**毕业论文**

1. 绪论
   1. 研究背景及意义

BIM产业的发展、BIM数据的增长、数据共享的重要性等等

* 1. 国内外研究现状

关系数据库、BIMServer（BerkeleyDb）、内存数据库redis、Hbase；

空间关系数据处理的不足，引出图数据库

* 1. 本文贡献

基于es存储属性数据、基于neo4j存储空间结构、改进neo4j路径查询

1.4 组织结构

第二章 相关技术基础

2.1 IFC相关技术

2.1.1 IFC标准

2.1.2 IFC文件格式

2.2 全文检索技术

2.2.1 全文检索基础理论

2.2.1 Lucence概述

2.2.2 ElasticSearch技术

2.3 图数据库技术

2.3.1 图的概念

2.3.3 Neo4j概述

第三章 基于ElasticSearch的IFC对象属性数据的存储和检索研究

3.1 IFC对象属性数据模型概述

3.1.1 IFC对象属性数据

3.2 数据提取技术

3.2.1 IFC文件解析

3.2.2 JSON数据生成

3.3 数据索引技术

3.3.1数据模式映射

3.3.2索引过程

3.3.3索引优化

3.4 数据检索技术

3.4.1 全文检索

3.4.2 关键词检索

3.5 自定义ID字段

第四章 基于Neo4j的IFC对象空间关系的存储和查询研究

4.1 IFC对象空间关系模型

4.2 图数据模型设计

4.3 图数据提取技术

4.3 图数据索引技术

4.4 图数据检索技术

4.5 基于BloomFilter的路径查询优化

第五章 系统实验

6.1 属性数据的索引与检索

6.1.1 索引效率

6.1.2 检索效率

6.2 空间关系数据的索引与检索

6.2.1 索引效率

6.2.2 检索效率

第六章 总结与展望

**目录**

[第一章 绪论 1](#_Toc492626076)

[1.1 研究背景及意义 1](#_Toc492626077)

[1.2 国内外研究现状 1](#_Toc492626078)

[1.3 本文贡献 2](#_Toc492626079)

[1.4 组织结构 3](#_Toc492626080)

[第二章 相关技术基础 4](#_Toc492626081)

[2.1 IFC相关技术 4](#_Toc492626082)

[2.1.1 IFC标准 4](#_Toc492626083)

[2.1.2 IFC文件格式 6](#_Toc492626084)

[2.2 全文检索技术 7](#_Toc492626085)

[2.2.1 全文检索基础理论 7](#_Toc492626086)

[2.2.1 Lucence概述 8](#_Toc492626087)

[2.2.2 ElasticSearch技术 9](#_Toc492626088)

[2.3 图数据库技术 11](#_Toc492626089)

[2.3.1 图数据与图数据库 11](#_Toc492626090)

[2.3.2 Neo4j概述 11](#_Toc492626091)

[第三章 基于ElasticSearch的IFC对象属性数据的存储和检索研究 14](#_Toc492626092)

[3.1 IFC对象属性数据模型概述 14](#_Toc492626093)

[3.1.1 属性 14](#_Toc492626094)

[3.1.2 属性集 15](#_Toc492626095)

[3.1.3 属性集与IFC对象关联 17](#_Toc492626096)

[3.2 数据提取技术 17](#_Toc492626097)

[3.2.1 IFC文件解析 18](#_Toc492626098)

[3.2.2 JSON数据映射 19](#_Toc492626099)

[3.3 数据索引技术 22](#_Toc492626100)

[3.3.1数据模式映射 22](#_Toc492626101)

[3.3.2索引过程 24](#_Toc492626102)

[3.3.3索引过程优化 26](#_Toc492626103)

[3.4 数据检索技术 27](#_Toc492626104)

[第四章 基于Neo4j的IFC对象空间关系数据的存储和查询研究 30](#_Toc492626105)

[4.1 IFC对象空间关系数据模型 30](#_Toc492626106)

[4.2 图数据模型设计 32](#_Toc492626107)

[4.3 图数据提取技术 34](#_Toc492626108)

[4.4 图数据存储设计 36](#_Toc492626109)

[4.5 图数据查询设计 37](#_Toc492626110)

[4.6 基于BloomFilter的路径查询优化 39](#_Toc492626111)

[4.6.1 特定路径查询场景 39](#_Toc492626112)

[4.6.2 BloomFilter原理 41](#_Toc492626113)

[4.6.3 构建BloomFilter 42](#_Toc492626114)

[4.6.4 基于BloomFilter的查询 43](#_Toc492626115)

[第五章 系统实验 44](#_Toc492626116)

[5.1 属性数据的索引与查询 44](#_Toc492626117)

[5.2 空间关系数据的索引与查询 49](#_Toc492626118)

[第六章 总结与展望 52](#_Toc492626119)

# 第一章 绪论

## 1.1 研究背景及意义

时光如驹，岁月如梭，随着互联网的日益发展，所产生的数据日益增加，已经逐渐进入了大数据时代，而大数据也渐渐的融入了各行各业。当今世界，一场新的制造业竞争已然拉开序幕：美国力促高端制造业回归、德国倾力打造工业4.0。而作为制造业大国，中国则推出了"中国制造2025"行动计划。"工业4.0"是以智能制造为主导的第四次工业革命，智能制造和智能工厂被列为工业4.0主攻方向。制造业的这一轮变革使得制造智能化成为一场全球范围的技术革命。之所以称其为技术革命，很大程度上是因为这场工业革命与人工智能、物联网、大数据、云计算等等这些最前沿的技术有着千丝万缕的联系。建筑行业作为制造业的重要组成部分，得到了广泛关注，作为建筑行业的行业数据的标准，BIM行业得到了广泛关注。

BIM系统是一种全新的信息化管理系统，目前正越来越多应用于建筑行业中。它的全称为Building Information Model，即建造信息模型，要求参建各方在设计、施工、项目管理、项目运营等各个过程中，将所有信息整合在统一的数据库中，通过数字信息仿真模拟建筑物所具有的真实信息，为建筑的全生命周期管理提供平台。在整个系统的运行过程中，要求业主、设计方、监理方、总包方、分包方、供应方多渠道和多方位的协调，并通过网上文件管理协同平台进行日常维护和管理。BIM为建筑行业提供了统一的数据标准。

BIM系统的核心是通过三维设计获得工程信息模型和几乎所有与设计相关的设计数据，可以持续及时地提供项目设计范围、进度以及成本信息。这些信息完整可靠，质量高并且完全协调。通过工程信息模型可以使得整个工程过程具有以下收益，交付速度加快（节省时间），协调性加强（减少错误），成本降低（节省资金），生产效率提高，工作质量上升，收益和商业机会增多，沟通时间减少，对整个建筑生产过程有着不可言喻的好处。

然而，随着时代的发展，智慧城市概念的提出，BIM数据有了长足的增长，以往的存储和查询手段已经渐渐越来越不能应付数据的长足提升。所以为海量BIM数据提供一种高效快速的存储和查询手段是有必要和有意义的。

## 1.2 国内外研究现状

1) 关系型BIM数据库

关系型BIM数据库,其主要采用关系型数据库存储BIM数据。一般根据IFC标准的逻辑大纲设计数据逻辑模式。譬如针对IFC中每个实体定义一个表，并根据该实体的每个属性定义一个表的字段。现有的关系型BIM数据库包括VTT Building and Transport And SECOM Co．开发的一个基于SQL Server的IFC数据库[1],清华大学张洋博士开发的“BIM信息集成平台”[2]，Cruz等开发的ACTIVe3D系统采用Oracle进行IFC数据存储[3][4][5]。由于IFC实体的具有大量可选属性，很多属性值为空(即稀疏性)，但关系数据库仍为其预留大量空间，从而导致存储空间的浪费。另外，关系数据库的扩展性较差，难以支持海量BIM数据的存储，所以关系型数据库已经不能满足BIM数据管理的需要。

2) 面向对象型BIM数据库

鉴于IFC模型的面向对象特性，一些学者提出应用面向对象数据库存储BIM数据。Faraj利用面向对象数据库ObjectStore存储IFC数据[6]。陆宁设计和实现了基于面向对象数据库Versant Object Database 8的IFC数据库，并通过与关系数据库对比，指出面向对象型IFC数据库效率更高[7]。但鉴于面向对象数据库本身理论不完善、价格昂贵等问题，存在难以推广的问题，且其可扩展性较NoSql数据库差。

3) 对象关系型BIM数据库

面向对象技术与传统关系型数据库技术相结合而形成的数据库系统，也可以说是一种扩展关系数据库，使它具有一定面向对象数据库特征。Kang等利用对象关系数据库Cubrid存储IFC数据[8]。但目前对象关系数据库尚不成熟，没有广泛使用的商业软件。

4) 键值型BIM数据库

当关系型数据库在可扩展方面几乎已经达到极限时，键值对数据库开始流行起来，它为解决大数据的需求提供了解决方案。这是一种NoSQL（非关系型数据库）模型，其数据按照键值对的形式进行组织、索引和存储。

开源的项目BIMServer采用键值对数据库Berkeley DB来进行数据的存储。Berkeley DB是一个开源的嵌入式数据库系统，提供了一系列直接访问数据库的函数，支持存储任意类型的键/值对，具有简单、小巧、可靠、高性能的特点。BIMServer赋予每一个IFC实体一个全局唯一的ID作为key，将IFC实体的所有字段值解析为字节数组作为value，将key/value对存储与Berkeley DB中，能够支持基于IFC实体ID的快速查询和应用。其局限在于键值对存储IFC实体信息的方式比较单一，不能够支持任何关系数据的存储，也不能支持基于字段的查询。

清华大学的刘强等为实现IFC模型在Web应用中的应用，使用分布式键值对存储系统Redis建立了IFC数据的高速缓存[9]。它将IFC文件以数据流方式写入Redis缓存，利用IFC文件的行编号进行索引，大幅提升了IFC文件的解析效率。同时结合内存中的EXPRESS字典使用JSON格式构建IFC对象模型，从而支持面向Web的IFC模型对象的传输和查询等应用。这种方式的局限类似于Berkeley DB，它只能支持基于IFC实体ID（在这里为行编号）的查询。

5) 列式数据库

清华大学的余芳强等基于分布式的开源列式数据库Hbase来解决基于列值的查询问题和关系数据join问题。不同于关系数据库的按行存储数据的方式，Hbase以列为单位存储数据，使用列族（Family）和列名共同定义一个列，擅长以列为单位存取数据。Hbase支持动态添加列，从而支持半结构化的BIM模型设计，以一定的冗余避免了多表连接操作，极大地提高了信息提取速度。

但是其半结构化的数据库设计只关注了IFC类型的可交换实体和非可交换实体，没有考虑到用户的实际需求。而这部分则是本文研究的重点。

## 1.3 研究内容和本文贡献

结合国内外BIM数据库研究的发展方向，可以看到，BIM数据库研究的目标是支持大数据量的存储和查询、支持更为灵活的查询方式。

基于此目标，本文选择了开源弹性搜索框架ElasticSearch并展开了研究。ElasticSearch 是分布式实时搜索、实时分析、实时存储引擎，使用Java开发并使用Lucene作为其核心来实现所有索引和搜索的功能。倒排索引保证了它的检索性能十分强劲。同时它还支持排序、聚合等复杂灵活的查询方式，满足BIM数据的存储和查询需求。

由于ElasticSearch是面向JSON文档存储的，本文将解决如何将IFC文件数据映射为JSON格式数据的问题。另一方面，考虑到IFC属性数据的重要性和用户对于属性数据的关注，本文定义了支持属性数据存储的半结构化JSON数据模式设计，将属性数据集成到IFC对象中去，避免了属性查询时的多表连接操作。对于基于ElasticSearch的属性数据存储的相关研究，本文将分为IFC数据解析、JSON数据映射、索引过程、索引优化等部分来进行详细介绍。

其次，本文还深入研究了IFC数据标准，提出了IFC对象空间关系数据模型的存储设计。这种空间关系模型与物理世界的建筑构件的组成、连接等关系相对应，往往是用户进行建筑构件查询的常常涉及到的模型。而这种关系模型可以理解成一种图模型，从而适合使用图数据库存储。

本文选择了开源的Neo4j图数据库进行IFC实体空间关系数据模型的存储，从而在属性值查询的基础上，为用户提供了另外一种维度的查询方式，即空间关系（或者成为路径）的查询方式。

IFC实体空间关系在IFC文件中被定义为引用关系。由于IFC标准中定义的实体种类众多，引用关系复杂。当数据量增长时，关系复杂度也跟着增长。本文将解决如何将这种关系数据提取出来、存储到Neo4j中去。具体的内容将分为图数据模型设计、图数据提取技术、存储和查询等部分来进行详细地介绍。

除此之外，本文进一步关注了一种特定的路径查询，并针对这种查询提出了基于BloomFilter的解决方案，使得查询效率得到了提升，尤其是在空值查询时能够提前失败，给用户响应。具体的内容将分为特定路径查询场景介绍、构建BloomFilter、基于BloomFilter查询等部分来进行详细地介绍。

## 1.4 组织结构

本文的组织结构大致如下：

第一章,描述了本文的研究背景与研究意义。

第二章,为本文的相关技术基础与背景知识给出了一个大致的介绍。主要涉及到IFC相关知识、全文检索Lucene和弹性搜索框架ElasticSearch、图数据和Neo4j图数据库等。

第三章，介绍了系统的总体架构。该系统结合ElasticSearch搜索框架和Neo4j图数据库，从IFC文件解析、数据模式映射、数据存储和查询等模块进行综述。

第四章,介绍了本文如何利用ElasticSearch处理和存储IFC实体的属性信息。以数据处理流程的先后顺序展开介绍，主要涉及到IFC属性和属性集的详细内容、IFC文件解析、JSON数据映射的定义和数据生成、索引过程、查询技术等部分。

