

Designing Data-Intensive Applications

- The Big Ideas Behind Reliable, Scalable, and Maintainable Systems

Part 1. Foundation of Dada Systems

1: Reliable, Scalable and Maintainable Applications (07/27)

2: Data Models and Query Languages(07/02)

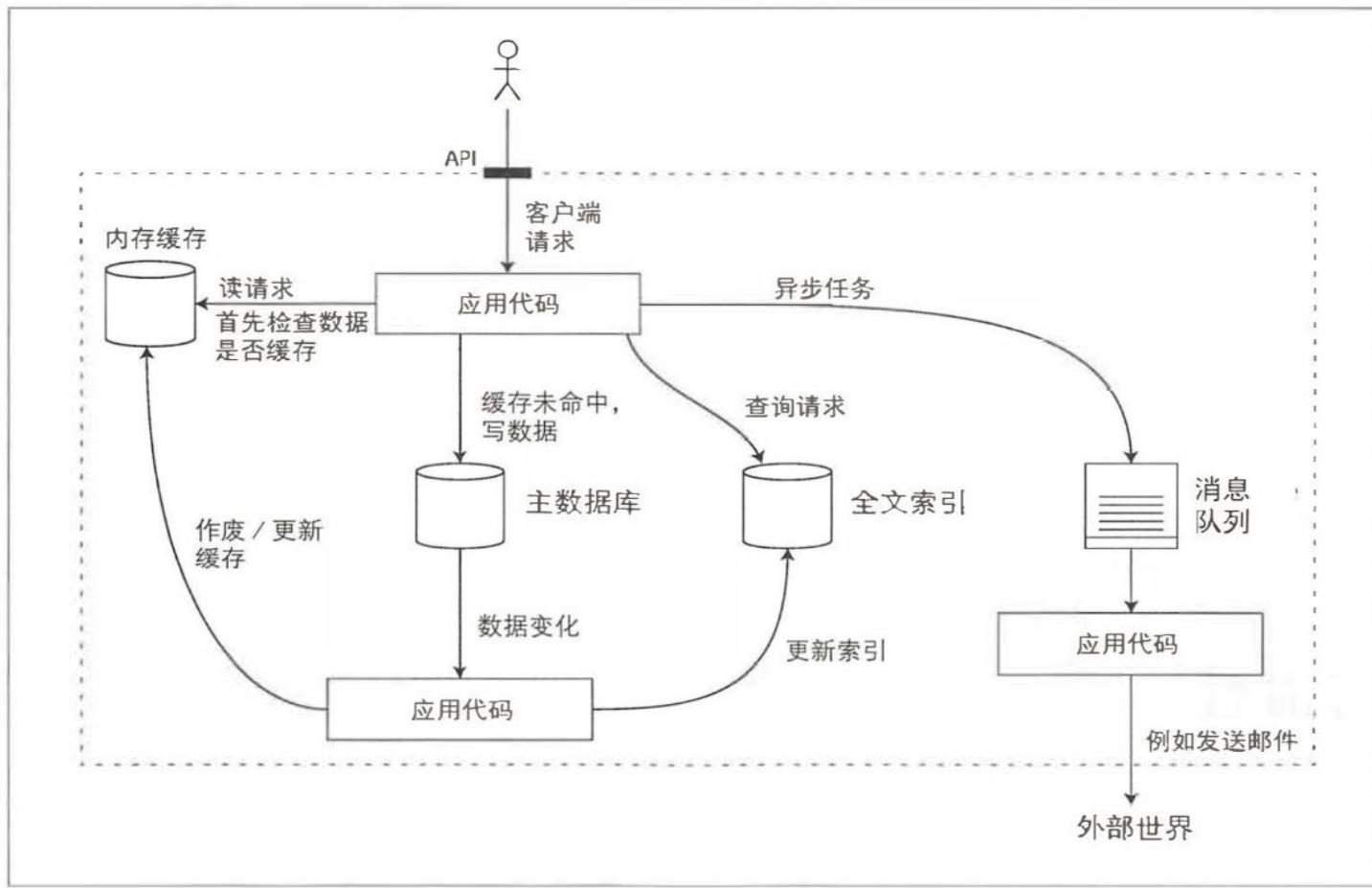
3: Storage and Retrieval

4: Encoding and Evolution

当今许多新型应用都属于数据密集型（data-intensive），而不是计算密集型（compute-intensive）。对于这些类型应用，CPU的处理能力往往不是第一限制性因素，关键在于数据量、数据的复杂度及数据的快速多变性。

数据密集型应用通常也是基于标准模块构建而成，每个模块负责单一的常用功能。例如，许多应用系统都包含以下模块：

- 数据库：用以存储数据，这样之后应用可以再次访问。
- 高速缓存：缓存那些复杂或操作代价昂贵的结果，以加快下一次访问。
- 索引：用户可以按关键字搜索数据并支持各种过滤。
- 流式处理：持续发送消息至另一个进程，处理采用异步方式。
- 批处理：定期处理大量的累积数据。





可靠性

每个人脑子里都有一个直观的认识，即什么意味着可靠或者不可靠。对于软件，典型的期望包括：

- 应用程序执行用户所期望的功能。
- 可以容忍用户出现错误或者不正确的软件使用方法。
- 性能可以应对典型场景、合理负载压力和数据量。
- 系统可防止任何未经授权的访问和滥用。

如果所有上述目标都要支持才算“正常工作”，那么我们可以认为可靠性大致意味着：即使发生了某些错误，系统仍可以继续正常工作。

Fault-tolerant System

Hardware Fault,

我们的第一个反应通常是为硬件添加冗余来减少系统故障率。例如对磁盘配置RAID，服务器配备双电源，甚至热插拔CPU，数据中心添加备用电源、发电机等。

当一个组件发生故障，冗余组件可以快速接管，之后再更换失效的组件。这种方法可能并不能完全防止硬件故障所引发的失效，但还是被普遍采用，且在实际中也确实可以让系统不间断运行长达数年。

