply, consume, and analyze data to produce information that is timely and suited to each user's individual context. The increasing sophistication of data sources and flows motivate correspondingly capable datacenters.

1.1 DATACENTERS DEFINED

This book defines datacenters broadly as large distributed systems that perform parallel computation for diverse users. Defined as such, datacenters exist in multiple forms. Private datacenters compute for users within a single organization. Within an organization, sharing increases utilization and improves efficiency by amortizing system costs over more work. Yet sharing requires mechanisms to distribute resources according to policies for fairness and priority, which are tailored to organizational needs. Internet services (e.g., search, mail) and software-as-a-service are representative of this type of datacenter computing.

In contrast, public datacenters compute for users that share no affiliation with the system operator. Users select machines from a menu and pay as these resources are reserved or consumed. This market for computation requires users to characterize their workloads and request resources explicitly. Infrastructure-as-a-service is representative of this type of datacenter computing.

Datacenters exist at varied scales. Industry's largest datacenters incorporate thousands of machines that compute for millions of tasks, yet smaller datacenters are far more common. This book is agnostic about scale, emphasizing the similarities between highly parallel and distributed clusters rather than the unique characteristics of warehouse-scale clusters. For example, recent research in search engines consider diverse document corpora, from the world wide web to online encyclopedia. At all scales, datacenters encounter fundamental challenges.

1.2 RESEARCH DIRECTIONS

Datacenter design and management is multifaceted, requiring the simultaneous pursuit of multiple objectives. First, system architects balance service quality and energy efficiency. Computational tasks must complete within tens of milliseconds for online services and should progress at the same rate as others in the job to avoid stragglers that lengthen the critical path. Yet the system

should colocate software to amortize hardware power, employ low-power servers when possible, and power only the servers required to meet service targets. Such efficiency would increase datacenter capability within the same footprint.

System architects must also balance service quality and fairness. The definition of fairness depends on the system context and is expressed in terms of equality in allocation or progress, in terms of domain-specific priorities, or in terms of game-theoretic desiderata. In cooperative settings, such as data- and task-parallel computation, equal progress reduces the variance in latency distribution and guards against outliers. In competitive settings, such as shared clusters and federated datacenters, economic notions of fairness incentivize sharing and guard against envy. Fairness, or the lack thereof, determines whether strategic users choose to participate in a shared system.

The pursuit of system desiderata requires advances in management mechanisms. Datacenter profiling must supply the requisite data for intelligent allocation and scheduling. Datacenter allocation must distribute physical hardware such as server, processor, or memory resources to diverse tasks. It must also distribute less tangible but no less critical resources such as time, priority, and power. Each of these resources are made scarce by design within efficient datacenters, making constrained management and optimization a rich vein of research.

1.3 RESEARCH CHALLENGES

Computer architects who wish to perform datacenter research encounter five broad challenges—applications, design, management, server simulation, system simulation—all of which demand strategies and methodologies that extend beyond their conventional tools.

Applications and Benchmarks. First, datacenter applications are increasingly built atop generalizable platforms and frameworks that implement multiple abstraction layers. The abstractions are intended to separate the programming model with which users develop distributed algorithms and the run-time system that breaks computation into many small, short tasks. MapReduce is an example of this strategy in practice—the programming model defines Map and Reduce functions and the run-time system creates individual Map and Reduce tasks that compute on small slices of the data. The abstractions are also intended to increase modularity, which permits sophisticated libraries for stream processing, machine learning, and graph analytics to leverage capabilities from lower in the stack of system software.

In this setting, computer architects must clearly define the workload of interest. They cannot simply benchmark an application because benchmarking datacenter applications is replete with subtle questions. They should clarify whether benchmarks evaluate the run-time system or specific types of tasks. Should a Spark benchmark include both the engine and the application tasks? What index of documents and set of queries produce a representative benchmark for search? To answer these questions, computer architects require increasingly precise measurements from increasingly large software systems.

Design. Second, hardware design is motivated by two competing objectives in datacenter computing—service quality and energy efficiency. Today's hardware architectures are designed for one or the other. High-performance design ensures low latency and high throughput accompanied by significant power costs whereas low-power design ensures energy efficiency accompanied by significant performance risks. During design, architects characterize hardware-software interactions to explore a rich space of high-performance and low-power design points. The selection of multiple, heterogeneous designs from this space dictates the balance between performance and efficiency. Moreover, the organization of these heterogeneous components at scale affects the competition and contention for preferred resources.

Management. Third, datacenter architects must design for manageability, which means anticipating the run-time consequences of decisions made at design-time. Datacenters co-locate multiple software tasks on a single hardware platform to amortize its fixed power costs, a strategy that improves efficiency and energy proportionality. At run-time, the datacenter must allocate hardware and schedule software to meet performance targets and ensure fairness. Architects consider a combination of throughput and latency when evaluating performance; the former describes system capacity whereas the latter describes application responsivenes. Furthermore, architects must adopt definitions of fairness that encourage users to employ shared datacenter hardware instead of procuring private systems. Studies in resource allocation might take a game-theoretic approach to account for strategic behavior. Because server and datacenter designs vary in management complexity, architects should optimize designs to manage performance risk—the probability of contention and poor outcomes—in shared systems.

Hardware Simulation. Fourth, hardware simulation for datacenter applications is complicated by the system software stack. Datacenter applications require operating systems, virtual machines, and libraries. Moreover, they perform network and disk I/O. For example, the open source engine for web search is implemented in Java and requires the Java virtual machine. To support the full system software stack and the hardware ecosystem, computer architects must rely on a sophisticated combination of emulation, which ensures functional correctness, and simulation, which provides timing models for greater insight.

Computer architects must apply application-specific insight to identify regions of interest for full system simulation with detailed timing models, which is prohibitively expensive under normal system operation. For tractability, systems and applications should initialize and warm up in emulation mode before performing specific tasks of interest in simulation mode. Application tasks are short and getting shorter, requiring only tens or hundreds of milliseconds of computation. If architects could start simulation right before the task of interest, simulating a task to completion would be tractable.

System Simulation. Finally, system simulation for datacenter applications is complicated by system scale. Datacenter applications increasingly rely on run-time systems to schedule and distribute tasks to workers. Hardware performance interacts with queueing dynamics to determine

distributions for task response time. Datacenter operators often optimize latency percentiles (e.g., 99th percentile) and mitigate stragglers in the distribution's long tail.