第五章,介绍了本文如何利用Neo4j处理和存储IFC实体空间关系数据。以数据处理流程的先后顺序展开介绍，主要涉及到IFC空间关系数据模型的详细内容、图数据模型、存储技术、查询技术以及基于BloomFilter的查询优化等。

第六章,对本文提出的方案进行了实验验证，证明了本文方法的可行性和有效性。

# 第二章 相关技术基础

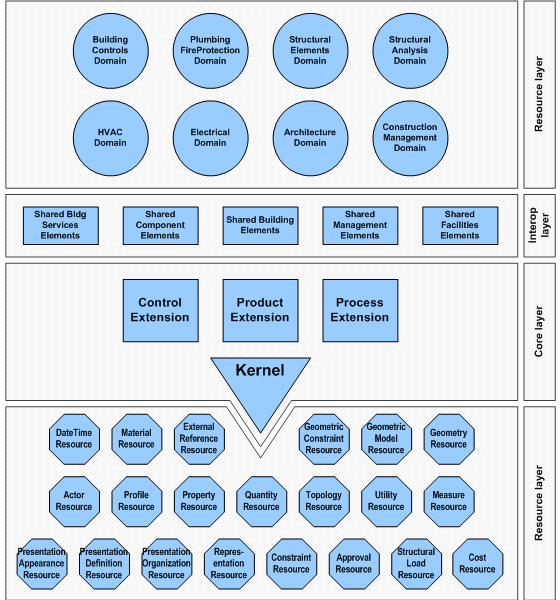
## 2.1 IFC相关技术

### 2.1.1 IFC标准

IFC是由buildingSMART以工业的产品资料交换标准STEP编号ISO-10303-11的产品模型信息描述,用EXPERSS语言为基础，基于BIM中AEC/FM相关领域信息交流所指定的资料标准格式。有专家认为IFC如同网络通信标准HTML一样，IFC不属于任何BIM软件专有，而加入IFC标准认证的各领域及不同软件也日益增加。许多公司或教育单位也加入研究并开发相应的应用，同时提供免费试用源代码，以此吸引更多人参与IFC的研究与发展。基于BIM的IFC标准已经发展10年有余，渐渐受到学术界与业界重视，IFC不断发展会是AEC相关信息交换的重要标准。

IFC标准同样也是一个类似面向对象的建筑数据模型。IFC 模型包括建筑整个生命周期内的各方面的信息，其中包含的信息量非常大而且涵盖面很广。IFC标准的目的是支持用于建筑的设计、施工和运行的各种特定的软件的协同工作。正因为如此，IFC标准是目前对建筑物信息描述最全面、最详细的规范。这证明了IFC 模型是建筑工业和设备制造工业之间的数据模型交换的最好方法。

为此，IFC标准的开发人员充分地应用了面向对象分析和设计方法，并设计了一个总体框架和若干原则将这些信息包容进来并加以很好地组织，这就形成了IFC的整体框架。IFC的总体框架是分层和模块化的，整体可分为四个层次，从下到上依次为资源层、核心层、共享层、领域层。每个层次内又包含若干模块，每个模块内又包含了不少信息。图1是IFC模型总体结构图。



图一，IFC模型总体结构图

其中：

1. 信息资源层（Resource layer）描述标准中用到的基本信息，如几何、尺寸、材料等基本元素信息，是整个信息模型的基础。这些信息可与其上层（核心层、共享层和领域层）的实体连接，用于定义上层实体的特性。
2. 核心层（Core Layer）定义了建筑工程信息的整体框架。这个层次提炼了一些适用于整个建筑行业的抽象概念，不仅包括建筑对象的位置和几何形状等，同时也定义了建筑对象之间的关系，其可以反映现实世界的结构。
3. 共享层（Interoperability layer）分类定义了一些适用于建筑项目各领域（如建筑设计、施工管理、设备管理等）的通用概念，以实现不同领域间的信息交换。
4. 领域层（Domain Layer）：定义了一个建筑项目不同领域（如建筑、结构、暖通、设备管理等）特有的概念和信息实体，形成领域内的专题信息。

### 2.1.2 IFC文件格式

IFC标准采用了EXPRESS语言作为描述语言，来描述IFC模型信息。一个完整的IFC模型包括类型定义、函数、规则、及预定义属性集组成。其中，类型定义是IFC模型的主要组成部分。类型定义分为定义类型（Defined Types）、枚举类型（Enumeration）、选择类型（Select Types）和实体类型（Entities）。IFC标准的当前版本（IFC 2X3）中包含了117个定义类型、164个枚举类型、46个选择类型和653个实体类型。实体采用面向对象的方式构建，比如门、窗，都是一个实体的实例。而定义类型、枚举类型、选择类型通常作为属性值出现在实体的实例中。下面这个例子中展示的是IfcDoor实体类型的EXPRESS定义。



首行“Entity”表示是实体类型的定义。第二行表示它是IfcBuildingElement类型的子类。由于实体是采用面向对象的方式构建的，当然具有继承的特性。第三行和第四行则定义了两个属性OverallHeight（高度）和OverallWidth（宽度）。“OPTIONAL”表示这个属性是可选的，“IfcPositiveLengthMeasure”是一个定义类型（Defined type），它用来表示OverallHeight（高度）和OverallWidth（宽度）这个属性的值是大于0的长度测量值。但是IfcDoor的属性绝不是只有这两个，它也继承了IfcBuildingElement的属性，同时IfcBuildingElemnt也继承了更上一层父类的属性。所以，IfcDoor的属性是自身的定义的属性与所有父类的属性的集合，且严格按照顺序。

那么一个IFC实体的实例的信息是如何在文件中存储的呢？这就要引出IFC SPF文件。

IFC SPF(STEP Physical File)文件是IFC标准所使用的主要数据交换文件。其内容紧凑，无冗余信息，大大地减少了数据存储占用空间。一个完整的IFC SPF文件包含两个部分：

1. 标头段，包含一些文件信息。
2. 数据段，包含IFC整个模型的信息。





上图中的解释是：

1. 行1是文件开始语句，表示ISO发布的STEP标准。对应文件最后一行（第92行）的END-ISO-10303-21。
2. 行2-6，表示文件信息，以HEADER关键字开头，以ENDSEC关键字结束。包括文件描述、文件名称、文件使用的IFC标准。
3. 行7-92是数据段。以DATA关键字开头，以ENDSEC关键字结尾。中间的每一行表示一个IFC实体的实例数据。#后面的数字是一个唯一的编号，可以不连续，而且不一定是增序，标识一个IFC实体的实例。等号右端首先是IFC实体的类型名，括号内是按照EXPRESS定义的包括继承属性的所有属性的属性值，值的类型可能是定义类型、选择类型、枚举类型。

## 2.2 全文检索技术

### 2.2.1 全文检索基础理论

全文检索技术，就是以数据诸如文字，声音，图像等为主要内容，以检索文献资料的内容而不是外表特征的一种检索技术，全文检索是信息检索技术的一种，主要是把用户的查询请求和全文中的每一个词进行比较，不考虑查询请求与文本语义上的匹配。在信息检索工具中，全文检索是最具通用性和实用性的。



图2 全文搜索一般流程

全文搜索的一般流程如同上图2所示。首先，从各类文档和数据库当中抽取数据，经过文本分析以后，将多种异构数据转化成同构的结构化或者半结构化数据，创建全文索引，建立索引数据库。目前应用最广的全文索引就是以倒排索引为核心的一系列的技术。当索引数据库建立完全以后，人们就可以通过查询接口查询文档。

由图中可以看出，全文检索系统中最核心、最关键的部分是全文检索引擎部分，这部分从功能模块上可以划分为文本分析模块、创建索引模块、查询索引模块。索引的准备工作和搜索的应用都是建立在这个引擎之上。由此可见，一个全文检索应用的优异程度，根本上是由全文检索引擎来决定。因此提升全文检索引擎的效率即是我们提升全文检索应用效率的根本。目前应用最广泛的数据库就是Lucence以及其一系的ElasticSearch与Solr。

### 2.2.1 Lucence概述

Lucene是非常优秀的成熟的开源的免费的纯Java语言的全文索引检索工具包。Lucene是一个高性能、可伸缩的信息搜索库（Information Retrieval Library）。它使你可以为你的应用程序添加索引和搜索能力。Lucene的作者Doug Cutting是资深的全文索引/检索专家。他最开始将此项目发布在他本人的主页上，2001年10月贡献给APACHE，成为APACHE基金的一个子项目。

Lucene作为一个全文检索引擎，其具有如下突出的优点：

1. 索引文件格式独立于应用平台。Lucene定义了一套以8位字节为基础的索引文件格式，使得兼容系统或者不同平台的应用能够共享建立的索引文件。
2. 在传统全文检索引擎的倒排索引的基础上，实现了分块索引，能够针对新的文件建立小文件索引，提升索引速度。然后通过与原有索引的合并，达到优化的目的。
3. 优秀的面向对象的系统架构，使得对于Lucene扩展的学习难度降低，方便扩充新功能。
4. 设计了独立于语言和文件格式的文本分析接口，索引器通过接受Token流完成索引文件的创立，用户扩展新的语言和文件格式，只需要实现文本分析的接口。
5. 已经默认实现了一套强大的查询引擎，用户无需自己编写代码即使系统可获得强大的查询能力，Lucene的查询实现中默认实现了布尔操作、模糊查询（Fuzzy Search）、分组查询等等。

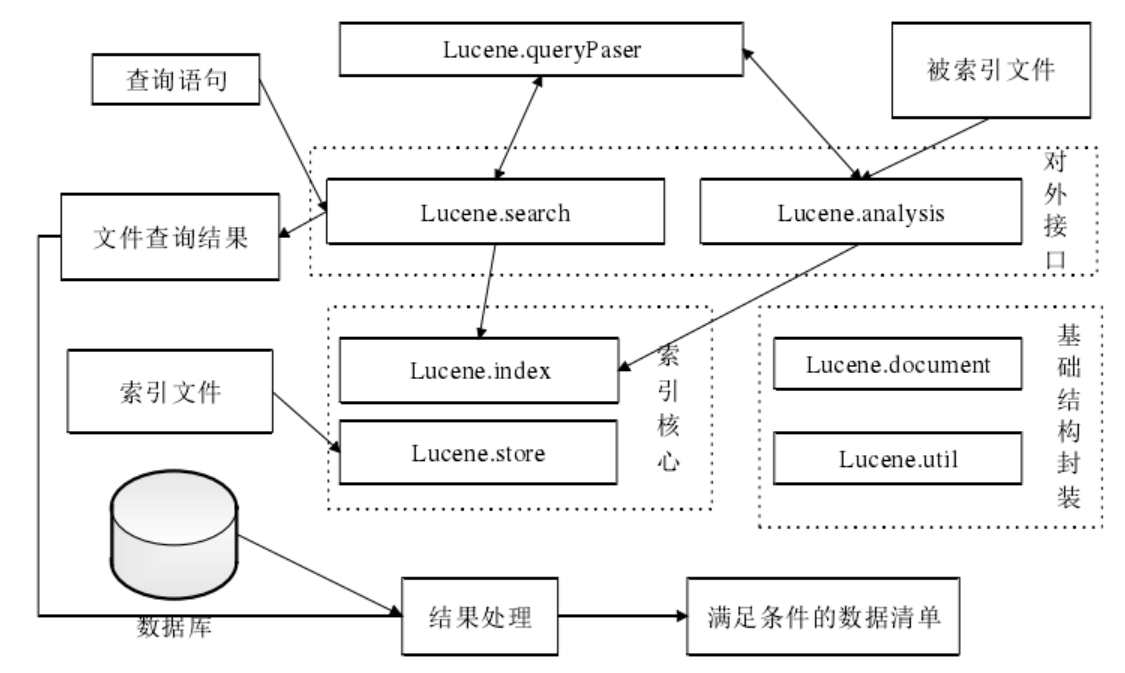


图3 lucence代码结构

Lucene的代码结构大致如上图所示，主要分为索引建立，与文档查询两个主要流程。首先，在索引建立的过程中，先将需要被索引的文件通过分析器分析好以后建立好倒排索引，放入索引核心的索引数据库当中。而当用户发起一个查询流程的时候，通过查询，将通过查询器，查询索引结果，通过查询语句的相应逻辑给出查询结果。

### 2.2.2 ElasticSearch技术

ElasticSearch 是分布式实时搜索、实时分析，实时存储引擎，简称（es)。ElasticSearch使用Java开发并使用Lucene作为其核心来实现所有索引和搜索的功能，通过简单RESTful API来隐藏Lucene的复杂性，从而让全文搜索变得简单。它的检索效率超出你的想象，从10亿的数据查出一条只需要1-2秒内。此外，ElasticSearch还具有支持实时分析 、实时搜索 、可分布、可扩展到上百台PB机器的特性。

下面主要来介绍一些ElasticSearch中的一些相关术语：

1. 索引（index）：一个索引就是一个拥有几分相似特征的文档的集合。比如说，你可以有一个客户数据的索引，一个产品目录的索引，还有一个订单数据的索引。一个索引由一个名字来标识（必须全部是小写字母），并且当我们要对这个索引中的文档进行索引、搜索、更新和删除的时候，都要使用到这个名字。
2. 类型（type）：在一个索引中，你可以定义一种或多种类型。一个类型是你的索引的一个逻辑上的分类/分区。
3. 文档（document）：一个文档是一个可被索引的基础信息单元。比如，你可以拥有某一个客户的文档，某一个产品的一个文档，当然，也可以拥有某个订单的一个文档。文档以JSON（JavaScript Object Notation）格式来表示。如下图4所示。

