论文部分：（目标1500+字）

|  |
| --- |
| 参考目录：  <https://en.wikipedia.org/wiki/Graph_database>  《Graph Database》 |

图数据库简介：

|  |
| --- |
| 划重点：  重点特性：高度**关联**的数据-复杂而动态的**联系**  主要领域：社交网络，主数据管理，地理空间，推荐系统。  图数据库是处理复杂的，**半结构化**的，紧密**关联**的数据的最好的技术。 |

|  |
| --- |
| 作为NoSQL的一员大将，图数据库近几年被广泛的讨论，其使用者也逐渐增多。不得不说在研究过它的思想后，我也爱上了这种形象自然的建模思路和设计方法。  首先，让我们想想什么是图？图就是节点和联系的集合，其中比较主流的图模型主要是**属性图**，资源描述框架（RDF）三元图和超图。拿最流行的属性图举例，它主要有两个特性。一：包含节点和联系，节点有自己的属性。二：联系有名字和方向，也可以有属性。这个模型在我看来是如此的通用，以至于我认为生活中任何东西都可以用图来建模表示。既然它是一个符合人们思维习惯的建模思路，那它必然也可以抽象出来用于计算机世界里的数据存储。而我们要讨论的图数据库就是建立在这样的基础上的。图数据库使用节点，边/关系(relationship)和属性。节点代表实体，关系代表实体间的联系，属性是节点的相关信息。引用维基百科对图数据库的定义“In [computing](https://en.wikipedia.org/wiki/Computing), a graph database (GDB) is a [database](https://en.wikipedia.org/wiki/Database) that uses [graph structures](https://en.wikipedia.org/wiki/Graph_(data_structure)) for [semantic queries](https://en.wikipedia.org/wiki/Semantic_query) with [nodes](https://en.wikipedia.org/wiki/Node_(graph_theory)), [edges](https://en.wikipedia.org/wiki/Edge_(graph_theory)) and properties to represent and store data. A key concept of the system is the graph (or edge or relationship), which directly relates data items in the store.”  知道了什么是图数据库后，我们就可以对比分析图数据库建模的方法和特点。传统的关系建模流程大致是：草图，er图，映射成表。在过程中需要把数据间的联系以逻辑的形式存储下来。同时应该使用范式，约束等技术整理模型结构，减少数据库中数据冗余，增进数据。也就是规范化。如此，建模过程看上去已经完成了，但实际项目中，需求总是变化的，所以实际生活中常常会继续使用**反规范化技术**。也就是说，在实现关系型解决方案的过程中，为了适应关系模型，我们对物理模型强加了很多变化，如为了获得查询性能，人为制造重复数据和重复结构。这些变化使得概念模型和数据真实的物理布局之间产生了差异。这种概念上的失调，使得将业务需求的变化转化为底层的稳固的关系结构变得非常复杂和繁琐。同时，需求的大量变更，使得**迁移**花销巨大和风险极高。总之，在实际的项目过程中，我们需要经历：设计-规范化-反规范化的过程。  而图建模就去掉了使用复杂数据管理代码来规范化和反规范化数据的这一步骤。我们可以很轻松的将复杂的层次结构抽象出来，又因为其建模思路和人的思维习惯相符，整个过程非常的易理解而又高保真。虽然在建模初期，常见一个陷阱是错误的将本应是节点的数据用联系+联系的属性表示出来，反而丢失了节点本身的内容。但这种错误随着对图的熟悉会越来越少，这是一个设计上容易遇到的问题，但错误并不在图数据库。同样值得注意的一点是：不要小看图的天然可扩展性。相比关系型需要不断确认逻辑设计，图数据库在修改实体和联系上方便了太多，例如，当我们需要新增实体并添加于已有实体间的关系时，我们不需要修改本来实体的内容，只需要新增节点然后添加一条指向已有节点的边即可。  这样分析了图建模的方式后，我们可以总结出图数据库的两大优势： 1.**灵活性**，可扩展性高。不必要一开始就把整个系统的结构设计出来，可以在开发的过程中低成本的增删节点和联系。2.**敏捷性**。图数据库不需要**schema**,所以它缺少以schema为导向的数据管理机制。他的管理通常作用于编程方式，利用测试来驱动数据模型和查询，以及依靠图来断言业务规则。适用于敏捷开发。  那么，图在性能方面的表现足够好吗？让我们继续分析模型建好后的操作。  关系模型使用数据中的信息关联数据。例如，当需要查找所有电话号码包含区号“311”的用户时，它通过遍历整个表，在电话号码字段中查找字符串“311”。这在大型表中是一个耗时的过程，因此关系数据库提供了数据库索引的概念，以提高查询速度。  相比之下，图形数据库直接存储记录之间的关系。同样是查找某用户的电话号码，图数据库使用户有直接指向电话记录的指针，而不是通过用户id查找用户表里电话号码字段的内容。也就是说，选择一个用户后，可以通过指针直接得到对应的电话号码，而不需要搜索用户表来查找匹配的记录。这可以消除昂贵的连接（join）操作。这种优势在进行深度较大的搜索时更加明显。当系统需要查找和用户a的电话号码前六位相同的其他用户最近在某地的购买的物品的生产地的最普遍的姓时，关系型需要多次搜索匹配，而图数据库只需要查找节点，顺着边继续进入下一个深度即可。从技术上讲，图数据库共用了O（log（ n））+ O（1）时间，即大致相对于数据大小的对数。相比之下，关系版本将是多个 O（log（ n））查找。由此我们可以得出结论：图数据库具有良好的查询**性能**。因为哪怕在数据集增大的情况下，查询也总是和图的一部分相关，因此，每个查询的执行时间只和满足查询条件的那部分遍历的图的大小而不是整个图的大小成正比。（使用免索引链接，图数据库可以将复杂的join查询转化为快速的图遍历，因此无论数据集有多大，都可以维持毫秒级的性能。）  那么，总结以上的分析。图数据库适用于处理**半结构化**的，紧密**关联且变动性大的**的数据。现实生活中，最适合使用图数据库的领域包括：社交网络，推荐引擎，地理空间（基于位置的服务），授权和访问控制。不适用更新全部或某个子集的实体。  接下来我们尝试**将图数据库与场景结合起来**。在过程中继续分析其优劣。  根据拿到的资料，我们简单拿方便面多维度属性和定量描述举例。具体的场景如下：有品牌A,B,C生产了口味a,b,c,d等n种方便面，每种方便面有自己表层物品属性，如编号，生产日期，规格，风味等。厂商为了更好的分析食品口味，研究市场对口味的喜好，请了m名测试员从色泽，光滑性，复水性，韧性，黏性，耐泡性等多个角度进行打分。同时，在口味方面，厂商进行了细化研究，通过多种评价标准，如强度等级，麻度等级，斯科维尔指数，描述词，根据方便面在不同时间的表现进行评级，得出了味道的时间强度评价和时间频率记录。当食物投放市场后，厂商进行了反馈调查，拿到了o位用户针对n种方便面的喜恶数据。  分析以上场景，我认为大致可以分为三个模型。  模型一：方便面的属性记录。这里主要包括表层属性，分数情况。  模型二：测试员分析过程中的打分记录。这里数据模型简单，数据量大。需要注意的是如何把时间因素记录得当。  模型三：用户的反馈记录。这里可以为后续的推荐功能，产量设计提供数据支持。  分析完数据模型后，我们可以结合图数据库的特点来看如何设计数据存储模型。在以上提到的食物领域，我认为源数据的主要特点是结构简单，数据量大，变动性高。那么，个人比较遗憾的发现，图数据库最大的优点——联系在这个场景里并没有很好的得到表现。因为食物之间并不会有很多的事件交集，显然无法像社交网络那样完美的体现图数据库中联系一等公民的地位。  那么，这就意味着图数据库不适合了吗？我认为也不是这样的。虽然事物间的联系不多，但从以上描述中可以清晰看到一个食物的属性是很多的。特别是在口味这里，作为食物最重要的特性，我们可以很明显的看到厂商在这里用了多种角度分析，在味觉粒度和时间粒度上都做了细化。也就是说，一个物品的属性之间是有联系的。这样无疑也可以让图数据库发挥特长。而且，在反馈分析这里，我认为图数据库是非常适合的。不同人对不同事物有不同的喜好。这个过程可以看出有3部分的联系。人和食物（是否喜爱），人和人（口味是否类似），食物和食物（是否是统一风格大类）值得注意的是，我们在介绍图数据库时提到过它擅长的领域，正如当时分析的，做推荐系统图数据库具有天然的优势，它带来的对性能的提高不容小觑。  同时，正如之前提到的，本情景下，需求变更快，源数据的变动性高，换句话说就是对数据模型的可扩展性有很高的要求。如果整体模型太过僵化，那么食物每增加一种评价方式，每发生一次某属性的突然增删，测试员每提供不完全统一的数据，数据库每涉及到迁移（无论大小），无疑都是一次痛苦的重新设计，系统的维护成本升高不说，万一由于操作者（开发者）经验问题，很可能导致冗余数据/丢失数据/模型臃肿等恶性情况的发生。这大大增大了项目的风险，也有很大概率降低系统的性能。  考虑到如上的可能性，图数据库瞬间变得非常亲切可人了。它天生良好的可扩展性无疑非常适合于需求变更快的敏捷开发模式。它没有那些复杂的，恼人的物理模型，概念模型转换。建立表结构是如此的简单，在原有表上新增属性和联系只需新增节点并画上线即可。实在是非常的高效方便。  再其次，在数据的查找方面，图数据库更有的可说。介于它的查询只发生在图的局部，不需要遍历整个库，而且由于它联系分明，在深层次的数据查询方面比传统的好上太多，传统关系型查询性能随深度增加是指数级减小的，而对于图数据库，这基本是线性的，当深度大于3时，其中的差距实在需要引起注意。  然而，虽然我这么喜爱这种“又聪明又懒惰”的图数据库建模方式，总体来说，它也还是有表现不佳的地方。这里主要体现在对时间流的掌控上，图的亮点在于实体间的联系，而时间是流式的，每个时间都有新的不同的数据，个人认为，这个变化的过程用图数据库来记录实在是卡在短板上了。  这里画出图数据库的模型设计：  模型一：    性能测试： |