Computer architects can turn to queueing models for rules of thumb and back-of-the-envelope calculations to assess system throughput, queueing times, and waiting times. When certain assumptions are satisfied, M/M/1 queueing models provide an elegant analytical framework for reasoning about system dynamics. But under arbitrary and general settings with non-parametric distributions for task inter-arrival and service times, architects should rely on discrete event simulation. Both analytical and empirical approaches can play a role when assessing system performance at scale, a setting in which physical measurements on a deployed system can be impractical.

CHAPTER 2

Applications and Benchmarks

Datacenter applications share a number of characteristics. First, they compute on big data, which motivates new architectures that blend the capacity and durability of storage with the performance of memory. Increasingly, data resides in distributed memory and is accessed via the network according to Zipfian popularity distributions [14]. Such distributions have long tails such that the most popular pieces of data are accessed far more frequently than less popular pieces. Although Zipfian distributions' temporal locality facilitates memory caching, their long tails produce irregular requests and demand high-capacity memory.

Second, datacenter applications extend to warehouse-scale with task parallelism. Data analysis is partitioned into many small pieces to form tasks, which the system queues and schedules. Third, applications are implemented atop programming models and run-time systems, such as MapReduce, to expose and manage task parallelism. Distributed computing frameworks implement abstractions such that programmers need not reason explicitly about a computing cluster's physical implementation when specifying program functionality. Collectively, these application characteristics have aided the proliferation of distributed computing.

2.1 BENCHMARK SUITES

As interest in datacenter research grows, benchmark suites have proliferated. A number of studies survey datacenter workloads, providing computer architects multiple perspectives on benchmarking. Many of these workloads are open source and widely available. The difficulty is exercising them with realistic input data and computational kernels, and measuring their hardware activity precisely.

Benchmark suites identify representative applications and provide software targets for research in hardware systems. Datacenter benchmarks are distinguished by their fine-grained tasks that facilitate "scale-out" computation in large distributed systems. In contrast, conventional benchmarks often focus on "scale-up" computation, which benefits from single-threaded performance. Scale-out and scale-up systems require different benchmarks and demand different design decisions. During design space exploration, for example, scale-out workloads might trade fewer resources (e.g., smaller cores and caches) for the power efficiency that is required for tightly integrated datacenter servers.

Lim et al. study representative datacenter workloads and design server architectures that balance performance, power, and cost [101]. The benchmark suite includes search, mail, content serving, and map reduce. As scale-out workloads, these benchmarks exhibit fine-grained task par-

allelism, which demands less capability from any one hardware component. Lim et al. find that datacenter workloads can benefit from low-end and mobile-class hardware, which offer performance with attractive total-cost-of-ownership (TCO). This seminal study illustrates new metrics and methodologies for the design of server architectures.

Ferdman et al. present CloudSuite, a benchmark suite for scale-out workloads. The suite includes data serving, MapReduce, media streaming, SAT solver, web frontend, and web search [49]. The difficulty with datacenter workloads is not obtaining the software but deploying it on realistic hardware platforms and exercising it with representative tasks. CloudSuite documents its system settings and makes several observations about microarchitectural activity—datacenter workloads miss often in the instruction cache, exhibit little instruction- or memory-level parallelism, work with data that does not fit in on-chip caches, and require little inter-core communication. The CloudSuite infrastructure and its insights enable a number of server design studies [47, 48, 56, 104].

Wang et al. present BigDataBench with data stores, relational databases, search engines, and graph analysis [159]. BigDataBench is notable for broad coverage of applications and data inputs, which affect microarchitectural activity and pose methodological challenges for cycle-level simulation for computation on realistic datasets. BigDataBench documents its system settings and makes several observations about system behavior—datacenter workloads exhibit low application intensity as defined by the number of instructions per byte of data transferred from memory, miss often in the instruction cache, and benefit from larger last-level caches. Characterizing and identifying a modest number of representative workloads and data inputs from the complete set improves tractability [75, 76].

In this chapter, we survey four representative datacenter applications—web search, memory caching and storage, MapReduce, and graph analysis. Search and caching represent latency-sensitive applications in which computation must complete in tens of milliseconds to guarantee the user experience. MapReduce and graph analysis represent batch applications in which computation may complete in the background. We consider these applications from a computer architect's perspective, describing the computation and its demands on the system architecture.

2.2 SEARCH

Data volumes are growing exponentially, driven by a plethora of platforms for content creation, from web pages and blogs to news feeds and social media. Users, confronted with this massive data corpus, turn to search engines for tractability. These engines identify and deliver relevant data in response to a user's query. Search engines have become increasingly sophisticated, partly to keep pace with data diversity. For example, early web search engines used the PageRank algorithm to identify popular pages that are often cited and linked by other popular pages [24]. Today's web search combines PageRank with many other algorithms that use contents from pages and queries from users to determine relevance [55]. Future search engines may consider hundreds of features,

which describe multiple types of content and users, and employ statistical machine learning to estimate relevance from this high-dimensional space [55, 137, 138].

Search is a natural starting point for research in datacenter architecture. Search is a high profile application that needs little motivation for the broader research community. Moreover, it embodies several fundamental characteristics of datacenter applications. First, search is relevant at multiple scales. Although Google search and Microsoft Bing dominate mind share for web search in the United States, researchers can study realistic search engines at more experimentally accessible scales by indexing and querying data on a personal machine, a subset of the Internet (e.g., Wikipedia), or any other well-defined data corpus of interest. Second, search is parallelizable. A strategy that distributes indexed data across multiple machines and then sends a query to each of those machines scales well on commodity hardware. In this setting, a deep analysis of one search engine node generalizes to others in the datacenter. Finally, search is extensible. Atop a search engine's primitive functionality—quantifying relevance given a query specification—researchers can explore variants that anticipate user questions by translating simple queries into more sophisticated ones or estimate relevance with novel machine learning methods.

Search Engine. The search engine performs two types of computation. First, the engine crawls and indexes content with batch computation that aggregates and organizes data. Crawls may be performed at many scales—although Google search crawls the world wide web, Wikipedia search crawls only the entries in its online encyclopedia. The engine creates an index from crawled pages, which contains information about words and their locations in the data corpus. Google describes its search index much like an index in the back of a book; a reader locates the search term within the index to find the appropriate page(s) [55]. An index may become quite large, especially if it spans the entire web. The engine typically partitions an index such that each part fits in a server's main memory. In a multi-server system, each query is distributed across servers and accesses the index in parallel.

