选择的书籍是：



**1. 总说内容**

全书分为三大部分：

第一部分，主要讨论有关增强数据密集型应用系统所需的若干基本原则。首先开篇第1章即瞄准目标：可靠性、可扩展性与可维护性，如何认识这些问题以及如何达成目标。第2章我们比较了多种不同的数据模型和查询语言，讨论各自的适用场景。接下来第3章主要针对存储引擎，即数据库是如何安排磁盘结构从而提高检索效率。第4章转向数据编码（序列化）方面，包括常见模式的演化历程。

第二部分，我们将从单机的数据存储转向跨机器的分布式系统，这是扩展性的重要一步，但随之而来的是各种挑战。所以将依次讨论数据远程复制（第5章）、数据分区（第6章）以及事务（第7章）。接下来的第8章包括分布式系统的更多细节，以及分布式环境如何达成一致性与共识（第9章）。

第三部分，主要针对产生派生数据的系统，所谓派生数据主要指在异构系统中，如果无法用一个数据源来解决所有问题，那么一种自然的方式就是集成多个不同的数据库、缓存模块以及索引模块等。首先第10章以批处理开始来处理派生数据，紧接着第11章采用流式处理。第12章总结之前介绍的多种技术，并分析讨论未来构建可靠、可扩展和可维护应用系统可能的新方向或方法。

**2. 分章节论述**

**第1章 可靠、 可扩展与可维护的应用系统**

**关键概念：**数据系统

**可靠性**：即使发生了某些错误，系统仍可以继续正常工作。有硬件故障，软件错误，人为失误；

**可靠性是很重要的**，因为可靠性绝不仅仅针对的是核电站和空中交管软件之类的系统，很多应用都需要可靠工作。商业软件中的错误会导致效率下降（如数据报告错误，甚至带来法律风险），电子商务网站的暂停会对营收和声誉带来巨大损失。

**可扩展性**：是用来描述系统应对负载增加能力的术语；

**需要关注描述负载**，具体来说，负载可以用称为负载参数的若干数字来描述。参数的最佳选择取决于系统的体系结构，它可能是Web服务器的每秒请求处理次数，数据库中写入的比例，聊天室的同时活动用户数量，缓存命中率等。用twitter的例子来说明了系统数据架构与负载吞吐量密切相关。

**针对负载**，我们可以对千特定应用来说，扩展能力好的架构通常会做出某些假设，然后有针对性地优化设计，如哪些操作是最频繁的，哪些负载是少数情况。如果这些假设最终发现是错误的，那么可扩展性的努力就白费了，甚至会出现与设计预期完全相反的情况。对千早期的初创公司或者尚未定型的产品，快速迭代推出产品功能往往比投入精力来应对不可知的扩展性更为重要。可扩展架构通常都是从通用模块逐步构建而来，背后往往有规律可循。

可维护性：软件的大部分成本并不在最初的开发阶段，而是在千整个生命周期内持续的投入，这包括维护与缺陷修复，监控系统来保持正常运行、故障排查、适配新平台、搭配新场景、技术缺陷的完善以及增加新功能等。