|  |
| --- |
| Neo4j是面向对象基于[Java](http://lib.csdn.net/base/17)的 ，被设计为一个建立在Java之上、可以直接嵌入应用的数据存储。此后，其他语言和平台的支持被引入，Neo4j社区获得持续增长，获得了越来越多的技术支持者。目前已支持.NET、Ruby、Python、[Node.js](http://lib.csdn.net/base/30" \o "Node.js知识库" \t "_blank)及PHP等。因此，不管是什么项目，没有理由不引入Neo4j。 |

|  |
| --- |
| **图数据库的明显优点：**   1. **性能**提升，特别在全局查询中。因为哪怕在数据集增大的情况下，查询也总是和图的一部分相关，因此，每个查询的执行时间只和满足查询条件的那部分遍历的图的大小而不是整个图的大小成正比。 2. **灵活性**，可扩展性高。不必要一开始就把整个系统的结构设计出来，可以在开发的过程中低成本的增删。 3. **敏捷性**。图数据库不需要**schema**,所以它缺少以schema为导向的数据管理机制。他的管理通常作用于编程方式，利用测试来驱动数据模型和查询，以及依靠图来断言业务规则。试用与敏捷开发。   图数据库的特点：   1. 联系   只有图数据库在处理联系上鹤立鸡群  关系型数据库缺少联系。（表连接，稀疏行，非空检查逻辑，复杂关联层数越多时空复杂度激增，需要处理schema…etc）而其他NoSQL缺少联系（这里主要指除了图数据库外的文件系统，列族，键值对）聚合存储模型中大量添加外键（内嵌映射结构），无反向指针。 |
| **对比关系建模与图建模。**   1. 关系建模（草图，er图，映射成表）但需求总是变化的，所以实际生活中常常使用的是**反规范化技术**（我个人觉得这部分的阐述是本书的精华之一）。总体的过程是“设计-规范化-反规范化”。   反规范化中会为了获得查询性能，人为制造重复数据。会造成大量**冗余数据**和**复杂结构**，也就是说，在实现关系型解决方案的过程中，为了适应关系模型，我们对白板上的模型强加了很多变化，这些变化使得概念模型和数据真实的物理布局之间产生了鸿沟，这种概念上的失调，使得将业务需求的变化转化为底层的稳固的关系结构变得非常复杂和繁琐。  有人认为这个过程虽然花时间不短，但建造一个高性能的关系模型所花费的经历相对开发整个项目的工作来说只是较小的一部分。但，实际生活中，需求的大量变更，使得**迁移**花销巨大和风险极高。  而图建模就去掉了使用复杂数据管理代码来规范化和反规范化数据的这一步骤。   1. 图的建模：高保真，易理解，对于跨域得心应手。但常见一个陷阱是错误的将本应是节点的数据用联系+联系的属性表示出来，反而丢失了节点本身的内容。简单的说，在建模时注意区别实体和联系，不要将实体误认为联系敷衍过去。一般来说，不要把实体建模成联系。应该用联系来传达实体之间是如何联系的，以及这些联系的质量。 2. 同样值得注意的一点是：不要小看图的天然可扩展性。图数据库的建模方式（一个个建立节点，建立联系）在关系型数据库看来可能是往db里猛灌大量详细数据，似乎是很粗糙的。但在图里，这样做是很自然的。 |

|  |
| --- |
| 如何**将图数据库与场景结合起来**。过程中继续分析其优劣。  在食物领域，根据拿到的资料，我进行了整理和模型设计。这里简单拿方便面多维度属性和定量描述举例。具体的场景如下：有品牌A,B,C生产了口味a,b,c,d等n种方便面，每种方便面有自己表层物品属性，如编号，生产日期，规格，风味等。厂商为了更好的分析食品口味，研究市场对口味的喜好，请了m名测试员从色泽，光滑性，复水性，韧性，黏性，耐泡性等多个角度进行打分。同时，在口味方面，厂商进行了细化研究，通过多种评价标准，如强度等级，麻度等级，斯科维尔指数，描述词，根据方便面在不同时间的表现进行评级，得出了味道的时间强度评价和时间频率记录。当食物投放市场后，厂商进行了反馈调查，拿到了o位用户针对n种方便面的喜恶数据。  分析以上场景，我认为大致可以分为三个模型。  模型一：方便面的属性记录。这里主要包括表层属性，分数情况。  模型二：测试员分析过程中的打分记录。这里数据模型简单，数据量大。需要注意的是如何把时间因素记录得当。  模型三：用户的反馈记录。这里可以为后续的推荐功能，产量设计提供数据支持。  分析完数据模型后，我们可以结合图数据库的特点来看如何设计数据存储模型。在以上提到的食物领域，我认为源数据的主要特点是结构简单，数据量大，变动性高。那么，个人比较遗憾的发现，图数据库最大的优点——联系在这个场景里并没有很好的得到表现。因为食物之间并不会有很多的事件交集，显然无法像社交网络那样完美的体现图数据库中联系一等公民的地位。  那么，这就意味着图数据库不适合了吗？我认为也不是这样的。虽然事物间的联系不多，但从以上描述中可以清晰看到一个食物的属性是很多的。特别是在口味这里，作为食物最重要的特性，我们可以很明显的看到厂商在这里用了多种角度分析，在味觉粒度和时间粒度上都做了细化。也就是说，一个物品的属性之间是有联系的。这样无疑也可以让图数据库发挥特长。而且，在反馈分析这里，我认为图数据库是非常适合的。不同人对不同事物有不同的喜好。这个过程可以看出有3部分的联系。人和食物（是否喜爱），人和人（口味是否类似），食物和食物（是否是统一风格大类）值得注意的是，我们在介绍图数据库时提到过它擅长的领域，正如当时分析的，做推荐系统图数据库具有天然的优势，它带来的对性能的提高不容小觑。  同时，正如之前提到的，本情景下，需求变更快，源数据的变动性高，换句话说就是对数据模型的可扩展性有很高的要求。如果整体模型太过僵化，那么食物每增加一种评价方式，每发生一次某属性的突然增删，测试员每提供不完全统一的数据，数据库每涉及到迁移（无论大小），无疑都是一次痛苦的重新设计，系统的维护成本升高不说，万一由于操作者（开发者）经验问题，很可能导致冗余数据/丢失数据/模型臃肿等恶性情况的发生。这大大增大了项目的风险，也有很大概率降低系统的性能。  考虑到如上的可能性，图数据库瞬间变得非常亲切可人了。它天生良好的可扩展性无疑非常适合于需求变更快的敏捷开发模式。它没有那些复杂的，恼人的物理模型，概念模型转换。建立表结构是如此的简单，在原有表上新增属性和联系只需新增节点并画上线即可。实在是非常的高效方便。  再其次，在数据的查找方面，图数据库更有的可说。介于它的查询只发生在图的局部，不需要遍历整个库，而且由于它联系分明，在深层次的数据查询方面比传统的好上太多，传统关系型查询性能随深度增加是指数级减小的，而对于图数据库，这基本是线性的，当深度大于3时，其中的差距实在需要引起注意。  然而，虽然我这么喜爱这种“又聪明又懒惰”的图数据库建模方式，总体来说，它也还是有表现不佳的地方。这里主要体现在对时间流的掌控上，图的亮点在于实体间的联系，而时间是流式的，每个时间都有新的不同的数据，个人认为，这个变化的过程用图数据库来记录实在是卡在短板上了。  这里画出图数据库的模型设计：  模型一： |