Second, the search engine executes interactive queries, identifying related pages with the index and calculating relevance scores to determine the pages to display. The engine calculates a static score for each page, independently of any query, based on attributes such as popularity (e.g., PageRank) and freshness. Upon receiving a query, the engine accounts for query terms and operators to calculate a dynamic score. The engine orders pages according to relevance scores, prepares captions or snippets for the most relevant pages, and serves these pages to the user. In a multi-server system, each server sends the most relevant pages in its index to an aggregator, which produces the final set of results.

Figure 2.1 illustrates the organization of the Microsoft Bing Search engine [137, 138]. Queries enter the system via the top-level aggregator, which caches results for popular or repeated queries [154]. The aggregator distributes queries to index-serving nodes, which calculate dynamic page ranks (using a neural network in this example) for their respective parts of the index. Finally, the aggregator integrates nodes' results to serve links and page snippets to users.

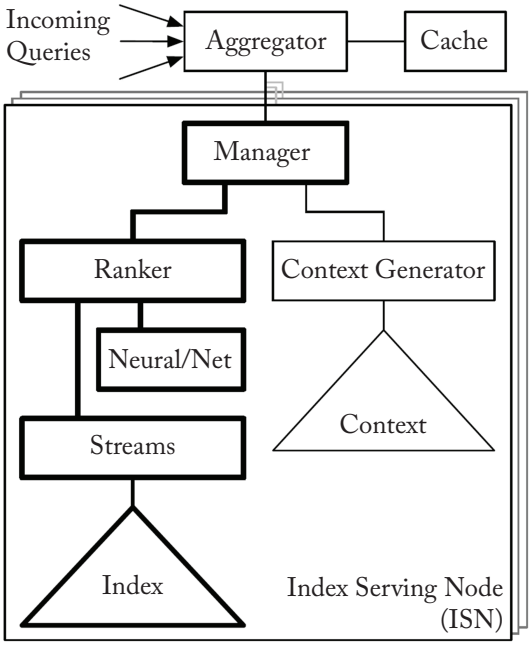

Figure 2.1: Overview of Microsoft Bing and the scoring computation within an index serving node [137, 138].

Query Complexity. A search engine must serve relevant content for heterogeneous and diverse queries. Figure 2.2 illustrates the space of possible query constructions. A query in this space is defined by the number of terms, the logical operators that integrate them into a single query, and the wildcard operators that allow a query to express rich permutations of any one term. Many queries execute only one term and do not use operators. Yet the majority of queries are more sophisticated.

The frequency distribution of query complexity is well studied. Silverstein et al. performed an early study using Altavista logs from 1999 [147] and Table 2.1 summarizes their findings. More than 55% of queries specify multiple terms. And 20% of queries use one or more operators to connect these terms together. Empty queries contain no terms and usually result from technical misunderstandings. For example, users fail to enter terms into an advanced search interface, which supports the use of query operators.

These early observations about user behavior and query complexity have proven durable. Jansen et al. performed a later study in 2006 to show that query lengths have increased over time as users and search engines have become more sophisticated [74]. Indeed, search engines themselves increasingly introduce complexity on the user's behalf. For example, the engine may

translate a user search for "rabbit" into a broader search that includes variants (e.g., "rabbits") and synonyms (e.g., "hare," "bunny") for more comprehensive and relevant results [41].

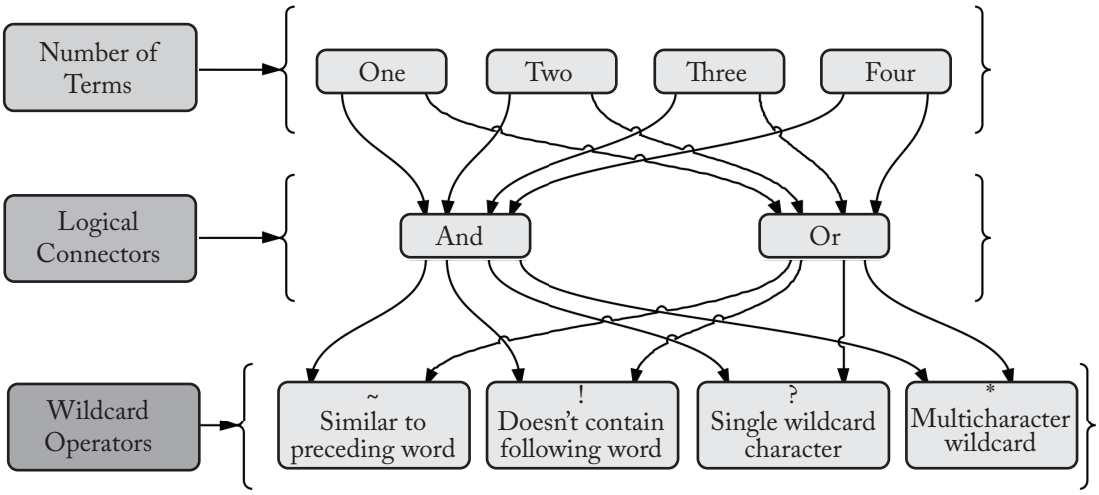

Figure 2.2: Determinants of query complexity.

Table 2.1: Distribution of query complexity [147]

No. of Terms (%)	No. of Operators (%)
0 (20.5%)	0 (79.6%)
1 (25.8%)	1 (9.7%)
2 (26.0%)	2 (6.0%)
3 (15.0%)	3 (2.6%)
>3 (12.6%)	>3 (2.1%)

A study of web search must account for query complexity, exercising the search engine with queries that combine terms and operators in ways that users might. We describe a query generator that produces diverse queries for a search engine benchmark [23]. Inputs to the generator are frequency distributions on query length and operator count, such as the one in Table 2.1, and indexed documents that provide a dictionary from which words are sampled. Outputs are queries that reflect the complexity of user behavior and the popularity of content within a data corpus.

Specifically, the generator creates a dictionary of words using contents for indexed documents. The dictionary tracks the frequency of each word's appearance in these documents. To produce a single-term query, the generator randomly samples a word from the dictionary. The probability of sampling a particular word is proportional to its popularity. However, single-term

queries do not reflect the sophistication of modern searches. Even in 1999, more than half of all queries included two or more terms. And this trend toward multi-term queries has increased over the years [74, 176].

To produce a multi-term query, the generator performs conditional sampling. The probability of sampling a particular word for the n-th term depends on the $n - 1$ terms already sampled for the same query; such dependences reflect true user behavior. Specifically, the generator produces a multi-term query by sampling a starter word from indexed documents. Then the generator examines the documents that contain this starter word. Within these documents, the starter word appears in context with other, related words. The generator creates a set of neighboring words. From this set, the generator randomly selects the next term. Selection probability is proportional to each word's frequency in the set. By recursively applying this process, the generator produces a query with the desired length and with a coherent mix of semantically related words.

