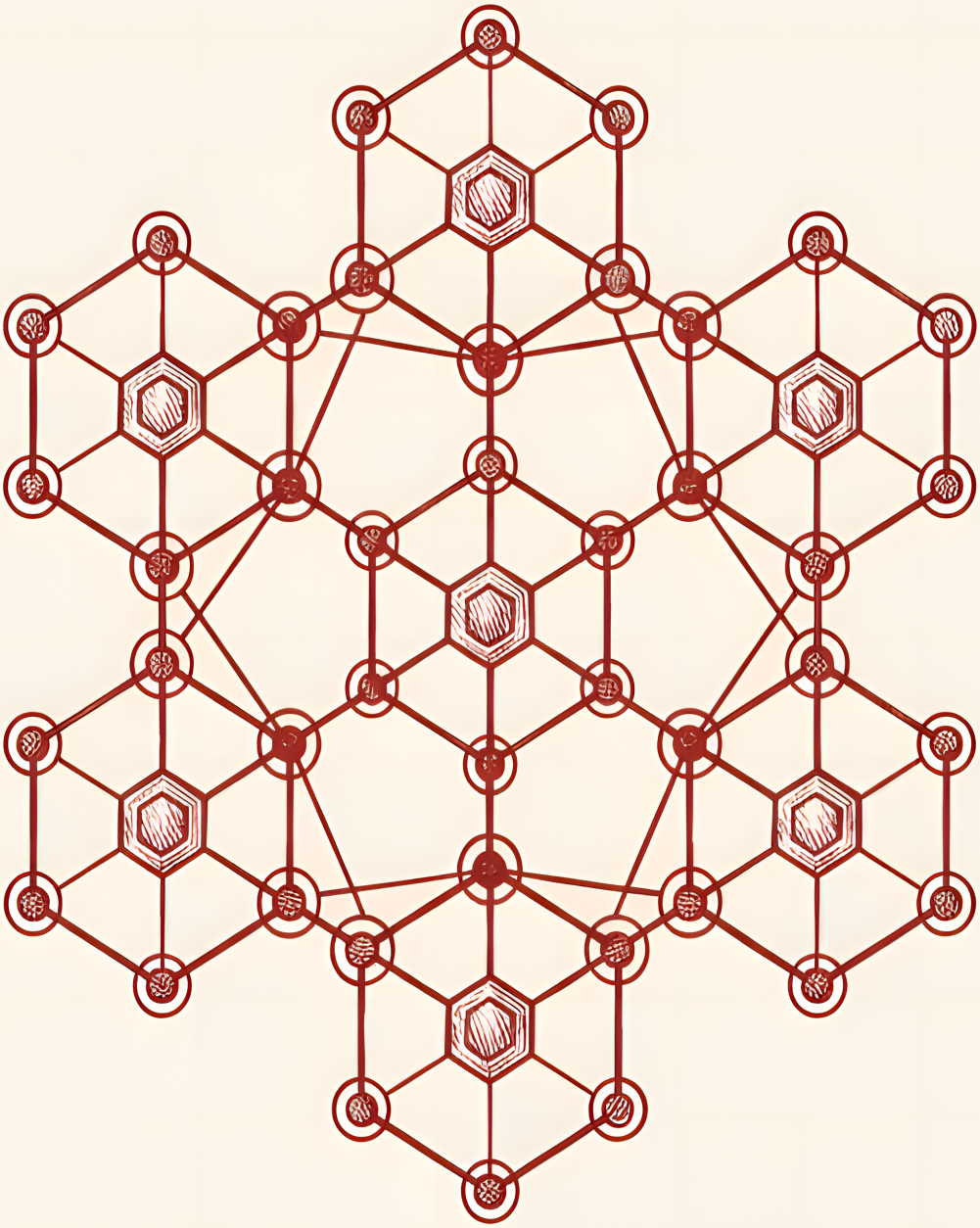
《木鸟杂记》出品

DDIA 逐章精读

Designing Data-Intensive Applications



ddia.qtmuniao.com

DDIA 逐章精读

木鸟杂记

DDIA 木鸟序

早就在知乎上听人推荐过这本书，之前偶尔翻过第二部分 Replication 和 Partition 两章，感觉讲的还不错，但对于做分布式存储和数据库人员来说，稍微有点泛泛而谈。初次见面，只觉其好而不神。

今年（2022）建了一个分布式系统和数据库爱好者的微信群，如果不组织大家做点什么，微信群迟早沦为僵尸群，“好”一点的可能变成吹水群，但这显然不是我的初衷。念念不忘，必有回响，心里又掠过了这本书，更兼在北美华人群里见过本书被分享过，还挺受欢迎。一拍大腿，就是他了：组织大家一块过一遍 DDIA 把，于是有了这个读书会。

一精读，便不可收拾。这才发现了此书之妙：数据系统方方面面，知识线索极为庞杂，本书却能以极为合理的脉络将其勾连在一起，形成环环相扣的知识体系。至于泛泛而谈？自然是真真香：这本就是框架式书籍，更何况，每章附录列出参考引用、论文列表，都是非常经典的深入阅读材料。

那么它的组织妙在何处？

现在（20220425）刚精读到第五章，仅以现在认识来简单聊聊我的看法。

第一，全书分三个部分。分别是单机，多机，衍生。从单机开始聊数据系统，可以摒除分布式庞杂理论的影响，专注在数据系统本身相关理论；到第二个部分放开单机限制，着重讲将数据系统扩展到多机所面临的问题和一般解决方案；最后一部分笔锋一转，着眼数据处理，以数据系统视角看，无非是一个数据集的变换，也即数据的派生。三个部分，层层递进，相互正交。这种行文思路，正是大型工业代码组织思路：将复杂度拆解到几个正交、但又相互连结的模块，从而使每个部分都相对内聚而简洁。

第二，具体到第一部分，开篇就给了三个总纲式的“心法”：可靠性、可伸缩性、可维护性。然后，从上到下，由离用户最近的数据模型（比如关系模型）和查询语言（比如 SQL），到稍微底层一点的存储引擎（比如 B+ tree 和 lsm tree）和查询引擎，再到最底层的编码（数据结构的降维）和演化，层层下探，令人拍案叫绝。我之前工作和兴趣之余所接触到的零碎知识，至此百川入海，万法归一。

第三，具体到每一章，也是节节递进，读起来无比丝滑。比如第三章，在讲存储引擎时，从一个仅由两个 shell 函数组成的“kv 引擎”起，到一个简单的日志结构的存储（Bitcask），再到经典的 LSM-Tree。这又是工程中惯用思路：从一个最小可用原型开始，不断增加需求、解决瓶颈，最终得到一个工业可用的存储引擎。

我们如何认识世界？不断归纳然后演绎。我们如何处理复杂度？不断拆解然后勾连。将汪洋恣肆的复杂度合理疏导，渐次递进，本书无愧神书！

> 如果你也对此书感兴趣，但苦于无人交流，欢迎参加我们的读书会：https://docs.qq.com/sheet/DWHFzdk5lUWx4UWJq?tab=BB08J2。里面有往期分享录屏、进群方式和之后安排，共勉~

DDIA 逐章精读（一）：可靠、可扩展、可维护

本书为什么以数据系统为主题

数据系统（data system）是一种模糊的统称。在信息社会中，一切皆可信息化，或者，某种程度上来说——数字化。这些数据的采集、存储和使用，是构成信息社会的基础。我们常见的绝大部分应用背后都有一套数据系统支撑，比如微信、京东、微博等等。

因此，作为 IT 从业人员，有必要系统性的了解一下现代的、分布式的数据系统。学习本书，能够学习到数据系统的背后的原理、了解其常见的实践、进而将其应用到我们工作的系统设计中。

常见的数据系统有哪些

- 存储数据，以便之后再次使用——数据库

- 记住一些非常“重”的操作结果，方便之后加快读取速度——缓存
- 允许用户以各种关键字搜索、以各种条件过滤数据——搜索引擎
- 源源不断的产生数据、并发送给其他进程进行处理——流式处理
- 定期处理累积的大量数据——批处理
- 进行消息的传送与分发——消息队列

这些概念如此耳熟能详以至于我们在设计系统时拿来就用，而不用去想其实现细节，更不用从头进行实现。当然，这也侧面说明这些概念抽象的多么成功。

数据系统的日益复杂化

但这些年来，随着应用需求的进一步复杂化，出现了很多新型的数据采集、存储和处理系统，它们不拘泥于单一的功能，也难以生硬的归到某个类别。随便举几个例子：

1. Kafka：可以作为存储持久化一段时间日志数据、可以作为消息队列对数据进行分发、可以作为流式处理组件对数据反复蒸馏等等。
2. Spark：可以对数据进行批处理、也可以化小批为流，对数据进行流式处理。
3. Redis：可以作为缓存加速对数据库的访问、也可以作为事件中心对消息的发布订阅。

我们面临一个新的场景，以某种组合使用这些组件时，在某种程度上，便是创立了一个新的数据系统。书中给了一个常见的对用户数据进行采集、存储、查询、旁路等操作的数据系统示例。从其示意图中可以看到各种 Web Services 的影子。