但是，换个角度，我们可以从软件设计时开始考虑，尽可能较少维护期间的麻烦，甚至避免造出容易过期的系统。为此，我们将特别关注**软件系统的三个设计原则：**

**可运维性**：方便运营团队来保持系统平稳运行。

**简单性**：简化系统复杂性，使新工程师能够轻松理解系统。注意这与用户界面的简单性并不一样。

**可演化性**：后续工程师能够轻松地对系统进行改进，并根据需求变化将其适配到非典型场景，也称为可延伸性、易修改性或可塑性

**小结：**这一章我们探讨了一些关千数据密集型应用的基本原则，这些原则将指导如何阅读本书的其余部分。一个应用必须完成预期的多种需求，主要包括功能性需求（即应该做什么，比如各种存储、检索、搜索和处理数据）和一些非功能性需求（即常规特性、例如安全性、可靠性、合规性、可伸缩性兼容性和可维护性）。本章我们着重梳理讨论了可靠性、可扩展性和可维护性。可靠性意味着即使发生故障，系统也可以正常工作。故障包括硬件（通常是随机的，不相关的）、软件（缺陷通常是系统的，更加难以处理）以及入为（总是很难避免时不时会出错）方面。容错技术可以很好地隐藏某种类型故障，避免影响最终用户。可扩展性是指负载增加时，有效保持系统性能的相关技术策略。为了讨论可扩展性，我们首先探讨了如何定釐描述负载和性能。简单地以Twitter浏览时间线为例描述负载，并将响应时间百分位数作为衡批性能的有效方式。对于可扩展的系统，增加处理能力的同时，还可以在高负载情况下持续保持系统的高可靠性。可维护性则意味着许多方面，但究其本质是为了让工程和运营团队更为轻松。良好的抽象可以帮助降低复杂性，并使系统更易于修改和适配新场景。良好的可操作性意味着对系统健康状况有良好的可观测性和有效的管理方法。然而知易行难，使应用程序可靠、可扩展或可维护并不容易。考虑到一些重要的模式和技术在很多不同应用中普遍适用，在接下来的几章中，我们就一些数据密集系统例子，分析它们如何实现上述这些目标。

**第2章 数据模型与查询语言**

**关键概念：**

**关系模型与文档模型**：现在最著名的数据模型可能是SQL,它基于Edgar Codd千1970年提出的关系模型lIJ:数据被组织成关系(relations),在SQL中称为表(table),其中每个关系都是元组(tuples)的无序集合（在SQL中称为行）。

**关系数据库与文档数据库现状**在**比较**关系数据库与文档数据库时，需要考虑很多方面的差异，包括它们的容错性和并发处理。本章将只关注数据模型中的差异。

支持文档数据模型的主要论点是模式灵活性，由千局部性而带来较好的性能，对千某些应用来说，它更接近千应用程序所使用的数据结构。关系模型则强在联结操作、多对一和多对多关系更简洁的表达上，与文档模型抗衡。

**数据查询语言**：Web上的声明式查询；MapReduce查询

**图状数据模型**：多对多关系是不同数据模型之间的重要区别特征。如果数据大多是一对多关系（树结构数据）或者记录之间没有关系，那么文档模型是最合适的。但是，如果多对多的关系在数据中很常见呢？关系模型能够处理简单的多对多关系，但是随着数据之间的关联越来越复杂，将数据建模转化为图模型会更加自然。

属性图

**Cypher查询语言**：Cypher是一种用千属性图的声明式查询语言，最早为Neo4j图形数据库而创建；

**SQL里的图查询**；

关系型数据库：强调SQL数据库的经典设计和ACID属性。适用于事务处理和复杂查询的应用，数据按表格格式组织。

非关系型数据库：包括文档型数据库（如MongoDB）、列式存储（如HBase）和键值存储（如Redis）。这些数据库在某些场景下提供更高的灵活性和扩展性，特别适用于大规模的数据处理和分布式环境。