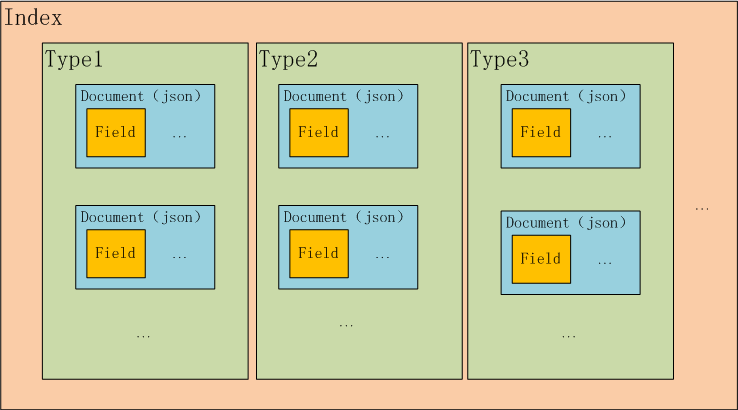


图4 ElasticSearch存储逻辑

1. 分片(shard)：一个索引保存了大量的文档数据，那这些数据是如何存储的呢？其实这是把很多数据，分布的放在每个分片中，分片又被放到集群中的各个机器上。每个分片都独立的“索引”（即可以增加、删除，修改、查询）。这样做的好处有两点:1.横向扩展，水平分割数据容量;2.可以在分片上并行的进行操作。如图5所示，每个索引的分片均匀的分布到了整个集群当中，同时每个分片还有一个副本，保证数据的高可用性。

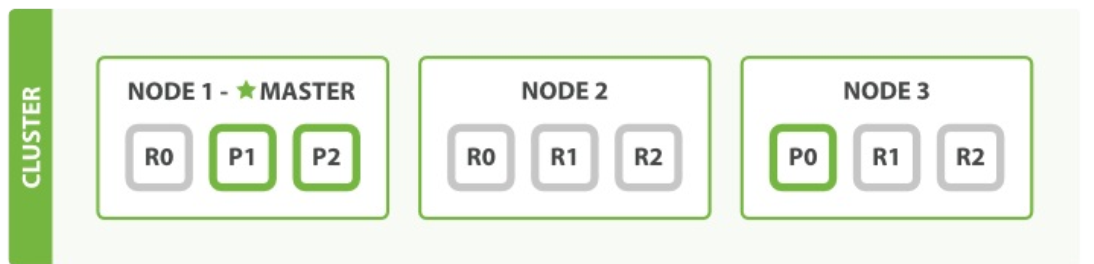


图5 ElasticSearch集群逻辑图

另外，ElasticSearch支持REST API对索引进行CRUD（增删改查）操作。curl工具是一种可以在命令行访问url的工具，支持get和post请求方式。-X指定http请求的方法，-d指定要传输的数据。要使用curl访问ElasticSearch索引的数据，其语法格式如下：



其中，<REST Verb>是REST风格的语法谓词；<Node>指定集群中的某个节点的IP；<port>指定节点端口号，默认9200；<Index>指定索引名；<Type>指定文档类型；<ID>指定文档的ID。

所以ElasticSearch为Lucene提供了一种高效的分布式数据存储方案，提供了高效的数据吞吐能力，以及处理大数据的能力。同时REST API使得整个集群更加易用，为开发者提供了快速开发应用的条件。

## 2.3 图数据库技术

### 2.3.1 图数据与图数据库

图数据库源起欧拉和图理论，也可称为面向/基于图的数据库，对应的英文是Graph Database。图数据库的基本含义是以“图”这种数据结构存储和查询数据，而不是存储图片的数据库。它的数据模型主要是以节点和关系（边）来体现，也可处理键值对。它的优点是快速解决复杂的关系问题。

图具有如下特征： 包含节点和边；节点上有属性（键值对）；边有名字和方向，并总是有一个开始节点和一个结束节点；边也可以有属性。图可以说是顶点和边的集合，或者说更简单一点儿，图就是一些节点和关联这些节点的联系（relationship）的集合。图将实体表现为节点，实体与其他实体连接的方式表现为联系。

图将实体表现为节点，实体与其他实体连接的方式表现为联系。我们可以用这个通用的、富有表现力的结构来建模各种场景，从宇宙火箭的建造到道路系统，从食物的供应链及原产地追踪到人们的病历，甚至更多其他的场景。

通常，在图计算中，基本的数据结构表达就是：

图数据库存储一些顶点、边与表中的数据。它们用最有效的方法来寻找数据项之间、模式之间的关系，或多个数据项之间的相互作用。

一张图里数据记录在节点，或包括的属性里面。最简单的图是单节点的，记录了一些属性。一个节点可以从单属性开始，成长为成千上亿，虽然会有一点麻烦。从某种意义上讲，将数据用关系连接起来分布到不同节点上才是有意义的。

图计算是在实际应用中比较常见的计算类别，当数据规模大到一定程度时，如何对其进行高效计算即成为迫切需要解决的问题。大规模图数据，例如支付宝的关联图，仅好友关系已经形成超过1600亿节点、4000亿边的巨型图。要处理如此规模的图数据，传统的单机处理方式显然已经无能为力，必须采用由大规模机器集群构成的并行图数据库。

在处理图数据时，其内部存储结构往往采用邻接矩阵或邻接表的方式，图4就是这两种存储方式的简单例子。在大规模并行图数据库场景下，邻接表的方式更加常用，大部分图数据库和处理框架都采用了这一存储结构。

随着数据量的不断变大，图数据库也变成了大数据时代的研究热点。

### 2.3.2 Neo4j概述

Neo4j是基于Java开发的高性能的、高可靠的、高可扩展的NOSQL图形数据库。Neo4j支持所有的图数据变更操作放在事务中处理，确保数据的一致性。Neo4j单个服务器实例可以应对有着大量节点和关系的复杂图数据模型，处理能力可达到数十亿级别。此外，Neo4j还提供了非常快的图算法，如图遍历、最短路径等。Neo4j可以作为嵌入式的图形引擎用于各种需要快速开发的图应用当中，其轻量、高性能的优势使其越来越受到关注。

下面分别介绍Neo4j的基本概念、查询语言和索引。

1. Node、Relationship、Property、Label

在Neo4j中，有两种基本数据类型，Node（节点）和Relationship（关系）。Node通过Relationship所定义的关系连接起来。同时可以在Node和Relationship上赋予key/value形式的属性，来表示详细的信息，以便对Node和Relationship进行查询。同一类型的节点还可以被赋予Label（标签）。Neo4j会根据Label将节点组织为不同的set（集合），基于Label来进行节点的查询将使得查询不再是面向整个图，而是面向某些set。Label是可选的，一个节点可以没有Label。同时，一个节点也可以有多个Label。通过节点、关系属性和标签可以构建出一个大型的图结构，再通过一些列的图操作来进行数据的管理和应用。这就是Neo4j的图数据模型。下图是一个简单的图模型：



图7 简单图模型

图7是一个含有3个节点的图模型,两个带有“Person”标签的节点，分别有两个属性，“name”和“born”。还有一个带有“Movie”标签的节点，有两个属性“title”和“released”。“Tom Hanks”节点（我们暂且以属性值标识这个节点）与“Forrest Gump”有一个“ACTED\_IN”的关系，且关系具有属性“roles”，其语义是Tom Hanks出演了电影Forrest Gump，出演的角色是“Forrest”。另一个关系“DIRECTED”，表示“Robert Zemeckis”执导了影片“Forrest Gump”。

Neo4j提供了图遍历的traverse API，以某一个节点为起始节点，可以采用深度优先或者广度优先的遍历算法遍历整个图中的节点。

1. Cypher查询语言

Neo4j提供了Cypher查询语言。Cypher是一种声明式的图查询语言，允许对图数据进行有效的查询和更新。Cypher是一个比较简单但语言非常强大的语言。非常复杂的数据库查询可以很容易地通过Cypher来表达，这很大程度上方便了开发者或用户对Neo4j做点对点模式（ad-hoc）的查询操作和其它数据库操作。Cypher参考了SQL的结构，使用各种子句建立查询。子句连接在一起，并且彼此之间提供中间的结果集。例如，一个MATCH子句的匹配结果是下一个子句的上下文。下面是部分查询子句：

MATCH：匹配模式，这是从图形中获取数据的最常用方法。

WHERE：过滤条件。

RETURN：返回所需要的。

基于图7的模型，如果要查找Tom Hanks演过的电影，Cypher语句可以写为如下形式：



需要注意的是，Cypher并不是一个静态的语言，不断发展的新版本会加入一些新的功能，有时候还会删除一些旧的功能。例如在Cypher 2.2之前的版本中，START子句被用来查找指定的ID的节点作为起始节点，但是在2.2及以后的版本中，START被完全禁止了，取而代之的是用MATCH子句来查找起始节点。每一个版本的Neo4j支持某些版本的Cypher，如Neo4j 3.2支持的Cypher版本有3.2、3.1、2.3。当升级Neo4j版本时，需要注意查看相应的Cypher版本是否支持。

1. Legacy Index和Schema Index

Neo4j提供了属性上的索引，使得用户可以通过属性值快速找到节点。索引包含Legacy Index 和Schema Index两种类型，它们都是基于Lucene实现的在Neo4j 2.0版本之前的索引被称为Legacy Index。Legacy Index能够提供全文检索的能力，而在Schema Index当中没有。这也是Neo4j 2.0版本及以后的版本还保留着Legacy Index的原因之一。使用Legacy Index往往需要一个“起始节点”，使用Cypher语句时需要借助START子句来调用。

在Neo4j 2.0 及以后的版本中Legacy Index不再是官方推荐的索引，而是推荐使用Schema Index。Schema Index需要基于Label和Property使用。它进行了一些优化，使得MATCH子句中查询节点变得更快。

创建Schema Index的Cypher语法如下：



其中的Label和Property分别对应具体的标签和属性。

# 第三章 系统总体架构

# 第四章 基于ElasticSearch的IFC对象属性数据的存储和检索研究

本文在绪论中已经介绍了IFC标准。在本章节将详细介绍IFC对象的属性数据。

## 4.1 IFC对象属性数据模型概述

### 4.1.1 属性

属性就是对事物以及概念描述的一个基本单位。在IFC标准中，除了IFC对象类型本身的各个字段可以称之为属性，IFC标准还定义了不同的属性类，为描述IFC对象提供了一种更加灵活的可扩展的方式。这些属性类均继承自抽象的基类IfcProperty。根据buildingsmart官网的文档， IfcProperty的定义如下图：

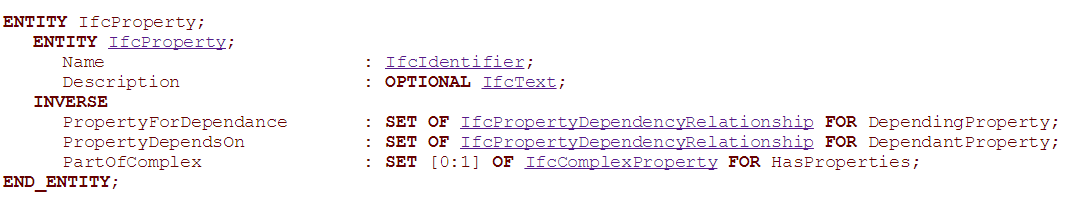


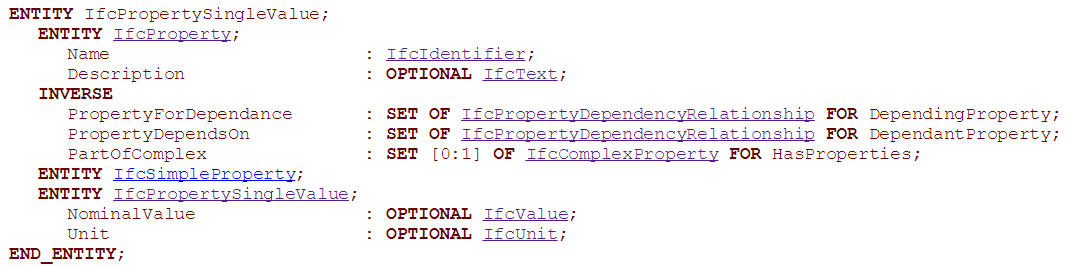
图1 IfcProperty定义

其中Name字段存储属性的名称，对应的值类型是IfcIdentifier，Description存储属性的说明，对应的值类型是IfcText。IfcProperty是抽象基类，它的派生子类如下图：

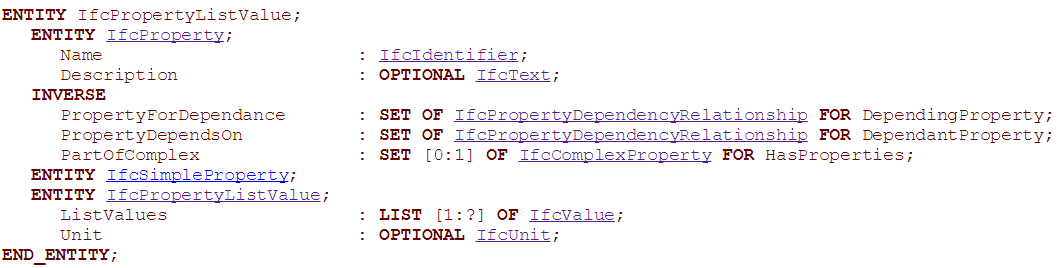


图2 IfcProperty派生子类图

其中，IfcSimpleProperty是简单属性。根据属性值类型的不同又有6个派生子类。比如，IfcPropertySingleValue面向单值属性，它的定义是：