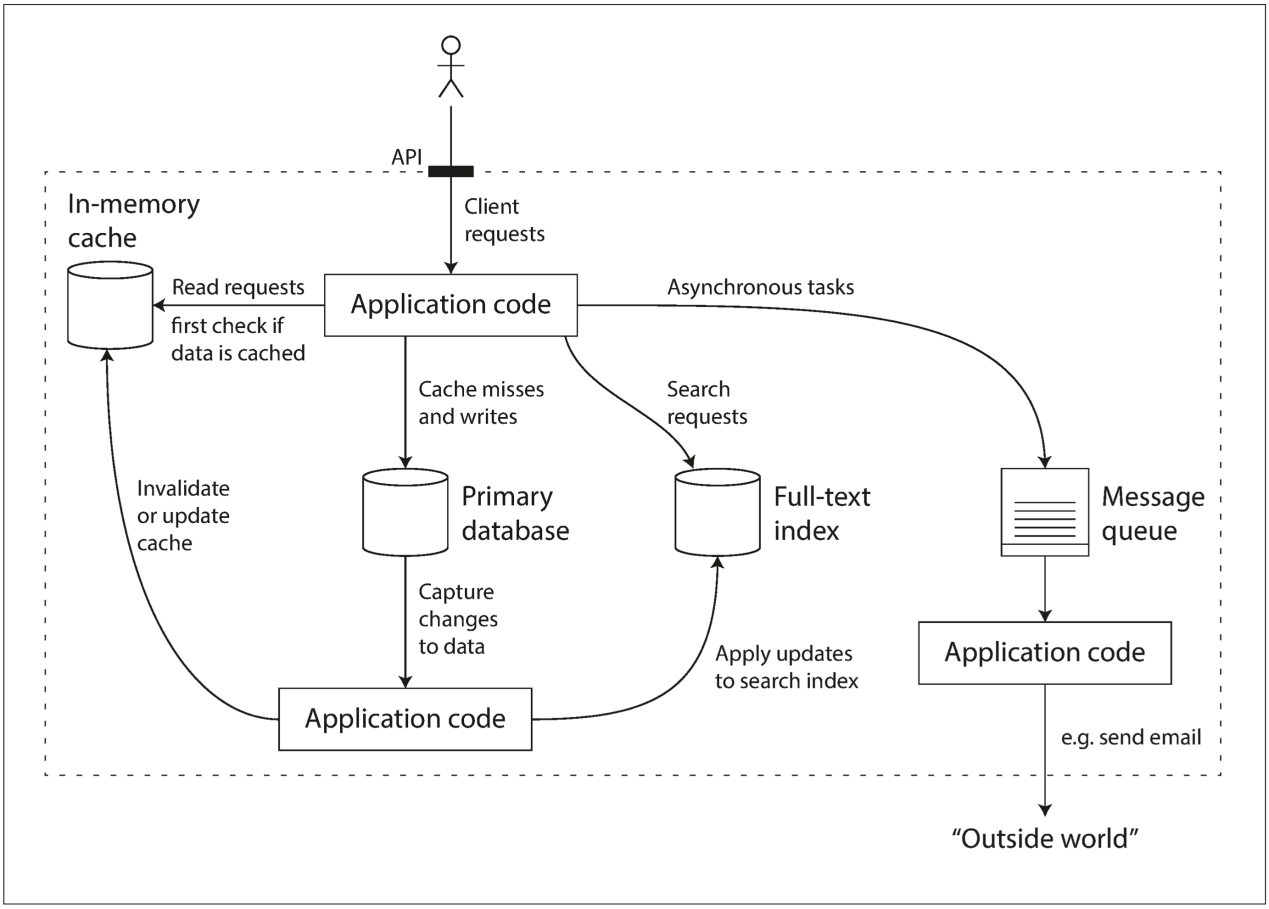


Figure 1-1. One possible architecture for a data system that combines several components.

但就这么一个小系统，在设计时，就可以有很多取舍：

1. 使用何种缓存策略？是旁路还是写穿透？
2. 部分组件机器出现问题时，是保证可用性还是保证一致性？
3. 当机器一时难以恢复，如何保证数据的正确性和完整性？
4. 当负载增加时，是增加机器还是提升单机性能？
5. 设计对外的 API 时，是力求简洁还是追求强大？

因此，有必要从根本上思考下如何评价一个好数据系统，如何构建一个好的数据系统，有哪些可以遵循的设计模式？有哪些通常需要考虑的方面？

书中用了三个词来回答：可靠性（Reliability）、可扩展性（Scalability）、可维护性（Maintainability）

可靠性

如何衡量可靠性?

- 功能上
- 正常情况下，应用行为满足 API 给出的行为
- 在用户误输入/误操作时，能够正常处理
- 性能上 在给定硬件和数据量下，能够满足承诺的性能指标。
- 安全上 能够阻止未授权、恶意破坏。

可用性也是可靠性的一个侧面，云服务通常以多少个 9 来衡量可用性。

两个易混淆的概念：Fault（系统出现问题） 和 Failure（系统不能提供服务）

不能进行 Fault-tolerance 的系统，积累的 fault 多了，就很容易 failure。

如何预防？混沌测试：如 Netflix 的 chaosmonkey。

硬件故障

在一个大型数据中心中，这是常态：

1. 网络抖动、不通
2. 硬盘老化坏道
3. 内存故障
4. 机器过热导致 CPU 出问题
5. 机房断电

数据系统中常见的需要考虑的硬件指标：

- MTTF mean time to failure 单块盘 平均故障时间 5 ~10 年，如果你有 1w+ 硬盘，则均匀期望下，每天都有坏盘出现。当然事实是硬盘会一波一波坏。

解决办法，增加冗余度：机房多路供电，双网络等等。

对于数据：

- 单机：可以做 RAID 冗余。如：EC 编码。
- 多机：多副本 或 EC 编码。

软件错误

相比硬件故障的随机性，软件错误的相关性更高：

1. 不能处理特定输入，导致系统崩溃。
2. 失控进程（如循环未释放资源）耗尽 CPU、内存、网络资源。
3. 系统依赖组件变慢甚至无响应。
4. 级联故障。

在设计软件时，我们通常有一些环境假设，和一些隐性约束。随着时间的推移、系统的持续运行，如果这些假设不能够继续被满足；如果这些约束被后面维护者增加功能时所破坏；都有可能让一开始正常运行的系统，突然崩溃。

人为问题

系统中最不稳定的是人，因此要在设计层面尽可能消除人对系统影响。依据软件的生命周期，分几个阶段来考虑：

- 设计编码
- 尽可能消除所有不必要的假设，提供合理的抽象，仔细设计 API
- 进程间进行隔离，对尤其容易出错的模块使用沙箱机制
- 对服务依赖进行熔断设计
- 测试阶段
- 尽可能引入第三方成员测试，尽量将测试平台自动化
- 单元测试、集成测试、e2e 测试、混沌测试
- 运行阶段
- 详细的仪表盘
- 持续自检
- 报警机制
- 问题预案

- 针对组织
- 科学的培训和管理

可靠性有多重要？

事关用户数据安全，事关企业声誉，企业存活和做大的基石。

可扩展性

可扩展性，即系统应对负载增长的能力。它很重要，但在实践中又很难做好，因为存在一个基本矛盾：只有能存活下来的产品才有资格谈扩展，而过早为扩展设计往往活不下去。

但仍是可以了解一些基本的概念，来应对可能会暴增的负载。

衡量负载

应对负载之前，要先找到合适的方法来衡量负载，如负载参数（load parameters）：

- 应用日活月活
- 每秒向 Web 服务器发出的请求
- 数据库中的读写比率
- 聊天室中同时活跃的用户数量

书中以 Twitter 2012 年 11 月披露的信息为例进行了说明：

1. 识别主营业务：发布推文、首页 Feed 流。
2. 确定其请求量级：发布推文（平均 4.6k 请求/秒，峰值超过 12k 请求/秒），查看其他人推文（300k 请求/秒）

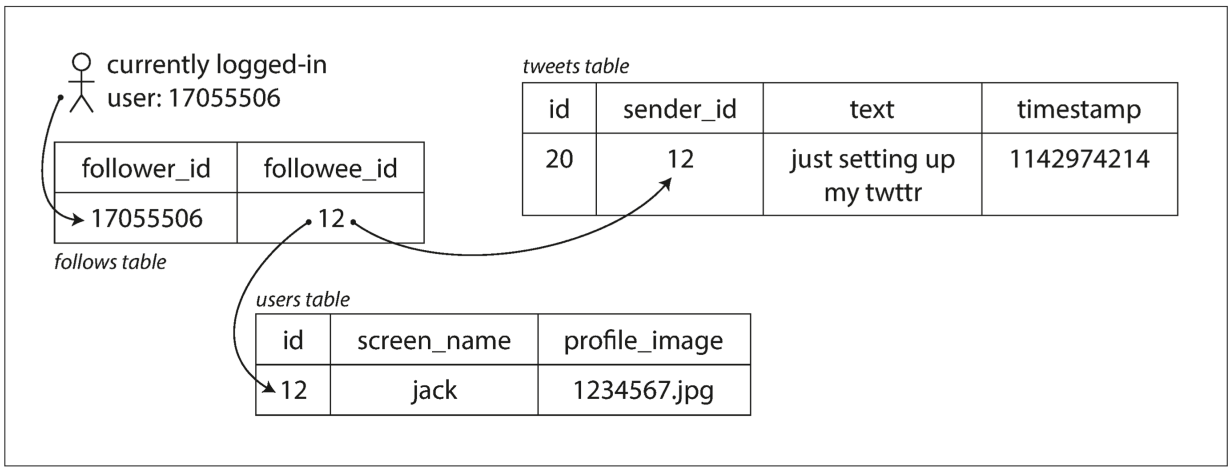


Figure 1-2. Simple relational schema for implementing a Twitter home timeline.

单就这个数据量级来说，无论怎么设计都问题不大。但 Twitter 需要根据用户之间的关注与被关注关系来对数据进行多次处理。常见的有推拉两种方式：

1. 拉。每个人查看其首页 Feed 流时，从数据库现拉取所有关注用户推文，合并后呈现。
2. 推。为每个用户保存一个 Feed 流视图，当用户发推文时，将其插入所有关注者 Feed 流视图中。

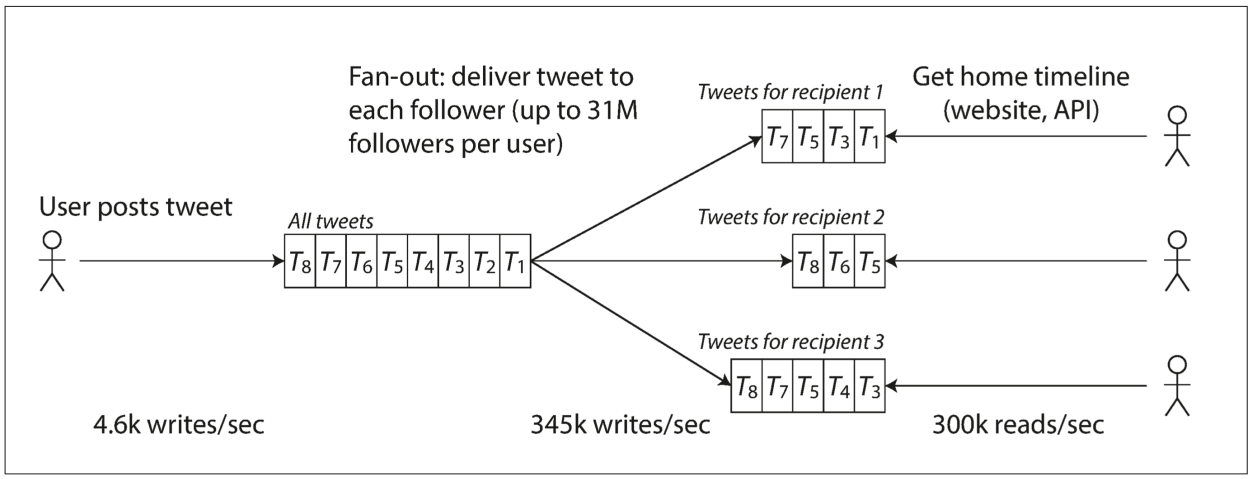


Figure 1-3. Twitter's data pipeline for delivering tweets to followers, with load parameters as of November 2012 [16].