代码：

Cmd:

cd G:\GITHUB\GraduationDBS\neo4j-community-3.0.6-windows\neo4j-community-3.0.6\bin

neo4j.bat console

然后访问<http://localhost:7474/browser/>

Neo4j-test

|  |
| --- |
| 划重点：  重点特性：高度**关联**的数据-复杂而动态的**联系**  主要领域：社交网络，主数据管理，地理空间，推荐系统。  图数据库是处理复杂的，**半结构化**的，紧密**关联**的数据的最好的技术。 |

问题：

|  |
| --- |
| 1. 图不需要schema?那它用的是什么管理机制？ 2. Schema具体体现在哪些方面？ 3. 什么叫聚合存储？ |

《图数据库》笔记：

|  |
| --- |
| 第一二章：   1. 图领域主要划分为：图数据库和图计算引擎。 2. 主流的图模型主要是**属性图**（最主流），资源描述框架（RDF）三元图和超图。 3. 图数据库需要注意的两大特性：底层存储和处理引擎。    1. 图数据库运用的底层存储可能是**原生图存储**，或者是将图数据库序列化，保存到其他通用的数据存储中。    2. 使用**原生图处理**来代表使用**免索引链接**的图数据库。 4. 市场上有的一些图数据库： 5. 图数据库的明显优点：    1. **性能**提升，特别在全局查询中。因为哪怕在数据集增大的情况下，查询也总是和图的一部分相关，因此，每个查询的执行时间只和满足查询条件的那部分遍历的图的大小而不是整个图的大小成正比。    2. **灵活性**，可扩展性高。不必要一开始就把整个系统的结构设计出来，可以在开发的过程中低成本的增删。    3. **敏捷性**。图数据库不需要**schema**,所以它缺少以schema为导向的数据管理机制。他的管理通常作用于编程方式，利用测试来驱动数据模型和查询，以及依靠图来断言业务规则。试用与敏捷开发。 6. 只有图数据库在处理联系上鹤立鸡群    1. 关系型数据库缺少联系。（表连接，稀疏行，非空检查逻辑，复杂关联层数越多时空复杂度激增，需要处理schema…etc）    2. NoSQL缺少联系（这里主要指除了图数据库外的文件系统，列族，键值对）聚合存储模型中大量添加外键（内嵌映射结构），无反向指针。 |
| 第三章：使用图进行数据建模   1. 属性图，Cypher基础用法（start, match, return, where, create unique…etc） 2. 对比关系建模与图建模。    1. 关系建模（草图，er图，映射成表）但需求总是变化的，所以实际生活中常常使用的是**反规范化技术**（我个人觉得这部分的阐述是本书的精华之一）。总体的过程是“设计-规范化-反规范化”。   反规范化中会为了获得查询性能，人为制造重复数据。会造成大量**冗余数据**和**复杂结构**，也就是说，在实现关系型解决方案的过程中，为了适应关系模型，我们对白板上的模型强加了很多变化，这些变化使得概念模型和数据真实的物理布局之间产生了鸿沟，这种概念上的失调，使得将业务需求的变化转化为底层的稳固的关系结构变得非常复杂和繁琐。  有人认为这个过程虽然花时间不短，但建造一个高性能的关系模型所花费的经历相对开发整个项目的工作来说只是较小的一部分。但，实际生活中，需求的大量变更，使得**迁移**花销巨大和风险极高。  而图建模就去掉了使用复杂数据管理代码来规范化和反规范化数据的这一步骤。   * 1. 图的建模：高保真，易理解，对于跨域得心应手。但常见一个陷阱是错误的将本应是节点的数据用联系+联系的属性表示出来，反而丢失了节点本身的内容。简单的说，在建模时注意区别实体和联系，不要将实体误认为联系敷衍过去。一般来说，不要把实体建模成联系。应该用联系来传达实体之间是如何联系的，以及这些联系的质量。   2. 同样值得注意的一点是：不要小看图的天然可扩展性。图数据库的建模方式（一个个建立节点，建立联系）在关系型数据库看来可能是往db里猛灌大量详细数据，似乎是很粗糙的。但在图里，这样做是很自然的。 |
| 第四章：构建基于图数据库的应用 |
|  |
|  |
| 第七章：使用图论预分析：   1. 深度优先搜索（带有决策的）和广度优先搜索（Dijkstra算法，A\*算法） 2. 图论和预测建模（三元闭包，结构平衡的三元闭包，局部桥） |