Software Errors,

软件系统问题有时没有快速解决办法，而只能仔细考虑很多细节，包括认真检查依赖的假设条件与系统之间交互，进行全面的测试，进程隔离，允许进程崩溃并自动重启，反复评估，监控并分析生产环节的行为表现等。如果系统提供某些保证，例如，在消息队列中，输出消息的数量应等于输入消息的数量，则可以不断地检查确认，如发现差异则立即告警^[12]。

Human Errors

- 以最小出错的方式来设计系统。例如，精心设计的抽象层、API以及管理界面，使“做正确的事情”很轻松，但搞坏很复杂。但是，如果限制过多，人们就会想法来绕过它，这会抵消其正面作用。因此解决之道在于很好的平衡。
- 想办法分离最容易出错的地方、容易引发故障的接口。特别是，提供一个功能齐全但非生产用的沙箱环境，使人们可以放心的尝试、体验，包括导入真实的数据，万一出现问题，不会影响真实用户。
- 充分的测试：从各单元测试到全系统集成测试以及手动测试^[3]。自动化测试已被广泛使用，对于覆盖正常操作中很少出现的边界条件等尤为重要。
- 当出现人为失误时，提供快速的恢复机制以尽量减少故障影响。例如，快速回滚配置改动，滚动发布新代码（这样任何意外的错误仅会影响一小部分用户），并提供校验数据的工具（防止旧的计算方式不正确）。
- 设置详细而清晰的监控子系统，包括性能指标和错误率。在其他行业称为遥测（Telemetry），一旦火箭离开地面，遥测对于跟踪运行和了解故障至关重要^[14]。监控可以向我们发送告警信号，并检查是否存在假设不成立或违反约束条件等。这些检测指标对于诊断问题也非常有用。
- 推行管理流程并加以培训。这非常重要而且比较复杂，具体内容已超出本书范围。

可扩展性

即使系统现在工作可靠，并不意味着它将来一定能够可靠运转。发生退化的一个常见原因是负载增加：例如也许并发用户从最初的10 000个增长到100 000个，或从100万到1000万；又或者系统目前要处理的数据量超出之前很多倍。

可扩展性是用来描述系统应对负载增加能力的术语。但是请注意，它并不是衡量一个系统的一维指标，谈论“X是可扩展”或“Y不扩展”没有太大意义。相反，讨论可扩展性通常要考虑这类问题：“如果系统以某种方式增长，我们应对增长的措施有哪些”，“我们该如何添加计算资源来处理额外的负载”。

- 将发送的新tweet插入到全局的tweet集合中。当用户查看时间线时，首先查找所有的关注对象，列出这些人的所有tweet，最后以时间为序来排序合并。如果以图1-2的关系型数据模型，可以执行下述的SQL查询语句：

```
SELECT tweets.*, users.* FROM tweets
JOIN users ON tweets.sender_id = users.id
JOIN follows ON follows.followee_id = users.id
WHERE follows.follower_id = current_user
```

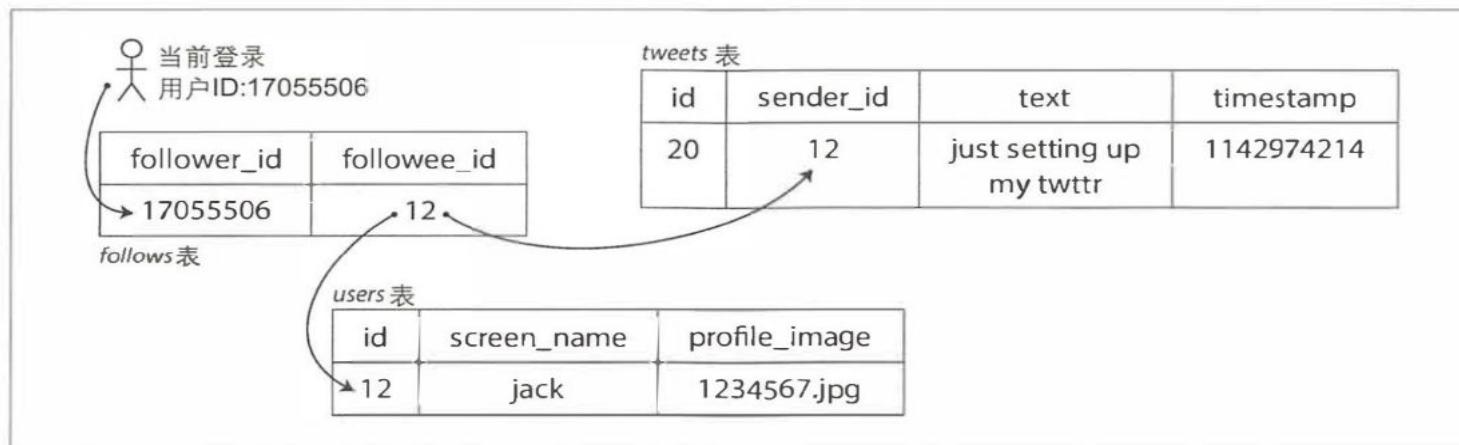


图1-2：采用关系型数据模型来支持时间线

2. 对每个用户的时间线维护一个缓存，如图1-3所示，类似每个用户一个tweet邮箱。当用户推送新tweet时，查询其关注者，将tweet插入到每个关注者的时间线缓存中。因为已经预先将结果取出，之后访问时间线性能非常快。