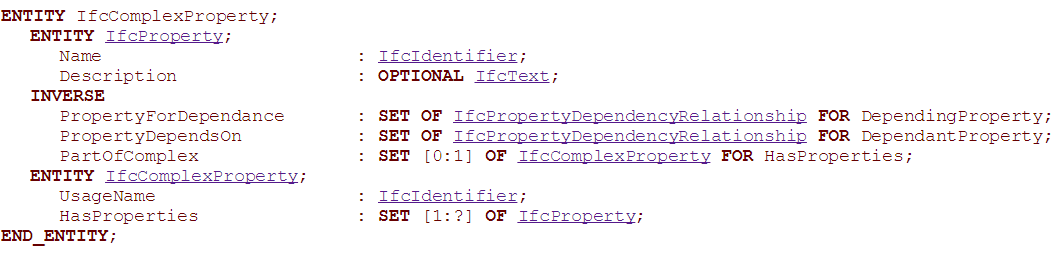


其中的NormalValue就是Name属性字段所对应的属性值。又比如IfcPropertyListValue，面向具有列表值类型的属性，它的定义是：



其中的ListValue就是Name字段所对应的属性值，是一个列表类型。其它如IfcPropertyBoundedValue、IfcPropertyEnumeratedValue、IfcPropertyReferenceValue、IfcPropertyTableValue分别面向上下边界区间类型、枚举类型、实体引用类型、二维表格结构类型的属性，再次不再一一介绍。

相对于IfcSimpleProperty的简单属性类型而言，IfcComplexProperty是复杂属性。它的定义如下图：



其中HasProperties存储了IfcProperty的集合，而UsageName是使用说明。IfcComplexProperty是可以嵌套的，集合中包含的IfcProperty也可以是一个IfcComplexProperty。

### 4.1.2 属性集

IFC标准用属性集来组织对象的属性。所有的属性集都继承自抽象基类IfcPropertySetDefinition。它的派生子类图如下：

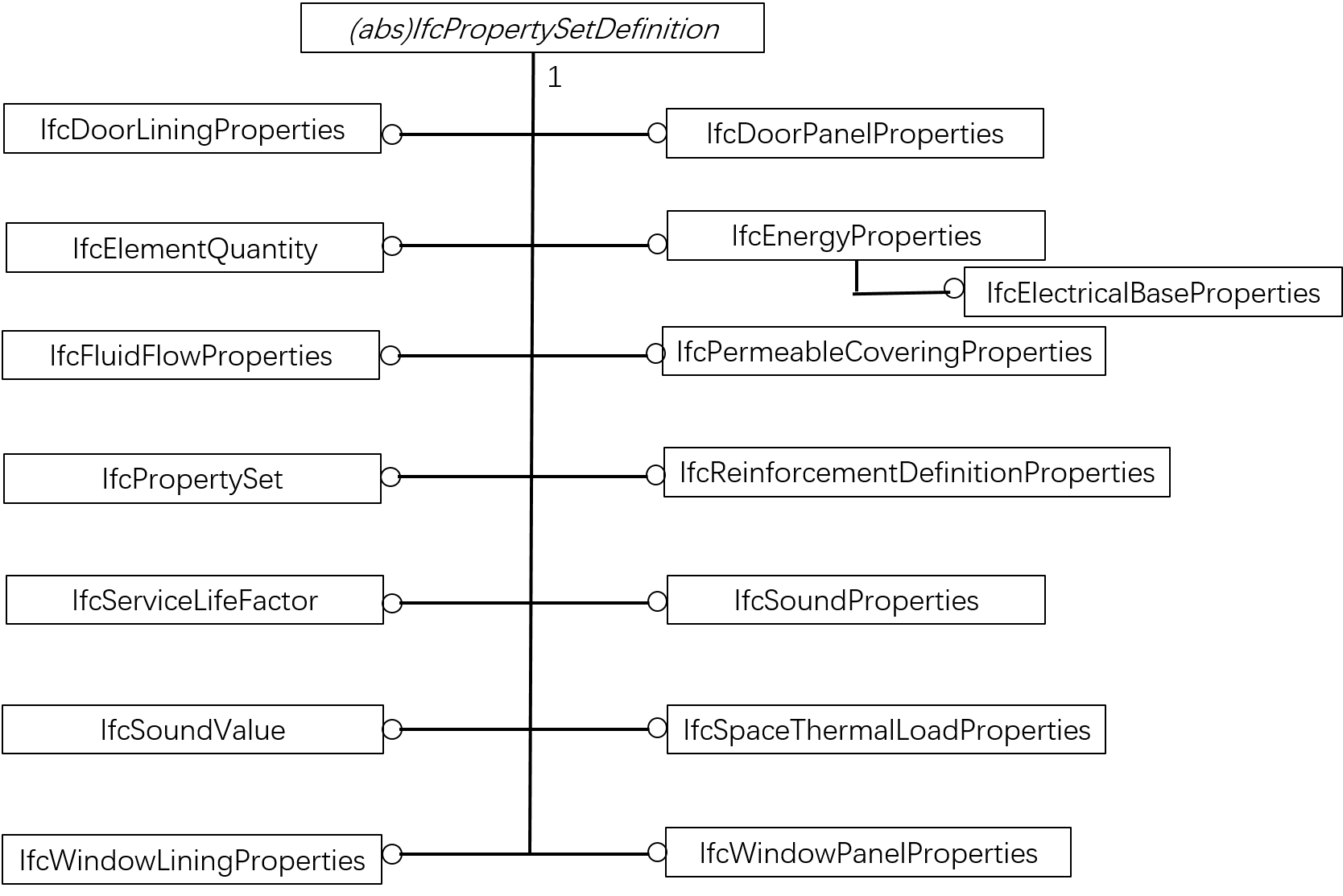
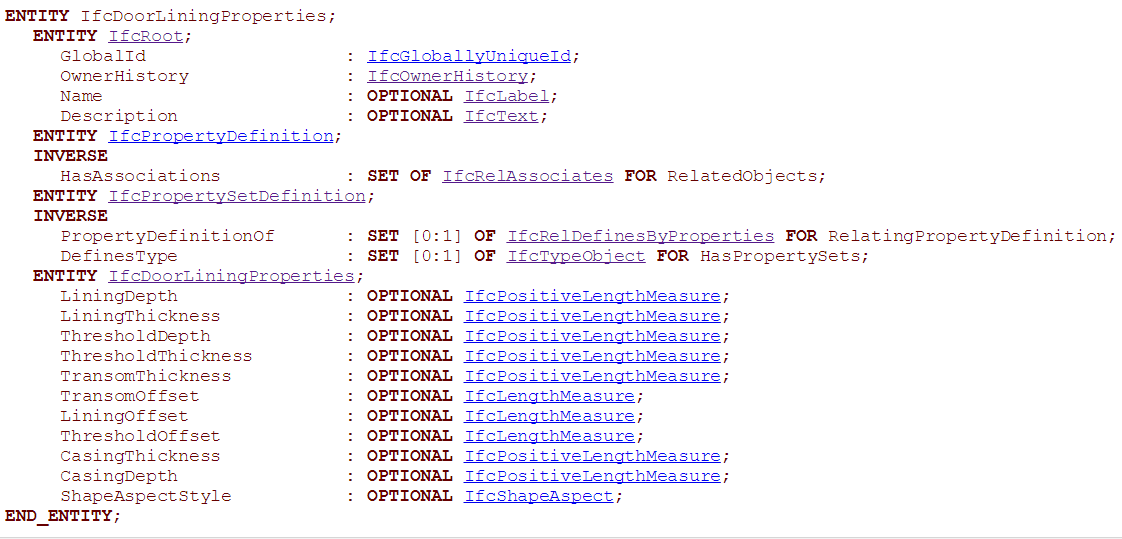


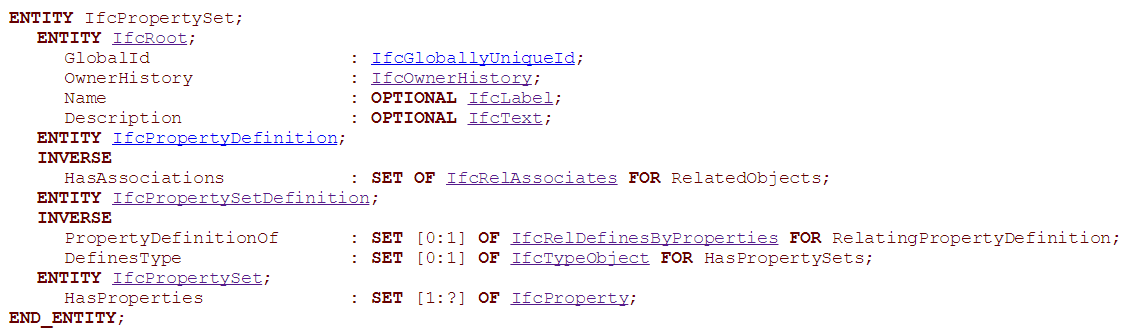
图3 IfcPropertySetDefinition派生类图

所有的这些属性集分为静态属性集和动态属性集。静态属性集往往面向特定的IFC实体类型，如IfcDoorLiningProperties应用于IfcDoorStyle类型，IfcWindowLiningProperties应用于IfcWindowStyle类型。举例IfcDoorLiningProperties的定义如下：



可以看到它包含了一系列的属性如LiningDepth、LiningThickness等来描述门框的信息。所以它是面向IfcDoorStyle的静态属性集。

事实上，以上派生类中除了IfcPropertySet之外都是静态属性集，在此不做一一介绍。IfcPropertySet是动态属性集。它的定义是：



其中HasProperties存储了一个IfcPropertySet的集合。

不要混淆了IfcComplexProperty与IfcPropertySet。前者作为IfcProperty的子类，属于属性范畴，后者是属性集。在IFC标准中需要通过IfcRelDefinesByProperties关系类型来将IFC对象类型和IfcPropertySet关联。也就是说，而IfcComplexProperty必须包含在一个属性集IfcPropertySet中才能关联到IFC对象。

### 4.1.3 属性集与IFC对象关联

在IFC标准中通过IfcRelDefinesByProperties关系实体将IfcPropertySetDefinition描述的信息与IFC对象关联。这种关联是一对多的映射，多个具有相同属性集的IFC对象可以通过同一个IfcRelDefinesByProperties关联到一个属性集。当然，一个IFC对象可以通过多个IfcRelDefinesByProperties来关联到不同的属性集。如图3所示：

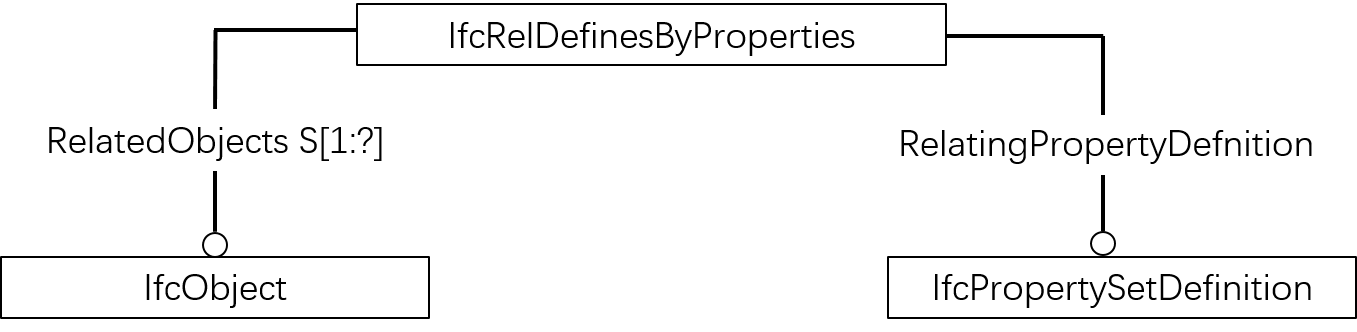


图3 IfcRelDefinesByProperties关联机制

需要举一个具体的例子。

以上便是IFC对象属性数据模型的介绍。对于如何将属性数据存储狭隘，接下来将会讲述。

## 4.2 数据提取技术

本小节将详细介绍从解析IFC文件到内存中的JAVA对象，到ElasticSearch所支持的JSON数据的过程。

### 4.2.1 IFC文件解析

本文在2.1小节中对比了IFC模型解析工具IFC Java ToolBox和BimServer解析模块，最后选择前者来进行IFC文件的解析。

IFC Java ToolBox主要有三个组成部分：

1. 针对IFC实体类型建立的Java类；
2. IFC文件解析器，将文件内容解析为内存中的Java类型；
3. IFC对象模型IfcModel，提供所有IFC对象的访问；

原生的IFC Java ToolBox API并不能访问属性字段，所以我们需要定义一个attributes数组，存储每个对象类型的属性名，并通过Java反射机制来进行属性字段的访问。

举例分析如下：

下面以IfcWallStandardCase实例来进行JSON映射的介绍：

#155=IFCWALLSTANDARDCASE('0o\_qQXI0r5UQtbU$t93HI7',#41,'\X2\57FA672C5899\X0\:CW 102-50-215p:309591',$,'\X2\57FA672C5899\X0\:CW 102-50-215p:412',#124,#151,'309591');

#264=IFCPROPERTYSINGLEVALUE('\X2\65CF\X0\',$,IFCLABEL('\X2\57FA672C5899\X0\: CW 102-50-215p'),$);

#265=IFCPROPERTYSINGLEVALUE('\X2\65CF4E0E7C7B578B\X0\',$,IFCLABEL('\X2\57FA672C5899\X0\: CW 102-50-215p'),$);

#266=IFCPROPERTYSINGLEVALUE('\X2\7C7B522B\X0\',$,IFCLABEL('\X2\5899\X0\'),$);

#267=IFCPROPERTYSINGLEVALUE('\X2\7C7B578B\X0\',$,IFCLABEL('\X2\57FA672C5899\X0\: CW 102-50-215p'),$);

#268= IFCPROPERTYSINGLEVALUE('\X2\7C7B578B\X0\ ID',$,IFCLABEL('\X2\

57FA672C5899\X0\: CW 102-50-215p'),$);