《Neo4j实战》

维基百科中文机翻

|  |
| --- |
| 在[计算中](https://en.wikipedia.org/wiki/Computing" \o "计算)，**图形数据库**（**GDB**[[1]](https://en.wikipedia.org/wiki/Graph_database#cite_note-1)）是一个[数据库](https://en.wikipedia.org/wiki/Database" \o "数据库)，它使用[图形结构](https://en.wikipedia.org/wiki/Graph_(data_structure))进行[语义查询](https://en.wikipedia.org/wiki/Semantic_query)，并使用[节点](https://en.wikipedia.org/wiki/Node_(graph_theory))，[边](https://en.wikipedia.org/wiki/Edge_(graph_theory))和属性来表示和存储数据。系统的一个关键概念是*图形*（或*边缘*或*关系*），它直接关联商店中的数据项。这些关系允许商店中的数据直接链接在一起，并且在很多情况下通过一个操作进行检索。  这与[关系型数据库](https://en.wikipedia.org/wiki/Relational_database)形成了鲜明对比，[关系型数据库](https://en.wikipedia.org/wiki/Relational_database)借助关系数据库管理系统，允许管理数据而不强加物理记录链等实施方面; 例如，数据之间的链接以逻辑级存储在数据库本身中，并且可以使用关系代数运算（例如*[连接](https://en.wikipedia.org/wiki/Join_(SQL)" \o "加入（SQL）)*）来操作并返回相关逻辑格式的相关数据。借助数据库管理系统在物理层面（如使用索引）可以执行关系查询，这可以在不修改数据库的逻辑结构的情况下提高性能。  图形数据库，通过设计，可以简单快速地检索[[*需要的*](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]复杂的层次结构，这些结构很难建模[[*根据谁来*](https://en.wikipedia.org/wiki/Wikipedia:Manual_of_Style/Words_to_watch#Unsupported_attributions)建模[*？*](https://en.wikipedia.org/wiki/Wikipedia:Manual_of_Style/Words_to_watch#Unsupported_attributions)]在关系系统中。图形数据库与20世纪70年代的[网络模型](https://en.wikipedia.org/wiki/Network_model" \o "网络模型)数据库相似，都表示通用图形，但网络模型数据库在较低的抽象层次上运行[[2]，](https://en.wikipedia.org/wiki/Graph_database#cite_note-Gutierrez-2)并且在边缘链上缺乏简单的遍历。[[3]](https://en.wikipedia.org/wiki/Graph_database#cite_note-3)  图形数据库的底层存储机制可能会有所不同。一些依赖于关系引擎并将图形数据“存储”在[表格中](https://en.wikipedia.org/wiki/Table_(database)" \o "表（数据库）)（尽管表格是逻辑元素，因此该方法在图形数据库，图形数据库管理系统和数据的物理设备之间施加了另一个抽象级别实际上是存储的）。其他人使用[键值存储](https://en.wikipedia.org/wiki/Key-value_store" \o "键值存储)或[面向文档的数据库](https://en.wikipedia.org/wiki/Document-oriented_database)进行存储，这使得它们本身就是[NoSQL](https://en.wikipedia.org/wiki/NoSQL)结构。大多数[[*根据谁？*](https://en.wikipedia.org/wiki/Wikipedia:Manual_of_Style/Words_to_watch#Unsupported_attributions)]基于非关系存储引擎的图形数据库也增加了*标签*或*属性*的概念，它们本质上是指向另一个文档的指针。这可以将数据元素分类以便于*集体*检索。  从图形数据库中检索数据需要使用除[SQL](https://en.wikipedia.org/wiki/SQL)以外的[查询语言](https://en.wikipedia.org/wiki/Query_language)，该[语言](https://en.wikipedia.org/wiki/Query_language)专为处理关系系统中的数据而设计，因此不能“优雅地”处理遍历图。截至2017年，没有一种统一的图形查询语言像[SQL一样](https://en.wikipedia.org/wiki/SQL)被用于关系数据库，并且存在多种系统，通常与一种产品紧密相关。一些标准化工作已经发生，导致像[Gremlin](https://en.wikipedia.org/wiki/Gremlin_(programming_language))，[SPARQL](https://en.wikipedia.org/wiki/SPARQL" \o "SPARQL)和[Cypher](https://en.wikipedia.org/wiki/Cypher_Query_Language" \o "Cypher查询语言)这样的多厂商查询语言。除了具有查询语言接口之外，还可以通过[应用程序编程接口](https://en.wikipedia.org/wiki/Application_programming_interface)访问某些图形数据库 （蜜蜂）。  [https://upload.wikimedia.org/wikipedia/commons/thumb/3/3a/GraphDatabase_PropertyGraph.png/308px-GraphDatabase_PropertyGraph.png](https://en.wikipedia.org/wiki/File:GraphDatabase_PropertyGraph.png)  图形数据库使用节点，属性和边。  图数据库基于[图论](https://en.wikipedia.org/wiki/Graph_theory)，并使用节点，边和属性。   * *节点*表示实体，例如人员，企业，账户或任何其他要跟踪的项目。他们大致相当于*纪录*，*关系*，或*排*在关系数据库或*文件*中的文档数据库。 * *边缘*（也称为*图*或*关系*）是将节点连接到其他节点的线; 他们代表了他们之间的关系。检查节点，属性和边的连接和互连时会出现有意义的模式。边缘是图形数据库中的关键概念，代表了其他系统中不直接实现的抽象。 * *属性*是节点的相关信息。例如，如果*维基百科*是节点之一，它可能与诸如*网站*，*参考资料*或*以字母w开头的单词之类的*属性相关，这取决于*维基百科的*哪些方面与给定数据库密切相关。   关系模型使用数据中的信息收集数据。例如，可以查找所有电话号码包含区号“311”的“用户”。这可以通过搜索选定的数据存储区或[表格来](https://en.wikipedia.org/wiki/Table_(database))查找，在所选的电话号码字段中查找字符串“311”。这在大型表中可能是一个耗时的过程，因此关系数据库提供了[数据库索引](https://en.wikipedia.org/wiki/Database_index)的概念，它允许像这样的数据存储在较小的子表中，仅包含选定的数据和[唯一的键](https://en.wikipedia.org/wiki/Unique_key)（或*主键*）的记录是它的一部分。如果电话号码被编入索引，则在较小的索引表中会发生相同的搜索，收集匹配记录的密钥，然后使用这些密钥在主数据表中查找记录。一般情况下，这些表格是物理存储的，因此这些键上的查找速度很快。[[4]](https://en.wikipedia.org/wiki/Graph_database#cite_note-from-4)  关系数据库*本质上*并不包含记录之间固定关系的想法。相反，相关数据通过将一个记录的唯一键存储在另一个记录的数据中而彼此链接。例如，包含用户电子邮件地址的表可能userpk包含一个名为的数据项，其中包含与其关联的用户记录的主键。为了链接用户及其电子邮件地址，系统首先查找选定的用户记录主键，userpk在电子邮件表的列中查找这些键（或者更可能是索引），提取电子邮件数据，然后然后链接用户和电子邮件记录以制作包含所有选定数据的复合记录。这个操作被称为[连接](https://en.wikipedia.org/wiki/Join_(SQL))，可能在计算上是昂贵的。根据查询的复杂程度，连接数量以及各种键的索引，系统可能需要搜索多个表和索引，收集大量信息，然后对其进行排序以将其匹配在一起。[[4]](https://en.wikipedia.org/wiki/Graph_database#cite_note-from-4)  相比之下，图形数据库直接存储记录之间的关系。而不是通过查找其中的用户密钥找到电子邮件地址userpk列，用户记录具有直接指向电子邮件地址记录的指针。也就是说，选择一个用户后，指针可以直接跟在电子邮件记录上，不需要搜索电子邮件表来查找匹配的记录。这可以消除昂贵的连接操作。例如，如果搜索地区代码为“311”的用户的所有电子邮件地址，则引擎将首先执行常规搜索以在“311”中查找用户，但是通过遵循在“311”中找到的链接来检索电子邮件地址那些记录。关系数据库首先会找到“311”中的所有用户，提取pk列表，用这些pk对电子邮件表中的任何记录执行另一次搜索，并将匹配的记录链接在一起。对于这些类型的常见操作，[[4]](https://en.wikipedia.org/wiki/Graph_database#cite_note-from-4)  当执行多于一个级别的搜索时，图表方法的真实价值变得明显。例如，考虑在“311”区域代码中搜索具有“订户”（将用户链接到其他用户的表）的用户。在这种情况下，关系数据库必须首先查找区号为“311”的所有用户，然后在订户表中查找任何这些用户，然后查看用户表以检索匹配的用户。相反，图形数据库将查找“311”中的所有用户，然后通过订户关系跟随后向链路来查找订户用户。这避免了几次搜索，查找以及内存占用所有来自构建输出所需的多个记录的临时数据。从技术上讲，[*O*](https://en.wikipedia.org/wiki/Big_O_notation)（log（*n*））+ O（1）时间，即大致相对于数据大小的对数。相比之下，关系版本将是多个*O*（log（*n*））查找，加上更多时间来加入所有数据。[[4]](https://en.wikipedia.org/wiki/Graph_database#cite_note-from-4)  图检索的相对优势随着查询的复杂性而增加。例如，有人可能想要知道“那部与潜艇相关的电影，与那位在”飘“中扮演主角的其他演员在电影中的演员。这首先要求系统找到“ [飘”中](https://en.wikipedia.org/wiki/Gone_with_the_Wind_(film))的演员，找到他们所有的电影，找到所有这些电影中不是“飘”的主角的所有演员，然后找到他们所在的所有电影，最后将该列表过滤为那些含有“潜艇”。在关系数据库中，这将需要通过电影和演员表进行几次单独搜索，对潜艇电影进行另一次搜索，找到这些电影中的所有演员，然后比较（大）收集的结果。相比之下，图形数据库将简单地从“飘”到“ [克拉克盖博”](https://en.wikipedia.org/wiki/Clark_Gable)，收集他所在电影的链接，将这些电影中的链接收集到其他演员，然后按照这些演员的链接回到电影列表。然后可以搜索结果列表的电影“潜艇”。所有这些都可以通过一次搜索完成。[[5]](https://en.wikipedia.org/wiki/Graph_database#cite_note-examples-5)  属性为这个结构添加了另一个抽象层，这也改进了许多常见的查询。属性本质上是可以应用于任何记录的标签，或者在某些情况下还可以应用于边缘。例如，可以将Clark Gable标记为“演员”，然后让系统快速找到所有的演员记录，而不是导演或摄像机操作员。如果边缘上的标签被允许，还可以将“飘”和“克拉克盖博”之间的关系标记为“主角”，并且通过搜索电影“飘”中的“主角”“演员”数据库将产生[Vivien Leigh](https://en.wikipedia.org/wiki/Vivien_Leigh)，[Olivia de Havilland](https://en.wikipedia.org/wiki/Olivia_de_Havilland)和克拉克盖博。等价的SQL查询将不得不依赖表中链接人员和电影的表中添加的数据，从而增加查询语法的复杂性。这些标签可以在某些情况下提高搜索性能，但通常在为最终用户提供额外语义数据方面更为有用。