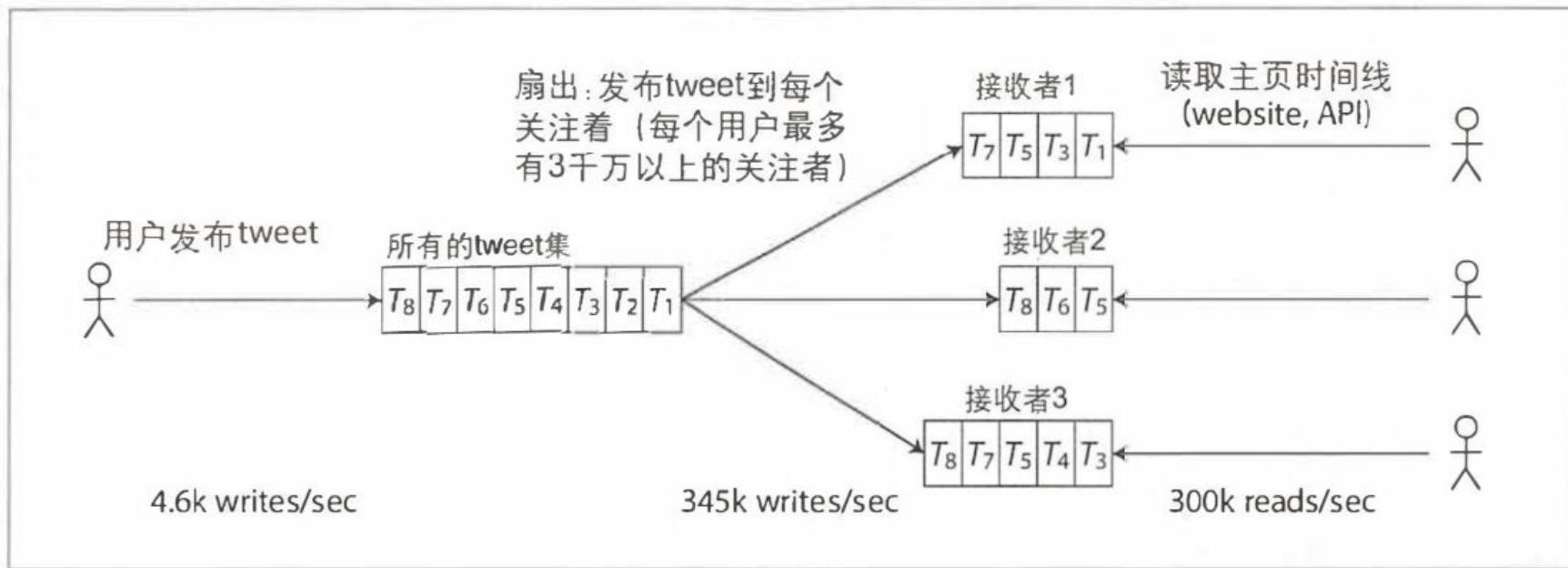


图1-3：Twitter的数据流水线方式来推送tweet，相关参数来自2012.11^[16]

描述性能

描述系统负载之后，接下来设想如果负载增加将会发生什么。有两种考虑方式：

- 负载增加，但系统资源（如CPU、内存、网络带宽等）保持不变，系统性能会发
生什么变化？
- 负载增加，如果要保持性能不变，需要增加多少资源？

Throughput, Response Time, Vertical Scaling, Horizontal Scaling

注3：理想情况下，批量作业的运行时间是数据集的总大小除以吞吐量。在实践中，由于倾斜
(数据在多个工作进程中不均匀分布) 问题，系统需要等待最慢的任务完成，所以运行时
间往往更长。

可维护性

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。

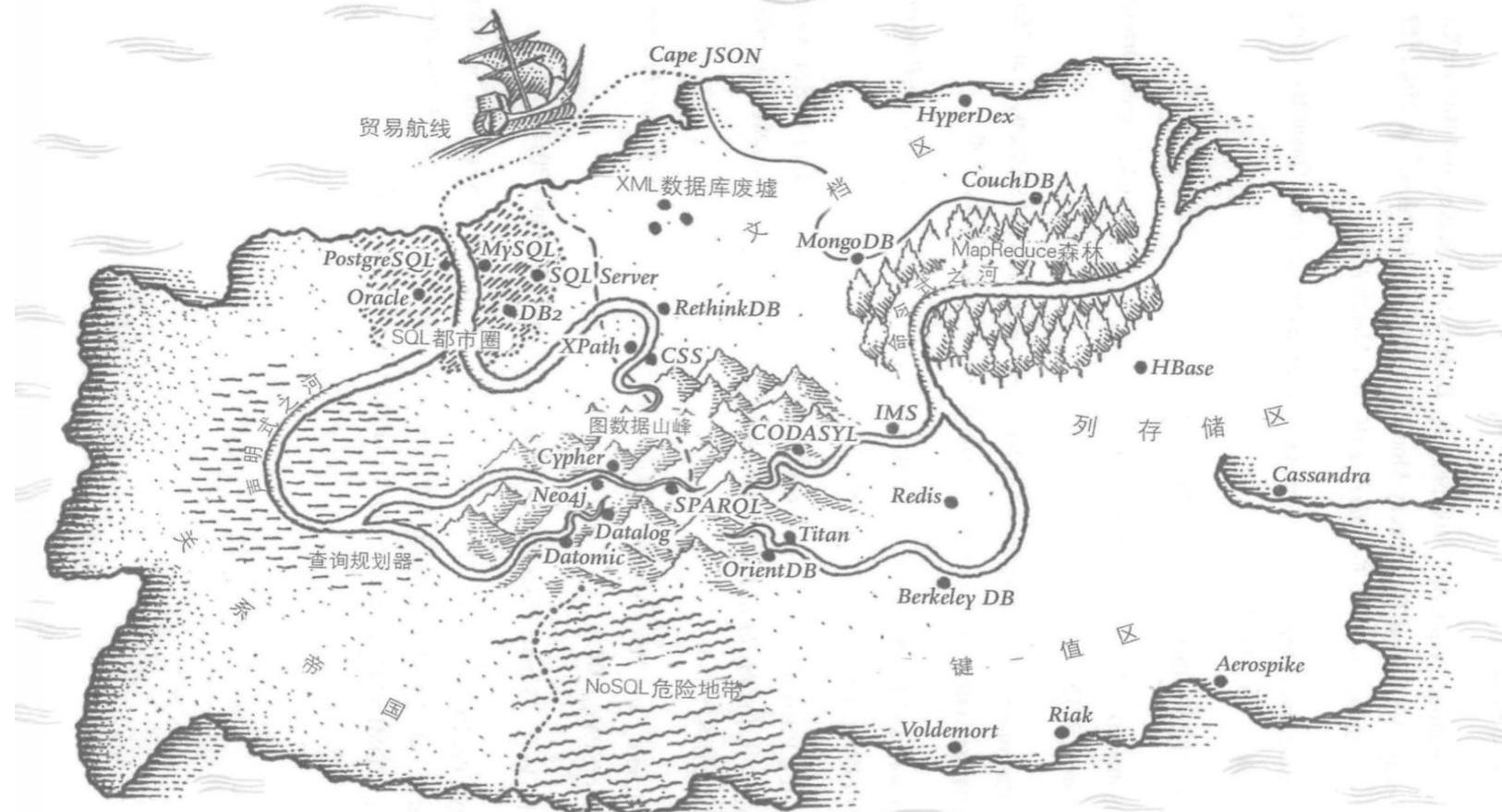
良好的可操作性意味着使日常工作变得简单，使运营团队能够专注于高附加值的任务。数据系统设计可以在这方面贡献很多，包括：