前者是 Lazy 的，用户只有查看时才会去拉取，不会有无效计算和请求，但每次需要现算，呈现速度较慢。而且流量一大也扛不住。

后者事先算出视图，而不管用户看不看，呈现速度较快，但会引入很多无效请求。

最终，使用的是一种推拉结合的方式，这也是外国一道经典的系统设计考题。

描述性能

注意和系统负载区分，系统负载是从用户视角来审视系统，是一种客观指标。而系统性能则是描述的系统的一种实际能力。比如：

1. 吞吐量（throughput）：每秒可以处理的单位数据量，通常记为 QPS。
2. 响应时间（response time）：从用户侧观察到的发出请求到收到回复的时间。
3. 延迟（latency）：日常中，延迟经常和响应时间混用指代响应时间；但严格来说，延迟只是指请求过程中排队等休眠时间，虽然其在响应时间中一般占大头；但只有我们把请求真正处理耗时认为是瞬时，延迟才能等同于响应时间。

响应时间通常以百分位点来衡量，比如 p95，p99 和 p999，它们意味着 95%，99%或 99.9%的请求都能在该阈值内完成。在实际中，通常使用滑动窗口滚动计算最近一段时间的响应时间分布，并通常以折线图或者柱状图进行呈现。

应对负载

在有了描述和定义负载、性能的手段之后，终于来到正题，如何应对负载的不断增长，即使系统具有可扩展性。

1. 纵向扩展（scaling up）或 垂直扩展（vertical scaling）：换具有更强大性能的机器。e.g. 大型机机器学习训练。
2. 横向扩展（scaling out）或 水平扩展（horizontal scaling）：“并联”很多廉价机，分摊负载。e.g. 马斯克造火箭。

负载扩展的两种方式：

- 自动 如果负载不好预测且多变，则自动较好。坏处在于不易跟踪负载，容易抖动，造成资源浪费。
- 手动 如果负载容易预测且不长变化，最好手动。设计简单，且不容易出错。

针对不同应用场景：

首先，如果规模很小，尽量还是用性能好一点的机器，可以省去很多麻烦。

其次，可以上云，利用云的可扩展性。甚至如 Snowflake 等基础服务提供商也是 All In 云原生。

最后，实在不行再考虑自行设计可扩展的分布式架构。

两种服务类型：

- 无状态服务 比较简单，多台机器，外层罩一个 gateway 就行。
- 有状态服务 根据需求场景，如读写负载、存储量级、数据复杂度、响应时间、访问模式，来进行取舍，设计合乎需求的架构。

不可能啥都要，没有万金油架构！但同时：万变不离其宗，组成不同架构的原子设计模式是有限的，这也是本书稍后要论述的重点。

可维护性

从软件的整个生命周期来看，维护阶段绝对占大头。

但大部分人都喜欢挖坑，不喜欢填坑。因此有必要，在刚开就把坑开的足够好。有三个原则：

- 可维护性（Operability） 便于运维团队无痛接手。
- 简洁性（Simplicity） 便于新手开发平滑上手：这需要一个合理的抽象，并尽量消除各种复杂度。如，层次化抽象。
- 可演化性（Evolvability） 便于后面需求快速适配：避免耦合过紧，将代码绑定到某种实现上。也称为可扩展性（extensibility），可修改性（modifiability） 或可塑性（plasticity）。

可维护性（Operability）：人生苦短，关爱运维

有效的运维绝对是个高技术活：

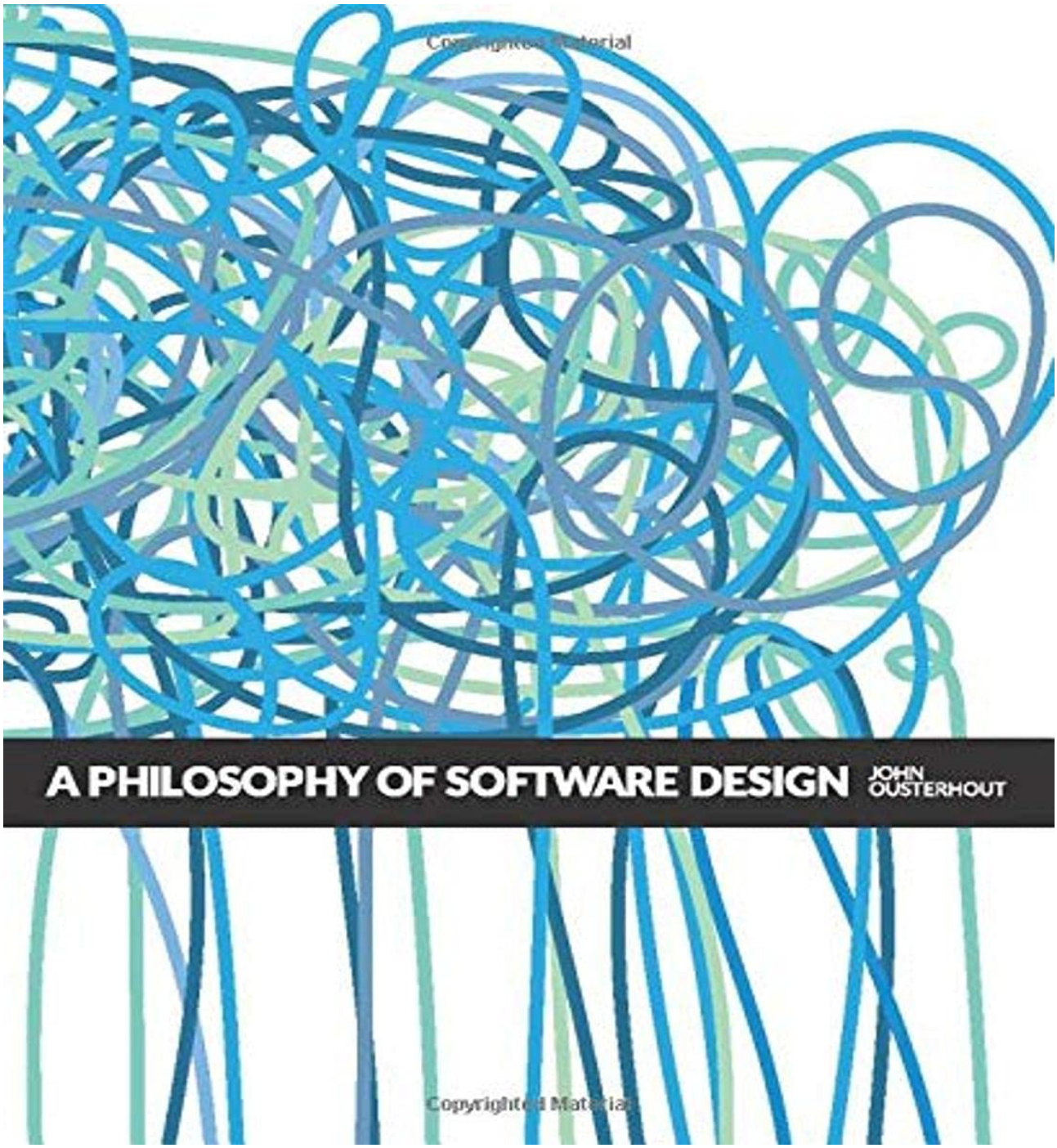
1. 紧盯系统状态，出问题时快速恢复。
2. 恢复后，复盘问题，定位原因。
3. 定期对平台、库、组件进行更新升级。

4. 了解组件间相互关系，避免级联故障。
5. 建立自动化配置管理、服务管理、更新升级机制。
6. 执行复杂维护任务，如将存储系统从一个数据中心搬到另外一个数据中心。
7. 配置变更时，保证系统安全性。

系统具有良好的可维护性，意味着将可定义的维护过程编写文档和工具以自动化，从而解放出人力关注更高价值事情：

1. 友好的文档和一致的运维规范。
2. 细致的监控仪表盘、自检和报警。
3. 通用的缺省配置。
4. 出问题时的自愈机制，无法自愈时允许管理员手动介入。
5. 将维护过程尽可能的自动化。
6. 避免单点依赖，无论是机器还是人。

简洁性（Simplicity）：复杂度管理



推荐一本书：A Philosophy of Software Design，讲述在软件设计中如何定义、识别和降低复杂度。

复杂度表现：

1. 状态空间的膨胀。

2. 组件间的强耦合。
3. 不一致的术语和命名。
4. 为了提升性能的 hack。
5. 随处可见的补丁（workaround）。

需求很简单，但不妨碍你实现的很复杂😉：过多的引入了额外复杂度（accidental complexity）——非问题本身决定的，而由实现所引入的复杂度。

通常是问题理解的不够本质，写出了“流水账”（没有任何抽象，abstraction）式的代码。

如果你为一个问题找到了合适的抽象，那么问题就解决了一半，如：

1. 高级语言隐藏了机器码、CPU 和系统调用细节。
2. SQL 隐藏了存储体系、索引结构、查询优化实现细节。

如何找到合适的抽象?

