**电 子 科 技 大 学**

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

**学士学位论文**

**BACHELOR THESIS**



论文题目 **C++ 环境的轻量级图数据库存储引擎设计与实现**

专 业 **软件工程（软件技术）**

学 号 **2014220201020**

作者姓名 **李陈扬**

指导教师 **段翰聪**

摘 要

随着Web技术的不断演进与发展，人类先后经历了以文档互联为主要特征的“Web 1.0”与数据互联为特征的“Web 2.0”时代，正在迈向基于知识互联的“Web 3.0”时代[1]。信息技术发展迅猛，数据呈爆炸性增长。传统的基于磁盘的关系型数据库比如 Oracle、MySQL 已经不能够满足日益增长的数据的大规模存储和关系查询需求。传统关系型数据库描述实体之间关系时，需要创建一个关联表以记录这些数据之间的关联关系，数据库需要通过关联表间接地维护实体间的关系，随着实体之间复杂关系数量和关系层次的增加，关联表的数量急剧上升，数据库的执行效能低下性能大大降低。而在图数据库的应用场景下，数据以自然语言方式存储，具有无索引邻接关系的原生图形存储结构可加快数据关系的事务处理速度。

本文基于图数据库的存储场景，研究了基于Java的Neo4j图数据库存储系统，调研其存储引擎的底层数据组织方式和查询模式，提出了一种轻量级的图数据库引擎设计方案。该方案实现了存储引擎的一系列模块：Store，Transaction，TxState，Command，Operations，Lock，FileChannel，ByteBuffer，Record和StoreEngine等。基于这些模块设计，实现了基于 linux 平台的轻量级图数据库存储引擎，最后实施性能验证，分析方案的效率。

**关键词：**图数据库，Neo4j，图形存储结构，无索引邻接

ABSTRACT

With the continuous evolution and development of Web technology, human beings have experienced the "Web 2.0" era characterized by "Web 1.0" and data interconnection, which are characterized by document interconnection, and are now moving towards the "Web 3.0" era based on knowledge interconnection. [1]. The rapid development of information technology and the explosive growth of data. Traditional disk-based relational databases such as Oracle and MySQL are no longer able to meet the growing demand for large-scale storage and relational queries of data. When a traditional relational database describes the relationship between entities, an association table needs to be created to record the association relationship between these data. The database needs to maintain the relationship between the entities indirectly through the association table. With the number of complex relationships between the entities and the relationship level With the increase, the number of related tables has risen sharply, and the performance of the database has been greatly reduced. In the application scenario of the graph database, the data is stored in natural language. The native graphics storage structure with no index adjacency relationship can speed up the transaction processing speed of the data relation.

Based on the storage scenario of graph database, this paper studies the Neo4j graph database storage system based on Java, investigates the underlying data organization and query mode of its storage engine, and proposes a lightweight graph database engine design scheme. The program implements a series of modules of the storage engine: Store, Transaction, TxState, Command, Operations, Lock, FileChannel, ByteBuffer, Record, and StoreEngine. Based on the design of these modules, a lightweight map database storage engine based on the linux platform was implemented. Finally, performance verification was performed and the efficiency of the solution was analyzed.

**Keywords:** graph database, Neo4j, graphical storage structure, no index adjacency

目 录

[摘 要 I](#_Toc484422302)

[ABSTRACT II](#_Toc484422303)

[目 录 III](#_Toc484422304)

[第一章 绪 论 1](#_Toc484422305)

[1.1 研究工作的背景与意义 1](#_Toc484422306)

[1.2 国内外研究历史与现状 1](#_Toc484422307)

[1.3 本文的主要贡献与创新 2](#_Toc484422308)

[1.4 本论文的结构安排 3](#_Toc484422309)

[第二章 相关技术理论基础 4](#_Toc484422310)

[2.1 数据库系统技术 4](#_Toc484422311)

[2.2 内存数据库系统技术 5](#_Toc484422312)

[2.2.1 内存数据库的定义 5](#_Toc484422313)

[2.2.2 内存数据库技术成果 6](#_Toc484422314)

[2.3 内存数据库 vs. 磁盘数据库 7](#_Toc484422315)

[2.3.1 内存 vs. 磁盘 7](#_Toc484422316)

[2.3.1 MMDB 与 DRDB 对比 8](#_Toc484422317)

[2.4 GPU通用计算技术 9](#_Toc484422318)

[2.4.1 GPU并行机制 10](#_Toc484422319)

[2.4.2 CUDA体系结构 13](#_Toc484422320)

[2.4.3 CUDA编程模型 15](#_Toc484422321)

[2.4.2 CUDA执行模式 16](#_Toc484422322)

[2.4.2 CUDA程序优化 17](#_Toc484422323)

[2.5 本章小结 19](#_Toc484422324)

[第三章 GPU计算引擎设计与实现 20](#_Toc484422325)

[3.1 需求分析及设计目标 20](#_Toc484422326)

[3.2 整体架构 20](#_Toc484422327)

[3.3 原语设计与实现 21](#_Toc484422328)

[3.3.1 Map 21](#_Toc484422329)

[3.3.2 Scatter和Gather 21](#_Toc484422330)

[3.3.3 Reduce 23](#_Toc484422331)

[3.3.4 Scan 24](#_Toc484422332)

[3.3.5 Filter 26](#_Toc484422333)

[3.4 原语组合成SQL算子 27](#_Toc484422334)

[3.5 关键技术和改进点 27](#_Toc484422335)

[3.5.1 大数据量处理 28](#_Toc484422336)

[3.5.2 全局内存访问延迟解决 28](#_Toc484422337)

[3.6 本章小结 28](#_Toc484422338)

[第四章 性能测试 29](#_Toc484422339)

[4.1 测试方案 29](#_Toc484422340)

[4.1.1 测试环境 29](#_Toc484422341)

[4.1.2 测试数据生成 30](#_Toc484422342)

[4.1.3 CPU上的实现方式 30](#_Toc484422343)

[4.1.4 正确性检验 30](#_Toc484422344)

[4.2 测试结果与分析 30](#_Toc484422345)

[4.3 本章小结 32](#_Toc484422346)

[第五章 全文总结与展望 33](#_Toc484422347)

[5.1 全文总结 33](#_Toc484422348)

[5.2 后续工作展望 33](#_Toc484422349)

[致 谢 34](#_Toc484422350)

[参考文献 35](#_Toc484422351)

[外文资料原文 36](#_Toc484422352)

[外文资料译文 37](#_Toc484422353)

第一章 绪 论

1.1 研究工作的背景与意义

随着用户、系统和传感器产生的数据量呈指数增长，其增长速度因大部分数据量集中在Amazon、Google和其他云服务的分布式系统上而进一步加快，数据内部依赖和复杂度的增加，这一问题因互联网、Web2.0、社交网络，以及对大量不同系统的数据源开放和标准化的访问而加剧，关系数据库产生了很多的不适应性。从而导致大量解决这些问题中某些特定方面的不同技术出现，它们可以与现有RDBMS相互配合或代替它们——亦被称为混合持久化（Polyglot Persistence）。数据库替代品并不是新鲜事物，它们已经以对象数据库（OODBMS）、层次数据库（如LDAP）等形式存在很长时间了。但是，过去几年间，出现了大量新项目，它们被统称为NoSQL数据库（NoSQL-databases）[2]。其中，图数据库（Graph Database）从最近十年的表现来看已经成为关注度最高，也是发展趋势最明显的数据库类型。

图形数据库每个对象是一个节点，之间的关系是一条边。相对于关系数据库来说，图形数据库善于处理大量复杂、互连接、低结构化的数据，这些数据变化迅速，需要频繁的查询——在关系数据库中，由于这些查询会导致大量的表连接，从而导致性能问题，而且在设计使用上也不方便。图形数据库适合用于社交网络，推荐系统等专注于构建关系图谱的系统。

本文基于以上背景，提出一种基于 C++ 的轻量级图形数据库存储引擎的设计方案。基于 Linux 平台对数据的图形结构存储方式进行设计实现。并对方案的效率进行了验证。

1.2 国内外研究历史与现状

在[众多不同的数据模型](http://en.wikipedia.org/wiki/Data_model)里，关系数据模型自80年代就处于统治地位，而且有不少实现，如[Oracle](http://www.oracle.com/index.html)、[MySQL](http://www.mysql.com/)和[MSSQL](http://www.microsoft.com/sqlserver/2008/en/us/)，它们也被称为关系数据库管理系统（RDBMS）。然而，最近随着关系数据库使用案例的不断增加，一些问题也暴露了出来，这主要是因为两个原因：[数据建模中的一些缺陷和问题](http://www.scribd.com/doc/2670985/SQL-Antipatterns)，以及在大数据量和多服务器之上进行水平伸缩的限制。

在应对这些趋势时，关系数据库产生了更多的问题。这导致大量解决这些问题某些特定方面的不同技术的出现，它们可以与现有RDBMS相互配合或代替它们，亦被称为[混合持久化（Polyglot Persistence）](http://codemonkeyism.com/nosql-polyglott-persistence/)。数据库替代品并不是新鲜事物，它们已经以[对象数据库（OODBMS）](http://en.wikipedia.org/wiki/List_of_object_database_management_systems)、层次数据库（如LDAP）等形式存在很长时间了。但是，过去几年间，出现了大量新项目，它们被统称为NOSQL数据库（NOSQL-databases）。图形便是关系规范化的一种替代技术。领域模型在数据结构上的方案，有两个主流学派：RDBMS采用的关系方法和图，即网络结构，如语义网用到的。

尽管图结构在理论上甚至可以用RDBMS规范化，但由于关系数据库的实现特点，对于像文件树这样的递归结构和像社交图这样的网络结构有严重的查询性能影响。网络关系上的每次操作都会导致RDBMS上的一次“联结”操作，以两个表的主键集合间的集合操作来实现 ，这种操作不仅缓慢并且无法随着这些表中元组数量的增加而伸缩。

在图的领域，并没有一套被广泛接受的术语，存在着很多不同类型的图模型。但是，后来有人致力于创建一种属性图形模型（Property Graph Model），以期统一大多数不同的图实现。按照该模型，属性图里信息的建模使用3种构造单元：节点（即顶点）、关系（即边）、具有方向和类型（标记和标向）、节点和关系上面的属性（即特性）。更特殊的是，这个模型是一个被标记和标向的属性多重图（multigraph）。被标记的图每条边都有一个标签，它被用来作为那条边的类型。有向图允许边有一个固定的方向，从末或源节点到首或目标节点。属性图允许每个节点和边有一组可变的属性列表，其中的属性是关联某个名字的值，简化了图形结构。多重图允许两个节点之间存在多条边。这意味着两个节点可以由不同边连接多次，即使两条边有相同的尾、头和标记。

图论的巨大用途被得到了认可，它跟不同领域的很多问题都有关联。最常用的图论算法包括各种类型的最短路径计算、测地线（Geodesic Path）、集中度测量（如PageRank、特征向量集中度、亲密度、关系度、HITS等）。然而，在很多情况下，这些算法的应用仅限制于研究，因为实际中没有任何可用于产品环境下的高性能图形数据库实现。幸运的是，近些年情况有所改观。有几个项目已经被开发出来，而且目标直指24/7的产品环境：Neo4j - 开源的Java属性图形模型、AllegroGraph - 闭源，RDF-QuadStore、Sones - 闭源，关注于.NET、Virtuoso - 闭源，关注于RDF、HyergraphDB - 开源的Java超图模型、Others like InfoGrid、Filament、FlockDB等。

目前有多种图形数据库系统，其中Neo4j是一个用Java实现、完全兼容ACID的图形数据库。数据以一种针对图形网络进行过优化的格式保存在磁盘上。Neo4j的内核是一种极快的图形引擎，具有数据库产品期望的所有特性，如恢复、两阶段提交、符合XA等。自2003年起，Neo4j就已经被作为24/7的产品使用。Neo4j既可作为无需任何管理开销的内嵌数据库使用；也可以作为单独的服务器使用，在这种使用场景下，它提供了广泛使用的REST接口，能够方便地集成到基于PHP、.NET和JavaScript的环境里。开发者可以通过Java-API直接与图形模型交互，这个API暴露了非常灵活的数据结构。至于像JRuby/Ruby、Scala、Python、Clojure等其他语言，社区也贡献了优秀的绑定库。Neo4j的典型数据特征：数据结构不是必须的，甚至可以完全没有，这可以简化模式变更和延迟数据迁移；可以方便建模常见的复杂领域数据集，如CMS里的访问控制可被建模成细粒度的访问控制表，类对象数据库的用例、TripleStores以及其他例子；典型使用的领域如语义网和RDF、LinkedData、GIS、基因分析、社交网络数据建模、深度推荐算法以及其他领域。

甚至“传统”RDBMS应用往往也会包含一些具有挑战性、非常适合用图来处理的数据集，如文件夹结构、产品配置、产品组装和分类、媒体元数据、金融领域的语义交易和欺诈检测等。

围绕内核，Neo4j提供了一组可选的组件。其中有支持通过元模型构造图形结构、SAIL，它是一种SparQL兼容的RDF TripleStore实现或一组公共图形算法的实现。要是你想将Neo4j作为单独的服务器运行，还可以找到REST包装器。这非常适合使用LAMP软件搭建的架构。通过memcached、e-tag和基于Apache的缓存和Web层，REST甚至简化了大规模读负荷的伸缩。