- 提供对系统运行时行为和内部的可观测性，方便监控。
- 支持自动化，与标准工具集成。
- 避免绑定特定的机器，这样在整个系统不间断运行的同时，允许机器停机维护。
- 提供良好的文档和易于理解的操作模式，诸如“如果我做了X，会发生Y”。
- 提供良好的默认配置，且允许管理员在需要时方便地修改默认值。
- 尝试自我修复，在需要时让管理员手动控制系统状态。
- 行为可预测，减少意外发生。

简化系统设计并不意味着减少系统功能，而主要意味着消除意外方面的复杂性，正如Moseley和Marks^[32]把复杂性定义为一种“意外”，即它并非软件固有、被用户所见或感知，而是实现本身所衍生出来的问题。

消除意外复杂性最好手段之一是抽象。一个好的设计抽象可以隐藏大量的实现细节，并对外提供干净、易懂的接口。一个好的设计抽象可用于各种不同的应用程序。这样，复用远比多次重复实现更有效率；另一方面，也带来更高质量的软件，而质量过硬的抽象组件所带来的好处，可以使运行其上的所有应用轻松获益。

我们的目标是可以轻松地修改数据系统，使其适应不断变化的需求，这和简单性与抽象性密切相关：简单易懂的系统往往比复杂的系统更容易修改。这是一个非常重要的理念，我们将采用另一个不同的词来指代数据系统级的敏捷性，即可演化性^[34]。



大多数应用程序是通过一层一层叠加数据模型来构建的。每一层都面临的关键问题是：如何将其用下一层来表示？例如：

1. 作为一名应用程序开发人员，观测现实世界（其中包括人员、组织、货物、行为、资金流动、传感器等），通过对对象或数据结构，以及操作这些数据结构的API来对其建模。这些数据结构往往特定于该应用。
2. 当需要存储这些数据结构时，可以采用通用数据模型（例如JSON或XML文档、关系数据库中的表或图模型）来表示。
3. 数据库工程师接着决定用何种内存、磁盘或网络的字节格式来表示上述JSON/XML/关系/图形数据。数据表示需要支持多种方式的查询、搜索、操作和处理数据。
4. 在更下一层，硬件工程师则需要考虑用电流、光脉冲、磁场等来表示字节。

复杂的应用程序可能会有更多的中间层，例如基于API来构建上层API，但是基本思想相同：每层都通过提供一个简洁的数据模型来隐藏下层的复杂性。这些抽象机制使得不同的人群可以高效协作，例如数据厂商的工程师和使用数据库的应用程序开发人员一起合作。

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于1970年提出的关系模型^[1]：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）。

关系数据库的核心在于商业数据处理，20世纪60年代和70年代主要运行在大型计算机之上。从今天的角度来看，用例看起来很常见，主要是事务处理（包括输入销售和银行交易、航空公司订票、仓库库存）和批处理（例如客户发票、工资单、报告）。

当时的其他数据库迫使应用开发人员考虑数据的内部表示。关系模型的目标就是将实现细节隐藏在更简洁的接口后面。

多年来，在数据存储和查询方面存在着其他许多竞争技术。在20世纪70年代和80年代初期，网络模型和层次模型是两个主要的选择，但最终关系模型主宰了这个领域。对象数据库曾在20世纪80年代后期和90年代初期起起伏伏。XML数据库则出现在21世纪初，但也仅限于利基市场。关系模型的每个竞争者都曾聒噪一时，可惜无一持久^[2]。

采用NoSQL数据库有这样几个驱动因素，包括：

- 比关系数据库更好的扩展性需求，包括支持超大数据集或超高写入吞吐量。
- 普遍偏爱免费和开源软件而不是商业数据库产品。
- 关系模型不能很好地支持一些特定的查询操作。
- 对关系模式一些限制性感到沮丧，渴望更具动态和表达力的数据模型^[5]。

不同的应用程序有不同的需求，某个用例的最佳的技术选择未必适合另一个用例。因此，在可预见的将来，关系数据库可能仍将继续与各种非关系数据存储一起使用，这种思路有时也被称为混合持久化 [3]。

对象-关系不匹配

现在大多数应用开发都采用面向对象的编程语言，由于兼容性问题，普遍对SQL数据模型存在抱怨：如果数据存储在关系表中，那么应用层代码中的对象与表、行和列的数据库模型之间需要一个笨拙的转换层。模型之间的脱离有时被称为阻抗失谐^{注1}。

简历(类似LinkedIn profile)

例如，图2-1展示了如何在关系模式中表示简历（类似LinkedIn profile）。整个简历可以通过唯一的标识符user_id来标识。像first_name和last_name这样的字段在每个用户中只出现一次，所以可以将其建模为users表中的列。然而，大多数人在他们的职业（职位）中有一个以上的工作，并且可能有多个教育阶段和任意数量的联系信息。用户与这些项目之间存在一对多的关系，可以用多种方式来表示：

- 在传统的SQL模型（SQL: 1999之前）中，最常见的规范化表示是将职位、教育和联系信息放在单独的表中，并使用外键引用users表，如图2-1所示。
- 之后的SQL标准增加了对结构化数据类型和XML数据的支持。这允许将多值数据存储在单行内，并支持在这些文档中查询和索引。Oracle、IBM DB2、MS SQL Server和PostgreSQL都不同程度上支持这些功能^[6,7]。一些数据库也支持JSON数据类型，例如IBM DB2、MySQL和PostgreSQL^[8]。
- 第三个选项是将工作、教育和联系信息编码为JSON或XML文档，将其存储在数据库的文本列中，并由应用程序解释其结构和内容。对于此方法，通常不能使用数据库查询该编码列中的值。

<http://www.linkedin.com/in/williamhgates>



Bill Gates
Greater Seattle Area | Philanthropy

Summary

Co-chair of the Bill & Melinda Gates Foundation. Chairman, Microsoft Corporation. Voracious reader. Avid traveler. Active blogger.