Figure 2.3 illustrates the process for generating a multi-term query, using "rabbit" as the starter word. The generator identifies 632 unique documents that contain "rabbit." With these documents, the generator constructs a sub-dictionary that tracks the frequency of each word's appearance in rabbit-relevant documents. In this example, "Roger" and "framed" are the most typical words that appear with "rabbit." The generator performs weighted sampling on words in the sub-dictionary to produce the second word in the query. Suppose the second word is "lucky." The query is now comprised of the two words "rabbit" and "lucky." If a third word is needed, the generator identifies documents that match both terms to create a new sub-dictionary for sampling. Thus, the generator recursively adds terms to produce a query with the desired length. The query length itself is drawn from a probability distribution, such as the one in Table 2.1.

As query length increases, the number of relevant documents decreases sharply. For example, consider the frequency distribution of words within Wikipedia documents. The vast majority of words in these documents appear less than ten times. A much smaller number of words are common and appear in more than 45K of the 50K Wikipedia documents. Such a distribution has two related implications. During query generation, the size of sub-dictionaries decreases rapidly with query length. And during query processing, the number of results returned to the user decreases rapidly with query complexity.

Logical operators connect words in a multi-term query. The most common operators are AND and OR. Of these, AND is the default. These operators affect the number of documents that match a given query. Using the AND operator sharply reduces the number of documents returned to a user. However, returned documents are often more relevant since the AND operator prunes those that only contain one of the query's terms. Another class of operators, wildcards, increases query sophistication and complexity. Table 2.2 summarizes these operators. For example, the NEAR operator ($\sim$) finds words that are similar to the user-specified word. Similarity is defined by the Levenshtein distance—all words that differ in one character from the original word are also considered relevant (e.g., "text" and "test"). Wildcard operators increase processor intensity by broadening query scope. Although users rarely specify explicit wildcards, these operators

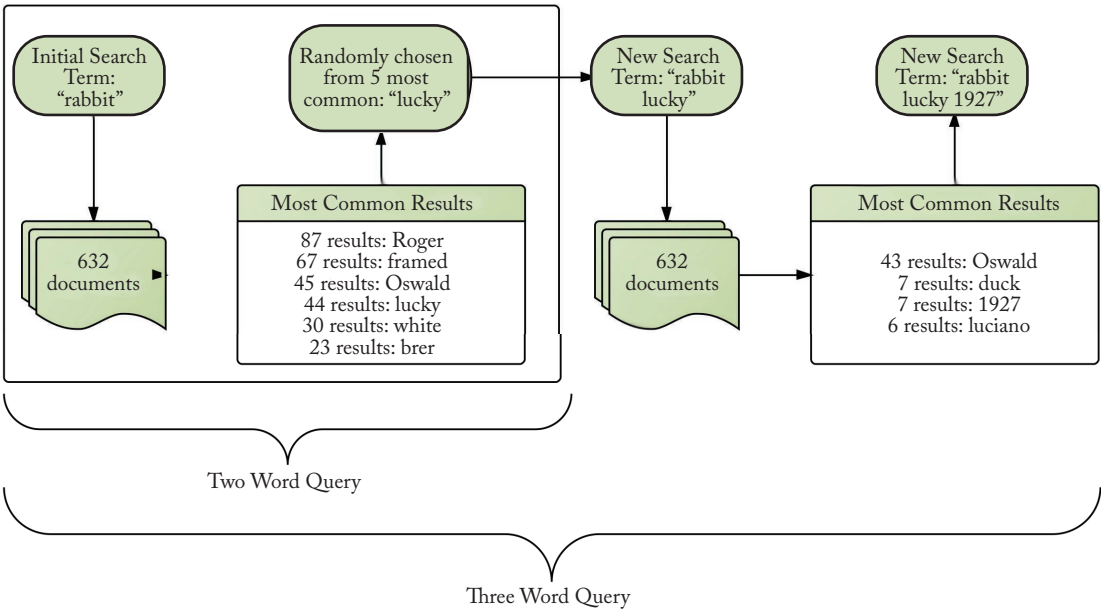

Figure 2.3: Example for multi-term query generation.

can be inserted transparently by the search engine to improve service quality. For example, the NEAR operator can be used to mitigate spelling errors and provide other user-friendly features.

Table 2.2: Wildcard operators with usage examples

Op	No. of Terms (%)	No. of Operators (%)
?	single-character wildcard	"thing?" matches "thing," "things," etc.
*	multi-character wildcard	"a*t" matches "at," "apt," "assistant," etc.
!	inverts following word/phrase	"!orange" returns any doc without "orange"
~	finds words within specied edit distance from preceding word	"text" matches "text," "test," "next," etc.

Benchmarking Strategies. Benchmarking web search for computer architecture research requires analysis in several dimensions—the data corpus, the queries that exercise the search engine, and the hardware design space. In several regards, web search is easy to simulate. First, the index is partitioned across many servers, which rank pages in parallel, and simulating a single server provides insights across all servers. Second, the index is sized to fit in memory, which reduces or eliminates the network and storage I/O that is not often accommodated by cycle-

level simulators. Finally, simulating end-to-end query execution is tractable because an individual query completes in tens or hundreds of milliseconds. Practically, researchers have simulated web search queries by using MARSSx86 and DRAMSim2, deploying Apache Nutch and Lucene, and executing search queries for Wikipedia's data corpus. Other methodologies for full system simulation may be equally effective.

2.3 MEMORY CACHING

Conventional approaches to storage and retrieval fail to scale as data volumes increase. Disk data transfer rates, which have failed to keep pace with disk capacity, limit how frequently each data block can be accessed, on average. Ousterhout et al. find that the ratio of disk to bandwidth capacity constrains the system, permitting access to a given 1 KB data block only once every 58 days [122]. Such poor performance, which is attributed to disk latency, raises the possibility of stranding data in the storage system.

The root cause of this performance challenge is poorly matched disk capacity and bandwidth. System architects might address this imbalance by distributing the same amount of data across many more disks, thereby increasing system bandwidth. Because data volume remains constant as the number of disks increases, disk utilization falls and the cost per bit rises. As disk-based storage becomes more expensive, today's DRAM and Flash memories become fast and viable alternatives. Such economic trends would also open the door to emerging technologies, such as phase change memory.

Table 2.3: Evolution in disk technology over 25 years. Reproduced from Ousterhout et al. [122].