1. 从计算机领域常见的抽象中找。
2. 从日常生活中常接触的概念找。

总之，一个合适的抽象，要么是符合直觉的；要么是和你的读者共享上下文的。

本书之后也会给出很多分布式系统中常用的抽象。

可演化性：降低改变门槛

系统需求没有变化，说明这个行业死了。

否则，需求一定是不断在变，引起变化的原因多种多样：

1. 对问题阈了解更全面
2. 出现了之前未考虑到的用例
3. 商业策略的改变
4. 客户爸爸要求新功能
5. 依赖平台的更迭
6. 合规性要求
7. 体量的改变

应对之道：

- 项目管理上 敏捷开发
- 系统设计上 依赖前两点。合理抽象，合理封装，对修改关闭，对扩展开放。

DDIA 逐章精读（二）：数据模型和查询语言

概要

本节围绕两个主要概念来展开。

如何分析一个数据模型：

1. 基本考察点：数据基本元素，和元素之间的对应关系（一对多，多对多）
2. 利用几种常用模型来比较：（最为流行的）关系模型，（树状的）文档模型，（极大自由度的）图模型。
3. schema 模式：强 Schema（写时约束）；弱 Schema（读时解析）

如何考量查询语言：

1. 如何与数据模型关联、匹配
2. 声明式（declarative）和命令式（imperative）

数据模型

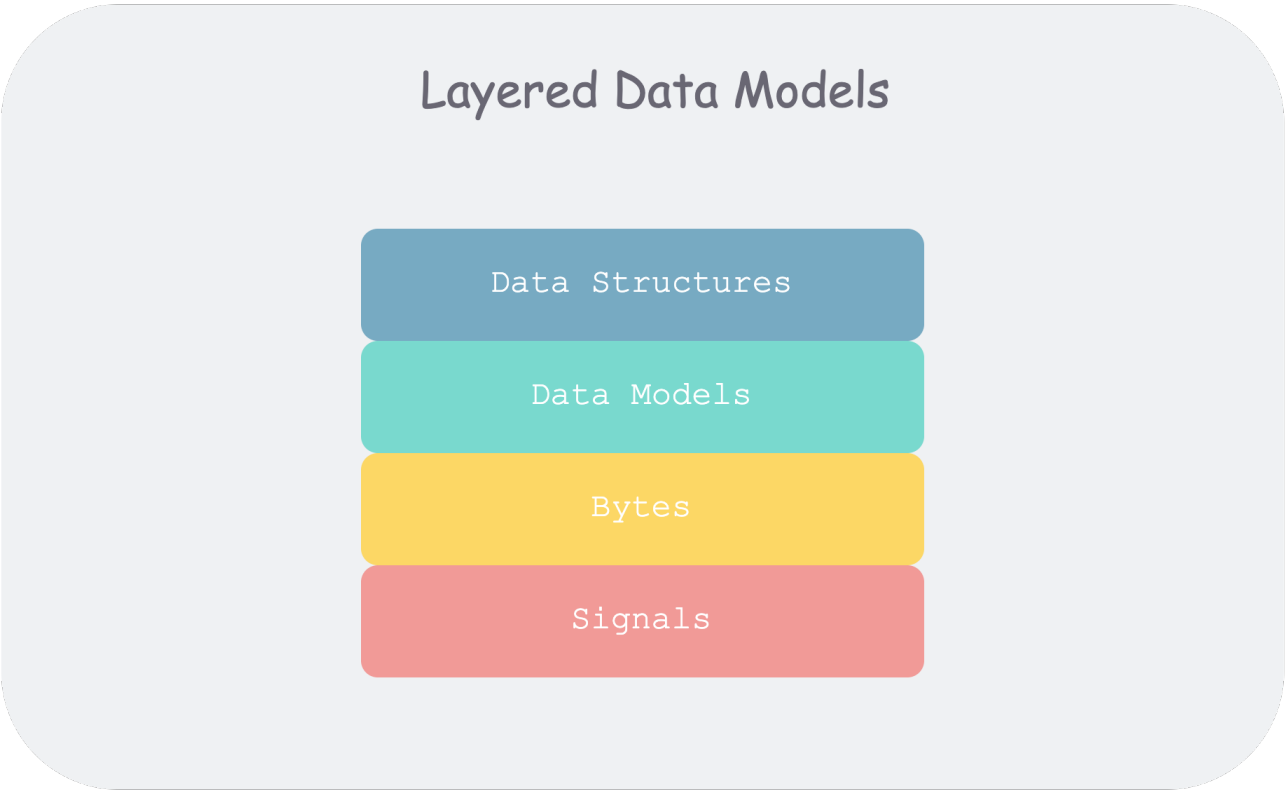
> A data model is an abstract model that organizes elements of data and standardizes how they relate to one another and to the properties of real-world entities. —https://en.wikipedia.org/wiki/Data_model

数据模型：如何组织数据，如何标准化关系，如何关联现实。

它既决定了我们构建软件的方式（实现），也左右了我们看待问题的角度（认知）。

作者开篇以计算机的不同抽象层次来让大家对泛化的数据模型有个整体观感。

大多数应用都是通过不同的数据模型层级累进构建的。



每层模型核心问题：如何用下一层的接口来对本层进行建模？

1. 作为应用开发者， 你将现实中的具体问题抽象为一组对象、数据结构（data structure） 以及作用于其上的 API。
2. 作为数据库管理员（DBA），为了持久化上述数据结构，你需要将他们表达为通用的数据模型（data model），如文档数据库中的 XML/JSON、关系数据库中的表、图数据库中的图。
3. 作为数据库系统开发者，你需要将上述数据模型组织为内存中、硬盘中或者网络中的字节（Bytes） 流，并提供多种操作数据集合的方法。
4. 作为硬件工程师，你需要将字节流表示为二极管的电位（内存）、磁场中的磁极（磁盘）、光纤中的光信号（网络）。

> 在每一层，通过对外暴露简洁的数据模型，我们隔离和分解了现实世界的复杂度。

这也反过来说明了，好的数据模型需有两个特点：

1. 简洁直观
2. 具有组合性

第二章首先探讨了关系模型、文档模型及其对比，其次是相关查询语言，最后探讨了图模型。

关系模型与文档模型

关系模型

关系模型无疑是当今最流行的数据库模型。

关系模型是 埃德加·科德（E. F. Codd）于 1969 年首先提出，并用“科德十二定律”来解释。但是商业落地的数据库基本没有能完全遵循的，因此关系模型后来通指这一类数据库。特点如下：

1. 将数据以关系呈现给用户（比如：一组包含行列的二维表）。
2. 提供操作数据集合的关系算子。

常见分类

1. 事务型（TP）：银行交易、火车票
2. 分析型（AP）：数据报表、监控表盘
3. 混合型（HTAP）：

关系模型诞生很多年后，虽有不时有各种挑战者（比如上世纪七八十年代的网状模型 network model 和层次模型 hierarchical model），但始终仍未有根本的能撼动其地位的新模型。

直到近十年来，随着移动互联网的普及，数据爆炸性增长，各种处理需求越来越精细化，催生了数据模型的百花齐放。

NoSQL 的诞生

NoSQL（最初表示 Non-SQL，后来有人转解为 Not only SQL），是对不同于传统的关系数据库的数据库管理系统的统称。根据 DB-Engines 排名，现在最受欢迎的 NoSQL 前几名为：MongoDB，Redis，ElasticSearch，Cassandra。

其催动因素有：

1. 处理更大数据集：更强伸缩性、更高吞吐量

2. 开源免费的兴起：冲击了原来把握在厂商的标准
3. 特化的查询操作：关系数据库难以支持的，比如图中的多跳分析
4. 表达能力更强：关系模型约束太严，限制太多

面向对象和关系模型的不匹配

核心冲突在于面向对象的嵌套性和关系模型的平铺性（？我随便造的）。

当然有 ORM 框架可以帮我们搞定这些事情，但仍是不太方便。

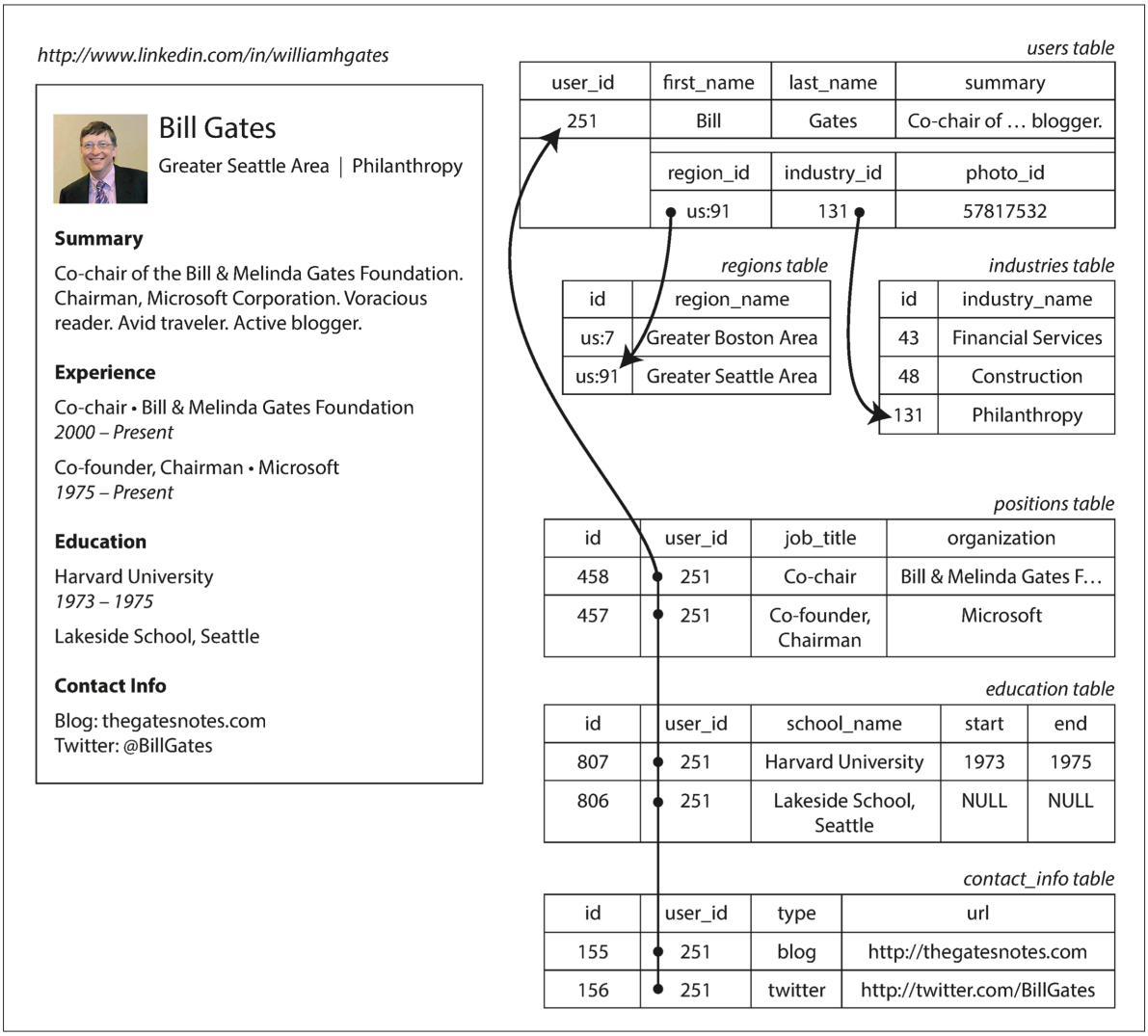


Figure 2-1. Representing a LinkedIn profile using a relational schema. Photo of Bill Gates courtesy of Wikimedia Commons, Ricardo Stuckert, Agência Brasil.