自适应规模的Neo4j无需任何额外的工作便可以处理包含数十亿节点、关系和属性的图。它的读性能可以很轻松地实现每毫秒（大约每秒1~2百万个遍历步骤）遍历2000关系，这完全是事务性的，每个线程都有热缓存。使用最短路径计算，Neo4j在处理包含数千个节点的小型图时，甚至比MySQL快1000倍，随着图规模的增加，差距也越来越大。

这其中的原因在于，在Neo4j里，图遍历执行的速度是常数，跟图的规模大小无关。不像在RDBMS里常见的联结操作那样，这里不涉及降低性能的集合操作。Neo4j以一种延迟风格遍历图：节点和关系只有在结果迭代器需要访问它们的时候才会被遍历并返回，对于大规模深度遍历而言，这极大地提高了性能。

如今，图数据库系统已经广泛应用于：社交网络、商品推荐、金融管理、欺诈识别等领域，这大大提高了实体间复杂关系的查找和分析效率。

1.3 本文的主要贡献

本论文主要研究了近几年几个流行的图数据库系统和图数据库技术进展，着重调研了基于Java环境的图数据库产品Neo4j，研究了其数据组织的方法和查询存储机制，并基于Neo4j的整体架构和设计模式，提出了一种基于C++的轻量级图数据库存储引擎设计方案。方案实现了一系列运行于Linux环境上的存储引擎模块：NodeProxy，RelationshipProxy，IdGenerator，Operations，Store，Transaction，TxState，FileLock和I/O。基于这些模块设计并实现了节点以及关系在内存中的创建表示以及其在物理磁盘中的序列化保存和反序列化读取。基于Linux平台对图形数据库存储引擎方案进行实现，实施性能验证，并分析方案的效率。

1.4 本论文的结构安排

本文的章节结构安排如下：

第一章描述研究工作的背景与意义、国内外研究历史与现状和本文的主要贡献与创新点。

第二章对现有的相关技术理论进行描述分析和总结。

第三章对本文所述方案的架构与实现进行描述分析和总结。

第四章描述对本文所述方案的测试和分析。

第五章是本方案的总结与展望。

第二章 相关技术理论基础

本章对数据库系统中已有的并且和本文相关的技术进行了介绍，最后对现有的技术的优缺点进行分析总结。

2.1 数据库系统

数据库系统（Database Management System, DBMS）是存储在计算机内，有组织，可共享的数据集合。数据库管理系统是数据管理的最新技术，是计算机科学发展最快的领域之一，已经成为计算机信息系统的核心和研究的热点。

第一个商用数据库管理系统产生于20世纪60年代末。这些系统是由文件系统演变而来的，提供了对大量数据存储和高效查询的部分支持。但是，文件系统并不能保证数据不丢失，如果没有备份的话；而且它们也不支持对数据项的高效存取，如果不知道它在特定的文件里的存储位置的话。文件系统不直接支持针对文件中数据的查询语言。它们对数据模式的支持仅限于创建文件的目录结构。持久性需求并不总被文件系统支持，你可能会丢失并没有备份的数据。最后，文件系统也不满足孤立性和原子性。虽然它们允许几个用户或进程并发地访问文件，但文件系统一般并不阻止两个用户同时修改同一个文件，从而导致一个用户的修改不能出现在文件中。早期的DBMS要求程序员直接面对数据的存储格式，并且它们不支持高级查询语言。例如，CODASYL查询语言具有允许用户通过数据元素间的指针从一个数据元素跳到另一个数据元素的语句。即使对于非常简单的查询，也需要相当大的工作量来写这样的程序。

数据模型是数据库系统的核心，根据数据模型数据库的发展可以分为三个阶段：

第一个阶段是以格式化系统统称的层次数据库和网状数据库。层次数据库用树形结构来表示各类实体和实体间的联系，对应于有根定向有序树，其代表系统是IMS(Information Management System)数据库系统；网状模型用有向图表示实体和实体间的联系，对应于有向图，其代表系统DBTG系统。这类系统用存取路径表示数据间的关系，数据库查询和数据操纵是导航式的过程化语言。

第二个阶段是使用关系模型的关系数据库系统。1970 年 IBM 公司 San Jose研究室的研究员 E.F.Codd [12]发表了“A Relational Model of Data for Large Shared Data Banks” 的论文，该论文提出了关系模型，其后又陆续发表了多篇关系数据库研究的论文，为关系数据库奠定了理论基础。以此为基础的典型系统有 System R 和 INGRES。80 年代攻克了关系数据库的一系列难题，涌现出很多优秀的关系数据库系统，如 DB2、Oracle、Sybase、Informix 等。

第三阶段是新一代数据库。关系数据库结构简单，操纵方便，但它只适合管理结构稳定、格式化和结构化的商业事务领域。对现代应用，如实时处理系统、计算机集成制造(CIM)、电话交换机、实时仿真、电信网管、电力调度，GIS，CAD/CAM 等具有很大的局限性。为了满足现代工程应用的需求，把数据库技术和其它的现代信息处理技术，如面向对象、人工智能、多媒体、实时处理、复杂建模等技术完美结合，产生新一代现代数据库已经成为必然。根据不同的应用环境和技术，现代数据库有：内存数据库(MMDB，Main Memory Database)、面向对象数据库(OODB，Object Oriented Database)、实时数据库(RTDB，Real Time Database)、主动数据库(ADB，Active Database)、多媒体数据库(MDB，Multimedia Database)、时态数据库(TDB，Temporal Database)、移动数据库(MDB)、空间数据库、知识库(KBMS，Knowledge Base Management System)等。其中面向对象的方法和技术对数据库的影响最为深远，数据库研究人员借鉴面向对象思想和技术，提出了面向对象的数据库(OODB，Object-Oriented Database)，以及在面向对象技术和在关系模型的基础之上提出了对象关系模型(ORDB，Object-Relational Database)。

2.2 图数据库系统

图形数据库（GDB[]）是一个数据库，它使用图形结构进行语义查询，并使用节点，边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。这些关系允许存储器（Store）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。

这与关系型数据库形成了鲜明对比，关系型数据库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操作（例如，连接）来以相关逻辑格式操作和返回相关数据。借助于物理层面的数据库管理系统（例如，使用索引），关系查询的执行是可能的，这允许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而快速地检索（关联系统中难以建模的复制层次结构）。图形数据库与20世纪70年代的网络模型数据库类似，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[]，并且在边缘链上缺乏简单的遍历。

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（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或面向文档的数据库进行存储，这使得它们本身就是NoSQL结构。大多数基于非关系型存储引擎的图表数据库都会添加标签或属性的概念，这些概念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以外的查询语言，该语言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，SPARQL和Cypher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（API）访问某些图形数据库。

图形数据库将数据存储在图形中，这是最通用的数据结构，能够以高度可访问的方式表示任何类型的数据。 Neo4j图基于属性图模型。下面是一个示例图：



2.2.1 节点

Neo4j中的节点（Node）是属性图模型中描述的节点，具有属性和标签。节点通常用于表示实体，但取决于域的关系也可用于此目的。最简单的图形是单个节点。例如下面的图，它由一个具有单个属性“title”的节点组成：



在前面的例子中，可以再添加两个节点，对每个节点再添加一个属性：



2.2.2 关系

Neo4j中的关系（Relationship）与属性图模型中描述的关系具有关系类型和属性两个特征。节点之间的关系是图数据库的关键特征，因为它们允许查找相关数据。关系连接两个节点，并保证有一个有效的源节点和目标节点。

关系将节点组织为任意结构，允许图形类似于列表、树、地图或复合实体。任何一个都可以组合成更复杂，更丰富的内部连接结构。一旦我们添加关系，我们的示例图将会变得更有意义：



我们的例子使用ACTED\_IN和DIRECTED作为关系类型。 ACTED\_IN关系上的roles属性具有一个数组类型的值，其中包含一个数组元素。

下面是一个ACTED\_IN关系，Tom Hanks节点作为源节点，而Forrest Gump作为目标节点：



我们观察到Tom Hanks节点有出度关系，而Forrest Gump节点有入度关系。

关系在任何一个方向都同样得到很好的遍历。

这意味着不需要在相反的方向上添加重复关系。虽然关系总是有方向的，但我们可以忽略在应用程序中某些无用的方向。值得注意的是，一个节点也可以与它自己有关系：



上面的例子意味着Tom Hanks KNOWS他自己。我们可以通过简单地遵循示例图中节点的关系找到以下数据：

|  |  |  |  |
| --- | --- | --- | --- |
| **What we want to know** | **Start from** | **Relationship type** | **Direction** |
| get actors in movie | :Movie node | :ACTED\_IN | Incoming |
| get movies with actor | :Person node | :ACTED\_IN | Outgoing |
| get directors of movie | :Movie node | :DIRECTED | Incoming |
| get movies directed by | :Person node | :DIRECTED | Outgoing |

2.2.3 属性

Neo4j中的属性（Property）是属性图模型中描述的属性。节点和关系都可以具有属性。属性是命名值，其中名称（name）（或键（key））是一个字符串。受支持的属性值为：

1. 数字值
2. 字符串值
3. 布尔值
4. 任何上述值的数组列表

null不是有效的属性值。 我们不将它存储在数据库中，取而代之的是，可以通过缺少属性键来表示null。

属性值种类如下：

|  |  |  |
| --- | --- | --- |
| **Type** | **Description** | **Value range** |
| Boolean | binary logic value | true/false |
| Integer | 64-bit integer | -9223372036854775808 to 9223372036854775807, inclusive |
| Float | 64-bit IEEE 754 floating-point number | - |
| String | sequence of Unicode characters | infinite |

2.2.4 标签

Neo4j中的标签（Label）是属性图模型中用以描述节点类型的标签。标签将角色或类型分配给节点。标签是一种命名的图形结构，用于将节点分组成集合，所有标有相同标签的节点都属于同一组。许多数据库查询可以使用这些集合而不是查询整个图，从而使查询更易于编写，并且更高效地执行。节点可以用任意数量的标签进行标记，也可以不标记标签，这使标签成为图的可选添加项。通常，定义约束并为属性添加索引时使用标签。

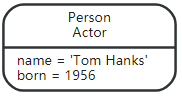
例如，代表用户的所有节点都可以用标签标记：用户（:User）。有了这些，您可以要求Neo4j仅在用户（User）节点上执行操作，例如查找具有给定名称的所有用户。

当然，我们也可以使用更多的标签。例如，由于标签可以在运行时添加和删除，因此可以用它们标记节点的临时状态。一个暂停标签（:Suspended）可以用来表示暂停的银行账户；一个季节标签（: Seasonal），可以用于表示当前正在销售的蔬菜等等。

在我们的例子中，我们将添加 :Person和 :Movie标签到我们的图中：



为了举例说明节点可能有多个标签，我们添加一个：Actor标签给Tom Hanks节点：



标签具有各自的标签名称。任何非空的Unicode字符串都可以用作标签名称。 在Cypher中，您可能需要使用反引号“`”语法来避免与Cypher语言标识符规则冲突或允许标签中的非字母数字字符。按照惯例，标签用驼峰书写符号书写，第一个字母为大写。例如User或CarOwner。

标签具有一个int值大小的空间，这意味着数据库可以包含的最大标签数量大约为20亿个。

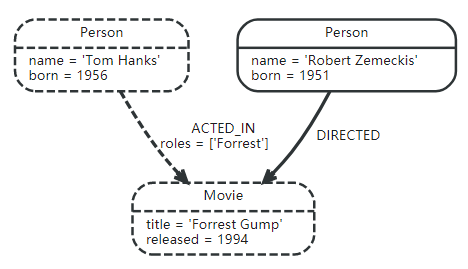
2.2.5 遍历

遍历（Traversal）通过遍历图来查找路径。遍历是从起始节点到相关节点的导航，找到像“我的朋友喜欢的我还没拥有的音乐”这样的问题的答案，或者为了解决“如果这种电力供应下降，什么Web服务会被影响到？”来查询图的过程。

遍历图意味着访问其节点，根据一些规则遵循关系。在大多数情况下，只有一个子图被访问，因为我们已经知道图中的哪些地方会找到需要的节点和关系。

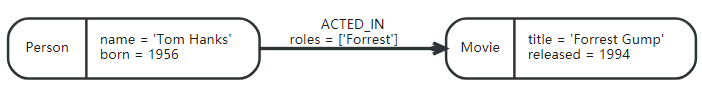
Cypher提供了一种声明方式来查询由遍历和其他技术支持的图。

如果我们想根据我们小型的示例数据库找出Tom Hanks所出镜的电影，则遍历将从Tom Hanks节点开始，然后执行以下任意操作：ACTED\_IN与节点连接，最后以电影名Forrest Gump作为结果（参见图中虚线部分）：

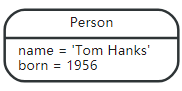


2.2.6路径

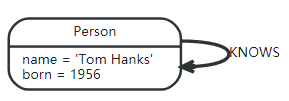
Neo4j中的路径（Path）是属性图模型中描述的路径。 从Cypher查询或遍历中检索路径。在前面的例子中，遍历结果可以作为路径返回：