	Mid-1980s	2009	Difference
Disk capacity	30 MB	500 GB	16,667 better
Maximum transfer rate	2 MB/sec	100 MB/s	50× better
Latency (seek + rotate)	20ms	10 nms	2× better
Capacity/bandwidth (large blocks)	15 s	5,000 s	333× worse
Capacity/bandwidth (1KB blocks)	600 s	58 days	8,333× worse

Datacenter servers with large main memories reduce access latency by caching popular data. Heavy-tailed Zipfian distributions on data popularity means that memory caches can significantly improve performance at moderate cost. At warehouse-scale, caching frameworks pool many servers' memories into a single, large, logical cache accessed via a key-value system. Industry uses a variety of key-value storage systems. Indeed, Facebook caches 75% of its non-media data in memory [115, 122].

Key-Value Storage Systems. We survey recent advances in key-value storage systems. Strikingly, many of these system software advances respond to hardware and technology trends. FAWN is a key-value store designed for low-power processors and Flash memory. RAMCloud

is a key-value store designed to ensure durability and availability given DRAM and low-latency networks. And because the network is on the critical path, bypassing the kernel or accelerating the network protocol with hardware support is compelling. Fundamentally, these research directions provide foundations for fine-grained, low-latency access to data at warehouse-scale.

A key-value storage system provides fine-grained access to large data sets. In each of these systems, data is represented as a collection of key-value pairs; each value is associated with a unique key. A hash function maps a key to the corresponding value's location, which identifies the server node and a physical location in memory or storage. The hash tables support a simple set of operations—`get(k)` retrieves a data value with key `k`, `set(k,v)` stores a data value `v` with key `k`, and `delete(k)` deletes a data value with key `k`. Atop these primitives, systems researchers can build sophisticated data models for consistency, reliability, etc. Internet service providers deploy such systems at scale—Amazon uses Dynamo [36], LinkedIn uses Voldemort [120], and Facebook uses memcached [118].

The **memcached** framework is a system for distributed memory caching [118]. Data requests issue queries to memcached before querying back-end databases. Consistent hashing directs each query to a memcached node by mapping a key to a unique server with the corresponding data value. Within each server, memcached divides memory into slabs, each of which are designated to hold data objects of a particular size. Slabs and their data sizes are chosen to reduce fragmentation, increase utilization, and increase cache hit rates [115].

Facebook's memcached deployment is dominated by `get` queries, serving thirty `get`'s for every `set` [14]. This balance suggests that its applications treat distributed memory as a persistent store rather than a temporary cache. The majority of its stored values are less than 500 bytes each and nearly all stored values are less than 1,000 bytes. Memcached values are often found in the cache and hit rates are high—greater than 90%.

Performance optimizations for distributed memory caching require a holistic approach that couples software data structures with hardware capabilities. Lim et al. design **MICA** (memory-store with intelligent concurrent access) to deliver high performance for in-memory, key-value stores [98]. First, MICA partitions data to exploit multi-core parallelism within a backend server. Second, it interfaces directly with the network interface card and avoids latency overheads in the operating system's network stack. Finally, it uses novel data structures for memory allocation and indexing.

Li et al. further pursue performance by coupling MICA's software advances with balanced server architectures [97]. The principle of balanced system design in which resources for compute, memory, and network are provisioned in the right proportions for the workload improves efficiency, reduces latency, and increases throughput to a billion requests per second on a single node. Moreover, modern platforms provide new capabilities, such as direct cache access, multi-queue network interface cards, and prefetching, which enhance request throughput when used properly.

Pursuing performance and durability, Ousterhout et al. propose **RAMCloud**, a key-value store that holds all data in memory and obviates the need for hard disks [122]. The case for

RAMCloud is based on software trends and hardware limitations. First, datacenters already hold a large majority of their data in memory and the marginal cost of holding the remainder in memory is no longer prohibitive. Second, datacenter applications can no longer tolerate disk's performance limitations—when disk is several orders of magnitude slower than memory even rare misses in the memory cache will have a large impact on average access time.

To implement the vision of placing all data in memory, RAMCloud addresses two challenges. For durability, it backs up data to disk. It distributes backups to hundreds or thousands of disks, which provide the requisite transfer rates for fast recovery [117]. For performance, RAMCloud must reduce network latency. DRAM latency ranges from tens to hundreds of nanoseconds. In contrast, Ethernet and the TCP/IP stack require hundreds of microseconds for round-trip transmission. Much of this time is spent in interrupt handling and protocol processing within the operating system. TCP/IP offload and kernel bypass can reduce round-trip software overheads to one microsecond [122].

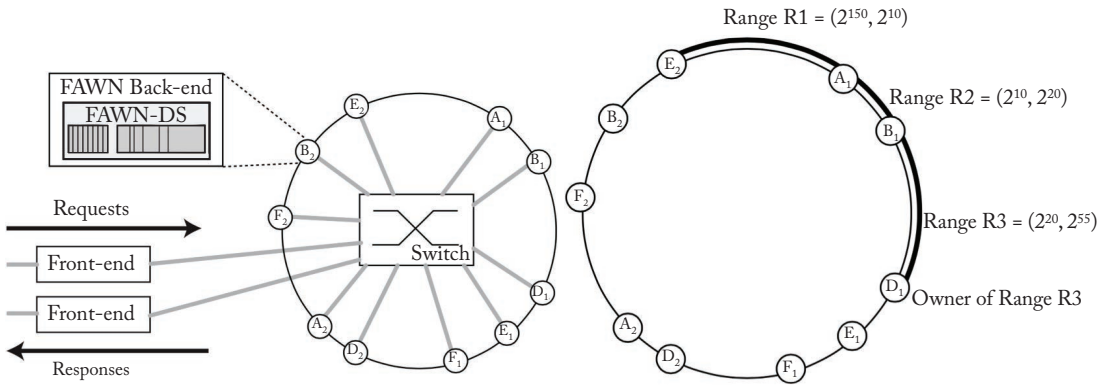

Figure 2.4: FAWN, a fast array of wimpy nodes, and its key-value store. Reproduced from Andersen et al. [2].

Pursuing energy efficiency for key-value stores, Andersen et al. propose **FAWN**, a key-value system designed for a fast array of wimpy nodes that employ low-power processors and solid-state disks [2, 3]. Compared to systems with general-purpose processors and hard disks, FAWN increases the number of queries per Joule by an order of magnitude or more for small, random accesses.