[[5]](https://en.wikipedia.org/wiki/Graph_database#cite_note-examples-5)  关系数据库非常适合平面数据布局，数据之间的关系是一层或两层深度。例如，会计数据库可能需要查找给定客户的所有发票的所有行项目，即三连接查询。图数据库旨在包含更多链接的数据集。它们特别适合[社交网络](https://en.wikipedia.org/wiki/Social_networking)系统，其中“朋友”关系基本上是无限的。这些属性使图形数据库自然适用于在线系统和[大数据](https://en.wikipedia.org/wiki/Big_data)环境中日益普遍的搜索类型。出于这个原因，图形数据库正在变得非常受[Facebook](https://en.wikipedia.org/wiki/Facebook)，[Google](https://en.wikipedia.org/wiki/Google)，[Twitter](https://en.wikipedia.org/wiki/Twitter)等大型在线系统的欢迎，以及记录之间有深层链接的类似系统。 属性[ [编辑](https://en.wikipedia.org/w/index.php?title=Graph_database&action=edit&section=2)] 与[关系数据库](https://en.wikipedia.org/wiki/Relational_database)相比，图数据库对于关联数据集[[*需要的引用*](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]往往更快，并且更直接地映射到[面向对象](https://en.wikipedia.org/wiki/Object-oriented_programming)应用程序的结构。他们可以扩展更多的自然[[*来源请求*](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]，因为他们通常不需要昂贵的大型数据集的[加入](https://en.wikipedia.org/wiki/Join_(SQL))操作（在逻辑和物理层面上执行非最佳设计的数据库时，这些代价昂贵）。由于他们对硬性架构的依赖程度较低，因此它们更适合通过演进架构管理临时数据和更改数据。相反，关系数据库管理系统在大量数据元素上执行相同操作时通常更快，从而允许以其自然结构操纵数据。  图形数据库是图形查询的强大工具。例如，计算图中两个节点之间的最短路径。其他类似图形的查询可以以自然的方式在图形数据库上执行（例如图形的直径计算或社区检测）。  在图形数据库的前期历史中，在20世纪60年代中期，[导航数据库](https://en.wikipedia.org/wiki/Navigational_database)（如[IBM](https://en.wikipedia.org/wiki/IBM)的[IMS](https://en.wikipedia.org/wiki/IBM_Information_Management_System)在其[分层模型中](https://en.wikipedia.org/wiki/Hierarchical_database_model)支持树形结构），但严格的树形结构可以通过虚拟记录进行规避。[[6]](https://en.wikipedia.org/wiki/Graph_database#cite_note-6)[[7]](https://en.wikipedia.org/wiki/Graph_database#cite_note-7)  图形结构可以在20世纪60年代后期的网络模型数据库中表示。1959年定义[COBOL的](https://en.wikipedia.org/wiki/COBOL)[CODASYL](https://en.wikipedia.org/wiki/CODASYL)在1969年定义了网络数据库语言。  从20世纪80年代中期开始，标记图可以用图形数据库表示，例如逻辑数据模型。[[2]](https://en.wikipedia.org/wiki/Graph_database#cite_note-Gutierrez-2)[[8]](https://en.wikipedia.org/wiki/Graph_database#cite_note-8)  图形数据库的一些改进在20世纪90年代初出现，并在20世纪90年代后期加速推行网页索引。  在2000年代中后期，商业*原子性，一致性，隔离性，耐久性*（[ACID](https://en.wikipedia.org/wiki/ACID)）图形数据库（例如[Neo4j](https://en.wikipedia.org/wiki/Neo4j)和[Oracle Spatial和Graph）](https://en.wikipedia.org/wiki/Oracle_Spatial_and_Graph)已经上市。  在2010年，可以[水平缩放的](https://en.wikipedia.org/wiki/Scalability#Horizontal_and_vertical_scaling)商业ACID图形数据库变得可用。此外，[SAP HANA](https://en.wikipedia.org/wiki/SAP_HANA)带来的[内存](https://en.wikipedia.org/wiki/In-memory_database)和[柱状](https://en.wikipedia.org/wiki/Column-oriented_DBMS)技术用于图形数据库。[[9]](https://en.wikipedia.org/wiki/Graph_database#cite_note-9)同样在2010年，支持图模型（以及其他模型，如关系数据库或[面向文档的数据库](https://en.wikipedia.org/wiki/Document-oriented_database)）的[多模型数据库](https://en.wikipedia.org/wiki/Multi-model_database)变得可用，例如[OrientDB](https://en.wikipedia.org/wiki/OrientDB)，[ArangoDB](https://en.wikipedia.org/wiki/ArangoDB)和[MarkLogic](https://en.wikipedia.org/wiki/MarkLogic)（从7.0版开始）。在此期间，各种类型的图形数据库特别受到[社交网络分析的欢迎](https://en.wikipedia.org/wiki/Social_network_analysis) 随着社交媒体公司的来临。 |