选择存储系统：选择合适的存储系统取决于数据模型、事务要求和可扩展性需求。需要根据应用的具体需求来决定采用何种存储系统（关系型还是非关系型）。

**第3章 数据存储与检索**

**关键概念：**

数据库核心-数据结构：

哈希索引：key-value存储与大多数编程语言所内置的字典结构非常相似，通常采用hash map (或者hash table,哈希表）来实现。

SSTables和LSM-Tree

构建和维护SSTables

其他索引结构：二级索引

事务处理与分析处理

**数据仓库**：

**列式存储**：面向列存储的想法很简单：不要将一行中的所有值存储在一起，而是将每列中的所有值存储在一起。如果每个列存储在一个单独的文件中，查询只需要读取和解析在该查询中使用的那些列，这可以节省大量的工作。

**列压缩**：除了仅从磁盘中加载查询所需的列之外，还可以通过压缩数据来进一步降低对磁盘吞吐量的要求。

**列存储的写操作**：

**小结**：概括来讲，存储引擎分为两大类：针对事务处理(OLTP)优化的架构，以及针对分析型(OLAP)的优化架构。它们典型的访问模式存在很大差异： • OLTP系统通常面向用户，这意味着它们可能收到大量的请求。为了处理负载，应用程序通常在每个查询中只涉及少显的记录。应用程序基千某种键来请求记录，而存储引擎使用索引来查找所请求键的数据。磁盘寻道时间往往是瓶颈。 • 由于不是直接面对最终用户，数据仓库和类似的分析型系统相对并不太广为入知，它们主要由业务分析师使用。处理的查询请求数目远低于OLTP系统，但每个查询通常要求非常苛刻，需要在短时间内扫描数百万条记录。磁盘带宽（不是寻道时间）通常是瓶颈，而面向列的存储对千这种工作负载成为日益流行的解决方案。在OLTP方面，由两个主要流派的存储引擎： • 日志结构流派，它只允许追加式更新文件和删除过时的文件，但不会修改己写入的文件。 BitCask、 SSTables、 LSM-tree、 LevelDB、 Cassandra、 HBase、 Lucene 等属于此类。数据存储与检索 I 101 • 原地更新流派，将磁盘视为可以覆盖的一组固定大小的页。 B-tree是这一哲学的最典型代表，它已用千所有主要的关系数据库，以及大輩的非关系数据库。日志结构的存储引擎是一个相对较新的方案。其关键思想是系统地将磁盘上随机访问写入转为顺序写入，由千硬盘驱动器和SSD的性能特性，可以实现更高的写入吞吐最。此外，简要介绍了一些更复杂的索引结构，以及为全内存而优化的数据库。然后，从存储引擎的内部间接地探索了典型数据仓库的总体架构。由此说明为什么分析工作负载与OLTP如此不同：当查询需要在大扯行中顺序扣描时，索引的关联性就会显著降低。相反，最重要的是非常紧凑地编码数据，以尽晕减少磁盘读取的数据量。我们讨论了列存储如何帮助实现这一目标。作为应用开发入员，掌握更多有关存储引擎内部的知识，可以更好地了解哪种工具最适合你的具体应用。如果还需要进一步调整数据库的可调参数，这些理解还可以帮助开发者正确评估调高或调低参数所带来的影响。

**第4章 数据编码与演化**

数据编码格式：

程序通常使用（至少）两种不同的数据表示形式：在内存中，数据保存在对象、结构体、列表、数组、哈希表和树等结构中。这些数据结构针对CPU的高效访问和操作进行了优化（通常使用指针）。将数据写入文件或通过网络发送时，必须将其编码为某种自包含的字节序列（例如JSON文档）。由于指针对其他进程没有意义，所以这个字节序列表示看起来与内存中使用的数据结构大不一样。

语言特定的格式：

JSON、 XML与二进制变体：

数据类型和模式演化：

Avro：Apache Avro是另外一种二进制编码格式

写模式与读模式：有了Avro, 当应用程序想要对某些数据进行编码（例如将其写入文件或数据库，以及通过网络发送）时，它使用所知道的模式的任何版本来编码数据；

**数据流模式**：通过数据库；通过服务调用；通过异步信息传递；

基千数据库的数据流：

归档存储：

基千消息传递的数据流：

消息代理：

小结：

本章，我们研究了将内存数据结构转换为网络或磁盘上字节流的多种方法。我们看到这些编码的细节不仅影响其效率，更重要的是还影响应用程序的体系结构和部署时的支持选项。特别地，许多服务需要支持滚动升级，即每次将新版本的服务逐步部署到几个节点，而不是同时部署到所有节点。滚动升级允许在不停机的情况下发布新版本的服务（因此鼓励频繁地发布小版本而不是大版本），并降低部署风险（允许错误版本在影响大釐用户之前检测并回滚）。这些特性非常有利于应用程序的演化和更改。在滚动升级期间，或者由千各种其他原因，必须假设不同的节点正在运行应用代码的不同版本。因此，在系统内流动的所有数据都以提供向后兼容性（新代码可以读取旧数据）和向前兼容性（旧代码可以读取新数据）的方式进行编码显得非常重要。