换另一个角度来说，关系模型很难直观的表示一对多的关系。比如简历上，一个人可能有多段教育经历和多段工作经历。

文档模型：使用 Json 和 XML 的天然嵌套。

关系模型：使用 SQL 模型就得将职位、教育单拎一张表，然后在用户表中使用外键关联。

在简历的例子中，文档模型还有几个优势：

1. 模式灵活：可以动态增删字段，如工作经历。
2. 更好的局部性：一个人的所有属性被集中访问的同时，也被集中存储。
3. 结构表达语义：简历与联系信息、教育经历、职业信息等隐含一对多的树状关系可以被 JSON 的树状结构明确表达出来。

多对一和多对多

是一个对比各种数据模型的切入角度。

region 在存储时，为什么不直接存储纯字符串：“Greater Seattle Area”，而是先存为 region_id → region name，其他地方都引用 region_id?

1. 统一样式：所有用到相同概念的地方都有相同的拼写和样式
2. 避免歧义：可能有同名地区
3. 易于修改：如果一个地区改名了，我们不用去逐一修改所有引用他的地方
4. 本地化支持：如果翻译成其他语言，可以只翻译名字表。
5. 更好搜索：列表可以关联地区，进行树形组织

类似的概念还有：面向抽象编程，而非面向细节。

关于用 ID 还是文本，作者提到了一点：ID 对人类是无意义的，无意义的意味着不会随着现实世界的将来的改变而改动。

这在关系数据库表设计时需要考虑，即如何控制冗余（duplication）。会有几种范式（normalization）来消除冗余。

文档型数据库很擅长处理一对多的树形关系，却不擅长处理多对多的图形关系。如果其不支持 Join，则处理多对多关系的复杂度就从数据库侧移动到了应用侧。

如，多个用户可能在同一个组织工作过。如果我们想找出在同一个学校和组织工作过的人，如果数据库不支持 Join，则需要在应用侧进行循环遍历来 Join。

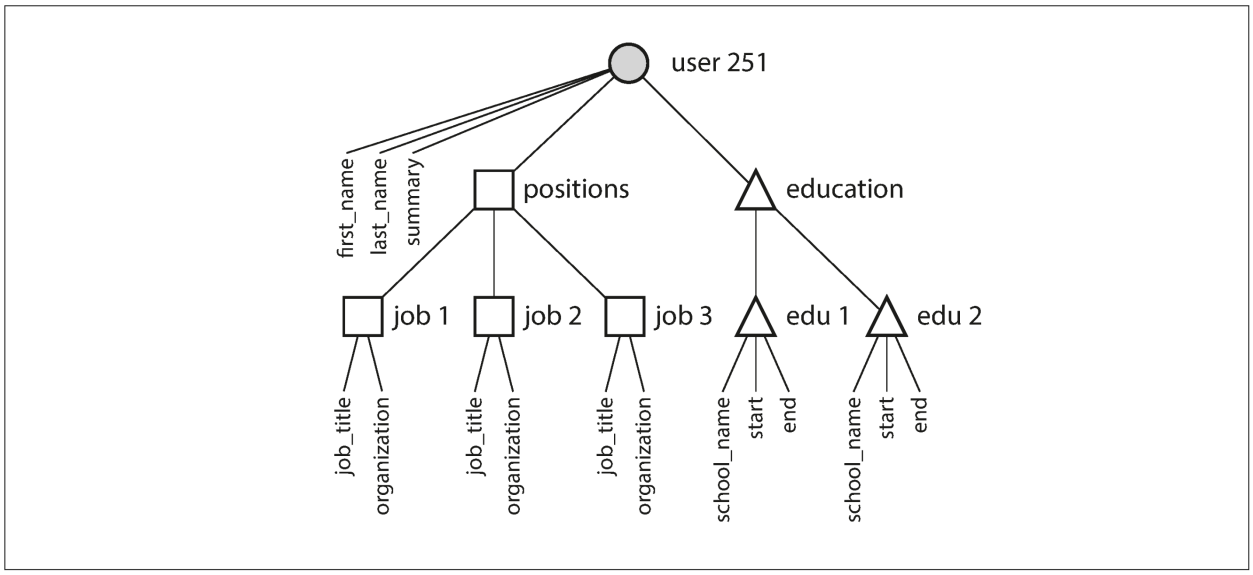


Figure 2-2. One-to-many relationships forming a tree structure.

文档 vs 关系

1. 对于一对多关系，文档型数据库将嵌套数据放在父节点中，而非单拎出来放另外一张表。
2. 对于多对一和多对多关系，本质上，两者都是使用外键（文档引用）进行索引。查询时需要进行 join 或者动态跟随。

文档模型是否在重复历史？

层次模型 （hierarchical model）

20 世纪 70 年代，IBM 的信息管理系统 IMS。

> A hierarchical database model is a data model in which the data are organized into a tree-like structure. The data are stored as records which are connected to one another through links. A record is a collection of fields, with each field containing only one value. The type of a record defines which fields the record contains. — wikipedia

几个要点：

1. 树形组织，每个子节点只允许有一个父节点
2. 节点存储数据，节点有类型

3. 节点间使用类似指针方式连接

可以看出，它跟文档模型很像，也因此很难解决多对多的关系，并且不支持 Join。

为了解决层次模型的局限，人们提出了各种解决方案，最突出的是：

1. 关系模型
2. 网状模型

网状模型（network model）

network model 是 hierarchical model 的一种扩展：允许一个节点有多个父节点。它被数据系统语言会议（CODASYL）的委员会进行了标准化，因此也被称为 CODASYL 模型。

多对一和多对多都可以由路径来表示。访问记录的唯一方式是顺着元素和链接组成的链路进行访问，这个链路叫访问路径 （access path）。难度犹如在 n-维空间中进行导航。

内存有限，因此需要严格控制遍历路径。并且需要事先知道数据库的拓扑结构，这就意味着得针对不同应用写大量的专用代码。

关系模型

在关系模型中，数据被组织成元组（tuples），进而集合成关系（relations）；在 SQL 中分别对应行（rows）和表（tables）。

- 不知道大家好奇过没，明明看起来更像表模型，为什叫关系模型？ 表只是一种实现。关系（relation）的说法来自集合论，指的是几个集合的笛卡尔积的子集。 $R \subseteq (D1 \times D2 \times D3 \cdots \times Dn)$ （关系用符号 R 表示，属性用符号 Ai 表示，属性的定义域用符号 Di 表示）

其主要目的和贡献在于提供了一种声明式的描述数据和构建查询的方法。

即，相比网络模型，关系模型的查询语句和执行路径相解耦，查询优化器（Query Optimizer 自动决定执行顺序、要使用的索引），即将逻辑和实现解耦。

举个例子：如果想使用新的方式对你的数据集进行查询，你只需要在新的字段上建立一个索引。那么在查询时，你并不需要改变的你用户代码，查询优化器便会动态的选择可用索引。

文档型 vs 关系型

根据数据类型来选择数据模型

	文档型	关系型
对应关系	数据有天然的一对多、树形嵌套关系，如简历。	通过外键 + Join 可以处理 多对一，多对多关系
代码简化	数据具有文档结构，则文档模型天然合适，用关系模型会使得建模繁琐、访问复杂。但不宜嵌套太深，因为只能手动指定访问路径，或者范围遍历	主键，索引，条件过滤
Join 支持	对 Join 支持的不太好	支持的还可以，但 Join 的实现会有很多难点
模式灵活性	弱 schema，支持动态增加字段	强 schema，修改 schema 代价很大
访问局部性	1. 一次性访问整个文档，较优 2. 只访问文档一部分，较差	分散在多个表中

对于高度关联的数据集，使用文档型表达比较奇怪，使用关系型可以接受，使用图模型最自然。

文档模型中 Schema 的灵活性

说文档型数据库是 schemaless 不太准确，更贴切的应该是 schema-on-read。

数据模型		编程语言		性能 & 空间
schema-on-read	写入时不校验，而在读取时进行动态解析。	弱类型	动态，在运行时解析	读取时动态解析，性能较差。写入时无法确定类型，无法对齐，空间利用率较差。
schema-on-write	写入时校验，数据对齐到 schema	强类型	静态，编译时确定	性能和空间使用都较优。

文档型数据库使用场景特点：

1. 有多种类型的数据，但每个放一张表又不合适。
2. 数据类型和结构由外部决定，你没办法控制数据的变化。

查询时的数据局部性

如果你同时需要文档中所有内容，把文档顺序存储，访问会效率比较高。

但如果你只需要访问文档中的某些字段，则文档仍需要将文档全部加载出。

但运用这种局部性不局限于文档型数据库。不同的数据库，会针对不同场景，调整数据物理分布以适应常用访问模式的局部性。

- Spanner 中允许表被声明为嵌入到父表中——常用关联内嵌（获得类似文档模型的结构）
- HBase 和 Cassandra 使用列族来聚集数据——分析型
- 图数据库中，将点和出边存在一个机器上——图遍历

关系型和文档型的融合

- MySQL 和 PostgreSQL 开始支持 JSON 原生支持 JSON 可以理解为，MySQL 可以理解 JSON 类型。如 Date 这种复杂格式一样，可以让某个字段为 JSON 类型、可以修改 Join 字段的某个属性、可以在 Json 字段中某个属性建立索引。

- RethinkDB 在查询中支持 relational-link Joins

科德（Codd）：nonsimple domains，记录中的值除了简单类型（数字、字符串），还可以一个嵌套关系（表）。这很像 SQL 对 XML、JSON 的支持。