#284=IFCPROPERTYSET('1jxqaKVd18p9OAJYxrc1qw',#41,'\X2\51764ED6\X0\',$,(#264,#265,#266,#267,#268));

#289=IFCRELDEFINESBYPROPERTIES('1j9gkIXEf1yvBd$sOSwe8K',#41,$,$,(#155),#284);

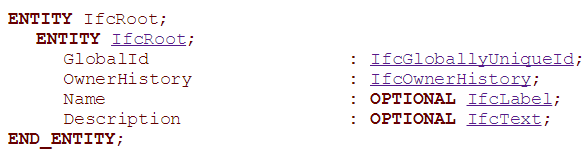
#155是IfcWallStandardCase对象。#264、#265、#266、#267、#268是四个IfcPropertySingleValue对象，分别表示单值属性。#284是IfcPropertySet对象，也就是一个动态属性集，根据IFC Schema定义，它的HasProperties字段存储了所包含的IfcProperty，在这里就是（#264,#265,#266,#267,#268）。#289是IfcRelDefinesByProperties对象，根据IFC Schema定义，RelatingPropertyDefinition表示一个属性集，在这里是#284，而RelatedObjects表示属性集关联到的IfcObject集合，在这里是只包含了一个元素（#155）的集合。

对这样的一个IFC实体，结合其自身Schema定义和属性集数据，将映射为下面的JSON格式：是不是可以将真是数据直接拿过来的？？

### 4.2.2 JSON数据映射

经过解析后的IFC对象数据需要转换为JSON数据，才能够通过Restful API被ElasticSearch索引。我们需要定义这个映射过程。

前面介绍过IFC标准，其分为四个功能层次，资源层、核心层、交互层、领域层。其中IfcRoot是核心层及以上层次中全部实体类型的抽象基类型。IfcRoot的定义如下：



其中GlobalId字段是一个全局唯一的GUID值。IfcRoot的所有派生类因为有了这个全局标识属性而具有了信息交换过程中的独立性。相比之下，资源层中的实体类型如IfcAddress（描述地址的实体），没有GlobalId属性，不能独立用于信息交换。而用户更为关系的是这些可用于交换的实体类型。IfcRoot的部分派生类如图4：

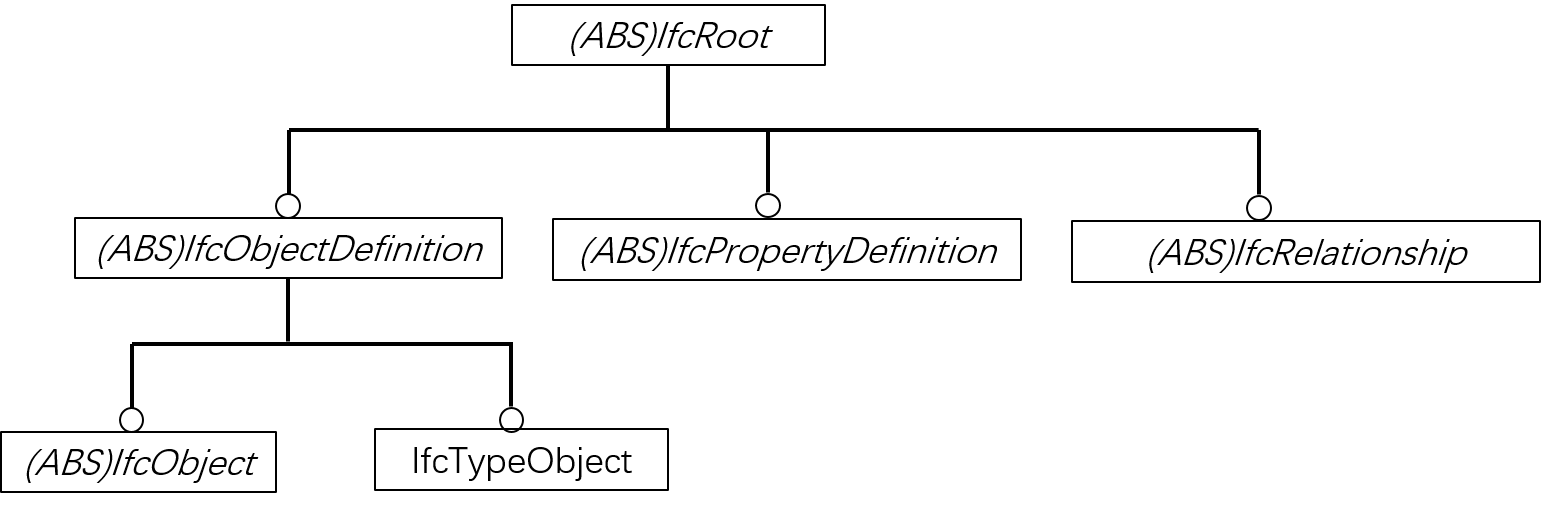


图4 IfcRoot部分派生类图

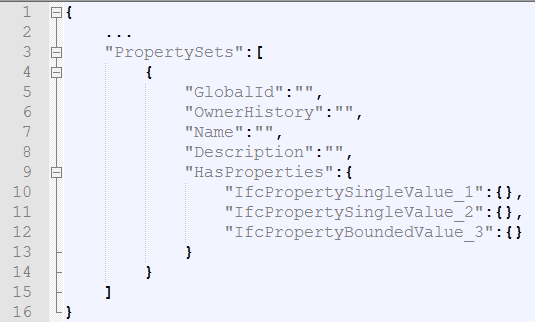
可以看到IfcRoot的直接派生类为IfcObjectDefinition、IfcPropertyDefinition、IfcRelationship三个类型。而3.1.3小节中讲到的IfcRelDefinesByProperties就是IfcRelationship的一个派生类，它将IfcObject类型与IfcPropertySetDefinition定义的属性集关联起来。由此，也就引出了本文ElasticSearch属性数据存储的重要研究对象：IfcObject。

本文的设计是将IfcObject的属性集数据集成到IfcObject本身，实现属性集的高效访问。而对于其它的IFC实体，按照IFC Schema定义中的字段一一映射为JSON格式。

设计的难点在于一个IfcObject可以关联多个IfcPropertySet实例，一个IfcPropertySet实例又关联了多个IfcProperty的派生类实例，而IfcProperty的派生类有6种，如3.1.1小节介绍，它们之间存在差异。

可能的设计方案：

1. 如果采用JSON对象嵌套的方法，如下：



其中行1表示省略IfcObject自身定义的字段，行3表示IfcPropertySet数组，行4-14表示一个IfcPropertySet对象，行9表示IfcPropertySet所关联的多个IfcProperty。这种方式能够完整地保留IfcPropertySet和IfcProperty的每一个字段。但是它的局限在于，需要为每一个IfcProperty实例命名，这个命名必须是数组内唯一的，这是非常困难的。另外，通过字段名访问数据也变得困难。

1. 第二种方案，侧重考虑用户对于字段的访问的便利。由于IfcPropertySet

的Name字段往往是用户关心的，所以可以把Name字段的值作为JSON中的字段名。对于IfcProperty也是一样，把Name字段的值作为JSON的字段名。如下图：



其中，“标识数据”是IfcPropertySet的Name字段的值，标识属性集名称，行4-6是IfcPropertySet的字段。“标高”是IfcPropertySingleValue的Name字段的值，表示属性名称，后面对应属性值，“偏移量”也是一样。这种方案使得用户能够根据日常知识方便地访问想要的数据。

但是它有一个问题是，Name字段不具有唯一性，也就是说可能有多个IfcPropertySet属性集名称都为标识数据。例如，属性集A，名称为“标识数据”，包含属性“标高”“偏移量”，属性集B，名称为“标识数据”，包含属性“类型名称”“部件代码”。在这种情况下希望得到的处理结果是，属性集A和属性集B的四个属性合并为一个属性集。如果是用第二种方案，那么会出现，行4、5、6的数据会因为重叠被覆盖掉。

经过对需求的考虑认为，GlobalId、Description、OwnerHistory在用户的查询中并无重要意义，所以去掉。

基于第二种方案，加上进行扩展，可以得到JSON格式如下：



图5 属性数据JSON格式

其中，行11-28是另一个名称为“Pset\_XXXXCommon”的属性集，行11-14是一个IfcPropertyBoundedValue属性实例，行15-17是一个

IfcPropertyEnumeratedValue属性实例，行18-20是一个IfcPropertyListValue属性实例，行21-23是一个IfcPropertyReferenceValue实例，行24-27是一个IfcPropertyTableValue属性实例。对于所有的字段，如果是引用类型，则只存储引用对象的ID值。另外，如果是IfcPropertySet的Name字段没有值，则存入“Others”JSON字段中。

同样对于IfcTypeObject也是如此。这段介绍应该放在属性集关联那里去。

如果是IfcComplexProperty类型，则嵌套表示。

其代码逻辑是：

1. 遍历IfcModel中的每一个IFC对象；
2. 针对每一个IFC类型创建JSON文件，以类型名为文件名称。由于解析IFC文件是逐行进行，所以需要维护一个Map结构存储文件句柄。
3. 针对解析后的每一个IFC对象，判断是否是IfcTypeObject或IfcRelDefinesByProperties的子类实例，如果都不是，则根据attributes属性名数组获取属性名XXXX，利用Java的反射机制执行getXXXX方法名，获取属性值，从而形成JSON格式字符串写入相应的文件。
4. 如果是IfcTypeObject的子类实例，那么可以对于“HasProperties”字段将遍历每一个所关联到的IfcPropertySet子集，按照上面图5所设计的JSON格式来进行字段和字段值的输出，写入到对应的IfcTypeObject子类类型的文件中去。
5. 如果是IfcRelDefinesByProperties的子类实例，那么将解析出RelatedObjects（是IFC对象实例集合）和RelatingPropertyDefinition（是一个IfcPropertySet的子类实例），获取IfcPropertySet的属性数据，然后依次获取RelatedObjects中的IFC对象类型的文件句柄，将属性数据写入到文件中去。

其代码部分如下：

因为bulk机制，所以所有的记录都应该是这个样子的。

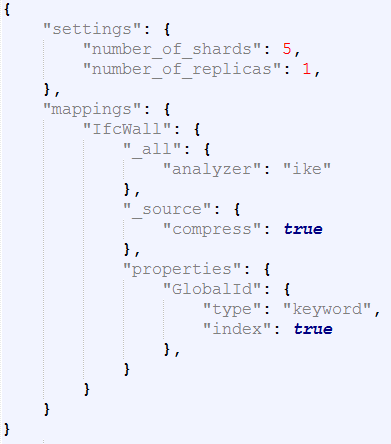
至此，已经将所有IFC对象的数据写入到了JSON文件。

支持中英文的问题。

## 4.3 数据索引技术

### 4.3.1数据模式映射

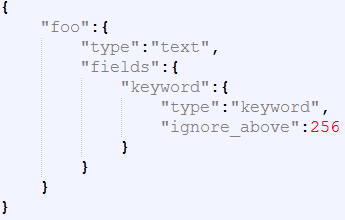
如果把ElasticSearch理解为一个数据管理平台，那么index就是数据库，type可以理解为表，mapping就是相关设置和表结构信息。下面是一个mapping示例：



可以将mapping信息大致分为settings和mappings两个部分，settings主要是作用于index的一些相关配置信息，如分片数、副本数等。mappings部分主要是主要是针对类型和字段的一些参数定义，其分为\_all、\_field\_names、\_id、\_index、\_meta、\_parent、\_routing、\_source、\_type、\_uid等元数据信息，和properties部分。Properties定义了索引字段的名称及其数据类型，类似于mysql中的表结构信息，例如“name”字段是String类型，“高度”字段是double类型等。Properties另外还定义了字段所使用的分词器、是否存储等信息。不过ElasticSearch的mapping比数据库灵活很多。当你索引一个包含新字段的文档时，ElasticSearch会使用动态映射机制。它可以根据数据格式动态识别它的数据类型，一般不需要指定mapping都可以。对于JSON中的基本数据类型，ElasticSearch会按照如下规则做映射：

|  |  |
| --- | --- |
| JSON类型 | ElasticSearch字段类型 |
| 布尔型：true/false | boolean |
| 整数：123 | long |
| 浮点数：123.45 | double |
| 字符串：日期格式2014-09-15 | Date |
| 字符串：foo bar | String |

关于String类型，需要特别说明，ElasticSearch 5.X之后的字段类型不再支持String，而是由text、keyword来取代。如果在手动定义mapping时仍然使用String，则会给出警告。如果是动态映射的话，ElasticSearch会为你创建下面的映射：



也就是说，字符串动态映射之后既有满足全文检索的text类型，又有满足关键字检索的keyword类型。

接下来主要讨论中文分词器。由于IFC文件中可能包含大量的中文，而且可能出现一个字段的值既有中文又有英文。而ElasticSearch默认的分析器只能支持英文所以需要ElasticSearch能够支持中文分词。

Es的默认分词器standard的中文分析是什么？中文单个？那英文呢？

<http://www.sojson.com/blog/82.html> （ik有默认词库）

Es的ik（中文分词器）是什么？那么对英文的分词怎么样？

Ik又无法满足英文分词的结果？所以需要进行设置。

<http://blog.csdn.net/hereiskxm/article/details/47441911>

各个分词器的性能还不一样：

<http://blog.csdn.net/xiaomin1991222/article/details/50981807>

如何处理JSON中转义字符的问题。

数据类型；

支持中英文的分词器介绍ik，jieba；安装；mapping设置分词器脚本与执行

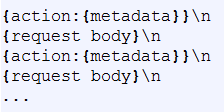
中文支持问题；

索引优化的策略有哪些涉及到了mapping

### 4.3.2索引过程

在将JSON数据导入到ElasticSearch的过程中，如果是一条一条的手动导入，效率低下。而使用bulk批量导入数据的机制，能够显著提高导入效率。ElasticSearch提供的bulk API允许一次请求执行批量的create、index、update或delete操作。