FAWN is comprised of a key-value system (KV) and a datastore (DS). The KV distributes data across datacenter nodes by mapping key ranges to DS nodes using consistent hashing. In this hashing scheme, a ring represents the continuum of key values. Each DS in the system is placed on the ring and its position determines the range of keys for which it is responsible. The KV routes each request, which uses a get/put interface, to the DS responsible for its key. For

example, requests for keys between 2^{10} and 2^{20} are all routed to the same DS. With consistent hashing, only a fraction of the keys are re-mapped when DS nodes join or leave the system.

The DS serves each request, reading data from and writing data to solid-state disks. The DS is a log-structured store, which appends writes to the end of a data log. Appends are particularly well suited for solid-state disks because they avoid the overheads of garbage collection and coarse-grained erasure. To read data from the log, the DS maintains an in-memory hash table that maps a key in its assigned range to the corresponding location. This location is specified as an offset to the append-only log.

Benchmarking Strategies. Benchmarking distributed memory caching frameworks requires a sequence of queries to the system (e.g., get, set, and delete). Moreover, it requires a distribution for data popularity, which specifies how often each key is used within a key-value store. Finally, the data object size matters and, in memcached, the typical size is small. Most objects are KBs in size and objects are rarely larger than 200 MBs [14, 100]. Small, fine-grained accesses illustrate the memory caching's advantages over disk. But they also highlight the network's latency overheads.

The design of recent key-value stores and the servers that deploy them suggest that further reductions in access latency require a holistic strategy that accelerates processing for hash index lookups, reduces network overheads, and exploits emerging memory technologies. Further research is needed to determine whether targeted microbenchmarks that measure separate progress toward each of these goals is possible. For example, network interfaces that bypass the operating system may be orthogonal to advances in resistive memory technologies. Collectively, recent performance optimizations have produced multiple, independent advances toward more responsive access to big data.

2.4 MAPREDUCE

Big data is cumbersome, especially when many small pieces of computation are needed on every piece of data. Datacenters must distribute computation across tens of thousands of servers, orchestrate communication between those servers, and ensure performance and reliability at scale. These challenges appear in varied settings, from counting words to indexing web pages. Software developers ought to share the programming models and infrastructure that address these challenges rather than re-architect their own solutions. In such a setting, Dean and Ghemawat devised MapReduce [35].

MapReduce is both a programming model and a run-time system. The model expresses computation as a series of map and reduce functions, which compute on key-value pairs. The system performs computation by partitioning input data, launching map and reduce tasks at scale, and ensuring resilience with mechanisms to mitigate straggling or failed tasks. By separating the programming model and the run-time system, MapReduce provides clean abstractions. Software developers specify only map and reduce functions, knowing that the run-time system will manage their deployment in the datacenter. Indeed, these abstractions have fostered a large ecosystem of

workloads built atop MapReduce for distributed databases, machine learning, graph analytics, and many others.

Programming Model. The MapReduce programming model is defined by its two constituent computational phases. Map applies a function to each element of the input data to produce a series of key-value pairs. Reduce applies a function to intermediate results in these key-value pairs to produce an integrated result. A MapReduce programmer specifies each of these functions and data types for key-value pairs to effect the desired computation.

```
map(key, value):
   // key: document name
   // value: document contents
   for each word w in value:
      EmitKeyValue(w, 1);
```

Figure 2.5: Map definition for word count, modified from [35].

```
reduce(keys, values):
   // keys: a list of words
   // values: a list of counts
   for each unique k in keys
      int result = 0;
      for each v in values with key k:
          result += v;
      Emit(k, result)
```

Figure 2.6: Reduce definition for word count, modified from [35].

Figures 2.5–2.6 reproduce an example from Dean and Barroso that counts words in a document. The map function consumes a key-value pair, which specifies data sources, and produces another pair, which indicates the occurrence of a word. The reduce function consumes these many key-value pairs and sums these results for each unique key. This example illustrates MapReduce in its simplest form, highlighting distinct map and reduce phases. These primitives extend naturally to varied settings, especially when many partial answers can be aggregated with commutative and associative reduce functions.

System Implementation. Implementations of MapReduce systems are notable for their scalability and resilience, both of which are derived from the nature of Map and Reduce primitives. The MapReduce programming model was inspired by primitives in functional programming languages, which treat functions as the fundamental building block for programs and treat computation as the composed evaluation of those functions. Purely functional programming languages

do not produce side effects or modify global state. In this spirit, the MapReduce framework makes no provision for functions with side effects, shifting the burden of ensuring atomicity and consistency to the programmer. In practice, applications that produce side-effects are a poor match for the MapReduce model.

The lack of side effects reduces the difficulty of parallelizing map and reduce functions for distributed computation in a datacenter. A MapReduce implementation includes a master, which orchestrates data movement, and workers that perform either map or reduce computation. MapReduce splits input data into pieces and distributes data to map workers. These workers compute intermediate key-value results, store them in local disk, and communicate their location to the master. The master communicates these locations to the reduce workers. These workers read the key-value pairs, sort them by key, and apply the reduce function.

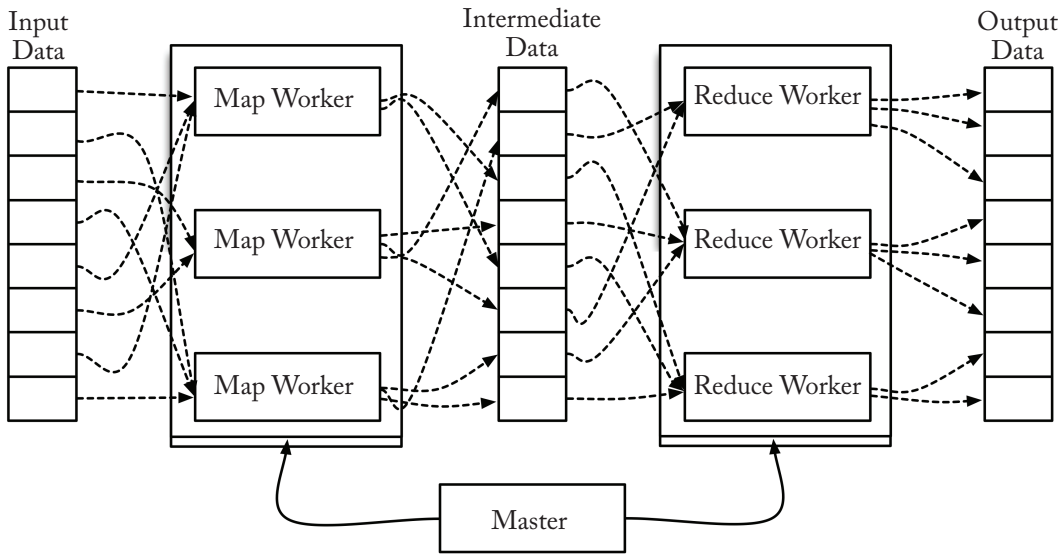

Figure 2.7: MapReduce overview.