数据查询语言

获取动物表中所有鲨鱼类动物。

```
function getSharks() {
  var sharks = [];
  for (var i = 0; i < animals.length; i++) {
    if (animals[i].family === 'Sharks') {
      sharks.push(animals[i]);
    }
  }
  return sharks;
}
```

```
SELECT * FROM animals WHERE family = 'Sharks';
```

	声明式（declarative）语言	**命令式（imperative）语言**
概念	描述控制逻辑而非执行流程	描述命令的执行过程，用一系列语句来不断改变状态
举例	SQL，CSS，XSL	IMS，CODASYL，通用语言如 C，C++，JS
抽象程度	高	低
		与实现耦合较深。

	声明式（declarative）语言	命令式（imperative）语言
解耦程度	与实现解耦。 可以持续优化查询引擎性能；	
解析执行	词法分析 → 语法分析 → 语义分析 生成执行计划 → 执行计划优化	词法分析 → 语法分析 → 语义分析 中间代码生成 → 代码优化 → 目标代码生成
多核并行	声明式更具多核潜力，给了更多运行时优化空间	命令式由于指定了代码执行顺序，编译时优化空间较小。

> Q：相对声明式语言，命令式语言有什么优点？
>
> 1. 当描述的目标变得复杂时，声明式表达能力不够。
> 2. 实现命令式的语言往往不会和声明式那么泾渭分明，通过合理抽象，通过一些编程范式（函数式），可以让代码兼顾表达力和清晰性。

数据库以外：Web 中的声明式

需求：选中页背景变蓝。

```
<ul>
  <li class="selected">
    <p>Sharks</p>
    <ul>
      <li>Great White Shark</li>
      <li>Tiger Shark</li>
      <li>Hammerhead Shark</li>
    </ul>
  </li>
  <li>
    <p>Whales</p>
    <ul>
      <li>Blue Whale</li>
```

```
      <li>Humpback Whale</li>
      <li>Fin Whale</li>
    </ul>
  </li>
</ul>
```

如果使用 CSS，则只需（CSS selector）：

```
li.selected > p {
  background-color: blue;
}
```

如果使用 XSL，则只需（XPath selector）：

```
<xsl:template match="li[@class='selected']/p">
    <fo:block background-color="blue">
        <xsl:apply-templates/>
    </fo:block>
</xsl:template>
```

但如果使用 JavaScript（而不借助上述 selector 库）：

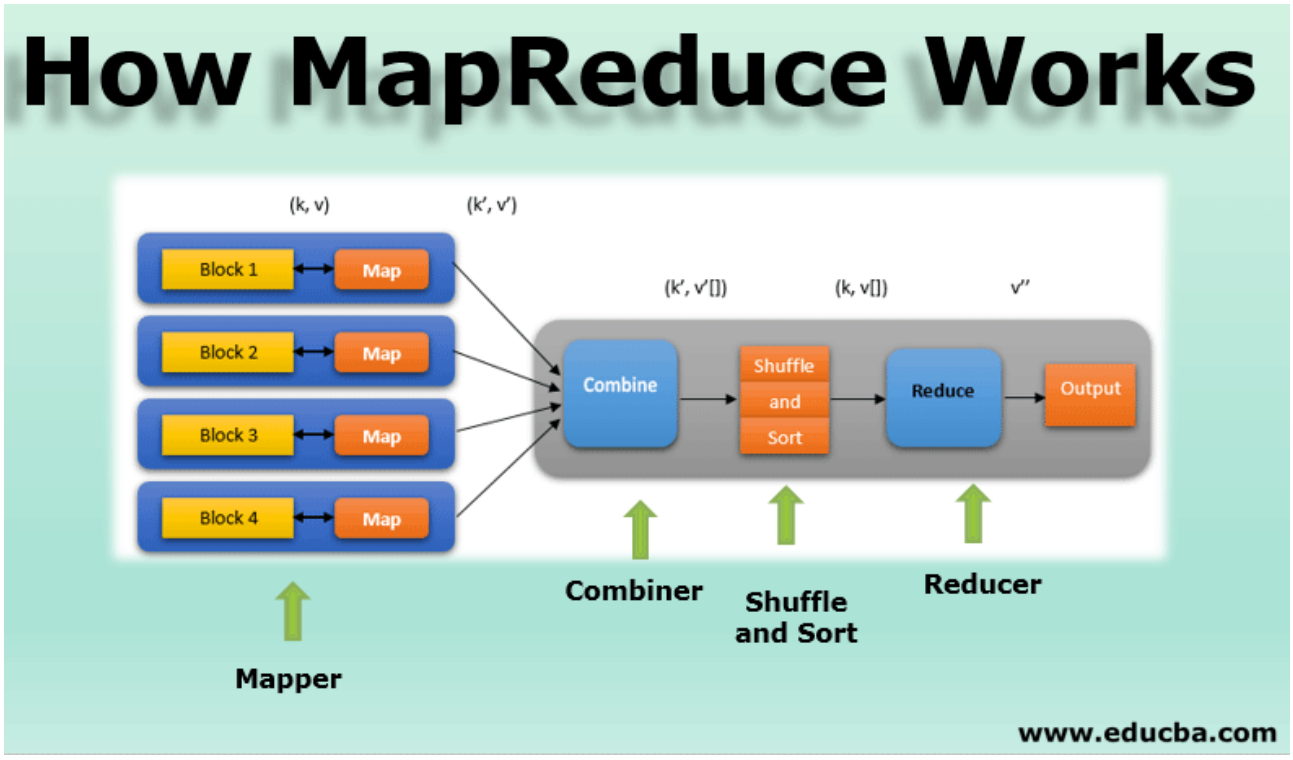
```
var liElements = document.getElementsByTagName('li');
for (var i = 0; i < liElements.length; i++) {
  if (liElements[i].className === 'selected') {
    var children = liElements[i].childNodes;
    for (var j = 0; j < children.length; j++) {
      var child = children[j];
      if (child.nodeType === Node.ELEMENT_NODE && child.tagName
=== 'P') {
        child.setAttribute('style', 'background-color: blue');
      }
    }
```

```
    }
}
```

MapReduce 查询

Google 的 MapReduce 模型

1. 借鉴自函数式编程。
2. 一种相当简单的编程模型，或者说原子的抽象，现在不太够用。
3. 但在大数据处理工具匮乏的蛮荒时代（03 年以前），谷歌提出的这套框架相当有开创性。



MongoDB 的 MapReduce 模型

MongoDB 使用的 MapReduce 是一种介于

1. 声明式：用户不必显式定义数据集的遍历方式、shuffle 过程等执行过程。
2. 命令式：用户又需要定义针对单条数据的执行过程。

两者间的混合数据模型。

需求：统计每月观察到鲨类鱼的次数。

查询语句：

PostgresSQL

```
SELECT date_trunc('month', observation_timestamp) AS
observation_month,
    sum(num_animals) AS total_animals
FROM observations
WHERE family = 'Sharks' GROUP BY observation_month;
```

MongoDB

```
db.observations.mapReduce(
  function map() {
    // 2. 对所有符合条件 doc 执行 map
    var year = this.observationTimestamp.getFullYear();
    var month = this.observationTimestamp.getMonth() + 1;
    emit(year + '-' + month, this.numAnimals); // 3. 输出一个 kv
pair
  },
  function reduce(key, values) {
    // 4. 按 key 聚集
    return Array.sum(values); // 5. 相同 key 加和
  },
  {
    query: { family: 'Sharks' }, // 1. 筛选
    out: 'monthlySharkReport', // 6. reduce 结果集
  }
);
```

上述语句在执行时，经历了：筛选（filter） → 遍历并执行 map → 对输出按 key 聚集（shuffle） → 对聚集的数据逐一 reduce → 输出结果集。

MapReduce 一些特点：

1. 要求 Map 和 Reduce 是纯函数。即无任何副作用，在任意地点、以任意次序执行任何多次，对相同的输入都能得到相同的输出。因此容易并发调度。
2. 非常底层、但表达力强大的编程模型。可基于其实现 SQL 等高级查询语言，如 Hive。

但要注意：

1. 不是所有的分布式 SQL 都基于 MapReduce 实现。
2. 不是只有 MapReduce 才允许嵌入通用语言（如 js）模块。
3. MapReduce 是有一定理解成本的，需要非常熟悉其执行原理才能让两个函数紧密配合。

MongoDB 2.2+ 进化版，aggregation pipeline:

```
db.observations.aggregate([
  { $match: { family: 'Sharks' } },
  {
    $group: {
      _id: {
        year: { $year: '$observationTimestamp' },
        month: { $month: '$observationTimestamp' },
      },
      totalAnimals: { $sum: '$numAnimals' },
    },
  },
]);
```

图模型

- 文档模型的适用场景？ 你的建模场景中存在着大量一对多（one-to-many）的关系。
- 图模型的适用场景？ 你的建模场景中存在大量的多对多（many-to-many）的关系。

基本概念

图数据模型（属性图）的基本概念一般有三个：点，边和附着于两者之上的属性。

常见的可以用图建模的场景：

例子	建模	应用
社交图谱	人是点，follow 关系是边	六度分隔，信息流推荐
互联网	网页是点，链接关系是边	PageRank
路网	交通枢纽是点，铁路/公路是边	路径规划，导航最短路径
洗钱	账户是点，转账关系是边	判断是否有环
知识图谱	概念是点，关联关系是边	启发式问答

> 同构（homogeneous）数据和异构数据 图模型中的点变可以像关系中的表一样都具有相同类型；但是，一张图中的点和变也可以具有不同类型，能够容纳异构数据是图模型善于处理多对多关系的一大原因。