使用bulk命令时，REST API以\_bulk结尾，批量操作写在JSON文件中，JSON的语法格式是：



需要注意两点：每行一定要以换行符（\n）结尾，包括最后一样；这些行不能包含未转义的换行符，也就是说，这个JSON不能使用pretty参数打印。

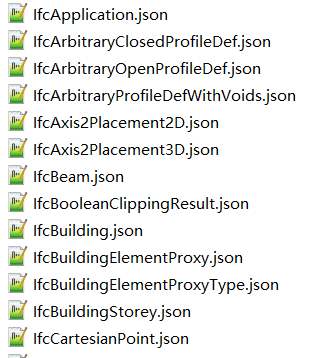
语法中action指定做什么操作，必须是create（创建新文档）、index（创建一个新文档或替换现有文档）、update（更新文档）、delete（删除文档）其中的一个。Metadada指定被创建、索引、更新或者删除的文档的\_index、\_type和\_id。例如在下面的例子中，索引一条IfcWall类型的JSON数据：



使用bulk的REST API的命令形式如下：



在IFC属性数据提取的章节中我们已经将JSON数据输出到以IFC类型名为文件名的各个JSON文件中，如下：



形成的JSON文件根据IFC文件中出现的不同的对象类型可能会有100多个到600多个，需要编写自动化脚本执行bulk命令。整个批量请求都需要由接受到请求的节点加载到内存中。该节点解析元数据（\_index、\_type、\_id），然后分发给其它的节点的分片，进行操作。对于每个节点来讲，该请求越大，其它请求所能获得的内存就越少。批量请求的大小有一个最佳值，大于这个值，性能将不再上升，甚至会下降。但是最佳值不是一个固定的值。它完全取决于硬件、文档的大小和复杂度、索引和搜索的负载的整体情况。ElasticSearch的官方文档建议，一个好的批量请求大小建议在5-15MB之间。同时，避免将bulk请求发送给单个节点，所以可以采用轮询机制访问所有节点。

采用轮询机制的bulk请求自动化脚本ifcbulk.sh如下：



### 4.3.3索引过程优化

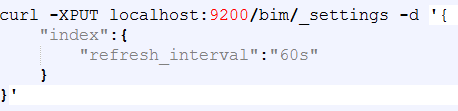
在上两小节已经对分词器和bulk对索引性能带来的影响进行了讨论。总结一下。当

ElasticSearch中配置文件elasticsearch.yml中有丰富的参数允许用户自由定义。下面将对一些影响到索引过程效率的参数进行介绍。

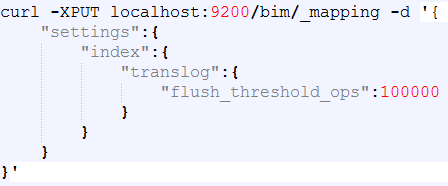
1. 提高ElasticSearch的内存。虽然理论上是分配的内存越大，ElasticSearch就有更多的内存空间可以提供更加高效的操作，但是建议是分配系统总的可用内存的50%。这样在做全文检索时，剩下的内存会被Lucene用作操作系统的文件系统缓存，从而加快全文检索。同时，分配的内存不要超过32G。这是因为，对于64位的系统而言，当内存小于32G时，JVM会采用一个内存对象指针压缩技术，使得内存指针依然使用32位来表示，大大减少内存占用。当内存大于32G时，不再使用指针压缩技术，而是采用64位的指针，这会造成内存的浪费。要修改ElasticSearch的内存分配，可以指定ES\_HEAP\_SIZE环境变量。服务进程在启动的时候会读取这个变量，并相应地设置堆大小，命令如下：



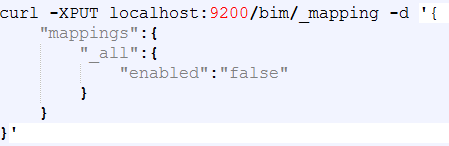
1. 刷新频率：index.refresh\_interval。刷新操作使得不同的分片之间的数据保持同步，默认的同步时间间隔是1s，同步的过程开销很大。在批量索引大量数据的过程中，可以将其修改的长一些，比如60s。完成索引之后，再修改为初始值。使用REST API动态配置的代码如下：



1. translog数据条数。每一个分片（shard）都有一个transaction log，它是一种Write Ahead Log。ElasticSearch在索引时中会将没有提交的数据记录在translog中。当进行flush操作的时候会将translog中的数据发送给Lucene做相关操作。默认的设置是数据达到5000条时进行一次flush操作。这个过程相对而言是比较浪费时间和资源的。所以我们可以将这个值调大。使用REST API动态配置的代码如下：



1. Mapping中的\_all域。Index中默认会有的\_all元数据，是为了支持对字段的查询。当不指定某个字段而是对所有字段进行检索时，会用到\_all这个元数据。但是它会增加索引时间。考虑到实际应用，如果没有这样的额查询需求的话，可以将\_all域禁止掉。使用REST API动态配置的代码如下：



1. 索引缓冲区大小。索引缓冲区用于存储新的索引文档，默认是JVM大小的10%。当缓冲区满之后，缓冲区的文件被写入磁盘上的一个段。增大缓冲区，可以使得一次写的效率更高。它的设置是静态的，并且必须在集群中的每个数据节点上配置。在elasticsearch.yml中的配置如下：



## 4.4 数据检索技术

ElasticSearch支持REST API对索引进行CRUD（增删改查）操作。curl工具是一种可以在命令行访问url的工具，支持get和post请求方式。结合ElasticSearch提供的\_search API可以方便地检索索引中的数据。

ElasticSearch对索引的检索分为结构化搜索和全文搜索。

1. 结构化搜索（Structured Search）：

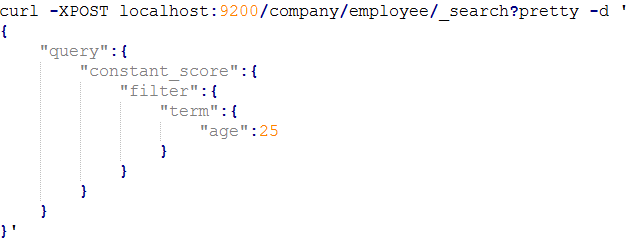
结构化检索用于对那些有精确的结构的数据的检索。比如日期、时间和数字。它们有着精确的格式，可以对这些数据进行逻辑上的操作，比如比较数字或时间的范围，或者判断两个值的大小。

结构化搜索的所得到的结果总是“是”或者“不是”，要么存在于集合之内，要么存在于集合之外。结构化搜索不关心文件的相关度或者评分。

对于这类查询，建议使用过滤器（filter）。过滤器的会跳过计算相关度的阶段，因此执行速度很快。同时，ElasticSearch会缓存filter过滤器的子句执行结果，在有大量查询时体现出性能上的优势。

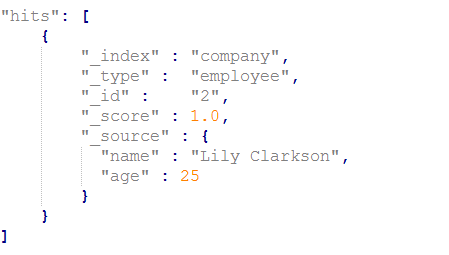
过滤器查询中合法的查询谓词有term（单精确值查询）、terms（多精确值查询）、bool（组合过滤器查询）、range（范围查询）、exists（存在查询）、missing（缺失查询）等。

举例：



其中，term查询是一个精确值查询，在这里要查询age为25的employee。使用filter过滤器来进行term查询将不会计算相关度得分，而外层的constant\_score则是表示以1作为统一的评分。

它的返回结果可能是这样的：



在应用当中，应尽量使用filter查询。

1. 全文检索（Full-text Search）

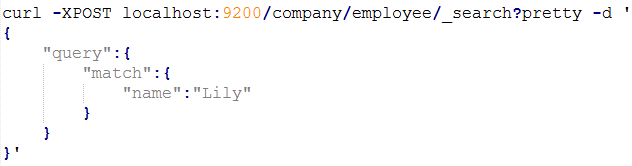
全文搜索的目标是在全文字段中搜索到最相关的文档。它包含两个重要的方面，相关性（Relevance）和分析（Analysis）。

相关性（Relevance）：它是评价查询与其结果间的相关程度，并根据这种相关程度对结果排名的一种能力。相关度计算采用 TF/IDF 算法。TF/IDF是指检索词频率/反向文档频率。检索词频率表示检索词在该字段出现的频率，出现频率越高，相关性也越高。 反向文档频率表示每个检索词在索引中出现的频率，频率越高，相关性越低。查询语句会为每个文档生成一个 \_score 字段来表示相关性计算结果。\_score评分越高，相关性越高。

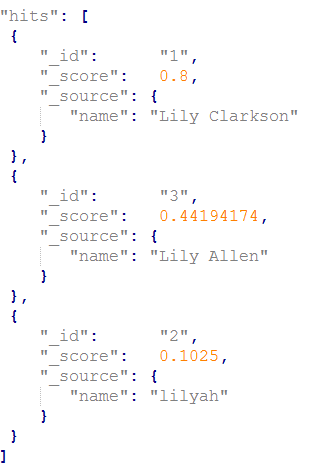
分析（Analysis）：它是将要索引的文本转换为有区别的、规范化的 token 的一个过程，目的是为了（a）创建倒排索引以及（b）查询倒排索引。当我们 索引一个文档时，它的文本被分析成词条以用来创建倒排索引。但是，当我们在字段搜索的时候，我们需要将查询字符串通过相同的分析过程，以保证我们搜索的词条格式与索引中的词条格式一致。

全文搜索比结构化搜索多了计算相关性的过程，同时搜索结果也不会被缓存。全文搜索的查询谓词有match\_all（所有文档的查询）、match（单字段查询）、multi\_match（多字段match查询）、bool（bool查询，有别于bool过滤器，bool查询会给每个bool子句进行评分）

举例：



其中，match查询作为对单字段的相似性查询，在这里是要找到与“Lily”相似的名字的employee。它的返回结果可能是这样的：



# 第五章 基于Neo4j的IFC对象空间关系数据的存储和查询研究

在绪论中本文对IFC标准进行了介绍。IFC标准的核心层中不仅定义了IFC对象的位置、几何形状等，同时也定义了IFC对象之间的关系。这些关系表现为空间结构上的组成、包含等多种关系，与IFC对象在物理世界中的关系一致。对此，本章将详细介绍IFC标准是如何定义IFC对象的空间关系的。

其次，当人们关心某一栋楼的某一层的某一面墙时，实际上指定了一个空间结构关系。为了满足用户对于IFC对象的空间关系的查询，本文采用了图数据库Neo4j来进行IFC对象的空间关系数据的存储和查询。具体的内容分为图数据模型设计、图数据提取技术、图数据存储技术和图数据查询技术4个部分来进行详细地介绍。

最后，本文基于BloomFilter技术提出了一种针对特定空间关系的查询优化，提高了查询效率，同时对于空值查询能够快速失败，提前给用户响应。

## 5.1 IFC对象空间关系数据模型

IFC标准划分为四个层次，包括资源层、核心层、共享层和领域层。资源层定义了基础信息，如几何、尺寸、材料等基本元素信息。核心层定义了IFC对象模型的整体框架，不仅包括IFC对象的位置和几何形状等，同时也定义了IFC对象之间的关系。这些关系表现为空间结构上的组成、包含等多种关系，与IFC对象在物理世界中的关系一致。例如一栋楼包含了许多楼层，而一层楼包含了许多建筑构件如墙、门、窗等。又比如一面墙上有两个空缺（OpeningElement），这个空缺对应了一面窗户等等。~~这些建筑构件对应IFC标准中的实体对象。而这些实体对象均由IfcRoot继承。~~

在IFC标准中，IfcRoot是核心层及以上层次中全部实体类型的抽象基类。IfcRoot的派生子类有三种类型，分别是IfcObjectDefinition、IfcPropertyDefinition和IfcRelationship。而IFC对象之间的空间关系就定义在IfcRelationship当中。

IfcRelationship的派生类有IfcRelAssigns、IfcRelAssociates、IfcRelConnects、IfcRelDecomposes和IfcRelDefines五类。这些类定义并非都是表示IFC对象之间的空间结构关系。它们的定义如下：

1. IfcRelAssigns及其派生类：表示Ifc实体（IfcObjectDefinition）与角色（IfcActor）、控制（IfcControl）、过程（IfcProcess）、资源（IfcResource）等之间的映射关系。
2. IfcRelAssociates及其派生类：表示Ifc实体（IfcObjectDefinition）与约束（IfcConstraint）、文档（IfcDocument）、库（IfcLibrary）、材料（IfcMaterial）等的映射关系。
3. IfcRelConnects及其派生类：表示Ifc实体对象（IfcObjectDefinition）之间基于某个标准的连接关系（connectivity）。例如，两面墙基于一条线相连接。
4. IfcRelDecomposes及其派生类：表示Ifc实体对象（IfcObjectDefinition）之间的组成关系（composition）。例如，一栋建筑有五层楼。
5. IfcRelDefines及其派生类：表示Ifc实体对象（IfcObjectDefinition）与属性类型（IfcProperty）、类型定义（Type）之间的映射关系。

根据这五类子类型的定义，只有IfcRelConnects和IfcRelDecomposes及其派生类涉及到Ifc对象的空间关系，因此在设计空间关系数据的存储模型时，只考虑这两类及其派生类。而其他类型的信息会存储在ElasticSearch当中以供检索。

下面将举一个例子来介绍Ifc实例对象之间的空间关系。

#94=IFCPROJECT('0DVPyye2z0deMtQa1pIata',#41,'\X2\987976EE7F1653F7\X0\',$,$,'\X2\987976EE540D79F0\X0\','\X2\987976EE72B66001\X0\',(#83,#91),#78);