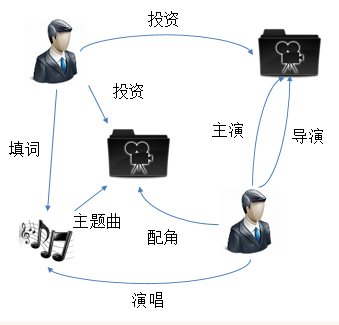
上面的路径长度为1。在Neo4j中，最短的路径长度为0，也就是说，它只包含一个节点并且没有关系，并且可以如下所示：



长度为1的路径：



2.3 关系型数据库 vs. 图数据库



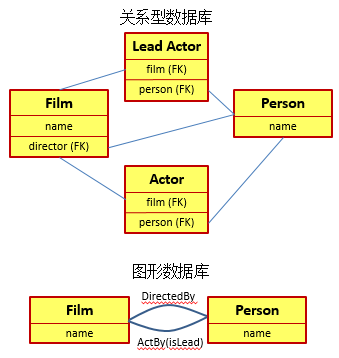
以一部电影中的各个实体关系为例，演员常常有主角配角之分，还要有导演，特效等人员的参与。通常情况下这些人员常常都被抽象为Person类型，对应着同一个数据库表。同时一位导演本身也可以是其它电影或者电视剧的演员，更可能是歌手，甚至是某些影视公司的投资者。而这些影视公司则常常是一系列电影，电视剧的资方。这种彼此关联的关系常常会非常复杂，而且在两个实体之间常常同时存在着多个不同的关系：

在尝试使用关系数据库建模时，我们首先需要建立表示各种实体的一系列表：表示人的表，表示电影的表，表示电视剧的表，表示影视公司的表等等（实体表）。这些表常常需要通过一系列关联表将它们关联起来：通过这些关联表来记录一个人到底参演过哪些电影，参演过哪些电视剧，唱过哪些歌，同时又是哪些公司的投资方。同时我们还需要创建一系列关联表来记录一部电影中哪些人是主角，哪些人是配角，哪个人是导演，哪些人是特效等。可以看到，我们需要大量的关联表来记录这一系列复杂的关系。在更多实体引入之后，我们将需要越来越多的关联表，从而使得基于关系型数据库的解决方案繁琐易错。

这一切的症结主要在于关系型数据库是以为实体建模这一基础理念设计的。该设计理念并没有提供对这些实体间关系的直接支持。在需要描述这些实体之间的关系时，我们常常需要创建一个关联表以记录这些数据之间的关联关系，而且这些关联表不用来记录除外键之外的其它数据。也就是说，这些关联表也仅仅是通过关系型数据库所已有的功能来模拟实体之间的关系。这种模拟导致了两个非常糟糕的结果：数据库需要通过关联表间接地维护实体间的关系，导致数据库的执行效能低下，同时关联表的数量急剧上升。

而图数据库却使用了人类自然语言认识的方式保存数据。我们经常在白板上画一些模型，用来讨论我们的项目，我们通常是画圈、画线。如果用关系型数据库来处理的话，就要把它映射成表格后再处理。如果用图数据库处理的话，就和白板上的模型没有什么差别，所以说白板模型就是物理模型。现在这个图就是我们的物理模型，在图数据库中，表现现实世界的关系就这么直观。在一个图形数据库中，数据库的最主要组成主要有两种，节点和关联这些节点的联系。每个节点代表一个实体，比如人、地点等，联系表示两个节点之间的连接关系。

2.3.1 图数据库与RDBMS的区别



关系型数据库：除了建立实体表外，还需要维护关联表。

图数据库：首先为这些事物定义其所对应的结点集，并定义该结点集所具有的各个属性。接下来辨识出它们之间的关系并创建这些关系的相应抽象。

从上图中可以看到，在需要表示多对多关系时，我们常常需要创建一个关联表来记录不同实体的多对多关系，而且这些关联表常常不用来记录信息。如果两个实体之间拥有多种关系，那么我们就需要在它们之间创建多个关联表。

而在一个图形数据库中，我们只需要标明两者之间存在着不同的关系，例如用DirectBy关系指向电影的导演，或用ActBy关系来指定参与电影拍摄的各个演员。同时在ActBy关系中，我们更可以通过关系中的属性来表示其是否是该电影的主演。而且从上面所展示的关系的名称上可以看出，关系是有向的。如果希望在两个结点集间建立双向关系，我们就需要为每个方向定义一个关系。

可以看出，相对于关系数据库中的各种关联表，图形数据库中的关系可以通过关系能够包含属性这一功能来提供更为丰富的关系展现方式。

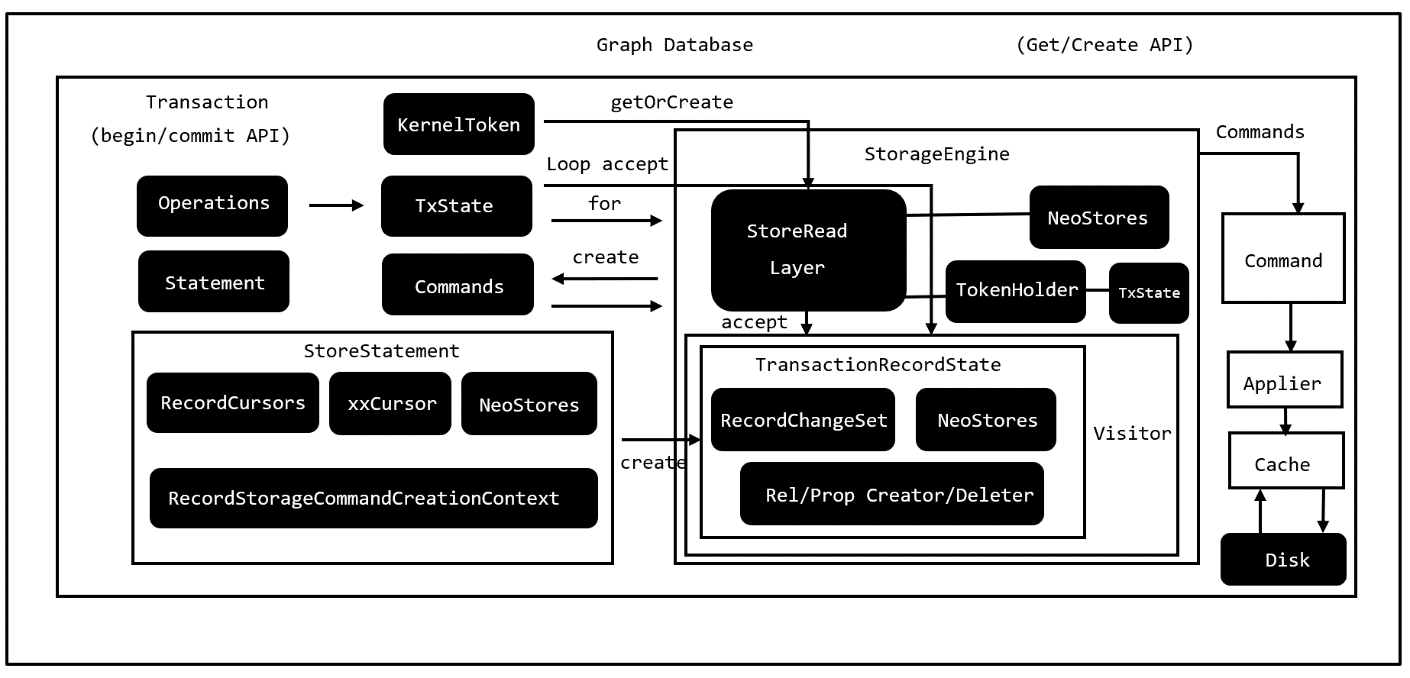
2.3.2 Neo4j vs. RDBMS

|  |  |  |
| --- | --- | --- |
|  | 关系型数据库 | 图数据库（ Neo4j） |
| 数据存储 | 有关联的数据存储在确定的、预定义的行列表中，使得其与表脱节，降低了查询效率。 | 无索引邻接关系的图存储结构，使得其有更快的事务和数据关系处理速度 |
| 数据建模 | 数据库建模必须由建模者开发，并由逻辑模型转化为物理模型（表格）。  由于数据类型和来源必须提前知道，任何变动可能需要重新设计模式、迁移等，延迟项目时间 | 灵活的、“白板友好 ”型数据模型；逻辑和物理模型间没有任何不匹配。数据的类型和来源可以在任何时候添加或更改，从而可以极大程度地缩短开发周期 |
| 查询性能 | 数据处理的性能受到了JOIN（或关系查询）的数目和深度的影响。 | 图处理可以保证低延迟和实时性能，与关系的数量和深度无关。 |
| 查询语言 | SQL：一种查询语句，其复杂度随着关联数据查询中的JOIN的数量的增加而增加。 | Cypher：原生图查询语句，提供描述关系查询的最有效的和最具有表现力的方式 |
| 事务支持 | ACID事务支持需要企业级程序应用提供一致且可靠的数据。 | 可以一直保证ACID数据的完全一致性和可靠数据。 |

2.3.3 图数据库 vs. 其他NOSQL

|  |  |  |
| --- | --- | --- |
|  | 其他NOSQL数据库 | 图数据库 |
| 数据存储 | 在数据库级别不支持连接的数据。 性能和数据可靠性随着连接的规模和复杂性而降低。 | 具有无索引邻接关系的原生图形存储结构可加快数据关系的事务处理速度。 |
| 数据建模 | 没有数据结构建模或存储数据关系。 需要应用程序来解决问题。 | 灵活的、“白板友好 ”型数据模型；逻辑和物理模型间没有任何不匹配。数据的类型和来源可以在任何时候添加或更改，从而可以极大程度地缩短开发周期 |
| 查询性能 | 没有数据关系的图形处理能力，因此必须在应用程序级别创建所有关系，需要“JOIN逻辑” | 原生图处理确保零延迟和实时性能，无论关系的数量或深度如何。 |
| 查询语言 | 查询语言各不相同，但是不存在用于表示数据关系的查询结构。. | Cypher：原生图形查询语言，提供描述关系查询的最有效和最具表现力的方式。 |
| 事务支持 | 在数据库级别对事物没有ACID支持，需要上层应用支持 | 可以一直保证ACID数据的完全一致性和可靠数据。 |

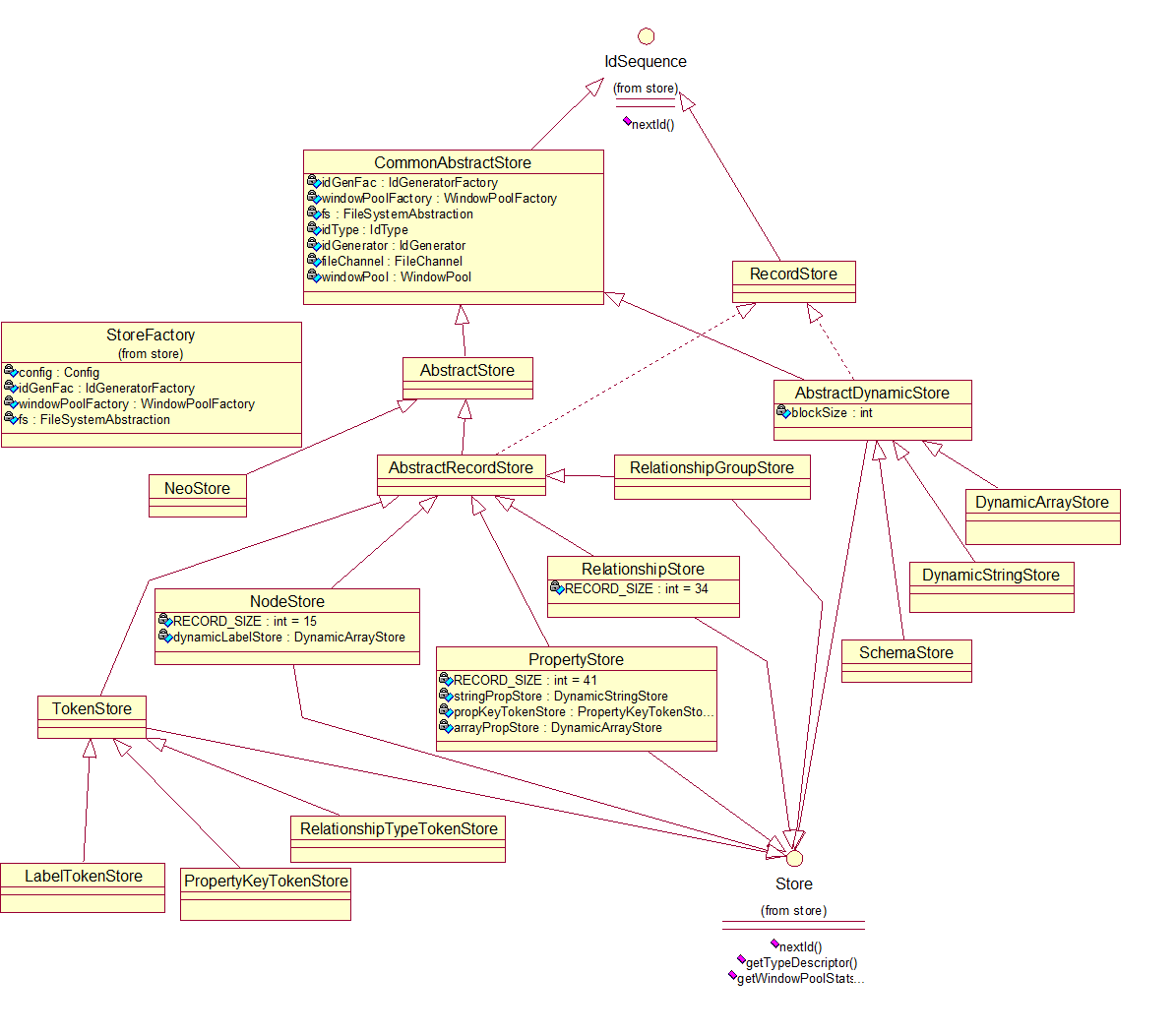
2.4 Neo4j架构设计



图数据库系统的事务提交流程架构主要包括这几个模块：Transaction、Operations、TxState、StoreStatement、StorageEngine、Commands、Applier、Cache。当用户进行一次事务的修改添加操作时，GraphDatabaseAPI接口会开始一个事务，该事务记录了当前事务的更改数据的临时状态（TxState）；将要写入页面缓存中的序列化记录项状态（RecordChangeSet）；被序列化的数据临时状态集列表（Commands）（按照增、改、删顺序排列）；页面缓存记录（PageCache）；当前事务中与数据修改关联文件的所有存储器（Store）。