本节都会以下图为例，它表示了一对夫妇，来自美国爱达荷州的 Lucy 和来自法国 的 Alain：他们已婚，住在伦敦。

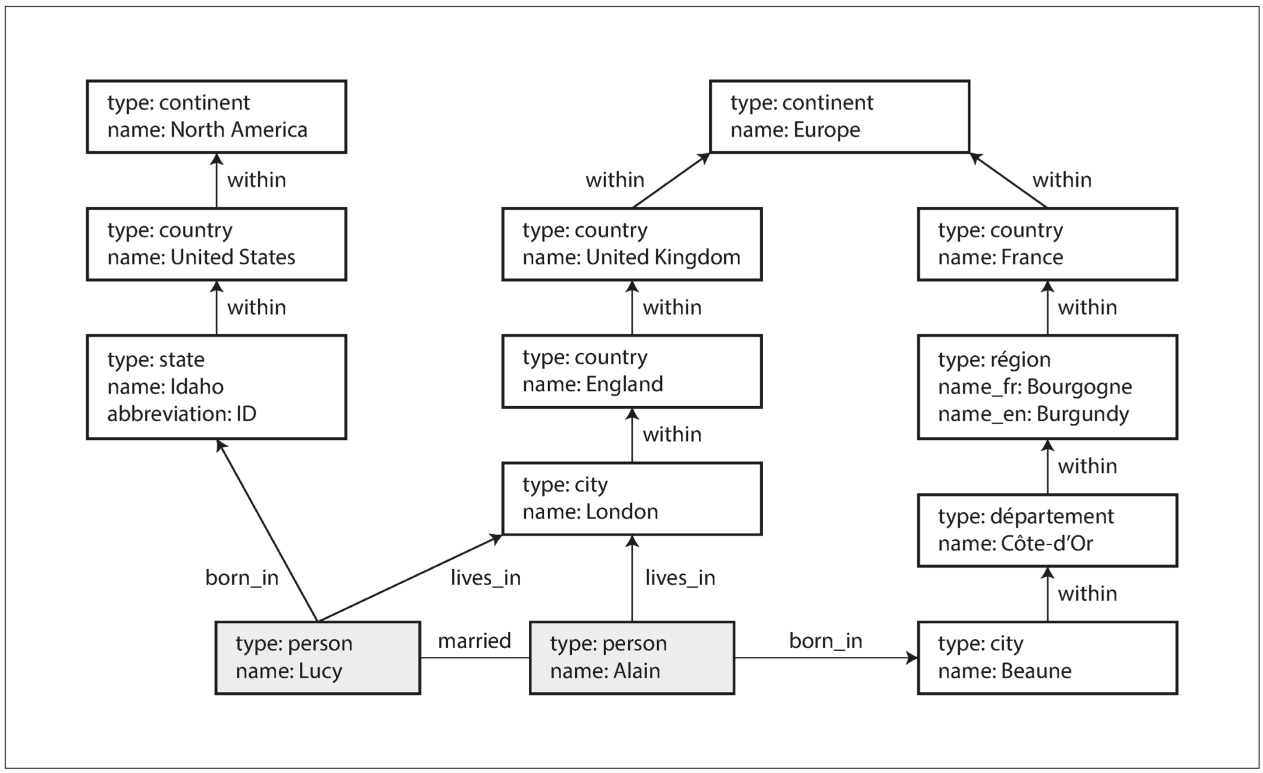


Figure 2-5. Example of graph-structured data (boxes represent vertices, arrows represent edges).

有多种对图的建模方式：

1. 属性图（property graph）：比较主流，如 Neo4j、Titan、InfiniteGraph
2. 三元组（triple-store）：如 Datomic、AllegroGraph

属性图（PG，Property Graphs）

点 (vertices, nodes, entities)	**边 (edges, relations, arcs)**
全局唯一 ID	全局唯一 ID
出边集合	起始点
入边集合	终止点
属性集（kv 对表示）	属性集（kv 对表示）
表示点类型的 type？	表示边类型的 label

> Q：有一个疑惑点，为什么书中对于 PG 点的定义中没有 Type？ 因为属性图具体实现时也可以分为强类型和弱类型，NebulaGraph 是强类型，好处在于效率高，但灵活性差；Neo4j 是弱类型。书中应该是用的弱类型，此时每个点都是一组属性集，不需要 type。

如果感觉不直观，可以使用我们熟悉的 SQL 语义来构建一个图模型，如下图。（Facebook TAO 论文中的单机存储引擎便是 MySQL）

```
// 点表
CREATE TABLE vertices (
    vertex_id integer PRIMARYKEY, properties json
);

// 边表
CREATE TABLE edges (
    edge_id integer PRIMARY KEY,
    tail_vertex integer REFERENCES vertices (vertex_id),
    head_vertex integer REFERENCES vertices (vertex_id),
    label text,
    properties json
);

// 对点的反向索引，图遍历时用。给定点，找出点的所有入边和出边。
CREATE INDEX edges_tails ON edges (tail_vertex);
CREATE INDEX edges_heads ON edges (head_vertex);
```

图是一种很灵活的建模方式：

1. 任何两点间都可以插入边，没有任何模式限制。
2. 对于任何顶点都可以高效（思考：如何高效？）找到其入边和出边，从而进行图遍历。
3. 使用多种标签来标记不同类型边（关系）。

相对于关系型数据来说，可以在同一个图中保存异构类型的数据和关系，给了图极大的表达能力！

这种表达能力，根据图中的例子，包括：

1. 对同样的概念，可以用不同结构表示。如不同国家的行政划分。
2. 对同样的概念，可以用不同粒度表示。比如 Lucy 的现居住地和诞生地。
3. 可以很自然的进行演化。

将异构的数据容纳在一张图中，可以通过图遍历，轻松完成关系型数据库中需要多次 Join 的操作。

Cypher 查询语言

Cypher 是 Neo4j 创造的一种查询语言。

Cypher 和 Neo 名字应该都是来自《黑客帝国》（The Matrix）。想想 Oracle。

Cypher 的一大特点是可读性强，尤其在表达路径模式（Path Pattern）时。

结合前图，看一个 Cypher 插入语句的例子：

```
CREATE
  (NAmerica:Location {name:'North America', type:'continent'}),
  (USA:Location      {name:'United States', type:'country'  }),
  (Idaho:Location    {name:'Idaho',         type:'state'    }),
  (Lucy:Person       {name:'Lucy' }),
  (Idaho) -[:WITHIN]->  (USA)  -[:WITHIN]-> (NAmerica),
  (Lucy)  -[:BORN_IN]-> (Idaho)
```

如果我们要进行一个这样的查询：找出所有从美国移居到欧洲的人名。

转化为图语言，即为：给定条件，BORN_IN 指向美国的地点，并且 LIVING_IN 指向欧洲的地点，找到所有符合上述条件的点，并且返回其名字属性。

用 Cypher 语句可表示为：

```
MATCH
  (person) -[:BORN_IN]->  () -[:WITHIN*0..]-> (us:Location
```

```
{name:'United States'}),
  (person) -[:LIVES_IN]-> () -[:WITHIN*0..]-> (eu:Location
{name:'Europe'})
RETURN person.name
```

注意到：

1. 点 `()`，边 `-[]→`，标签\类型 `:`，属性 `{}`。
2. 名字绑定或者说变量：`person`
3. 0 到多次通配符： `*0...`

正如声明式查询语言的一贯特点，你只需描述问题，不必担心执行过程。但与 SQL 的区别在于，SQL 基于关系代数，Cypher 类似正则表达式。

无论是 BFS、DFS 还是剪枝等实现细节，一般来说（但是不同厂商通常都会有不同的最佳实践），用户都不需要关心。

使用 SQL 进行图查询

前面看到可以用 SQL 存储点和边，以表示图。

那可以用 SQL 进行图查询吗?

Oracle 的 PGQL:

```
CREATE PROPERTY GRAPH bank_transfers
     VERTEX TABLES (persons KEY(account_number))
     EDGE TABLES(
                  transactions KEY (from_acct, to_acct, date,
amount)
                  SOURCE KEY (from_account) REFERENCES persons
                  DESTINATION KEY (to_account) REFERENCES
persons
                  PROPERTIES (date, amount)
       )
```

其中有一个难点，就是如何表达图中的路径模式（graph pattern），如多跳查询，对应到 SQL 中，就是不确定次数的 Join：

```
() -[:WITHIN*0..]-> ()
```

使用 SQL:1999 中 recursive common table expressions（PostgreSQL, IBM DB2, Oracle, and SQL Server 支持）的可以满足。但是，相当冗长和笨拙。

Triple-Stores and SPARQL

Triple-Stores，可以理解为三元组存储，即用三元组存储图。

其含义如下：

Subject	**对应图中的一个点**
Object	1. 一个原子数据，如 string 或者 number。 2. 另一个 Subject。
Predicate	1. 如果 Object 是原子数据，则 对应点附带的 KV 对。 2. 如果 Object 是另一个 Object，则 Predicate 对应图中的边。

仍是上边例子，用 Turtle triples（一种 Triple-Stores 语法）表达为：

```
@prefix : <urn:example:>.
_:lucy     a       :Person.
_:lucy     :name   "Lucy".
_:lucy     :bornIn _:idaho.
_:idaho    a       :Location.
_:idaho    :name   "Idaho".
_:idaho    :type   "state".
_:idaho    :within _:usa.
_:usa      a       :Location
_:usa      :name   "United States"
_:usa      :type   "country".
_:usa      :within _:namerica.
_:namerica a       :Location.
_:namerica :name   "North America".
_:namerica :type   "continent".
```

一种更紧凑的写法：

```
@prefix : <urn:example:>.
_:lucy     a: Person;   :name "Lucy";          :bornIn _:idaho
_:idaho    a: Location; :name "Idaho";         :type
"state";      :within _:usa.
_:usa      a: Location; :name "United States"; :type
"country";    :within _:namerica.
_:namerica a :Location; :name "North America"; :type
"continent".
```