Experience

Co-chair • Bill & Melinda Gates Foundation
2000 – Present

Co-founder, Chairman • Microsoft
1975 – Present

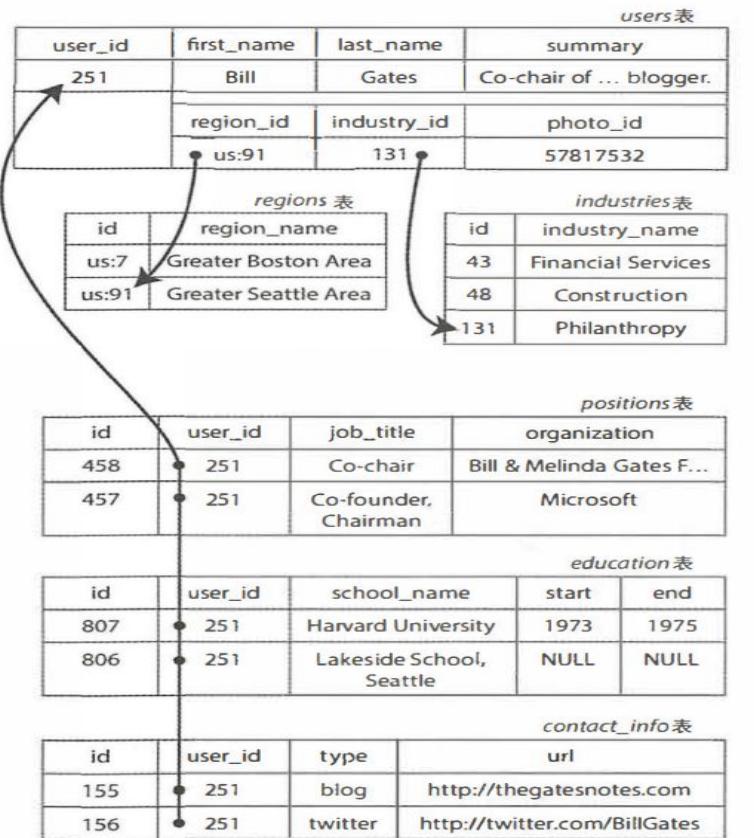
Education

Harvard University
1973 – 1975

Lakeside School, Seattle

Contact Info

Blog: thegatesnotes.com
Twitter: @BillGates



示例2-1：将LinkedIn简历表示为JSON文档

```
{  
    "user_id": 251,  
    "first_name": "Bill",  
    "last_name": "Gates",  
    "summary": "Co-chair of the Bill & Melinda Gates... Active blogger.",  
    "region_id": "us:91",  
    "industry_id": 131,  
    "photo_url": "/p/7/000/253/05b/308dd6e.jpg",  
    "positions": [  
        {"job_title": "Co-chair", "organization": "Bill & Melinda Gates Foundation"},  
        {"job_title": "Co-founder, Chairman", "organization": "Microsoft"}  
],  
    "education": [  
        {"school_name": "Harvard University", "start": 1973, "end": 1975},  
        {"school_name": "Lakeside School, Seattle", "start": null, "end": null}  
],  

```

多对一与多对多的关系

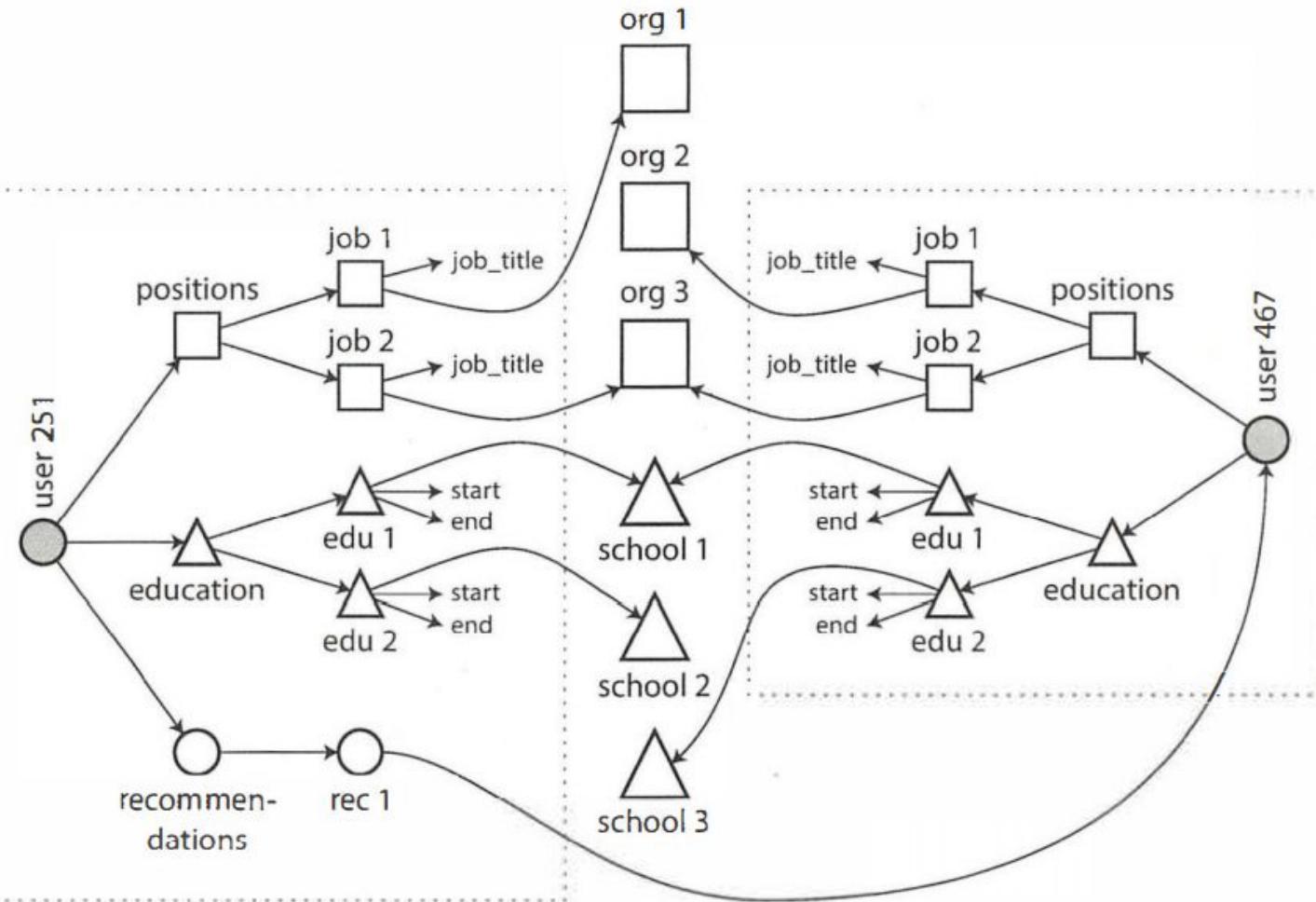
在示例2-1中，`region_id`和`industry_id`定义为ID，而不是纯文本字符串形式，例如“Greater Seattle Area”和“Philanthropy”。为什么这样做呢？