#592=IFCSITE('0DVPyye2z0deMtQa1pIatc',#41,'Default',$,'',#591,$,$,.ELEMENT.,(39,54,57,601318),(116,25,58,795166),0.,$,$);

#679=IFCRELAGGREGATES('35aC7nYvTEcBYUkz$D5i0v',#41,$,$,#94,(#592));

#104=IFCBUILDING('0DVPyye2z0deMtQa1pIatb',#41,'',$,$,#32,$,'',.ELEMENT.,$,$,#100);

#683=IFCRELAGGREGATES('30UiOLvXbBv9bj$ovhCyLU',#41,$,$,#592,(#104));

#113=IFCBUILDINGSTOREY('0DVPyye2z0deMtQa2CjRCZ',#41,'\X2\68079AD8\X0\ 1',$,$,#111,$,'\X2\68079AD8\X0\ 1',.ELEMENT.,0.);

#687=IFCRELAGGREGATES('27PCKGLxT4mxtV9cw6mgBW',#41,$,$,#104,(#113));

#155=IFCWALLSTANDARDCASE('0o\_qQXI0r5UQtbU$t93HI7',#41,'\X2\57FA672C5899\X0\:CW 102-50-215p:309591',$,'\X2\57FA672C5899\X0\:CW 102-50-215p:412',#124,#151,'309591');

#672=IFCRELCONTAINEDINSPATIALSTRUCTURE('3Zu5Bv0LOHrPC10066FoQQ',#41,$,$,(#155,#367,#430,#550),#113);

#94代表项目，#592代表场地，#679代表项目（#94）“包含”场地（#592）。#104代表建筑，#683代表场地（#592）包含建筑（#104）。#113代表楼层，#687代表建筑（#104）包含楼层（#113）。（#155）代表墙，#367、#430、#550代表其它建筑构件，#672代表楼层（#113）包含这四个建筑构件。

这是一个典型的Ifc对象之间的空间包含关系的例子。如果用户需要指定特定楼层的某个建筑构件，这就需要通过Ifc对象之间的空间关系数据来进行查找。由于这种空间关系是使用关系对象来间接表示的，所以在一定程度上使得空间关系的查询并不容易。但是这种关系天然适合图数据库来进行存储。Neo4j是一个高性能的图数据库，它将整个复杂的关联数据集存储在一个大型的网络结构中，再采用一系列的图操作来实现对数据的管理和应用。

接下来本文将介绍基于Neo4j图数据库的Ifc对象之间的空间关系数据的存储和查询相关技术。

## 5.2 图数据模型设计

在现实世界中，建筑元素的关系可以是：某个项目包含很多场所，每个场所包含很多建筑，每个建筑包含很多楼层，每个楼层包含很多建筑构件，如门、窗、墙、梁、板、洞等等。而建筑构件之间也有多元化的关系，比如某两面墙相连接、某面墙上有四扇窗等等。这些关系可以形象化地展示为图1.

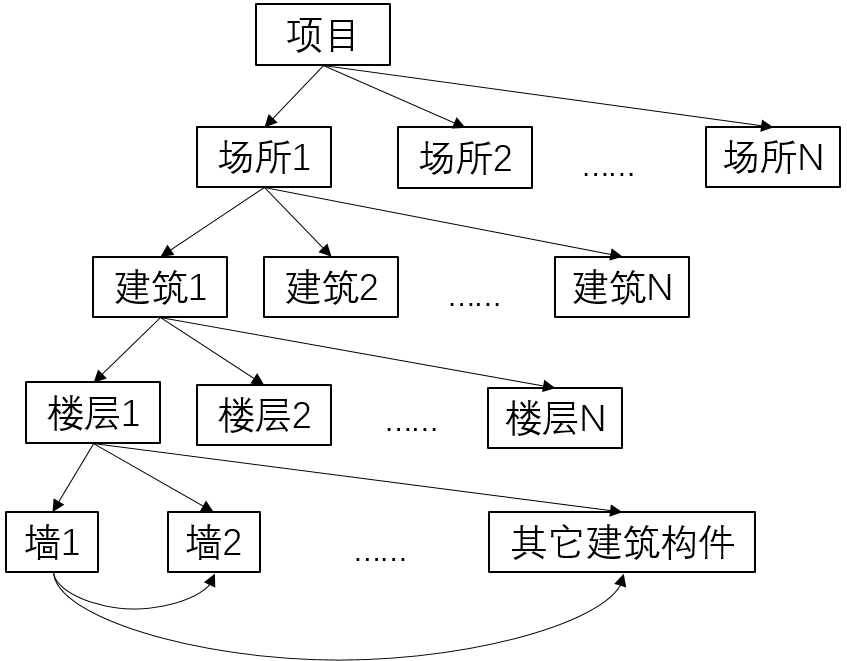


图1 现实世界中的建筑元素的空间关系图

依照4.1小节对于IFC标准中关系的介绍，如果用IFC对象来表示上面图中的建筑元素，则对应的是下面这张图2。

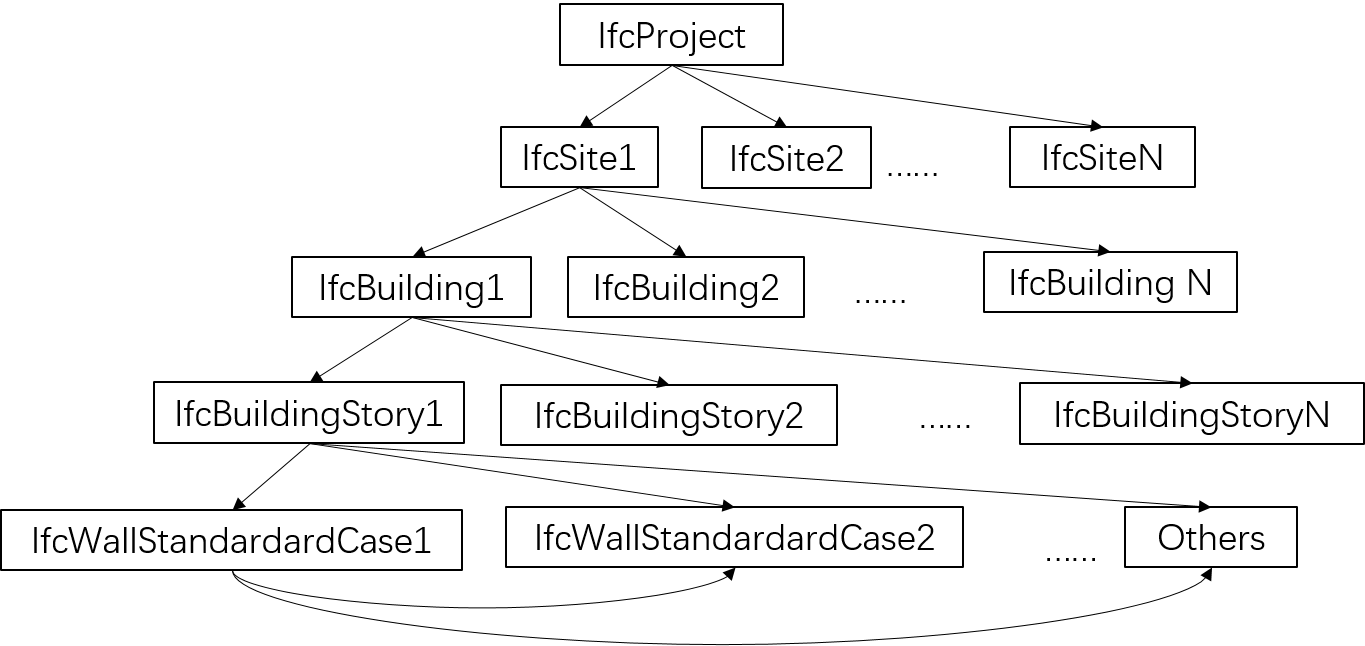


图2 IFC对象的空间关系示意图

在上图中我们用一条带箭头的连线来代表Ifc对象之间的空间关系。而这些关系在IFC标准中是用IfcRelationship的派生类来定义的。例如，IfcProject包含了很多IfcSite，这个包含关系用IfcRelAggregates来表示。而IfcWallStandardCase连接了另一个IfcWallStandardCase，这个连接关系用IfcConnects关系类型来定义。

如果使用IfcRelationship派生类来替代上图中带箭头的连线，就是一个完整的IFC对象的空间关系模型，如图3。为了简洁，只标出了部分关系。

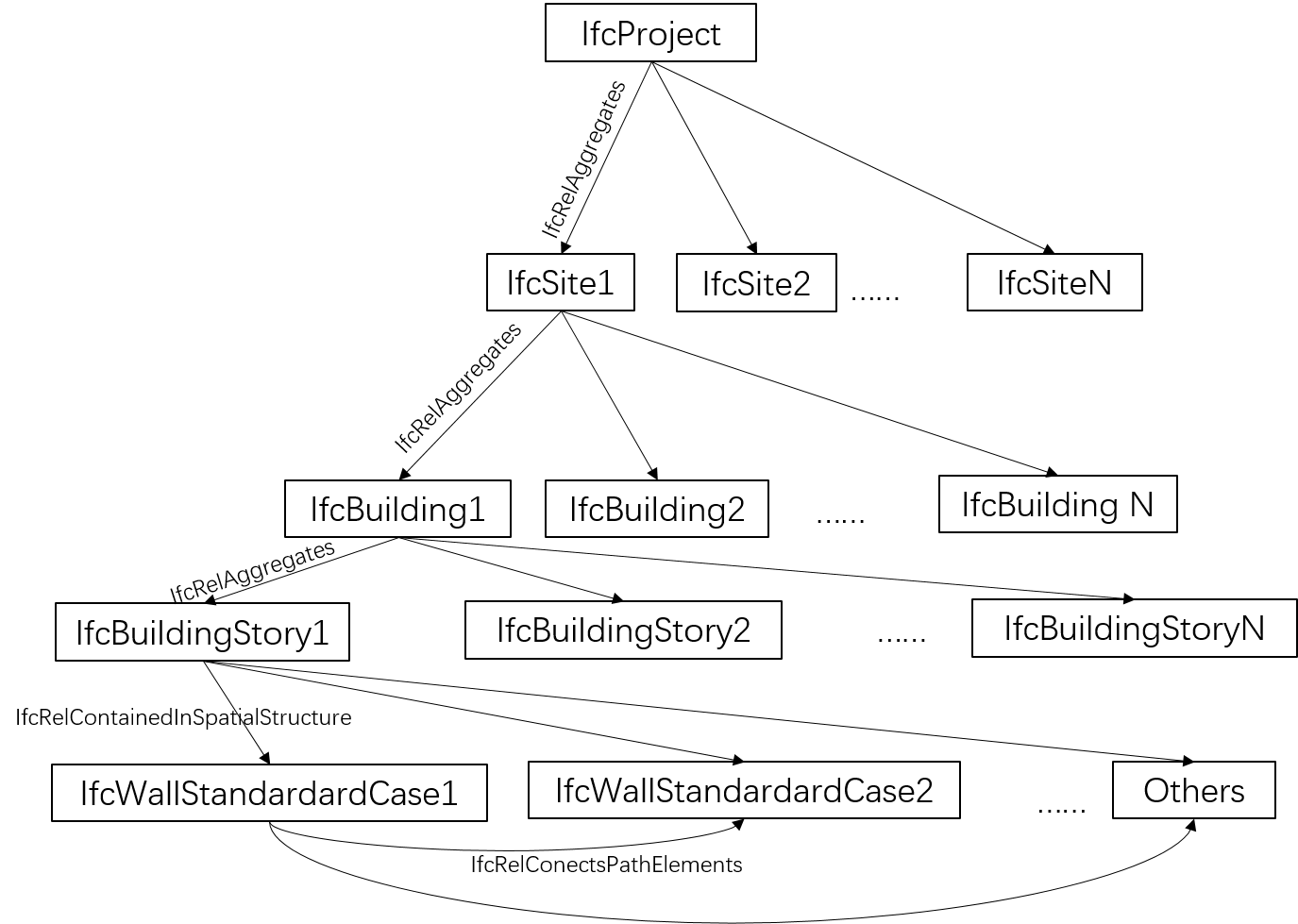


图3 IFC对象的空间关系数据模型

图3展示的模型可以直接对应到Neo4j的图数据模型。在Neo4j中，有两种基本数据类型，Node（节点）和Relationship（关系）。Node通过Relationship所定义的关系连接起来。同时可以在Node和Relationship上赋予key/value形式的属性，来表示详细的信息，以便对Node和Relationship进行查询。如果指定某个节点，从此节点出发，通过关系找到另外一个节点，这就形成了一条路径。这就是Neo4j的图数据模型。由此可知，图3中所展示的IFC对象的空间关系数据模型，天然适合Neo4j的图数据模型。每一个IFC对象对应图模型中一个Node，每一个IfcRelationship派生类对应图模型中的relationship。由此可以实现IFC对象的空间关系数据的存储，并依托Neo4j对于图数据的操作来进行IFC对象的空间关系的查询和应用。