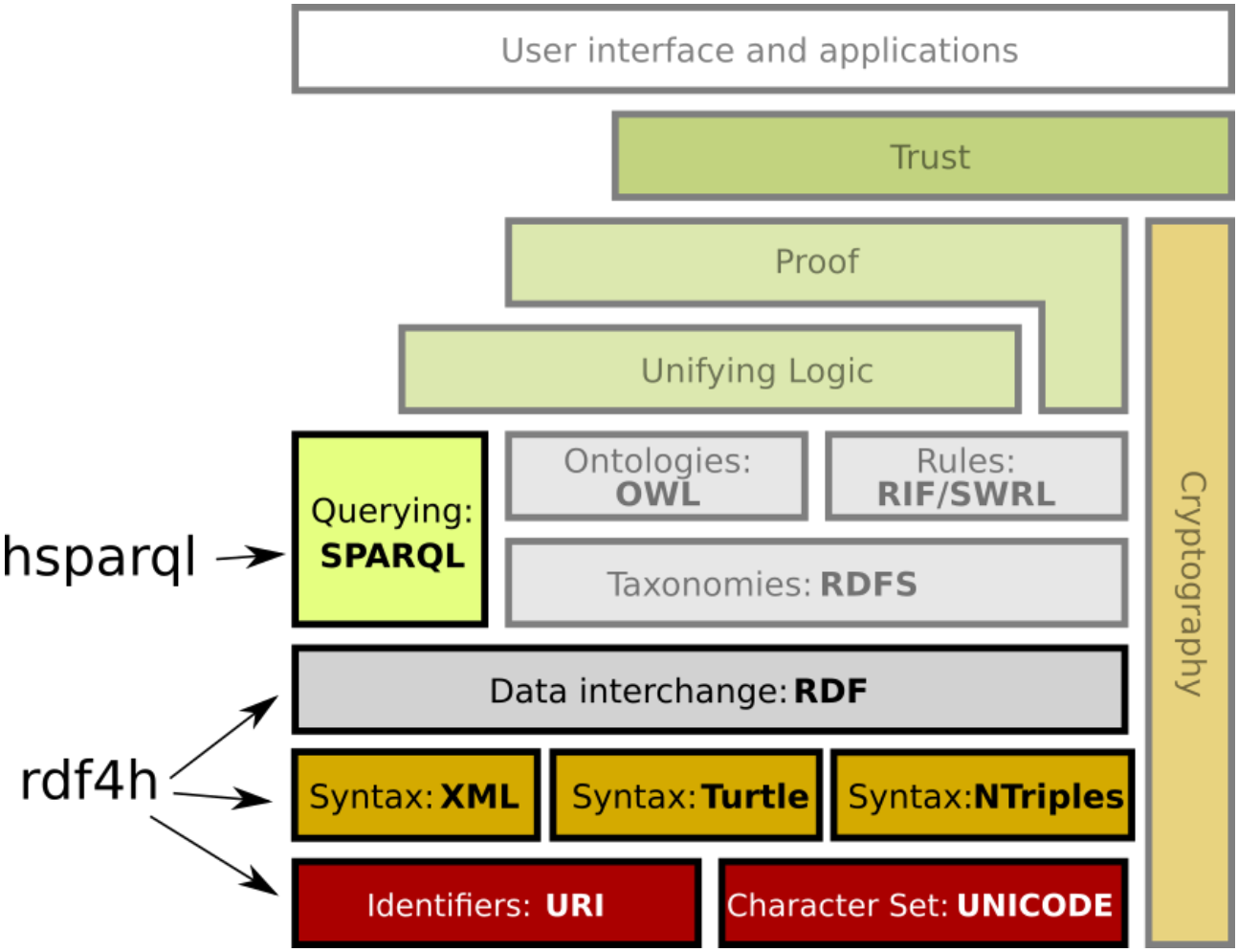
语义网（The Semantic Web）

万维网之父 Tim Berners Lee 于 1998 年提出，知识图谱前身。其目的在于对网络中的资源进行结构化，从而让计算机能够理解网络中的数据。即不是以文本、二进制流等非结构数据呈现内容，而是以某种标准结构化互联网上通过超链接而关联的数据。

语义：提供一种统一的方式对所有资源进行描述和结构化（机器可读）。

网：将所有资源勾连起来。

下面是语义网技术栈（Semantic Web Stack）：



其中 RDF （ ResourceDescription Framework，资源描述框架 ）提供了一种结构化网络中数据的标准。使发布到网络中的任何资源（文字、图片、视频、网页），都能以统一的形式被计算机理解。从另一个角度来理解，即，不需要资源使用方通过深度学习等方式来抽取语义，而是靠资源提供方通过 RDF 主动提供其资源语义。

感觉有点理想主义，但互联网、开源社区都是靠这种理想主义、分享精神发展起来的！

虽然语义网没有发展起来，但是其中间数据交换格式 RDF 所定义的 SPO 三元组 (Subject-Predicate-Object) 却是一种很好用的数据模型，也就是上面提到的 Triple-Stores。

RDF 数据模型

上面提到的 Turtle 语言（SPO 三元组）是一种简单易读的描述 RDF 数据的方式，RDF 也可以基于 XML 表示，但是要冗余难读的多（嵌套太深）：

```
<rdf:RDF xmlns="urn:example:"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <Location rdf:nodeID="idaho">
        <name>Idaho</name>
        <type>state</type>
        <within>
            <Location rdf:nodeID="usa">
                <name>United States</name>
                <type>country</type>
                <within>
                    <Location rdf:nodeID="namerica">
                        <name>North America</name>
                        <type>continent</type>
                </Location>
            </within>
      </Location>
    </within>
    </Location>
    <Person rdf:nodeID="lucy">
        <name>Lucy</name>
        <bornIn rdf:nodeID="idaho"/>
    </Person>
</rdf:RDF>
```

为了标准化和去除二义性，一些看起来比较奇怪的点是：无论 subject，predicate 还是 object 都是由 URI 定义，如

```
lives_in 会表示为 <http://my-company.com/namespace#lives_in>
```

其前缀只是一个 namespace，让定义唯一化，并且在网络上可访问。当然，一个简化的方法是可以在文件头声明一个公共前缀。

SPARQL 查询语言

有了语义网，自然需要在语义网中进行遍历查询，于是有了 RDF 的查询语言：SPARQL Protocol and RDF Query Language, pronounced“sparkle.”

```
PREFIX : <urn:example:>
SELECT ?personName WHERE {
  ?person :name ?personName.
  ?person :bornIn  / :within* / :name "United States".
  ?person :livesIn / :within* / :name "Europe".
}
```

他是 Cypher 的前驱，因此结构看起来很像：

```
(person) -[:BORN_IN]-> () -[:WITHIN*0..]-> (location)   # Cypher
?person   :bornIn /         :within*        ?location.   # SPARQL
```

但 SPARQL 没有区分边和属性的关系，都用了 Predicates。

```
(usa {name:'United States'})   # Cypher
?usa :name "United States".    # SPARQL
```

虽然语义网没有成功落地，但其技术栈影响了后来的知识图谱和图查询语言。

图模型和网络模型

图模型是网络模型旧瓶装新酒吗?

否，他们在很多重要的方面都不一样。

模型	图模型（Graph Model）	网络模型（Network Model）
连接方式	任意两个点之间都有可以有边	指定了嵌套约束
记录查找	1. 使用全局 ID 2. 使用属性索引。 3. 使用图遍历。	只能使用路径查询
有序性	点和边都是无序的	记录的孩子们是有序集合，在插入时需要考虑维持有序的开销
查询语言	即可命令式，也可以声明式	命令式的

查询语言前驱：Datalog

有点像 triple-store，但是变了下次序：(subject, predicate, object) → predicate(subject, object).
之前数据用 Datalog 表示为：

```
name(namerica, 'North America').
type(namerica, continent).

name(usa, 'United States').
type(usa, country).
within(usa, namerica).

name(idaho, 'Idaho').
type(idaho, state).
within(idaho, usa).

name(lucy, 'Lucy').
born_in(lucy, idaho).
```

查询从美国迁移到欧洲的人可以表示为：

```
within_recursive(Location, Name) :- name(Location, Name). /*
Rule 1 */
within_recursive(Location, Name) :- within(Location, Via), /*
Rule 2 */
                                    within_recursive(Via, Name).

migrated(Name, BornIn, LivingIn) :- name(Person, Name), /* Rule
3 */
                                    born_in(Person, BornLoc),
                                    within_recursive(BornLoc,
BornIn),
                                    lives_in(Person, LivingLoc),
                                    within_recursive(LivingLoc,
LivingIn).
?- migrated(Who, 'United States', 'Europe'). /* Who = 'Lucy'. */
```

1. 代码中以大写字母开头的元素是变量，字符串、数字或以小写字母开头的元素是常量。下划线（_）被称为匿名变量
2. 可以使用基本 Predicate 自定义 Predicate，类似于使用基本函数自定义函数。
3. 逗号连接的多个谓词表达式为且的关系。

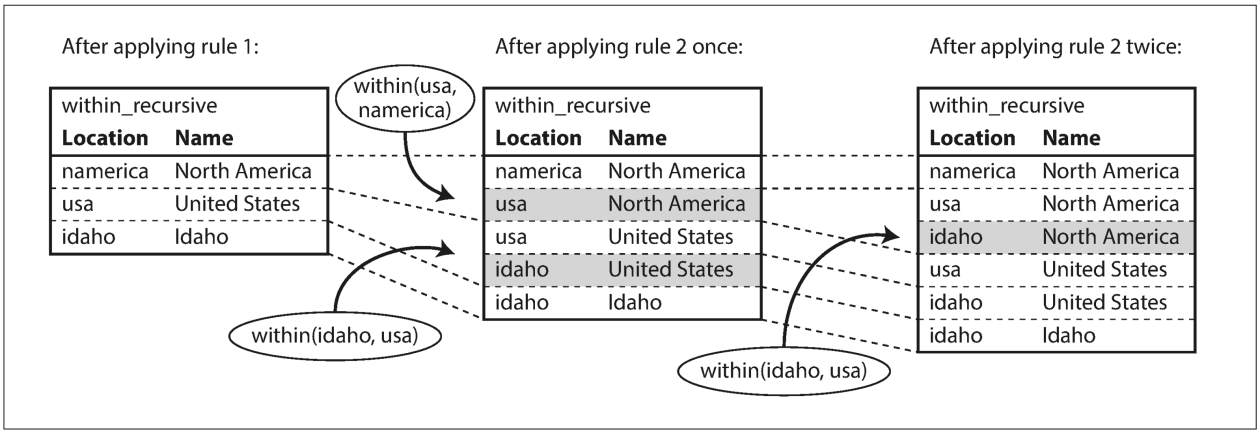


Figure 2-6. Determining that Idaho is in North America, using the Datalog rules from Example 2-11.

基于集合的逻辑运算：

1. 根据基本数据子集选出符合条件集合。
2. 应用规则，扩充原集合。
3. 如果可以递归，则递归穷尽所有可能性。

Prolog（Programming in Logic 的缩写）是一种逻辑编程语言。它创建在逻辑学的理论基础之上。

参考

1. 声明式 (declarative) vs 命令式 (imperative)：https://lotabout.me/2020/Declarative-vs-Imperative-language/
2. SimmerChan 知乎专栏，知识图谱，语义网，RDF：https://www.zhihu.com/column/knowledgegraph
3. MySQL 为什么叫“关系”模型：https://zhuanlan.zhihu.com/p/64731206