|  |
| --- |
| **和其它数据库合作**  　　在上面我们刚刚提到了不应该由Neo4J记录不适合它记录的数据，以保证Neo4J不被不合理的使用方式拉低其执行效率。那么这些数据应该记录在哪里呢？答案非常简单：适合记录这些数据的其它类型的数据库。  　　而我想在这里介绍的是，如何完成Neo4J和其它数据库之间的集成，从而使它们协同工作，向用户提供完整的服务。对于某些系统，我们可以允许这些数据库之间拥有一定程度的不一致；而对于另外一些系统，我们则需要时刻保证数据的一致性。  　　Neo4J所提出支持的技术方案主要有三种：Event-based Synchronization，Periodic Synchronization以及Periodic Full Export/Import of Data。Event-based Synchronization实际上就是通过同时向基于Neo4J的后台和基于其它数据库的后台发送相同的消息，并由这些后台完成对数据的写入。Periodic Synchronization则是定时地将一个数据库中对数据的更改同步到另一个数据库中。而Periodic Full Export/Import of Data则是通过将一个数据库中的所有数据导入到另外一个数据库中的方式来完成的。  　　这三种解决方案都是用来处理Neo4J所记录的数据与其它数据库相同的情况。而更为常见的情况则是，Neo4J记录实体关系比较复杂的图，其它数据库则用来记录具有其它类型表现形式的数据。Neo4J和这些数据库之间的数据只有一部分交集，而每个数据库都拥有自己所特有的数据。针对这种情况的处理方法则常常是多步提交。例如在一个交友网站中，用户可以在页面上完成自身账户的设置，如用户名，密码等，并可以在下一步添加好友界面中添加一系列好友以及有关于该好友的注释。那么在该系统中，用户自身的账户设置就可能记录在关系型数据库中，而有关好友的相关信息则记录在图形数据库中。如果将这两步中的所有信息作为一个请求发送到后台，那么就可能出现在某个数据库上成功保存而在另一个数据库上保存失败的情况。为了避免这种情况，我们就需要将填充这两部分资料的信息分为两个页面，而在每个页面下部提供一个”保存并进行下一步”的按钮。这样如果第一步设置账户的步骤无法正常保存，那么用户就没有办法进行下一步添加朋友的操作。而在添加朋友这步中，如果图形数据库无法正常保存，那么我们将可以明确地告诉用户添加朋友失败，从而允许用户重试。  　　其实很多时候，跨不同数据库保存数据的问题都可以通过调整设计的方式来解决，况且这些数据库所记录的数据常常具有非常不同的数据结构。因此就用户来说，分成多步提交常常是一个非常自然的使用方式。 |