如果用户界面是可以输入地区或行业的自由文本字段，则将其存储为纯文本字符串更有意义。但是，拥有地理区域和行业的标准化列表，并让用户从下拉列表或自动填充器中进行选择会更有优势，这样：

- 所有的简历保持样式和输入值一致。
- 避免歧义（例如，如果存在一些同名的城市）。
- 易于更新：名字只保存一次，因此，如果需要改变（例如，由于政治事件而更改城市名称），可以很容易全面更新。
- 本地化支持：当网站被翻译成其他语言时，标准化的列表可以方便本地化，因此地区和行业可以用查看者的母语来显示。
- 更好的搜索支持：例如，搜索华盛顿州的慈善家可以匹配到这个简历，这是因为地区列表可以将西雅图属于华盛顿的信息编码进来（而从“大西雅图地区”字符串中并不能看出来西雅图属于华盛顿）。

然而这种数据规范化需要表达多对一的关系（许多人生活在同一地区，许多人在同一行业工作），这并不是很适合文档模型。对于关系数据库，由于支持联结操作，可以很方便地通过ID来引用其他表中的行。而在文档数据库中，一对多的树状结构不需要联结，支持联结通常也很弱^{注3}。

如果数据库本身不支持联结，则必须在应用程序代码中，通过对数据库进行多次查询来模拟联结（对于上述例子，地区和行业的列表很小且一段时间内不太可能发生的变化，应用程序可以简单地将它们缓存在内存中。但无论如何，联结的工作其实从数据库转移到了应用层）。



网络模型

网络模型由一个称为数据系统语言会议(Conference on Data System Languages, 数据模型与查询语言I 41 CODASYL)的委员会进行标准化，并由多个不同的数据库厂商实施，它也被称为CODASYL模型

在网络模型中，记录之间的链接不是外键，而更像是编程语言中的指针（会存储在磁盘上）。访问记录的唯一方法是选择一条始于根记录的路径，并沿着相关链接依次访问。这条链接链条也因此被称为访问路径。

在最简单的情况下，访问路径像是遍历链表：从链表的头部开始，一次查看一个记录，直到找到所需的记录。但是在一个多对多关系的世界里，存在多条不同的路径通向相同的记录，使用网络模型的程序员必须在脑海中设法跟踪这些不同的访问路径。

关系模型

相比之下，关系模型所做的则是定义了所有数据的格式：关系（表）只是元组（行）的集合，仅此而已。没有复杂的嵌套结构，也没有复杂的访问路径。可以读取表中的任何一行或者所有行，支持任意条件查询。可以指定某些列作为键并匹配这些列来读取特定行。可以在任何表中插入新行，而不必担心与其他表之间的外键关系^{注4}。

在关系数据库中，[查询优化器](#)自动决定以何种顺序执行查询，以及使用哪些索引。这些选择实际上等价于“访问路径”，但最大的区别在于它们是由查询优化器自动生成的，而不是由应用开发人员所维护，因此不用过多地考虑它们。

如果想用新的方式查询数据，只需声明一个新的索引，查询会自动使用最合适的索引。不需要更改查询即可利用新的索引。因此，关系模型使得应用程序添加新功能变得非常容易

文档数据库的比较

- 1:文档数据库是某种方式的层次模型:即在其父记录中保存了嵌套记录(一对多关系,如图2-1中的positions、education和contact_info),而不是存储在单独的表中
- 2:在表示多对一和多对多的关系时,关系数据库和文档数据库并没有根本的不同:在这两种情况下,相关项都由唯一的标识符引用,该标识符在关系模型中被称为外键,在文档模型中被称为文档引用
- 3:支持文档数据模型的主要论点是模式灵活性,由局部性而带来较好的性能,对某些应用来说,它更接近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

哪种数据模型的应用代码更简单？

如果应用数据具有类似文档的结构（即一对多关系树，通常一次加载整个树），那么使用文档模型更为合适。而关系型模型则倾向于某种数据分解，它把文档结构分解为多个表（如图2-1中的positions、education和contact_info），有可能使得模式更为笨重，以及不必要的应用代码复杂化。

文档模型也有一定的局限性：例如，不能直接引用文档中的嵌套项，而需要说“用户251的职位列表中的第二项”（非常类似于层次模型中的访问路径）。然而，只要文档嵌套不太深，这通常不是问题。

在文档数据库中，对联结的支持不足是否是问题取决于应用程序。例如，在使用文档数据库记录事件发生时间的应用分析程序中，可能永远不需要多对多关系^[19]。

但是，如果应用程序确实使用了多对多关系，那么文档模型就变得不太吸引人。可以通过反规范化来减少对联结的需求，但是应用程序代码需要做额外的工作来保持非规范化数据的一致性。通过向数据库发出多个请求，可以在应用程序代码中模拟联结，但是这也将应用程序变得复杂，并且通常比数据库内的专用代码执行的联结慢。在这些情况下，使用文档模型会导致应用程序代码更复杂、性能更差^[15]。