本章还讨论了多种数据编码格式及其兼容性情况：

• 编程语言特定的编码仅限千某一种编程语言，往往无法提供向前和向后兼容性。

• JSON、 XML和CSV等文本格式非常普遍，其兼容性取决于你如何使用他们。它们有可选的模式语言，这有时是有用的，有时却是一个障碍。这些格式对某些数据类型的支持有些模糊，必须小心处理数字和二进制字符串等问题。

• 像Thrift、 Protocol Buffers和Avro这样的二进制的模式驱动格式，支持使用清晰定义的向前和向后兼容性语义进行紧凑、高效的编码。这些模式对千静态类型语言中的文档和代码生成非常有用。然而，它们有一个缺点，即只有在数据解码后才是人类可读的。

我们还讨论了数据流的几种模型，说明了数据编码在不同场景下非常重要：

• 数据库，其中写入数据库的进程对数据进行编码，而读取数据库的进程对数据进行解码。 • RPC和RESTAPI,其中客户端对请求进行编码，服务器对请求进行解码并对响应进行编码，客户端最终对响应进行解码异步消息传递（使用消息代理或Actor) ,节点之间通过互相发送消息进行通信，消息由发送者编码并由接收者解码。

最后，我们可以得出这样的结论，只要稍加小心，向后／向前兼容性和滚动升级是完全可以实现的。

**第5章 数据复制**

复制主要指通过互联网络在多台机器上保存相同数据的副本。正如第二部分开头所介绍的， 通过数据复制方案，人们通常希望达到以下目的：

• 使数据在地理位置上更接近用户，从而降低访问延迟。

• 当部分组件出现故障，系统依然可以继续工作，从而提高可用性。

• 扩展至多台机器以同时提供数据访问服务，从而提高读吞吐量。

主节点与从节点：

同步复制与异步复制：同步复制的优点是，一且向用户确认，从节点可以明确保证完成了与主节点的更新同步，数据已经处千最新版本。万一主节点发生故障，总是可以在从节点继续访问最新数据。缺点则是，如果同步的从节点无法完成确认（例如由千从节点发生崩溃，或者网络故障，或任何其他原因），写入就不能视为成功。主节点会阻塞其后所有的写操作，直到同步副本确认完成。

配置新的从节点：‘

读自己的写；单调读；前缀一致读

复制滞后问题：

多主节点复制：

无主节点复制：

小结：小结

本章， 我们详细探讨了复制相关的话题，复制或者多副本技术主要服务于以下目的：

• 高可用性：即使某台机器（或多台机器，或整个数据中心）出现故障，系统也能保持正常运行。

• 连接断开与容错：允许应用程序在出现网络中断时继续工作。

• 低延迟：将数据放置在距离用户较近的地方，从而实现更快地交互。

• 可扩展性：采用多副本读取，大幅提高系统读操作的吞吐量。

在多台机器上保存多份相同的数据副本，看似只是个很简单的目标，但事实上复制技术是一个非常烧脑的问题。需要仔细考虑并发以及所有可能出错的环节，并小心处理故障之后的各种情形。最最基本的，要处理好节点不可用与网络中断问题，这里甚至还没考虑一些更隐蔽的失效场景，例如由于软件bug而导致的无提示的数据损坏。