|  |
| --- |
| Overview  Very simply, a graph database is a database designed to treat the relationships between data as a first-class citizen in the data model. Why Graph Databases? We live in a connected world! There are no isolated pieces of information, but rich, connected domains all around us. Only a database that natively embraces relationships is able to store, process, and query connections efficiently. While other databases compute relationships at query time through expensive JOIN operations, a graph database stores connections as first class citizens.  Accessing nodes and relationships in a native graph database is an efficient, constant-time operation and allows you to quickly traverse millions of connections per second per core.  Independent of the total size of your dataset, graph databases excel at managing highly connected data and complex queries. Armed only with a pattern and a set of starting points, graph databases explore the larger neighborhood around the initial starting points — collecting and aggregating information from millions of nodes and relationships — leaving the billions outside the search perimeter untouched. 为什么图形数据库？ 我们生活在一个互联的世界！我们周围没有孤立的信息，但有丰富的连接域。只有本地拥抱关系的数据库才能够有效地存储，处理和查询连接。当其他数据库通过昂贵的JOIN操作在查询时计算关系时，图形数据库将连接存储为一等公民。  在本地图形数据库中访问节点和关系是一种高效的恒定时间操作，并允许您快速遍历每个内核每秒数百万个连接。  独立于数据集的总大小，图形数据库擅长管理高度连接的数据和复杂的查询。图形数据库只在一个模式和一组出发点上武装起来，围绕最初的起点探索更大的邻域 - 收集和汇总来自数百万节点和关系的信息 - 使得数十亿在搜索周界外不变。 属性图模型 如果您曾经使用过对象模型或实体关系图，那么标记的属性图模型看起来很熟悉。  **节点**是图中的实体。它们可以包含任何数量的属性（键值对），称为属性。节点可以标记代表您的域中不同角色的标签。除了上下文化节点和关系属性之外，标签还可以用于将元数据索引或约束信息附加到某些节点。  **关系**提供两个节点实体（例如员工 WORKS\_FOR公司）之间的定向的，命名的，语义上相关的连接。关系总是有方向，类型，起始节点和结束节点。像节点一样，关系也可以具有属性。在大多数情况下，关系具有量化属性，如权重，成本，距离，评级，时间间隔或强度。随着关系的有效存储，两个节点可以共享任意数量或类型的关系而不牺牲性能。请注意，虽然它们是定向的，但关系总是可以有效地在任一方向导航。  属性图的构建块  图数据库中有一个核心一致性规则：**“没有断开的链接”**。由于关系始终具有开始和结束节点，因此无法删除节点而又不删除其关联关系。您也可以始终假定现有关系永远不会指向不存在的终端。  ——neo4j官网介绍  ——https://neo4j.com/developer/graph-database/#\_why\_graph\_databases |