当开始一个事务后，事务通过Operations里定义的一系列操作进行添加、修改、删除节点和关系。所有涉及的修改将以临时状态保存到TxState状态集合里。当一个事务结束时，系统会选择提交（Commit）或是回滚（Rollback）所有的临时状态。当提交时，所有状态集合会被转换成序列化记录项状态存储在RecordChangeSet里，最后按照增、改、删的顺序将序列化数据依次加入Commands数组，等待ApplierVisitor访问者依次写入相应数据的Store。这里的Store可以关联到相关文件，读取缓存页面数据，也可以用内部的Cursor写入数据到页面缓存区。按照一定的规则，系统通过PageSwapper将某些页面调入页面缓存，或将暂时修改的页面缓存刷新同步到相对应的文件。相反，当系统因为某些事务冲突，如其他事务的执行影响，或者加锁失败等，事务将回滚，清空当前数据更改的临时状态，不进行后续的操作。

2.5 Neo4j底层存储架构



如图所示，底层架构分为多个存储器（Store），每一个存储器对应一种存储的数据类型，并与一个文件相对应，且包含一个id存储器。存储的数据类型分别为：实体id、节点（Node）、关系（Relationship）、标签（Label）、关系类型（RelationshipType）、属性（Property）、属性键（PropertyKey）、可变长值等。每个存储器（Store）继承自一个基类的Store即CommonAbstractStore，这个基类具有生成新的实体id，创建、读取文件，读取、写入页面缓存数据等公共功能。每个继承它的Store都有各自存储的格式化写入类和写入记录项的基本信息如：记录项大小。它们各自也因自己所需或者会包含其他的Store。这里我们可以看到，NodeStore由于保存的是节点数据，节点可能会包含属性、标签，对于此类字符串长度不一定的数据需要保存在变长存储器（DynamicArrayStore）中，所以NodeStore内含变长存储器类。

2.5.1 实体id存储结构



每一种资源类型的id分配 Neo4j 中是通过 id生成器（IdGeneratorImpl）来实现的，其功能是负责id管理分配和回收复用。对于节点，关系，属性等每一种资源类型，都可以生成一个IdGenerator 实例来负责其id管理分配和回收复用。id存储文件保存了当前实体可用的最大id（highId）和之前被删除后可复用的空闲id（FreeIds）数组。

Id存储器含有以下部分模块：

* IdGeneratorImpl（id操作接口实现类）
* IdContainer（id缓存、I/O类）
* freeIds（缓存队列）、readFromDisk（读队列）

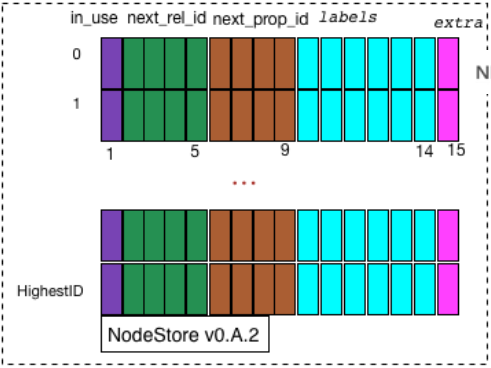
当我们创建一个节点或者一条边，事务处理部分会执行为该节点分配一个唯一的id。它会先初始化一个IdGeneratorImpl实例对象，将要写入节点id的文件fs传入并打开，传入需要读取文件中可复用id的数目n。如果文件还未创建，则创建对应的id文件，并将highId设置为0L，最高id值写入文件头；如果文件存在，初始化时将读取文件头部中的highId，并将临时变量设置为该值。

每次获取nextId的时候IdGeneratorImpl会在IdContainer中查找缓存的freeIds队列，如果有缓存可用的id值，就直接出队操作把id值返回给新节点；如果没有，就检查缓存readFromDisk队列，有就出队读取id，如果readFromDisk队列也为空，就用readFromDisk 读取文件中n个可复用id，并且入队到readFromDisk，把readFromDisk中新的id出队返回给节点。

如果 readFromDisk队列及读取id 文件都没有已释放且可复用的 id了，则把highId++的id值返回给节点。用户释放一个 id 后，先放入 freeIds队列，当freeIds队列中回收的 id 个数超过n个时，写入到 id 文件的末尾。

Close()把freeIds队列写入文件，再把readFromDisk队列写入freeIds队列，再写入文件。

2.5.2 节点存储结构



NodeStore的主文件是neostore.nodestore.db，其文件存储格式示意图如下，整个文件是有一个 RECORD\_SIZE=15Bytes 的定长数组构成。访问时，可以通过 node\_id 作为数组的下标进行访问。

**in\_use(1 Byte):**

* 第1 bit 表示 record 是否在 use
* 第2~4 bit 表示 node 的第1个 relationship\_id 的 高3位
* 第 5~8 bit表示 node 的第1个property\_id 的 高4位

**extra (1 Bytes):**

表示该node 是否 dense，node 的 relationshiop 的数量超过 50 个，这表示是 dense。

**next\_relationship\_id(4 Bytes):**

下一个出度、入度的关系索引 id。

**next\_prop\_id(4 Bytes):**

下一个属性的索引id。

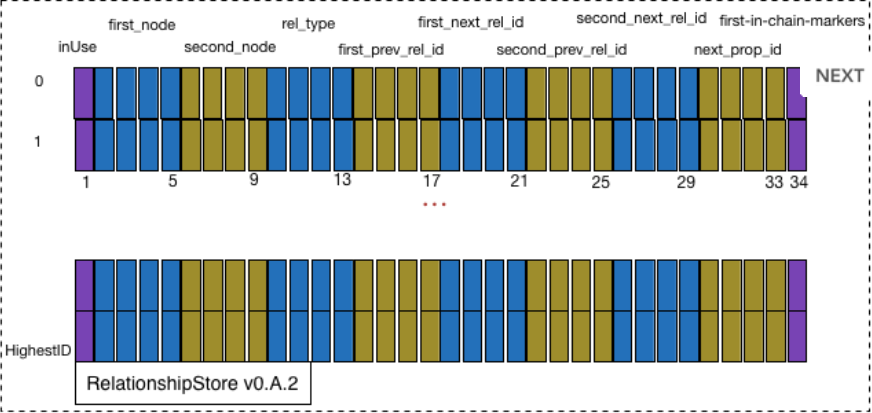
**labels (6 Bytes):**

第10~14字节是Node 的Label id，保存在 ArrayPropertyStore 这样的DynamicStore 中，是可变的存储区。

Neo4j 中, Node 的存储是由 NodeStore 和 ArrayPropertyStore 两种类型配合来完成的

Node 的 label 内容是存在 DynamicStore 中，如果长度超过一个块（block），则分block存储，并将其在ArrayPropertyStore中的第1个block 的 block\_id 保存到 NodeStore类型文件相应record 的labels字段中。

2.5.3 关系存储结构



**in\_use(1 Byte):**

* 第1 bit 表示 record是否在 use;
* 第2~4 bit 表示first\_node的node\_id的高3位；
* 第 5~8 bit表示 next\_prop\_id 的property\_id 的 高4位

**first\_node(4 Bytes) :**

第2~5字节是RelationShip的from\_node 的node\_id 的低32位. 加上inUse 字节的第 2~4 bit 作为高3位，构成一个完整的35位node\_id。

**second\_node(4 Bytes) :**

第6~9字节是RelationShip的to\_node 的node\_id 的低32位. 加上rel\_type的第29~31 bit作为高3位，构成一个完整的35位node\_id。

**rel\_type (4 Bytes):**

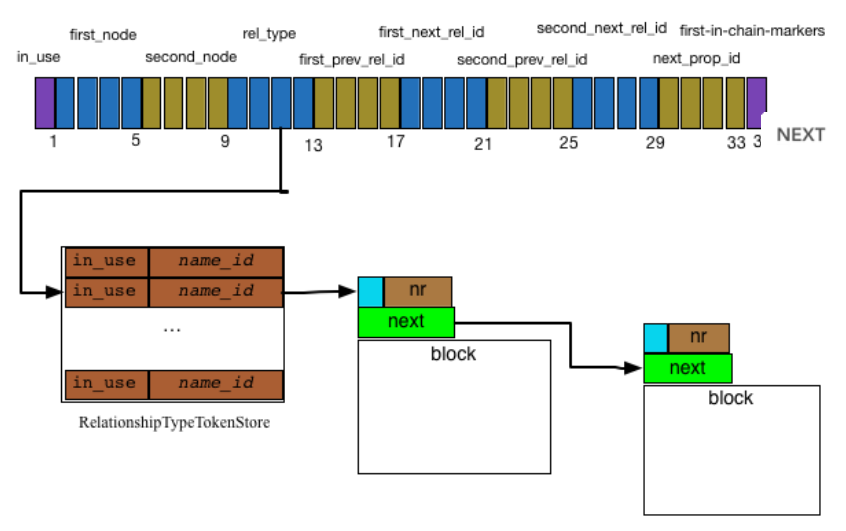
第 1~16 位 表示 RelationShipType，其他位对nodeid和relid进行高位扩展：

* + 第 10~13 字节, 分成6部分;
  + 第29~31 位是second\_node 的node\_id高3位;
  + 第26~28 位是first\_prev\_rel\_id 的 relationship\_id高3位;
  + 第23~25 位是first\_next\_rel\_id 的relationship\_id高3位;
  + 第20~22 位是second\_prev\_rel\_id 的relationship\_id高3位;
  + 第17~19 位是second\_next\_rel\_id 的relationship\_id高3位;
  + 第 1~16 位 表示 RelationShipType;

**first-in-chain-markers（1 byte）:**

表示是否是关系的第一条或者最后一条。

2.5.4 关系种类存储结构

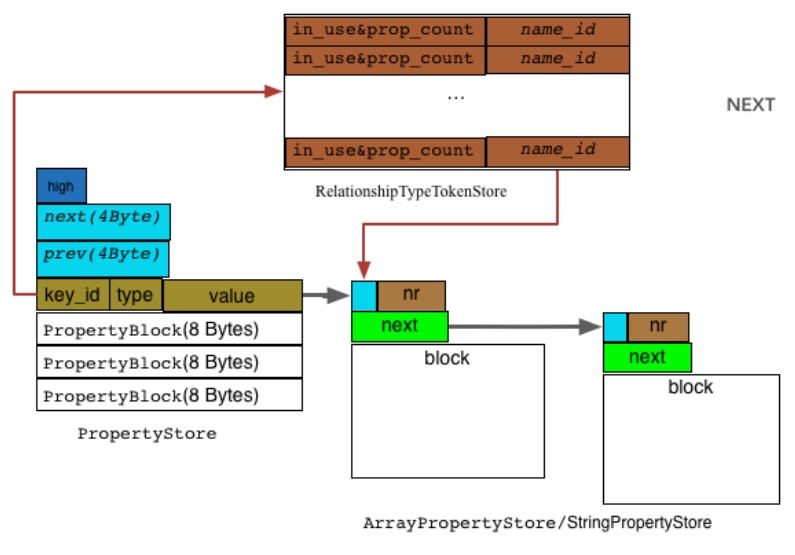


关系地址映射到token文件，读取name\_id后查找name文件返回关系名。

Neo4j 中, Relationship 的存储是由 RelationshipStore , RelationshipGroupStore, RelationshipTypeTokenStore和StringPropertyStore 4种类型的Store配合来完成的. 其中RelationshipStore是Relationship最主要的存储结构；RelationshipTypeTokenStore和StringPropertyStore 配合用来存储关系的类型。图中棕色存储块把同一种typeid映射到同一块typename地址，内部存储了in\_use和name\_id，通过name\_id查找右边的扩展长度的数组返回关系名。

关系的类型的字符串描述值是存在StringPropertyStore这样的DynamicStore 中，如果长度超过一个block ，则分block存储，并将其在StringPropertyStore中的第1个block 的 block\_id 保存到 RelationshipTypeTokenStore类型文件相应record 的name\_id字段中。