我们主要讨论了三种多副本方案：

主从复制：所有的客户端写入操作都发送到某一个节点（主节点），由该节点负责将数据更改事件发送到其他副本（从节点）。每个副本都可以接收读请求，但内容可能是过期值。

多主节点复制：系统存在多个主节点，每个都可以接收写请求，客户端将写请求发送到其中的一个主节点上，由该主节点负责将数据更改事件同步到其他主节点和自己的从节点。

无主节点复制：客户端将写请求发送到多个节点上，读取时从多个节点上并行读取，以此检测和纠正某些过期数据。

每种方法都有其优点和缺点。主从复制非常流行，主要是因为它很容易理解，也不需要担心冲突问题。而万一出现节点失效、网络中断或者延迟抖动等情况，多主节点和无主节点复制方案会更加可靠，不过背后的代价则是系统的复杂性和弱一致性保证。

复制可以是同步的，也可以是异步的，而一且发生故障，二者的表现差异会对系统行为产生深远的影响。在系统稳定状态下异步复制性能优秀，但仍须认真考虑一旦出现复制滞后和节点失效两种场景会导致何种影响。万一某个主节点发生故障，而一个异步更新的从节点被提升为新的主节点，要意识到最新确认的数据可能有丢失的风险。

我们还分析了由千复制滞后所引起的一些奇怪效应，并讨论了以下一致性模型，来帮助应用程序处理复制滞后；

写后读一致性保证用户总能看到自己所提交的最新数据。

单调读用户在某个时间点读到数据之后，保证此后不会出现比该时间点更早的数据。

前缀一致读保证数据之间的因果关系，例如，总是以正确的顺序先读取间题，然后看到回答。

最后，我们讨论了多主节点和无主节点复制方案所引入的并发问题。即由千多个写可能同时发生，继而可能产生冲突。为此，我们研究了一个算法使得数据库系统可以判定某操作是否发生在另一个操作之前，或者是同时发生。接下来，探讨采用合并并发更新值的方法来解决冲突。

**第6章 数据分区**

数据分区与数据复制：

键-值数据的分区：

分区与二级索引：

分区再平衡：

请求路由：

小结：

本章，我们探讨了将大规模数据集划分成更小子集的多种方法。数据量如果太大，单台机器进行存储和处理就会成为瓶颈，因此需要引入数据分区机制。分区的目地是通过多台机器均匀分布数据和查询负载，避免出现热点。这需要选择合适的数据分区方案，在节点添加或删除时重新动态平衡分区。

我们讨论了两种主要的分区方法：

• 基千关键字区间的分区。先对关键字进行排序，每个分区只负责一段包含最小到最大关键字范围的一段关键字。对关键字排序的优点是可以支持高效的区间查询，但是如果应用程序经常访问与排序一致的某段关键字，就会存在热点的风险。采用这种方法，当分区太大时，通常将其分裂为两个子区间，从而动态地再平衡分区。

• 哈希分区。将哈希函数作用千每个关键字，每个分区负责一定范围的哈希值。这种方法打破了原关键字的顺序关系，它的区间查询效率比较低，但可以更均匀地分配负载。采用哈希分区时，通常事先创建好足够多（但固定数量）的分区，让每个节点承担多个分区，当添加或删除节点时将某些分区从一个节点迁移到另一个节点，也可以支持动态分区。

混合上述两种基本方法也是可行的，例如使用复合键：键的一部分来标识分区，而另一部分来记录排序后的顺序。我们还讨论了分区与二级索引，二级索引也需要进行分区，有两种方法：

• 基于文档来分区二级索引（本地索引）。二级索引存储在与关键字相同的分区中，这意味着写入时我们只需要更新一个分区，但缺点是读取二级索引时需要在所有分区上执行scatter/gather。

• 基千词条来分区二级索引（全局索引）。它是基千索引的值而进行的独立分区。二级索引中的条目可能包含来自关键字的多个分区里的记录。在写入时，不得不更新二级索引的多个分区；但读取时，则可以从单个分区直接快速提取数据。最后，我们讨论了如何将查询请求路由到正确的分区，包括简单的分区感知负载均衡器，以及复杂的并行查询执行引擎。