通常无法一概而论哪种数据模型的应用代码更简单。这主要取决于数据项之间的关系类型。对于高度关联的数据，文档模型不太适合，关系模型可以胜任，而图模型（参阅本章后面的“图状数据模型”）则是最为自然的。

文档模型中的模式灵活性

可以将任意的键一值添加到文档中，并且在读取时，客户端无法保证文档可能包含哪些字段

读时模式（数据的结构是隐式的，只有在读取时才解释），与写时模式（关系数据库的一种传统方法，模式是显式的，并且数据库确保数据写入时都必须遵循）

相对应如果集合中的项由于某种原因（例如数据异构），并不都具有相同的结构，

例如：

- 有许多不同类型的对象，将每种类型的对象都保存在各自的表中不太现实。

- 数据的结构由无法控制的外部系统所决定，而且可能随时改变。

在这些情况下，模式带来的损害大于它所能提供的帮助，无模式文档可能是更自然的数据模型。但是，当所有记录都有相同结构时，模式则是记录和确保这种结构的有效机制。我

查询的数据局部性

局部性优势仅适用需要同时访问文档大部分内容的场景。由于数据库通常会加载整个文档，如果应用只是访问其中的一小部分，则对于大型文档数据来讲就有些浪费。对文档进行更新时，通常会重写整个文档，而只有修改量不改变源文档大小时，原地覆盖更新才更有效^[19]。因此，通常建议文档应该尽量小且避免写入时增加文档大小^[9]。这些性能方面的不利因素大大限制了文档数据库的适用场景。

mei kan dong

文档数据库与关系数据库的融合

大多数关系数据库系统(MySQL除外)都支持XML。其中包括对XML文档进行本地修改，在XML文档中进行索引和查询等，这样应用程序可以获得与文档数据库非常相似的数据模型

文档数据库方面，RethinkDB的查询接口支持和关系型类似的联结，而一些MongoDB驱动程序可以自动解析数据库的引用关系

图状数据模型

我们之前看到，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。

图由两种对象组成：顶点（也称为结点或实体）和边（也称为关系或弧）。很多数据可以建模为图。典型的例子包括：

社交网络

顶点是人，边指示哪些人彼此认识。

Web图

顶点是网页，边表示与其他页面的HTML链接。

小结

关系型数据库很多优点：

- 易于理解 - 二维表很接近我们逻辑上理解世界的方式
- 易于维护 - 社区成熟，服务稳定，数据稳定，高一致性 (Consistency)，读写实时
- 使用灵活 - SQL 语言功能丰富，可以在表间进行 JOIN

关系型数据库缺点也有不少：

- 高并发读写性能较弱 - 为了保证高一致性加锁造成读写性能的牺牲
- 表结构 (Schema) 改动成本高 - 修改表结构时会造成部分数据库服务不可用
- 水平扩展 (Horizontal Scaling) 较难 - 需要解决跨服务器 JOIN，分布式事务等问题

1. 选择 SQL vs NoSQL 考虑以下因素
 - a. ACID vs BASE
 - b. 结构化数据 vs 非结构化数据
 - c. 扩展性
2. 如选择 NoSQL，根据查询模式选择以下四种之一
 - a. Key-value Store
 - b. Wide-column Store
 - c. Document Store
 - d. Graph Store

NoSQL 的优势在于为现代应用提供更合适的数据库方案：

- 横向扩展 (Horizontal Scaling) - 支持海量读写的需求
- 数据复制 (Replication) 和分区 (Partition)
- 更高的可用性 (Availability) - 以牺牲一定的一致性 (Consistency) 为代价
- 提倡反正规化 (Denormalization) - 利用便宜的硬盘空间来提高查询性能
- 多样的数据存储方式 - 使用更符合应用特征的方案节约开发时间
- 简化应用设计初期的存储模型设计 - 支持存储信息更灵活地迭代

NoSQL 的缺点有几个：

- 一致性支持较弱 - 多数 NoSQL 数据库倾向于牺牲一致性来增强可用性
- 不支持 JOIN
- 社区较新

选择 SQL 还是 NoSQL

1:ACID vs BASE

关系型数据库支持 ACID (Atomicity, Consistency, Isolation, Duration) 即原子性, 一致性, 隔离性和持续性。相对而言, NoSQL 采用更宽松的模型 BASE (Basically Available, Soft state, Eventual Consistency) 即基本可用, 软状态和最终一致性。

2: 结构化数据 vs 非结构化数据

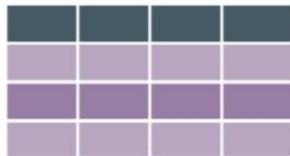
有时候我们的应用包含着非常结构化的数据, 可以简单地放入一个二维表中; 有时候我们的应用需要存储一个键值对, 一个图 (Graph), 一篇文章。前者我们可以采用 SQL 或者 NoSQL, 而后者我们优先考虑 NoSQL 并且选择合适类型的 NoSQL 数据库存储数据。

3: 扩展性 (Scalability)

NoSQL 数据库管理员可以使用数据库自带的功能, 更容易满足高并发的需求。查询时只需要访问一个表, 这样也使得横向扩展更加容易。相反, 如果采用关系型数据库, 传统上它的主要扩展方式是纵向扩展性 (Vertical Scaling) 即采用更大更强的机器。然而, 今天的很多关系型数据库也拥有了一定的横向扩展性支持。相对来说横向扩展对于关系型数据库会更难一点, 会更要求我们好好考虑如何设计表以及进行分片来满足查询的要求。

Types of Databases

Relational (SQL)

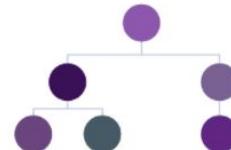


ORACLE

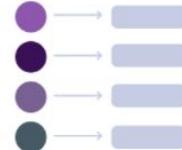


Non-relational (NoSQL)