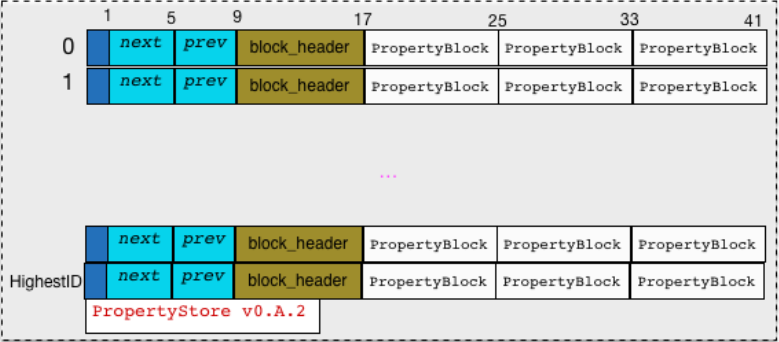
2.5.5 属性存储结构



其中PropertyStore是Property最主要的存储结构，当Property的Key-Value对的Value 是字符串或数组类型并且要求的存储空间比较大，在PropertyStore中保存不了，则会存在StringPropertyStore/ ArrayPropertyStore这样的变长DynamicStore 中如图。

如果长度超过一个块（block），则分block存储，并将其在StringPropertyStore/ ArrayPropertyStore中的第1个block 的 block\_id 保存到 PropertyStore类型文件相应record 的PropertyBlock字段中。

PropertyKeyTokenStore和StringPropertyStore 配合用来存储Propery的Key部分。Propery的Key是编码的，key 的 id 保存在 PropertyKeyTokenStore (即 neostore.propertystore.db.index)，key 的字符串名保存在对应的StringPropertyStore类型文件neostore.propertystore.db.index.keys 中。



neostore.propertystore.db文件存储格式示意图，整个文件是有一个 RECORD\_SIZE=41 Bytes 的定长数组构成。

访问时，可以通过关系或节点中的 prop\_id 作为数组的下标进行访问。

Node或Relationship 的属性是通过双向链表方式组织的。

**highByte(1 Byte):**

* 第1~4 bit 表示 next 的高4位；
* 第 5~8 bit表示 prev 的高4位

**prev(4 Bytes) :**

表示本属性在双向链表中的上一个属性的id。是prev property\_id的 低32位。加上highByte字节的第 5~8 bit作为高4位，构成一个完整的36位property\_id。

**next(4 Bytes) :**

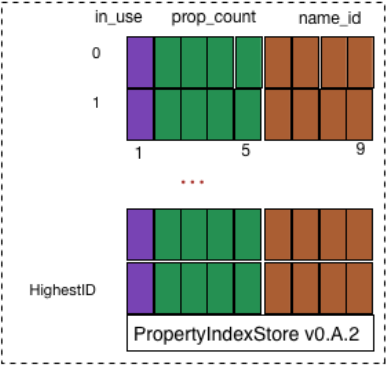
表示本属性在双向链表中的下一个属性的id。是next property\_id的 低32位。加上highByte字节的第 1~4 bit作为高4位，构成一个完整的36位property\_id。

**block\_header(8 Bytes):**

* + key\_id(24 bits) : 第1 ~24 bit , property 的key 的 id
  + type( 4 bits ): 第25 ~28 bit, property 的 value 的类型
  + payload(36 bits): 第29 ~64 bit

对于Interger, Boolean, Float, Byte, Character , Short 类型的值，直接保存在payload；对于long，如果36位可以表示，则直接保存在payload如果不够，则保存到第1个PropertyBlock中，如：double 类型，保存到第1个PropertyBlock中；对于 array/string ，如果编码后在 block\_header及3个PropertyBlock 能保存，则直接保存；否则，保存到（另外的可变存储区）ArrayDynamicStore/StringDynamicStore 中， payload 保存其在ArrayDynamicStore中的数组下标。

2.5.6 属性键存储结构



**Prop\_count (4 Bytes):**

该属性一共存储的数目

**name\_id (4 Bytes):**

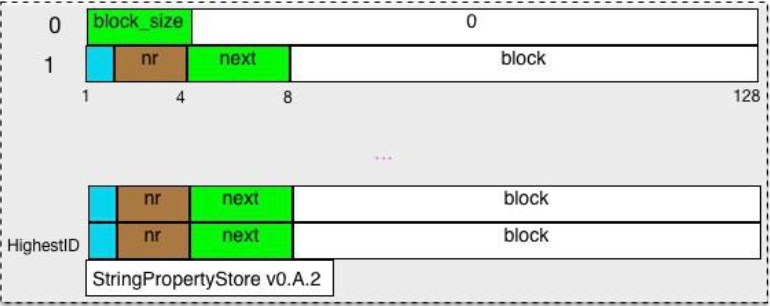
Key 名字的 id 号，该id可以对应propertystore.db.index.keys文件中数组下标查询该属性的key name

其对应的存储格式如上图所示: 是一个长度为 RECORD\_SIZE=9 Bytes 的 record 数组。

访问时，可以通过 key\_id 作为数组的下标进行访问。

Propery的key是编码的，key 的 id 保存在 PropertyKeyTokenStore (即 neostore.propertystore.db.index)，key 的字符串名保存在对应的StringPropertyStore类型文件neostore.propertystore.db.index.keys 中。

2.5.7 变长值保存策略



第1个 record 中(int)保存block\_size：

* Block\_header\_size(8Bytes)
* block\_content\_size的定长数组的size
* 一个字符串“StringPropertyStore v0.A.2”的size

第2个以后的 record开始保存实际的block数据,它由8个字节的block\_header和定长的 block\_content（可配置）构成。

Neo4j 中对于字符串等变长值的保存策略是用一组定长的 block 来保存，block之间用单向链表链接。类 AbstractDynamicStore 实现了该功能。

**inUse(1 Byte)**:

* 第1~4 bit 表示next\_block 的高4位;
* 第5 bit表示block 是否在 use;
* 第8 bit 表示 block 是否是单向链表的第1个 block；0 表示第1 个block, 1表示后续 block。

**nr\_of\_bytes(3Bytes):**

本 block 中保存的数据的长度**。**

**next\_block(4Bytes):**

* next\_block 的低 4 个字节，加上 inUse 的第1~4 位，next\_block 的实际长度共 36 bit。
* 以数组方式存储的单向链表的指针，指向保存同一条数据的下一个 block 的id。

2.6 本章小结

本章首先介绍图数据库事务部分的整体架构，接下来分别对数据库底层各个存储器模块进行介绍，详细阐述了：实体id、节点、关系、关系类型、属性、属性键、变长值等文件存储格式。加深了对图数据库序列化格式的认识。

第三章 图数据库存储引擎设计与实现

3.1 需求分析及设计目标

随着互联网的发展，越来越多的数据出现在了人们的生活当中。而数据中的价值也开始被人们越来越广泛的重视。传统的关系数据库比如 Oracle、MySQL 已经不能够满足日益增长的关系数据的存储和查询功能。无论是政府还是公司，都需要能够快速对关系查询业务进行响应，从而在最短的时间内分析数据内在联系。应运而生的图数据库正好解决查找数据之间关系的方案。

本文基于图数据库的存储场景，研究了基于Java的Neo4j图数据库存储系统，调研其存储引擎的底层数据组织方式和查询模式，提出了一种轻量级的图数据库引擎设计方案。该方案实现了存储引擎的一系列模块：Store，Transaction，TxState，Command，Operations，Lock，FileChannel，ByteBuffer，Record和StoreEngine等。基于这些模块设计，实现了基于 linux 平台的轻量级图数据库存储引擎，最后实施性能验证，分析方案的效率。

3.2 整体架构

各模块功能说明如下：

1、查询和计算：最终用户用于在此语言基础之上进行图的遍历和查询，最终返回运行结果。

2、操作和运维：用于系统实时监控，例如系统配置、安装、升级、运行时监控，甚至包括可视化界面等。

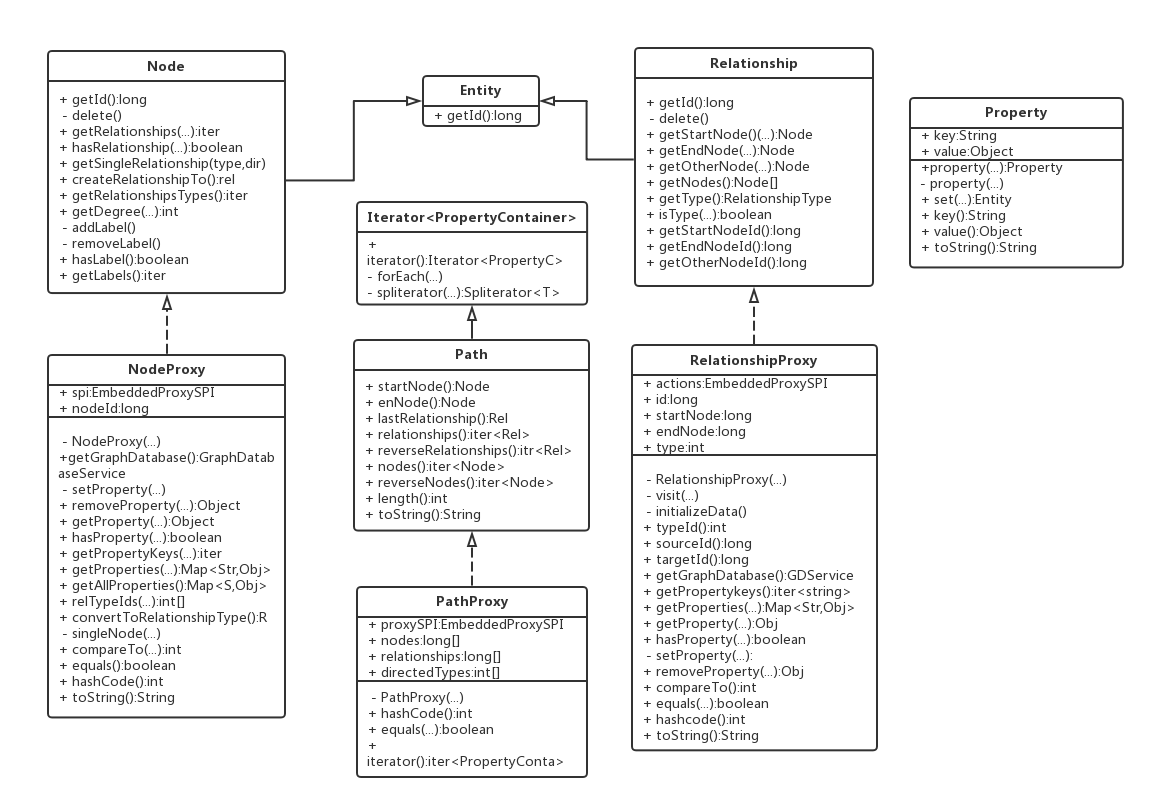
3、数据加载：包括离线数据加载和在线数据加载，既可以是批量的数据加载，也可以是流数据加载方式。

4、图数据库核心：主要包括图存储和图处理引擎这两个核心。图处理引擎负责实时数据更新和执行图运算；图存储负责将关系型数据及其他非结构化数据转换成图的存储格式；HA服务负责处理处理数据容错、数据一致性以及服务不间断等功能。



图 3-1 图数据库架构

3.3 实体的设计与实现



代码3-3-1 NodeProxy.h

class NodeProxy: public Node

{

public:

NodeProxy(EmbeddedProxySPI \*spi, const long &nodeId);

~NodeProxy();

long getId() const override;

void remove() override;

vector<Relationship\*> getRelationships() const;

vector<Relationship\*> getRelationships(RelationshipType type) const;

Relationship \*createRelationshipTo(Node \*otherNode, RelationshipType type);

string toString() const;

private:

EmbeddedProxySPI \*spi;

long nodeId;

};

节点代表的是图数据库关系中的一个实体（代码3-3-1），比如一个人、物。节点可以通过getId()来获取它在数据库中唯一标识它本身的id值，remove()函数可以通过改变临时缓存状态中的关于节点的状态，最后提交Transaction事务时统一将缓存状态中的所有改变的记录刷新到内存缓存中，进而再刷新到磁盘上。

每当用户操作生成一个节点构造关系时，系统会首先生成两个NodeProxy，节点具有一个唯一的id值，需要保存在缓存状态中，之后利用这两个节点进行createRelationshipTo()，继而再生成如下的RelationshipProxy实体类，它保存了两个点的id信息，同时关系中也保存了用户期待的部分属性，例如：LIKE、FRIEND等关系，这称为RelationshipType。关系实体代码如下：

代码3-3-2 RelationshipProxy.h

class RelationshipProxy: public Relationship

{

public:

RelationshipProxy();

~RelationshipProxy();

long getId() const override;

int typeId() const;

long sourceId() const;

long targetId() const;

void remove() override;

Node \*getStartNode() const;

Node \*getEndNode() const;

RelationshipType getType() const override;

bool isType(RelationshipType type) const override;

long getStartNodeId() const override;

long getEndNodeId() const override;

long getOtherNodeId(long id) const override;

private:

EmbeddedProxySPI \*actions;

long id = NO\_ID;

long startNode = NO\_ID;

long endNode = NO\_ID;

int type;

};

由此可知，一个关系（代码3-3-2）具有的操作包括获取两个节点的id，以及获取该关系的类型。关系实体的作用是通过调用getEndNode()来访问一个节点关联的节点，进而进行节点关系的传递，例如：A节点是张三，B节点是李四，他们之间建立了一个RelationshipType为“Friend”的关系，首先获取名字叫“张三”的节点，根据返回的A节点调用getRelationships()，系统返回过滤后的关系，通过该关系调用getEndNode(),最后找到B节点，返回李四给用户。这算是一个简单查询实体的朋友。