By separating the MapReduce programming model from its implementation with a clean abstraction layer, system architects can pursue designs that ensure performance and resilience. For performance, MapReduce identifies straggling tasks and preemptively re-schedules them on another, potentially faster, machine. MapReduce tasks retrieve data from a distributed file system that replicates data and distributes them on many machines, motivating sophisticated task placement strategies that pursue locality. These same system properties ensure resilience. The master resets and reschedules failed tasks, and the distributed file system replicates data to guard against loss.

Variants of MapReduce. Since Google's introduction of MapReduce as a programming model for parallel computing in 2004, the systems community has produced a number of im-

plementations. In 2007, Microsoft Research proposed Dryad for distributed data analysis on large clusters [73]. Dryad supports sophisticated communication topologies, expressed as directed acyclic graphs, that subsume MapReduce's restricted map-shuffle-reduce communication pattern. As in MapReduce, Dryad's implementation includes a run-time system that schedules tasks and manages resources.

In 2011, after six years of development, the Apache Software Foundation released Hadoop. Hadoop includes a distributed a file system (HDFS) and an implementation of MapReduce that mirrors Google's system software. Since its launch, Hadoop has provided a foundation for an ecosystem of software libraries and frameworks:

- **Hive** provides a database engine that transforms SQL-like queries into MapReduce tasks executed against HDFS [8]
- **HBase** provides a non-relational storage system that holds big data in sparse tables [7], as in Google's BigTable [29]
- **Pig** provides a scripting language to write sophisticated MapReduce programs [12]
- **Spark** and **Mahout** provide libraries for machine learning atop Hadoop [10, 13]

Spark is a particularly noteworthy addition to the Hadoop ecosystem, accelerating iterative computation for machine learning by orders of magnitude by using main memory more effectively. MapReduce and its variants communicate in an acyclic data flow model that, in practice, requires expensive I/O operations to a distributed file system. Such I/O is highly inefficient for the iterative computation found in many statistical machine learning applications (e.g., logistic regression). For efficiency, Spark introduces resilient distributed datasets (RDDs), a collection of data objects that can be cached in main memory across computational iterations. From a computer architect's perspective, Spark and other future advances present new opportunities. As software frameworks use hardware more efficiently, performance and energy efficiency constraints will shift from datacenter storage and networks to server processors and memory.

Finally, although we have focused our discussion of MapReduce on datacenter-scale implementations, the programming model can be implemented on shared memory multiprocessors as well [136, 153]. At such scale, the run-time system creates threads and distributes them across multiple cores. MapReduce's advantage is programmability and offers performance within competitive range of POSIX threads. Shared memory MapReduce implementations, which may become increasingly important with big memory servers, illustrate the value of clean abstraction layers between the programming model and its implementation.

Benchmarking Strategies. Benchmarking MapReduce for computer architecture research requires a data set and a sequence of Map and Reduce tasks. These tasks might arise from the simple applications that are described in Dean and Ghemawat's original study [35]—counting and sorting specific elements in a large data set. Alternatively, these tasks might arise from a sophisticated software framework built atop MapReduce for database queries or machine learning.

For example, the Spark framework supports a variety of operations on big data, which combine to implement broadly relevant statistical analyses such as logistic regression and alternating least squares.

Within sophisticated software frameworks built atop MapReduce, we can identify specific regions of interest for cycle-level simulation of processors and memory systems. Researchers have successfully simulated Phoenix MapReduce, an implementation for shared memory multiprocessors, and Spark MapReduce, a machine learning framework. At present, simulations model all hardware activity from the master, the worker(s), and other aspects of the run-time system. In the future, more precise measurements may target specific map or reduce tasks, and exclude run-time overheads.

2.5 GRAPH ANALYSIS

Datacenter workloads are more diverse than the applications and frameworks we have described thus far. For example, as frameworks for graph analysis mature, they may present specific demands for hardware. Graphs from datacenter computation tend to be large and sparse. Media companies (e.g., Netflix) may implement collaborative filtering to analyze users' movie preferences and tailor recommendations based on each user's viewing history. Social networking frameworks may implement page rank algorithms to analyze network structure and identify popular individuals or organizations in the graph. Several software frameworks provide clean programming abstractions and capable libraries for such computation.

Programming Models and Run-Time Systems. Low et al. devise the **GraphLab** framework for data mining and machine learning on graphs, a class of computation that does not perform well with traditional MapReduce frameworks [87, 105, 106]. GraphLab provides a programming model and an execution model for asynchronous, dynamic, graph-parallel computation. The programming model specifies a graph and functions on it. A data graph consists of $G = (V, E, D)$ consists of vertices, edges, and data. Update functions modify data associated with a vertex and schedule future updates on other vertices. Formally, an update is a function $f(v, S_v) \rightarrow (S_v, T)$ that consumes a vertex v and its associated scope S_v, which includes both its own data and data from its adjacent vertices, and produces an updated scope S_v and additional tasks T for future iterations in the analysis.

The execution model parallelizes updates on the graph by maintaining a list of tasks to perform on vertices. Workers compute in parallel, retrieving scheduled tasks from the list and placing new ones onto the list. GraphLab's run-time system can schedule tasks for performance by, for example, ordering tasks to maximize data locality and minimize network communication. Task (re-)ordering reveals parallelism and increases performance, but the extent that tasks can be re-ordered is dictated by the consistency model. GraphLab provides several consistency models, from full consistency, which requires concurrent updates on the graph to be at least two vertices apart, to vertex consistency, which allows any set of concurrent updates.