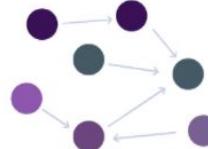
Document



Key-value



Graph



Wide-column



4.1 Key Value Store

这是 NoSQL 数据库的最简单的类型，数据即简单的键值对。这种类型的代表是 Redis，严格的来说，与其称 Redis 为数据库不如称其为缓存，其最合适的应用场景并不是长期存储数据而是将经常访问的请求结果暂时保存下来，避免反复向背后的数据库请求相同的信息。

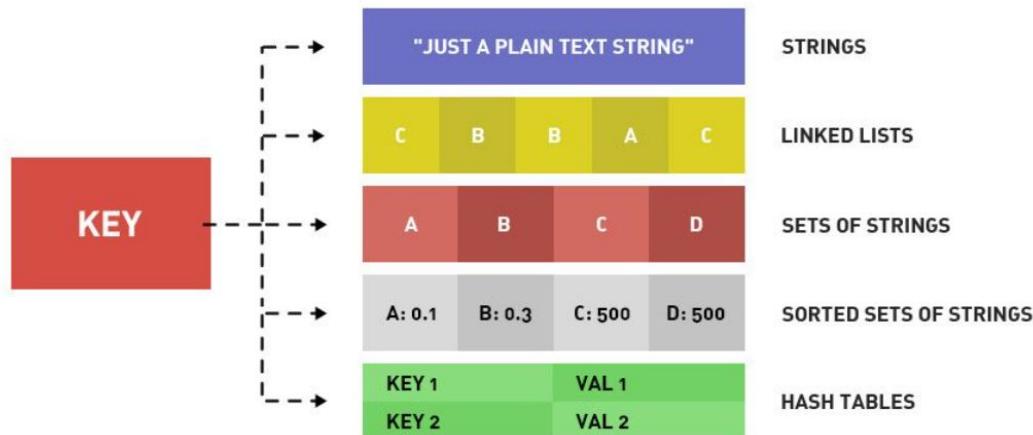


Image from [Deister Software](#)

这类数据库适合存储大量具有 Key-Value 简单查询模式的数据。

4.2 Wide-Column Store

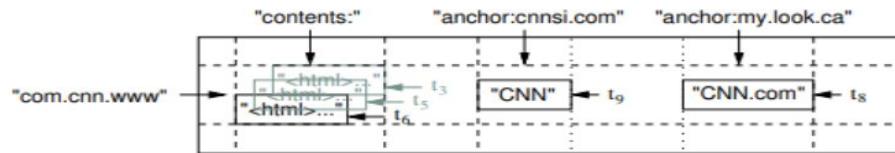


Image from Google BigTable Paper

Wide-column Store 的两个特点可以从上图中看出。

- 可以看做二维版的 Key-value Store，如上图中的“contents”列包含不同时间戳的 HTML 内容。
- 支持 Column Family，如上图中的 anchor Column Family 包含两个包含 CNN 关键词的网站名。

BigTable 是这类数据库的鼻祖，HBase 和 Cassandra 也是这个类型的代表，其中 HBase 相比 Cassandra 提供更强的一致性 (Consistency)。

此类数据库适合存储大量具有可预测的查询模式的数据。

4.3 Document Store

Relational

ID	first_name	last_name	cell	city	year_of_birth	location_x	location_y
1	'Mary'	'Jones'	'516-555-2048'	'Long Island'	1986	'-73.9876'	'40.7574'

ID	user_id	profession
10	1	'Developer'
11	1	'Engineer'

ID	user_id	name	version
20	1	'MyApp'	1.0.4
21	1	'DocFinder'	2.5.7

ID	user_id	make	year
30	1	'Bentley'	1973
31	1	'Rolls Royce'	1965

MongoDB

```
first_name: "Mary",
last_name: "Jones",
cell: "516-555-2048",
city: "Long Island",
year_of_birth: 1986,
location: {
    type: "Point",
    coordinates: [-73.9876, 40.7574]
},
profession: ["Developer", "Engineer"],
apps: [
    { name: "MyApp",
        version: 1.0.4 },
    { name: "DocFinder",
        version: 2.5.7 }
],
cars: [
    { make: "Bentley",
        year: 1973 },
    { make: "Rolls Royce",
        year: 1965 }
]
```

Image from MongoDB Documentation. What is a Document Database?

Document Store 以 JSON 的形式存储数据，这使得它的结构极其灵活，当我们有额外信息需要存储时只需要简单地将新的信息作为一对新的键值对加入 JSON 即可。在 JSON 的基础上，Document Store 同样支持添加 Index 来加速查询。

Document Store 的代表是 MongoDB 和 CouchDB。

此类数据库由于 JSON 的丰富表现力，对各类数据形式的适应性都比较好。

4.4 Graph Store

Connecting Paths

Find all paths, up to 6 degrees, between Joe and Billy

```
MATCH
  path = shortestPath(
    (p1:Person)-[:KNOWS*..6]-(p2:Person)
  )
WHERE
  p1.name = "Joe" AND p2.name = "Billy"
RETURN
  path
```

There's only one shortest path between Joe and Billy in this dataset, so that path is returned.

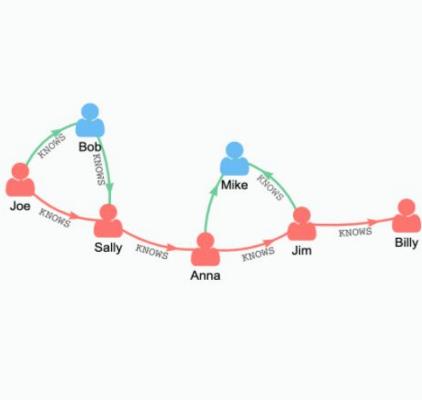


Image from neo4j.com Neo4J Code Sample for Social Use Case

Graph Store 顾名思义是用来建模图的。在 Graph Store 中信息以节点 (Node) 和边 (Edge) 的形式存储，其中边用来表达节点之间的关系。这种数据库可以实现非 Graph Store 数据库无法轻易实现的查询，如上图所表达的从两个节点（人）之间六个边（认识关系）之内的所有路径。

Graph Store 的代表是 Neo4J，很适合社交关系，网络管理和欺诈识别 (Fraud Detection) 等应用场景。

此类数据库专业处理分析图状数据。