与节点一样，新建的关系也通过某种方式把关系的所有信息保存在状态管理器中，提交Transaction时一并添加到缓存，进而刷新到磁盘。

3.4 实体id生成器

每一个实体类例如：节点、关系、属性等都具有一个Store，对应生成一个文件，用于对该类所有的实体数据保存，保存的单位是Record记录项。每一个Stroe里都含有一个ID生成器，即标识唯一实体的id生成器，系统利用该id可以索引到磁盘中的记录项，进行读写操作。

3.4.1 id生成器

代码3-4-1 获取可用id

long IdGenerator::nextId()

{

unique\_lock<mutex> lock(nextIdMutex);

assertStillOpen();

long nextDefragId = idContainer->getReusableId();

if (nextDefragId != IdContainer::NO\_RESULT)

{

return nextDefragId;

}

if (highId == IdGenerator::INTEGER\_MINUS\_ONE)

{

highId++;

}

if (highId == IdGenerator::INTEGER\_MINUS\_ONE)

{

throw new runtime\_error("ReservedIdException");

}

if (highId < 0) {

throw new runtime\_error("NegativeIdException");

}

if (highId > max)

{

throw new runtime\_error("IdCapacityExceededException");

}

return highId++;

}

对该临界区进行加同步锁，并获取一个新的id。id生成器可以从缓存队列中获取一系列连续的freeId，即之前被删除或释放的实体id，它们可以被重用。磁盘中还保存了当前使用的最大id值，如果没有可用的freeId，磁盘会返回最大id值。

3.4.2 id缓冲区

代码3-4-2 FreeIdKeeper.h

class FreeIdKeeper: public Closeable

{

public:

FreeIdKeeper();

~FreeIdKeeper();

FreeIdKeeper(StoreChannel \*channel, int batchSize, bool aggressiveMode);

static long countFreeIds(StoreChannel \*channel);

void freeId(long id);

void truncate(long position);

long getId();

long \* getIds(int numberOfIds);

long getCount();

// freeId -> disk; readFromDisk -> freeIds -> disk

void close();

private:

const static int ID\_ENTRY\_SIZE = sizeof(long);

PrimitiveLongArrayQueue \*freeIds = new PrimitiveLongArrayQueue();

PrimitiveLongArrayQueue \*readFromDisk = new PrimitiveLongArrayQueue();

StoreChannel \*channel;

int batchSize;

bool aggressiveMode;

long freeIdCount;

long stackPosition;

long initialPosition;

// IdFromDisk 缓冲区出队，不够直接从磁盘读取 batchSize 个id

// 磁盘没有 freeId 则返回 NO\_RESULT

long getIdFromDisk();

// 从磁盘读 <= batchSize 个 id

void readIdBatch();

void readIdBatch0();

// freeIds 超过 batchSize 则写入文件

long flushFreeIds(ByteBuffer \*writeBuffer);

long flushFreeIds0(ByteBuffer \*writeBuffer);

// 把文件中的空洞填满（数据前移）

void compact(ByteBuffer \*writeBuffer);

};

每次获取nextId的时候IdContainer会在FreeIdKeeper中查找缓存的freeIds队列，如果有缓存可用的id值，就直接出队操作把id值返回给新节点，如果没有，就检查缓存readFromDisk队列，有就出队继续从文件读取id，如果readFromDisk队列也为空，就用readFromDisk 读取文件中n个可复用id，并且入队到readFromDisk，把readFromDisk中新的id出队返回给节点。

如果 readFromDisk队列 及 读取id 文件都没有已释放且可复用的 id了，则把highId++的id值返回给节点。用户释放一个 id 后，先放入 freeIds队列，当freeIds队列 中回收的 id 个数超过 n 个时，写入到 id 文件的末尾。

Close()把freeIds队列写入文件，再把readFromDisk队列写入freeIds队列，再写入文件。

3.5 记录项

代码3-5 AbstractBaseRecord

class AbstractBaseRecord

{

public:

AbstractBaseRecord();

AbstractBaseRecord(long id);

virtual ~AbstractBaseRecord();

virtual void clear();

virtual long getId() const;

virtual void setId(long id) final;

virtual bool inUse() const final;

virtual void setInUse(bool inUse);

virtual void setCreated() final;

virtual bool isCreated() const final;

const static int NO\_ID = -1;

protected:

long id;

bool inUseFlag;

bool created;

virtual AbstractBaseRecord initialize(bool inUse);

};

记录项（Record）是保存实体序列化数据的类，通常可直接参与磁盘I/O。所有记录项可以通过get/set id来对id进行操作。inUse()标明了记录项是否正在使用，如该记录已被删除，则inUse()返回false，并且会同步到磁盘。

节点记录项、关系记录项等都继承自AbstractBaseRecord。此外，节点记录项保存了下一条关系的id和下一个标签的id。相应的，关系记录项保存了起始节点id和终结点id，以及上下条关系的id等。

3.6 存储模块

代码3-6 AbstractBaseRecord.cpp

template<class RECORD>

inline void CommonAbstractStore<RECORD>::updateRecord(RECORD \* record)

{

long id = record->getId();

// 验证id是否合法

IdValidator::assertValidId(getIdType(), id, recordFormat->getMaxId());

//

long blockId = blockIdForRecord(id);

try

{

fileSwapper.write(record);

// 删除后，释放id

if (!record->inUse())

{

freeId(id);

}

}

catch (const std::exception& e)

{

cerr << e.what();

}

}

每种实体对应一个存储器（Store），这些存储器与各自的存储文件相关联。每个Store内部含有一个id生成器，可对该类型的实体进行id生成和更新。Store通过getRecord(long id, …)来索引获取文件中相应偏移量的记录项。Store通过updateRecord(RECORD \* record)（代码3-6）更新文件中对应记录项。该函数先获取record保存的id值，验证id合法性。然后通过fileSwapper将序列化数据写入磁盘，并检测record的inUse标志位是否关闭，是则释放该id，且将id放进空闲id队列，以便下次复用。

3.7 事务临时状态

代码3-7 TxState.h

class TxState

{

public:

TxState();

~TxState();

void nodeDoCreate(long nodeId);

RemovalsCountingDiffSets getNodes();

RemovalsCountingDiffSets getRelationships();

void accept(TxStateVisitor visitor);

void changed();

void dataChanged();

bool ifHasChanges();

void nodeDoDelete(long nodeId);

SuperDiffSets<int> getLabelStateNodeDiffSets(long nodeId);

bool nodeIsDeletedInThisTx(long nodeId);

bool nodeIsAddedInThisTx(long nodeId);

void relationshipDoCreate(long id, int relationshipTypeId, long startNodeId, long endNodeId);

void relationshipDoDelete(long id, int type, long startNodeId, long endNodeId);

private:

static LabelState LABEL\_STATE;

static NodeState NODE\_STATE;

static RelationshipState RELATIONSHIP\_STATE;

// 保存当前节点、标签、关系的状态，用id（long）索引

PrimitiveLongObjectMap<LabelState> labelStatesMap;

PrimitiveLongObjectMap<NodeState> nodeStatesMap;

PrimitiveLongObjectMap<RelationshipState> relationshipStatesMap;

// 保存创建的标签、属性键、关系类型的Token状态，用id(int)索引

PrimitiveIntObjectMap<string> createdLabelTokens;

PrimitiveIntObjectMap<string> createdPropertyKeyTokens;

PrimitiveIntObjectMap<string> createdRelationshipTypeTokens;

NodeState \*getOrCreateNodeState(long nodeId);

RelationshipState \*getOrCreateRelationshipState(long relationshipId);

RemovalsCountingDiffSets nodes;

RemovalsCountingDiffSets relationships;

bool hasChanges;

bool hasDataChanges;

};

事务的数据变更状态（TxState）（代码3-7）保存了临时增、删、改的数据信息。成员变量labelStatesMap、nodeStatesMap、relationshipStatesMap分别用以保存以各自实体id为索引的数据信息。createdLabelTokens、createdPropertyKeyTokens、createdRelationshipTypeTokens保存的是添加的标签、关系类型、属性键的文本（string）信息。nodes、relationships保存的是节点id和关系id临时的集合。TxState调用accept(TxStateVisitor visitor)，遍历内部的节点id和关系id集合，索引到各自Map中所有的状态，以此转到记录项状态集合（RecordChangeSet）里面，并用记录项（Record）存储。

3.7.1节点状态

代码3-7-1 NodeState.h

class NodeState :public PropertyContainerState

{

public:

NodeState();

~NodeState();

NodeState(long id, TxState \*state);

SuperDiffSets<int> getOrCreateLabelDiffSets();

void addRelationship(long relId, int typeId, Direction direction);

void removeRelationship(long relId, int typeId, Direction direction);

void clear();

void accept(NodeState.Visitor visitor);

bool hasAddedRelationships();

bool hasRemovedRelationships();

PrimitiveIntSet relationshipTypes();

bool hasRelationshipChanges();

SuperDiffSets<int> getLabelDiffSets();

set<long> getAddedRelationships(Direction direction);

set<long> getAddedRelationships(Direction direction, int \*relTypes);

set<long> getAddedRelationships();

set<long>\* getAddedRelationships(RelationshipDirection direction, int relType);

private:

SuperDiffSets<int> labelDiffSets;

RelationshipChangesForNode relationshipsAdded;

RelationshipChangesForNode relationshipsRemoved;

TxState \*state;

};

节点状态（代码3-7-1）中，labelDiffSets保存了节点标签的临时id集合，relationshipsAdded、relationshipsRemoved分别保存了对该节点添加、删除的关系改变的数据信息。节点所有相关的添加、修改信息都保存在节点状态模块中。

3.7.2 关系状态

代码3-7-2 RelationshipState.h

class RelationshipState: public PropertyContainerState

{

public:

RelationshipState();

RelationshipState(long id);

~RelationshipState();

void setMetaData(long startNode, long endNode, int type);

bool accept(RelationshipVisitor visitor);

private:

long startNode = -1;

long endNode = -1;

int type = -1;

};

关系状态（代码3-7-2）保存了添加、修改该关系时起始节点（startNode）和终结点（endNode）的id值，此外，关系种类id（type）也作为关系状态的一部分保存。

3.8 记录项状态

记录项状态保存改变的record（代码3-8-1）。它是由事务临时数据修改状态转换而来的，通过提取TxState中关键数据变化，经过序列化后保存到记录项状态。记录项状态集通过实体id索引。

代码3-8 RecordChanges.h

template <class RECORD>

class RecordChanges

{

public:

RecordChanges();

RecordChanges(Loader<RECORD> \*loader, IntCounter \*globalCounter);

~RecordChanges();

RecordChange<RECORD> \*getIfLoaded(long key);

RecordChange<RECORD> \*getOrLoad(long key);

void setTo(long key, RECORD newRecord);

RecordChange<RECORD> \*setRecord(long key, RECORD record);

int changeSize();

void close();

RecordChange<RECORD> \*create(long key);

vector<RecordChange<RECORD>\*> changes();

private:

PrimitiveLongObjectMap<RecordChange<RECORD>\*> \*recordChanges;

Loader<RECORD> \*loader;

IntCounter \*changeCounter;

};

3.9 事务

数据库事务(Database Transaction) ，是指作为单个逻辑工作单元执行的一系列操作，要么完全地执行，要么完全地不执行。事务处理可以确保除非事务性单元内的所有操作都成功完成，否则不会永久更新面向数据的资源。通过将一组相关操作组合为一个要么全部成功要么全部失败的单元，可以简化错误恢复并使应用程序更加可靠。

3.9.1事务特性

事务四大特性(简称ACID) ：

原子性(Atomicity)：事务中的全部操作在数据库中是不可分割的，要么全部完成，要么均不执行。

一致性(Consistency)：几个并行执行的事务，其执行结果必须与按某一顺序串行执行的结果相一致。

隔离性(Isolation)：事务的执行不受其他事务的干扰，事务执行的中间结果对其他事务必须是透明的。

持久性(Durability)：对于任意已提交事务，系统必须保证该事务对数据库的改变不被丢失，即使数据库出现故障。

3.9.2 事务提交

代码3-9-1 KernelTransaction.cpp

long KernelTransaction::commit()

{

bool success = false;

// Convert changes into commands and commit

if (hasChanges())

{

try

{

vector<Command> \*extractedCommands = new vector<Command>();

storageEngine->createCommands(

extractedCommands,

txState,

storageStatement,

commitLocks);

if (!extractedCommands->isEmpty())

{

success = true;

for (size\_t i = 0; i < extractedCommands->size(); i++)

{

extractedCommands[i].handle(applier);

}

TransactionToApply batch = new TransactionToApply(transactionRepresentation);

txId = transactionId = commitProcess.commit(batch);

}

}

catch (const std::exception& e)

{

cerr << e.what();

}

finally

{

if (!success)

{

rollback();

}

}

}

success = true;

return 0;