For example, consider the PageRank algorithm within the GraphLab framework. The algorithm recursively defines the rank $R(v)$ of a vertex v:

$$R(v) = \sum_{u|(u,v)\in E} R(u)j/\deg(u),$$

where $R(u)$ is the current rank of neighboring node u and $\deg(u)$ is the degree of that node. Recall that a graph $G = (V, E, D)$ consists of vertices, edges, and user-defined data. Suppose we compute PageRank for web pages, counting links between pages and identifying popular pages. The data graph is obtained directly from web links. Each vertex is a web page, each edge is a link, and each piece of data is the current estimate of a page's rank. GraphLab initializes the PageRank computation by enqueueing a task to compute pagerank for every vertex v in the graph. If the update function changes the rank of a vertex, the ranks for neighboring vertices must also change and the corresponding update tasks enqueue for future computation. Figure 2.8 illustrates the pseudo-code for the PageRank update function.

```
pageRank(v):
  R_old(v) = R(v)
  for each u in Neighbors(v)
     R(v) = R(v) + R(u)
  if |R(v) - R_old(v)| > eps
     queue pageRank(u) for each u in Neighbors(v)
```

Figure 2.8: Pagerank implementation within GraphLab, modified from [105].

GraphLab is a relatively mature starting point for researchers in computer architecture who wish to study graph analytics. But it is only one of several graph analysis frameworks. Satish et al. survey multiple frameworks [142] and find that GraphLab outperforms Giraph [5], a framework built directly atop Hadoop MapReduce. This survey also includes Combinatorial BLAS [26] and SociaLite [143], which are more specialized frameworks for sparse linear algebra and graph queries, respectively. Although these frameworks perform better than GraphLab, they may be less accessible to the lay user. Finally, the Spark framework for high-performance, iterative machine learning atop Hadoop provides extensions for graph analytics in the form of GraphX [6, 167]. Spark's ability to cache data, in memory, across loop iterations may improve performance and reduce communication between workers.

Computational Kernels and Data. Satish et al. identify four representative and recurring computational kernels for graphs—PageRank, Breadth First Search, Triangle Counting, and Collaborative Filtering—and we summarize them here to illustrate computation that is well suited for graph analytic frameworks.

- **Page Rank** identifies popular graph vertices by iteratively calculating the rank PR for each vertex i, which increases with the rank of its neighbors j, such that $PR^{t+1}(i) = \sum_{j|(j,i)\in E} PR^t/\deg(j)$.
- **Breadth First Search** is a classic graph traversal algorithm, which identifies the minimum distance between a starting vertex and other vertices in the graph. The traversal may be top-down, bottom-up, or a hybrid of the two [19]. After initializing distances to infinity, BFS calculates the distance for each vertex i based on the distance calculated for its neighbors j, such that $D(i) = \min_{j|(j,i)\in E} D(j) + 1$.
- **Triangle Counting** enumerates the number of triangles in the graph. A triangle consists of two neighboring vertices that share a third neighbor. Each vertex communicates its list of neighbors with all adjacent vertices. Each recipient calculates the intersection between its neighbors' lists and its own, such that $N = \sum_{i,j,k,i<j<k} E_{ij} \cap E_{jk} \cap E_{ik}$.
- **Collaborative Filtering** fits a model that estimates a user's rating for an item given ratings for other users and items. Suppose the matrix R quantifies the known ratings such that rows enumerate users and columns enumerate items. Collaborative filtering factors the matrix $R = PQ$ by solving the following minimization problem: $\min_{p,q} \sum_{(u,v)\in R}(R_{uv} - p_u^T q_v)^2$, where u and v are indices over users and items.

Programming models and run-time systems, such as GraphLab and GraphX, provide ample support for these and other kernels. These computational kernels are general and applicable to a variety of large networks. Leskovec et al. have collected large datasets, with graphs from social and web networks, as part of the Stanford Network Analysis Project [94, 95]. These graphs vary in size but are invariably sparse. For example, a Google+ social network has 107.6 K nodes and only 13.6 M edges.

2.6 ADDITIONAL CONSIDERATIONS

Multi-tier Workloads. In addition to the software frameworks presented in this chapter, conventional applications for enterprise computing continue to be relevant. The Transaction Processing Performance Council (TPC) provides relevant workloads for online transaction processing [155]. For example, its TPC-C benchmark simulates users as they issue queries and transactions against a database. Transactions are representative of those in electronic commerce and multi-tiered datacenters. In tier one, front-end clients run the TPC-C application and issue transactions. In tier two, back-end servers run the database management system. For experimental infrastructure, tier zero comprises machines that emulate human users and generate a mix of requests to exercise the system.

Similarly, the Standard Performance Evaluation Corporation (SPEC) provides workloads for enterprise computing, such as web-based order processing (SPECjEnterprise [151], SPECjbb [150]), file servers (SPEC SFS [149]). These workloads exercise different parts of a

high-performance, commercial server and are especially relevant when benchmarking cache and memory hierarchy performance. SPECjbb is particularly interesting for its focus on multi-tier, Java business applications. In tier one, a transaction injector issues requests and services to the back-end. In tier two, a back-end server implements business logic. As in TPC benchmarks, tier zero comprises a controller that directs the execution of the workload.

Emerging Workloads. The provision of computing as a service expands the scope of datacenter workloads. Mobile devices increasingly rely on the cloud for personal assistance and contextual analysis. At present, these workloads compute on general-purpose server architectures. In the future, however, these workloads could benefit from hardware acceleration. Hauswald et al. benchmark analytical workloads for intelligent personal assistants (Sirius) and port them to varied platforms such as graphics processors and FPGA-based accelerators [66]. Similarly, as neural networks gain popularity for approximating sophisticated tasks, future datacenters may supply neural networks as a service and accelerate them [31, 65].

General-Purpose Workloads. Moreover, democratized access to cloud computing means that traditional, general-purpose benchmarks should play a role in datacenter research. Elastic cloud computing offers virtual machines to users who can launch arbitrarily diverse computation. Although this computation often deploys open source frameworks for distributed, task-parallel computing, it may also deploy more conventional applications such as those in the SPEC CPU [148], SPEC OMP, PARSEC [20], and SPLASH [163] benchmark suites.

Regions of Interest. This chapter presented software frameworks for datacenter computing—web search, map reduce, memory caching, and graph analysis. Computer architects should be wary of treating these frameworks as benchmarks. Rather, architects should identify computational kernels and regions of interest within the frameworks, whether they be specific queries in web search, a machine learning library atop map reduce, a specific operation on a key-value store, or a particular analysis on a social network. When benchmarking datacenter workloads, input data and the computation on that data are as important, if not more so, than the programming model and run-time system for that computation.

A framework's run-time system is often conflated with the software application of interest. For example, Hadoop and Spark support machine learning computation, but the framework itself also exercises the hardware, with computation to split input data, route intermediate results, and sort data at the reduce workers. A computer architect who benchmarks a Hadoop job might capture this computation for management in addition to the specific computation of interest. More precise measurements require hardware counter APIs or simulation checkpoints that are flexible enough to demarcate regions of interest in the application.