}

提交事务（代码3-9-1）时，事务模块首先判断TxState中是否有状态改变，如果有，则调用存储引擎（storageEngine）进行createCommands(…)操作。存储引擎会遍历TxState临时状态中的数据，将之转换为记录项状态，最后遍历这些记录项状态，以增、改、删的顺序将记录项状态中的修改后的序列化record数据入队列extractedCommands。通过访问者模式，系统调用extractedCommands[i].handle(applier)；传入访问者，对每一个命令（command）进行访问处理，进而通过存储器将record写入缓存区或磁盘。

try/catch模块捕获提交事务过程中发生的错误，例如数据错误或者加锁错误，捕获到最后会回滚（rollback()），把所有的临时数据改变状态清空，取消之前所有的修改。

3.10 事务锁

锁（Lock）是数据库引擎用来同步多个用户同时对同一个数据块的访问的一种机制。

在事务获取数据块当前状态的依赖关系（比如通过读取或修改数据）之前，它必须保护自己不受其他事务对同一数据进行修改的影响。事务通过请求锁定数据块来达到此目的。锁有多种模式，如共享或独占。锁模式定义了事务对数据所拥有的依赖关系级别。如果某个事务已获得特定数据的锁，则其他事务不能获得会与该锁模式发生冲突的锁。如果事务请求的锁模式与已授予同一数据的锁发生冲突，则数据库引擎实例将暂停事务请求直到第一个锁释放。

3.10.1读锁

代码3-10-1 FileLock.cpp

int FileLock::readLockReg(int fd, int type, off\_t offset, int whence, off\_t len)

{

struct flock lock, savelock;

lock.l\_type = type;

lock.l\_start = offset;

lock.l\_whence = whence;

lock.l\_len = len;

savelock = lock;

if (fcntl(fd, F\_GETLK, &lock) < 0)

{

err\_sys("fcntl error");

}

// 查看发现有写锁

if (lock.l\_type == F\_WRLCK)

{

return -1;

}

// l\_type == F\_UNLCK, 没有写锁

else

{

if(fcntl(fd, F\_SETLK, &savelock) < 0)

err\_sys("fcntl error");

return 0;

}

}

读取记录项时加读锁，当有读锁时，其他线程可加读锁读取，只有当所有读锁解锁后才可以加写锁进行写入。同时，加有写锁时不可进行加读锁或读取。读锁（代码3-10-1）使用了Linux系统中 “<fcntl.h>”头文件里的fcntl()函数进行加锁，它支持对文件某区域进行加锁，我们利用它对每次读写的记录项区域进行加锁。flock锁结构封装了加锁类型（type）、文件偏移量（offset）、文件起始位（whence）、加锁长度（len）。调用(fcntl(fd, F\_GETLK, &lock)，其中F\_GETLK表示获取当前区域的加锁情况，如果等于F\_WRLCK，表明该区域的记录项已有写锁，则返回-1。否则调用fcntl(fd, F\_SETLK, &savelock)（F\_SETLK表示操作为设置锁），设置读锁，以便后续读取数据。

3.10.2 写锁

代码3-10-2 FileLock.cpp

int FileLock::writeLockReg(int fd, int type, off\_t offset, int whence, off\_t len)

{

struct flock lock, savelock;

lock.l\_type = type;

lock.l\_start = offset;

lock.l\_whence = whence;

lock.l\_len = len;

savelock = lock;

if (fcntl(fd, F\_GETLK, &lock) < 0)

{

err\_sys("fcntl error");

}

// 查看发现有读/写锁

if (lock.l\_type == F\_WRLCK || lock.l\_type == F\_RDLCK)

{

return -1;

}

// l\_type == F\_UNLCK, 没有读/写锁

else

{

if (fcntl(fd, F\_SETLK, &savelock) < 0)

err\_sys("fcntl error");

return 0;

}

}

当有读锁时，不可加写锁或写入，只有当所有读锁解锁后才可以加写锁进行写入。同时，加有写锁时不可再进行加写锁或写入。写锁（代码3-10-2）使用了Linux系统中 “<fcntl.h>”头文件里的fcntl()函数进行加锁，它支持对文件某区域进行加锁，我们利用它对每次读写的记录项区域进行加锁。flock锁结构封装了加锁类型（type）、文件偏移量（offset）、文件起始位（whence）、加锁长度（len）。调用(fcntl(fd, F\_GETLK, &lock)，获取当前区域的加锁情况，如果等于F\_WRLCK或F\_RDLCK，表明该区域的记录项已有写锁或读锁，则返回-1。否则调用fcntl(fd, F\_SETLK, &savelock)，设置写锁，以便后续写入数据。

3.11 I/O类的设计实现

I/O中的文件流和缓冲流模块仿照了Java自带的FileChannel和ByteBuffer类。FileChannel与文件关联，可以读取文件指定内容到ByteBuffer中，再提取缓冲流中的内容。同样的，写入时，将指定内容先写入ByteBuffer，再写入FileChannel关联的文件。读取、写入缓冲流时，格式化部分由RecordFormat模块进行处理。

3.11.1文件流- FileChannel

代码3-11-1 FileChannel.h

class FileChannel

{

public:

FileChannel();

~FileChannel();

FileChannel(char \*storeDir, const char \*fileName);

FileChannel(char \*path);

FileChannel \*open(char \*mode);

char \* getParentFile();

FILE \* getFilePtr();

char \* getFileName();

char \* getMode();

// -- Channel operations --

int read(ByteBuffer \*dst);

long read(ByteBuffer \*\*dst, int offset, int length);

int read(ByteBuffer \*dst, long position);

int write(ByteBuffer \*src);

long write(ByteBuffer \*\*src, int offset, int length);

int write(ByteBuffer \*src, long position);

// -- Other operations --

long position();

FileChannel \*position(long newPosition);

long size();

FileChannel \*truncate(long size);

void flush();

FileLock \*tryLock();

void close();

bool isOpen();

bool fileExists();

private:

//file pointer

FILE \* pFile;

char \* path;

char \* mode;

bool ifOpen;

char \*storeDir;

mutex closeMutex;

};

文件流（代码3-11-1）把文件句柄（pFIle）、文件名（path）、打开模式（mode）封装在内部，向外层暴露读、写接口，分别将文件内容读进ByteBuffer和将ByteBuffer的内容写入文件。文件流支持文件指针偏移（position()），刷新文件（flush()）等操作。

3.11.2 缓冲区- ByteBuffer

代码3-11-2 ByteBuffer.h

class ByteBuffer

{

public:

ByteBuffer();

ByteBuffer(int mark, int pos, int lim, int cap, // package-private

char \* buf, int offset);

ByteBuffer(int mark, int pos, int lim, int cap);

~ByteBuffer();

ByteBuffer\* allocate(int capacity);

void nextReadBufSeq(const int &nb);

char\* nextWriteBufSeq(int \*realReadSize);

void setBuf(char \*buf);

void appendToBuf(char buf[]);

int getCapacity() const;

int getPosition() const;

int getLimit() const;

ByteBuffer \*setPosition(int newPosition);

ByteBuffer \*setLimit(int newLimit);

ByteBuffer \*setMark();

ByteBuffer \*reset();

ByteBuffer \*clear();

ByteBuffer \*flip();

ByteBuffer \*rewind();

int remaining();

bool hasRemaining();

int markValue();

void truncate();

void discardMark();

// private methods for bounds checking

int nextGetIndex();

int nextGetIndex(int nb);

int nextPutIndex();

int nextPutIndex(int nb);

int checkIndex(int i);

int checkIndex(int i, int nb);

static void checkBounds(int off, int len, int size);

// -- get put methods --

int ix(const int &i);

char get();

char get(const int &index);

ByteBuffer \*put(const char &b);

ByteBuffer \*put(const int &index, const char &b);

ByteBuffer \*get(char \* dst, const int &offset, const int &length);

ByteBuffer \*get(char \* dst);

ByteBuffer \*put(char \* src, const int &offset, const int &length);

ByteBuffer \*put(char \* src);

int getInt();

ByteBuffer \*putInt(const int &value);

int getInt(const int &index);

ByteBuffer \*putInt(const int &index, const int &value);

long getLong();

ByteBuffer \*putLong(const long &value);

long getLong(const int &index);

ByteBuffer \*putLong(const int &index, const long &value);

private:

// store data in bytes

char \* buf;

// mark a index for the position to return

int mark = -1;

// read or put cursor position

int position = 0;

// allocated size of this buf

int capacity;

// read/write limitation index of this buf

int limit;

// offset from this position or from the beginning

int offset;

};

缓冲区模块（代码3-11-2）可以将文件流中指定位置的内容按指定长度读入buf。缓冲区（ByteBuffer）通过position对当前缓冲区内容偏移量进行管理，以便写入文件时进行偏移控制。capacity表示该缓冲区读取的最大上限，读取record时，通常会设置成一个record的长度。读取完毕后，调用flip()，将position置零，同时将limit移动到之前的position位置表示buf的结尾，复位后可从position = 0的位置进行读取buf，反序列化到record内部。

3.11.3节点格式化I/O模块

代码3-11-3 NodeRecordFormat.cpp

void NodeRecordFormat::write(NodeRecord \* record, ByteBuffer \* buf, int recordSize)

{

buf->allocate(recordSize);

if (record->inUse())

{

long nextRel = record->getNextRel();

long nextProp = record->getNextProp();

short relModifier = nextRel == (Record(Record::NO\_NEXT\_RELATIONSHIP)).getIntType() ? 0 : (short)((nextRel & 0x700000000L) >> 31);

short propModifier = nextProp == (Record(Record::NO\_NEXT\_PROPERTY)).getIntType() ? 0 : (short)((nextProp & 0xF00000000L) >> 28);

// [ , x] in use bit

// [ ,xxx ] higher bits for rel id

// [xxxx, ] higher bits for prop id

short inUseUnsignedByte = Record(record->inUse() ? Record::IN\_USE : Record::NOT\_IN\_USE).getByteType();

inUseUnsignedByte = (short)(inUseUnsignedByte | relModifier | propModifier);

buf->put(inUseUnsignedByte);

buf->putInt((int)nextRel);

buf->putInt((int)nextProp);

// lsb of labels

long labelField = record->getLabelField();

buf->putInt((int)labelField);

// msb of labels

buf->put((char)((labelField & 0xFF00000000L) >> 32));

char extra = record->isDense() ? (char)1 : (char)0;

buf->put(extra);

}

// 只修改inUse部分

else

{

markAsUnused(buf);

}

delete buf;

}

节点格式化模块（代码3-11-3）将inUse的第1位标识为inUse（bool），将关系的id（long）第33位~第35写入inUse（1 Byte）的2~4位；将属性id（long）的第33位~第36位写入inUse的5~8位。并把关系id和属性id的低32位直接强制转化为int（32bit）型，写入文件。把标签id的低32位转换为int（32bit）型，写入文件后，其第33位到第40位转为char（1Byte）写入文件，即5个Bytes内容保存标签。

3.11.4 关系格式化I/O模块

代码3-11-3 RelationshipRecordFormat.cpp

void RelationshipRecordFormat::write(RelationshipRecord \* record, ByteBuffer \* buf, int recordSize)

{

buf->allocate(recordSize);

if (record->inUse())

{

long firstNode = record->getFirstNode();

short firstNodeMod = (short)((firstNode & 0x700000000L) >> 31);

long secondNode = record->getSecondNode();

long secondNodeMod = (secondNode & 0x700000000L) >> 4;

long firstPrevRel = record->getFirstPrevRel();

long firstPrevRelMod = firstPrevRel == (Record(Record::NO\_NEXT\_RELATIONSHIP)).getIntType() ? 0 : (firstPrevRel & 0x700000000L) >> 7;

long firstNextRel = record->getFirstNextRel();

long firstNextRelMod = firstNextRel == (Record(Record::NO\_NEXT\_RELATIONSHIP)).getIntType() ? 0 : (firstNextRel & 0x700000000L) >> 10;

long secondPrevRel = record->getSecondPrevRel();

long secondPrevRelMod = secondPrevRel == (Record(Record::NO\_NEXT\_RELATIONSHIP)).getIntType() ? 0 : (secondPrevRel & 0x700000000L) >> 13;

long secondNextRel = record->getSecondNextRel();

long secondNextRelMod = secondNextRel == (Record(Record::NO\_NEXT\_RELATIONSHIP)).getIntType() ? 0 : (secondNextRel & 0x700000000L) >> 16;

long nextProp = record->getNextProp();

long nextPropMod = nextProp == (Record(Record::NO\_NEXT\_PROPERTY)).getIntType() ? 0 : (nextProp & 0xF00000000L) >> 28;

// [ , x] in use flag

// [ ,xxx ] first node high order bits

// [xxxx, ] next prop high order bits

short inUseUnsignedByte = (short)((Record(record->inUse() ? Record::IN\_USE :

Record::NOT\_IN\_USE)).getByteType() | firstNodeMod | nextPropMod);

// [ xxx, ][ , ][ , ][ , ] second node high // order bits, 0x70000000

// [ ,xxx ][ , ][ , ][ , ] first prev rel high // order bits, 0xE000000

// [ , x][xx , ][ , ][ , ] first next rel high // order bits, 0x1C00000

// [ , ][ xx,x ][ , ][ , ] second prev rel high // order bits, 0x380000

// [ , ][ , xxx][ , ][ , ] second next rel high // order bits, 0x70000

// [ , ][ , ][xxxx,xxxx][xxxx,xxxx] type

int typeInt = (int)(record->getType() | secondNodeMod | firstPrevRelMod | firstNextRelMod | secondPrevRelMod | secondNextRelMod);

// [ , x] 1:st in start node chain, 0x1

// [ , x ] 1:st in end node chain, 0x2

long firstInStartNodeChain = record->isFirstInFirstChain() ? 0x1 : 0;

long firstInEndNodeChain = record->isFirstInSecondChain() ? 0x2 : 0;

char extraByte = (char)(firstInEndNodeChain | firstInStartNodeChain);

buf->put((char)inUseUnsignedByte);

buf->putInt((int)firstNode);

buf->putInt((int)secondNode);

buf->putInt(typeInt);

buf->putInt((int)firstPrevRel);

buf->putInt((int)firstNextRel);

buf->putInt((int)secondPrevRel);

buf->putInt((int)secondNextRel);

buf->putInt((int)nextProp);

buf->put(extraByte);

}

// 只修改inUse部分

else

{

markAsUnused(buf);

}

}

关系格式化模块（代码3-11-3）将inUse的第1位标识为inUse（bool），把firstNode的id的第33位~第35位保存到inUse的2~4位，低32位转换为int（32bit）型写入文件。nextProp 的id第33位~第35位保存到inUse的5~8位。typeInt（32bit）（关系种类）不仅保存了关系种类的id（16bit），还保存了以下数据：

* + 第29~31 位是second\_node 的node\_id高3位;
  + 第26~28 位是first\_prev\_rel\_id 的 relationship\_id高3位;
  + 第23~25 位是first\_next\_rel\_id 的relationship\_id高3位;
  + 第20~22 位是second\_prev\_rel\_id 的relationship\_id高3位;
  + 第17~19 位是second\_next\_rel\_id 的relationship\_id高3位;
  + 第 1~16 位 表示 RelationShipType;

以上变量的高3位（第33bit~第35bit）保存到typeInt中写入文件后，其低32位均强转成int（32bit）型依次写入文件。

First-in-chain-markers（2bit）表示是否是关系的第一条或者最后一条，加入extryByte的低2位，并写入文件。

3.12 本章小结

本章首先介绍了图数据库存储引擎的需求分析及设计目标，然后介绍了方案的整体架构以及具体实现方式，介绍了实体、id生成器、记录项、存储器、事务状态、记录项状态、事务、事务锁和I/O类等模块的设计与实现方式，以及它们如何组合为存储引擎服务。

第四章 性能测试

4.1 测试方案

测试环境如表4-1，本存储引擎实现了的一系列模块：Store，Transaction，TxState，Command，Operations，Lock，FileChannel，ByteBuffer，Record和StoreEngine等。基于这些模块设计，实现了基于 Linux 平台的轻量级图数据库存储引擎，最后实施插入节点和关系的时间性能验证，分析方案的效率。

4.1.1 测试环境

表4-1 测试环境

|  |  |  |  |
| --- | --- | --- | --- |
| 操作系统 | CPU | 内存 | JVM内存 |
| Ubuntu 14.04 LTS | I3 2.4Ghz 4核 | 8G | 1024M |

由于目前数据库的应用场景大部分在Linux服务器环境下，故测试环境如表4-1所示，其中CPU采用I3 2.4Ghz 4核。内存采用8G，JVM 1024M配置下，最多可插入3万多个节点和关系，再多就溢出。

4.1.2 测试数据

* + 4000节点，4000关系
  + 8000节点，8000关系
  + 16000节点，16000关系
  + 24000节点，24000关系
  + 32000节点，32000关系
  + 40000节点，40000关系

测试数据按插入节点/关系4000的倍数梯度上升，方便对比。期间测试记录插入时间分析插入数据量和时间的关系，以及对比Neo4j存储引擎和本论文研究的轻量存储引擎的插入时间开销。

4.2 测试结果与分析

Neo4j采用其原生接口操作：

4000 节点， 4000 关系： 1 秒 , 其间 cpu 使用率 25% ， 761M

8000 节点， 8000 关系： 2 秒 , 其间 cpu 使用率 25% ， 829M

16000 节点， 16000 关系： 5 秒 , 其间 cpu 使用率 25 ， 983M

24000 节点， 24000 关系： 9 秒 , 其间 cpu 使用率 25% ， 1079M

32000 节点， 32000 关系： 14 秒 , 其间 cpu 使用率 25% ， 1187M

40000 节点， 40000 关系：执行 1 多分钟以后直接报 outOfMemery: java heap space

由表4-2以及图4-2可以知道，在Neo4j原生接口和本论文的存储引擎（myStore）插入数据时，随着数据规模的增大，存储数量增加到一定的程度之后插入时间出现上升的趋势，这是由于数据规模增大，磁盘I/O的开销也越来越大，而且磁盘I/O开销占整个存储引擎插入时间的主要部分。

Neo4j的实现方式是先存入内存缓存页面，再利用页面交换模块将每页（8M）的数据分别写入文件。加锁的力度是按页加锁，锁粒度大，开销比较小，而且I/O的次数也较少。

myStore存储引擎由于没有内存缓存区设置，写入文件按记录项加锁，锁的粒度比较小，所以开销比较大。加上写入文件时是按记录项进行单独I/O，I/O次数极为频繁，耗费的I/O时间较大，所以跟插入时间比Neo4j的时间更多。

表4-2 插入时间比较

|  |  |  |
| --- | --- | --- |
| 时间（ms）  节点/关系量（个）  节点() | Neo4j（ms） | myStore（ms） |
| 4,000 | 1102 | 3520 |
| 8,000 | 2117 | 7387 |
| 16,000 | 5208 | 17302 |
| 24,000 | 9412 | 30210 |
| 32,000 | 14028 | 55026 |
| 40,000 | outOfMemery | outOfMemery |

图4-1 Neo4j和myStore插入时间对比

4.3 本章小结

本章首先介绍了采用的测试方案以及测试环境，最后得出测试数据结果，通过分析表格数据和折线图比较了两个存储引擎的插入性能测试的结果，进行了分析从而验证了方案的可行性。

第五章 全文总结与展望

5.1 全文总结

在科技不断发展的时代下，本图数据库存储引擎借鉴了时下流行的Neo4j图数据库系统提出了一种轻量级的图数据库引擎设计方案。该方案实现了存储引擎的一系列模块：Store，Transaction，TxState，Command，Operations，Lock，FileChannel，ByteBuffer，Record和StoreEngine等。基于这些模块设计，实现了基于 Linux 平台的轻量级图数据库存储引擎，最后实施性能验证，分析方案的效率。并且通过实施性能测试，验证了方案的效率。

5.2 后续工作展望

本方案虽然已经实现，但是还是有许多日臻完善的地方：

1、存储引擎只是实现了节点和关系插入的部分，对属性、标签等实体的实现还有待研究。

2、内存缓存部分还没有实现，只是数据直接与磁盘I/O，开销比较大，有待进一步完善。

3、加锁的粒度太小，锁的开销比较大，而且没有使用乐观锁等技术，并发量比较小。

要达到一个更加完善的图数据库存储引擎，等待解决和丰富的内容还有很多，这些都有待自己在接下来的时间里一一完成。

致 谢

大学四年转眼就要结束了，非常感谢在大学里认识的老师们、同学们。四年里，我总是不断的感到迷茫，老师们就像指路的明灯，一次又一次的为我指明了前进的方向。感谢身边的同学们一直以来支持和帮助，让我不断成长，同时也在四年的大学生活里收获了许多快乐而美好的回忆。也要感谢实验室的师兄师姐们一直依赖的帮助，是你们让我少走了许多的弯路。最后，我要感谢我的导师段翰聪老师，在本文完成的过程中总是耐心的指导，不断的从思路上进行点，从行为上进行督促，让我能更好的完成本篇论文。

再一次诚挚的对你们致谢，也祝各位老师工作顺利、身体健康。也祝自己给这四年的学习生涯画一个圆满的句号。

参考文献

1. De Witt D. J., Katz R. H., Olken F., et al. Implementation techniques for main memory database systems[A].In Proceedings of the 1984 ACM International Conference on Management of Data[C].Boston,USA:ACM,1984:1-8
2. Boncz P A, Zukowski M, Nes N. MonetDB/X100: Hyper-Pipelining Query Execution[C].CIDR. 2005, 5: 225-237.
3. Färber F, Cha S K, Primsch J, et al. SAP HANA database: data management for modern business applications[J]. ACM Sigmod Record, 2012, 40(4): 45-51.
4. Oukid I, Lehner W, Kissinger T, et al. Instant Recovery for Main Memory Databases[C]. 2015 7th Biennial Conference on Innovative Data Systems Research, Asilomar, California, USA. CIDR. 2015.
5. Yuan Y, Lee R, Zhang X. The Yin and Yang of processing data warehousing queries on GPU devices[J]. Proceedings of the VLDB Endowment, 2013, 6(10): 817-828.
6. Heimel M, Saecker M, Pirk H, et al. Hardware-oblivious parallelism for in-memory column-stores[J]. Proceedings of the VLDB Endowment, 2013, 6(9): 709-720.
7. Breß S, Beier F, Rauhe H, et al. Efficient co-processor utilization in database query processing[J]. Information Systems, 2013, 38(8): 1084-1096.
8. Breß S. The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS[J]. Datenbank-Spektrum, 2014, 14(3): 199-209.
9. Bakkum P, Chakradhar S. Efficient data management for GPU databases[J]. High Performance Computing on Graphics Processing Units, 2012.
10. Li P, Luo Y, Zhang N, et al. HeteroSpark: A heterogeneous CPU/GPU Spark platform for machine learning algorithms[C]. 2015 IEEE International Conference on Networking, Architecture and Storage (NAS). IEEE, 2015: 347-348.
11. Jia Y, Shelhamer E, Donahue J, et al. Caffe: Convolutional architecture for fast feature embedding[C]. Proceedings of the 22nd ACM international conference on Multimedia. ACM, 2014: 675-678.
12. Codd E F. A relational model of data for large shared data banks[J]. Communications of the ACM, 1970, 13(6): 377-387.

外文资料原文



外文资料译文

通过CUDA使用GPU加速SQL数据库算子

Peter Bakkum and Kevin Skadron

Department of Computer Science University of Virginia,

Charlottesville, VA 22904 {pbb7c, skadron}@virginia.edu

摘要

以前的工作已经显示出对GPU上各种数据库操作的显着加速，但是仅使用不属于常规数据库语言（如SQL）一部分的原语。 本文直接在GPU上实现了SQLite命令处理程序的一个子集。 这大大减少了实现GPU加速所需的努力，避免了数据库程序员需要使用新的编程语言（如CUDA）或修改其程序以使用非SQL库。

本文着重于加速SELECT查询，并在SQLite命令处理器的高效GPU实现中描述考虑因素。 根据结果集的大小，NVIDIA特斯拉C1060的结果可实现20-70倍的加速。

1.介绍

GPU通常被称为视频卡，是计算机在屏幕上呈现图形信息的手段。现GPU的并行架构在某些问题上提供了非常高的吞吐量，并且其在台式计算机中几乎普遍使用意味着它是一种便宜且无所不在的处理能力来源。通过诸如NVIDIA CUDA（一种应用程序编程接口）开发的框架，将这种功能应用于更一般的非图形化问题越来越受到人们的关注，为程序员提供了一种在NVIDIA GPU上执行通用逻辑的简单而标准的方法。程序员经常使用CUDA和类似的接口来加速计算密集的数据处理操作，通常执行速度提高了五十倍，程序员必须了解六个不同的存储空间，CUDA线程和线程块如何映射到GPU硬件，理解CUDA的线程内部通信等.CUDA已经使GPU开发更接近于主流，但程序员仍然必须为GPU执行的每个数据处理操作编写一个低级别的CUDA内核，这是一个频繁重复工作的时间任务。

SQL是用于操纵和查询数据库的行业标准通用声明性语言。能够执行非常复杂的数据集连接和聚合，SQL被用作过程程序和数据结构化表之间的桥梁。SQL查询的加速将使程序员能够通过很少或根本不改变其源代码来提高数据处理操作的速度。尽管需要GPU程序加速，但是即使在GPU上严格地仿真了SQL查询来证明并行架构对这种执行模式的适应性，SQL也不能自动访问GPU。

目前的GPU技术存在限制，这些技术影响了这样一种GPU SQL实现的潜在用户。两个最相关的技术限制是GPU内存大小和主机到GPU设备内存传输时间。虽然未来的显卡几乎肯定会有更大的内存，但目前的NVIDIA卡最多可以达到4 GB，是许多数据库大小的一小部分。在CPU和GPU之间传输内存块仍然是昂贵的。因此，将数据行分段到GPU和分段结果行返回需要很大的开销。尽管有这些限制，实际的查询执行可以通过GPU的高度并行组织并发运行，因此优于执行CPU查询。

尽管有上述限制，但仍有许多适用于该项目领域的应用程序。许多数据库，如用于研究的数据库，不频繁地修改数据，并在读取查询期间遇到最大的负载。另一组应用程序更关心特定查询的延迟，而不是严格遵守最新的数据，一个例子是互联网搜索引擎。对大尺寸数据集的许多查询仅处理总数据的一个子集，从而邀请将该子集分为GPU存储器。另外，虽然GPU的有限存储器大小是一个很大的限制，但只分配了Tesla C1060的4 GB的一半来存储数据集，为用户提供了超过1.34亿行4个整数的空间。