## 5.3 图数据提取技术

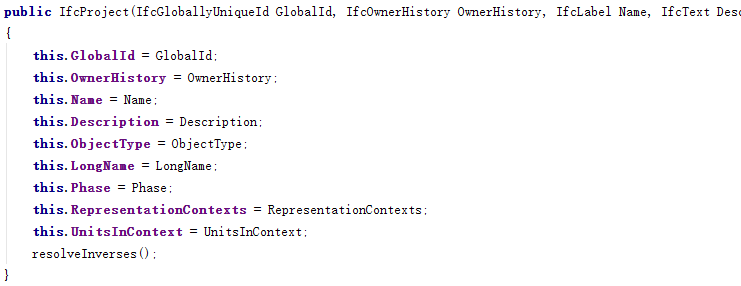
本小节将针对Ifc文件数据如何解析到内存中进行详细介绍。以三条IFC数据为例：

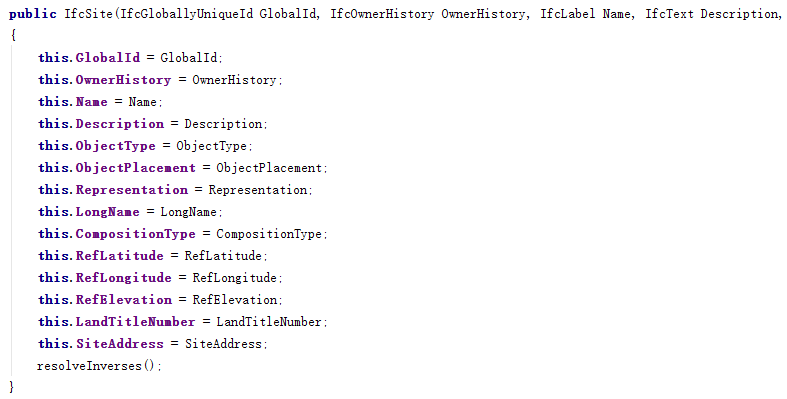
#94=IFCPROJECT('0DVPyye2z0deMtQa1pIata',#41,'\X2\987976EE7F1653F7\X0\',$,$,'\X2\987976EE540D79F0\X0\','\X2\987976EE72B66001\X0\',(#83,#91),#78);

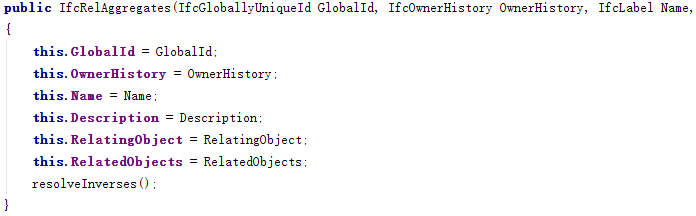
#592=IFCSITE('0DVPyye2z0deMtQa1pIatc',#41,'Default',$,'',#591,$,$,.ELEMENT.,(39,54,57,601318),(116,25,58,795166),0.,$,$);

#679= IFCRELAGGREGATES('35aC7nYvTEcBYUkz$D5i0v',#41,$,$,#94,(#592));

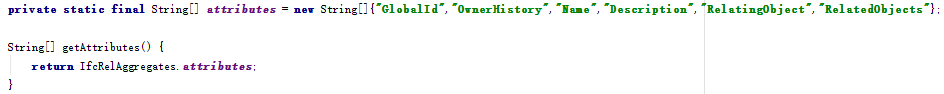
基于3.2.1小节介绍的IFC文件解析过程，这三条数据将被解析为内存中三个Java对象。







Java对象对应了IFC对象，Java对象的fields对应了IFC对象的字段。这些Java对象对应Neo4j的Node，Java对象的字段和字段值将作为key/value pair，存储到Neo4j数据库中，作为Node的属性。

由于IfcJavaToolBox解析器并不支持获取对象的字段名，所以需要为每一个对象类型定义一个attributes数组结构，如IfcRelAggregates.java代码中的定义：

在此需要说明，因为这些Java对象的fields是继承父类的，反射的方法在此行不通，所以才采用了这种方法。attributes数组中的值必须严格按照IFC标准的schema当中的属性定义的顺序，这样才能够解析出正确的key/value pair。

至于id字段可以借鉴bimserver的方法，内存中建立一个对象和id的映射的map，已达到自定义id的目的。

## 5.4 图数据存储设计

在上一小节中介绍了IFC文件中的三条数据解析到了内存中。这已经完成了数据准备。这就来到了Neo4j开发的部分。

本小节将介绍如何将内存中的Java对象数据存储到Neo4j当中。Neo4j提供了服务器版本和嵌入式版本。但由于后者更有利于接口自定义设计，且不受网络数据传输效率的影响，本文采用嵌入式版本来进行开发。

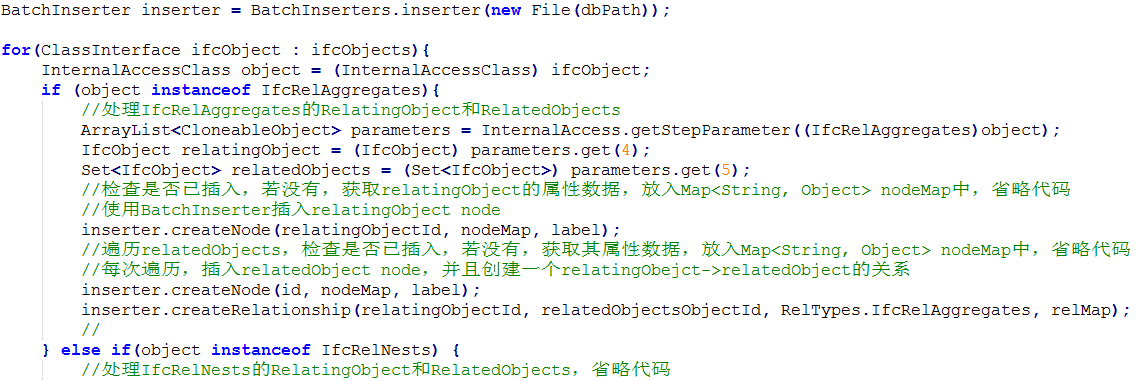
Neo4j存储图数据的步骤是：

1. 构造Map结构，存储属性的key/value pair。
2. 指定DB PATH，在该PATH下创建EmbeddedGraphDatabase实例。
3. 调用createNode()方法生成Node实例。
4. 实现Relationship接口，定义枚举类型表示关系。
5. Node实例调用createRelationship()方法，创建节点之间的关系。
6. 调用PropertyContainer 接口中的方法setProperty()将Map结构的key/value，赋值给Node和Relastionship实例，并可通过 getProperty()获取属性。

这是创建少量节点时可以采用的方式。在本文中需要存储的IFC对象数量巨大，所以采用批量插入——BatchInserter的方式，以提高存储效率。其算法思想是：

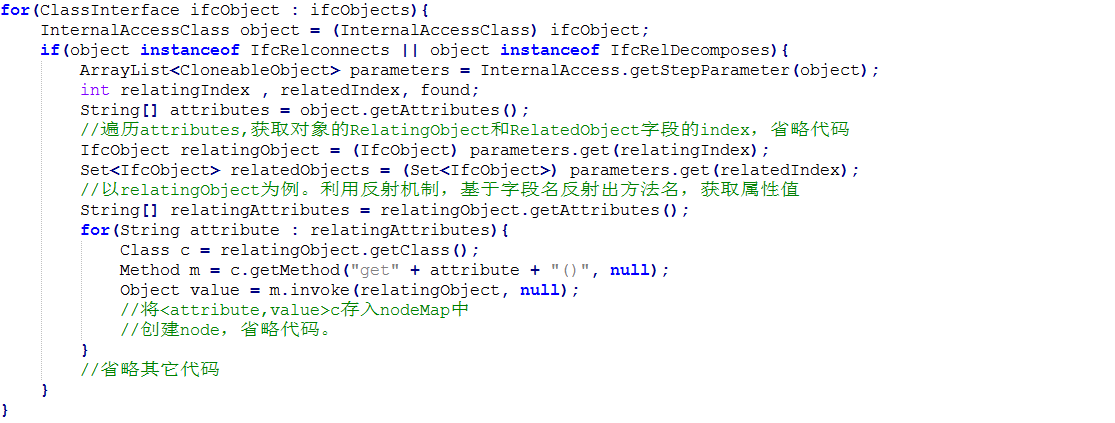
1. 遍历内存中构建的Java对象集合Collection<ClassInterface> ifcObjects，找到IfcRelConnects和IfcRelDecomposes的派生类实例（为什么限于这两类，已经在4.1小节介绍）。
2. 对于1中找到的每一个关系实例，其relatingObejct字段指向一个关系的开始节点，而relatedObjects指向一个关系的结束节点，结束节点可能是一个集合。此时可以使用Neo4j的批量插入接口BatchInserter来创建节点和关系了。
3. 需要注意的是，节点可能出现在多个关系实例里。所以，需要维护一个List(Long)类型的结构，存储已经创建的节点的ID。每次创建节点之前检查是否已经创建，从而避免重复创建节点。

其代码逻辑如下：



这是使用原生的IfcJavaToolBox的API所写的代码，因为本文关心的IfcRelConnects和IfcRelDecomposes的派生类有13种，而每一种的字段类型都不相同，要从中找到relatingObject和relatedObjects需要针对不同的类型各写一个if代码块。同时，每一个IfcObject的字段也千差万别，所以这样写代码实现起来十分冗杂。

因此想到，结合本文为每个IFC JAVA对象定义的attibutes数组，再使用JAVA反射机制，调用method.invoke(), 可以很容易获取字段值，从而大大简化代码。代码逻辑如下：



以上代码逻辑实现了将内存中已经解析出的IFC JAVA对象的数据存储到Neo4j的过程。

Batch方式的插入效率？

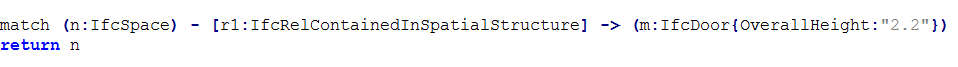
## 5.5 图数据查询设计

用户使用Neo4j的一个经典场景是依据节点或者关系的属性数据找到相关节点以及关联的节点。例如，用户需要找到已经安装了高度为2.2m的门的墙。在IFC对象关系数据模型中，门与墙的关系如图：



图4

于是其查询逻辑使用Cypher如表达式1：



表达式1

其查询执行逻辑是：

1. 访问节点文件和属性文件，找到Label为IfcDoor的节点，遍历IfcDoor类型节点链表，读取其属性数据，找到满足OverallHeight为2.2的节点。
2. 访问关系文件，以1）中找到的节点为起点，遍历关系链表，过滤条件是找到Label为IfcRelContainedInSpatialStructure的关系，且关系的终止节点的label为IfcSpace。
3. 返回符合条件的终止节点。

可以看到，整个查询过程可以分为两个部分，第一步是先找到起始节点，

第二步是以起始节点开始，遍历其关系，过滤目标节点。

关系遍历的规模是起始节点的出度。而第一步找到其实节点的遍历规模则是整个Label类型的节点。这时候需要大量的随机读写，如果没有索引，效率将非常低下。为了获得更好的性能，我们可以在字段属性上构建索引，这样任何查询操作都会使用索引，从而大幅度提升查询性能。

在第二章中我们已经介绍了Neo4j的索引。本文将使用Schema索引，来为节点和关系创建索引。下面是创建索引的Cypher语句示例：

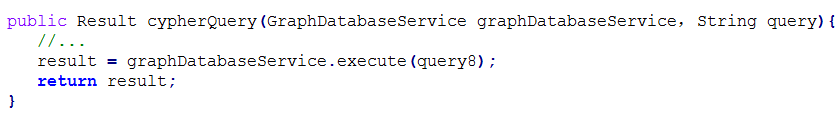


由于Schema索引是依据Label和属性字段来创建的，且IFC对象类型多、属性字段差异大，手动创建十分麻烦。所以本文采用的方法是，在初次向Neo4j中导入所有IFC对象的数据之后，依据IFC对象的数据模型一次性创建所有IFC类型的索引。

其代码逻辑是：

1. 维护一个已导入的IFC对象类型集合，可以防止对不存在的类型进行索引；
2. 针对每个IFC对象类型，获取属性名数组attributes，进行Cypher语句字符串的拼接；
3. 执行Cypher索引语句。

对于Cypher语言的执行，使用嵌入式Neo4j的GraphDatabaseService API，其代码如下：



创建索引以后，当查询指定属性值的节点时，Schema索引将自动被调用，以提高查询性能。

关系是否要被索引呢？

Cypher语言介绍放在前面的部分。

Neo4j是同时启用线程共享的。所以可以另开线程，将

使用cypher语言的代码，语句如何转化？

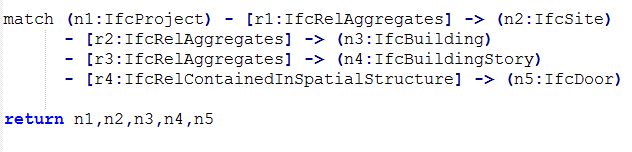
直接将代码搬出来？可是没有多长。需要将用户的查询语言转化为cypher语言进行查询。不用了。这个查询语言已经很简洁了。重点放在查询的快速失败上。如果是空值查询，是不是比较快呢？试验证明是的。

## 5.6 基于BloomFilter的路径查询优化

前面几个小节已经讲述了如何基于Neo4j图数据库来进行IFC对象的空间关系数据的存储和查询。这已经能够满足用户对于某个对象或者某些对象以及它们之间的空间关系的查询。本小节将针对一种特定的路径查询场景提供优化方案。

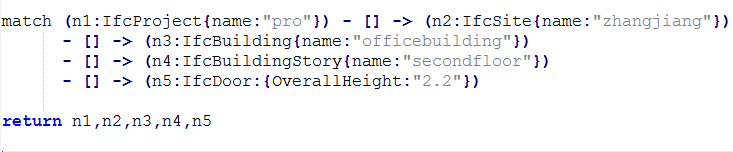
5.6.1 特定路径查询场景

IFC标准中有一种特定的查询是，用户需要依次确定项目、场地、建筑、楼层，然后再找到楼层中满足一定条件的建筑构件，如门、窗、家具或其它零件。图3展示了这种关系在neo4j中的表示。在Neo4j图数据模型的定义中，从项目节点到建筑构件节点形成了一个路径，这个查询是一个特定的路径查询。其Cypher语句的简单形式是表达式1：



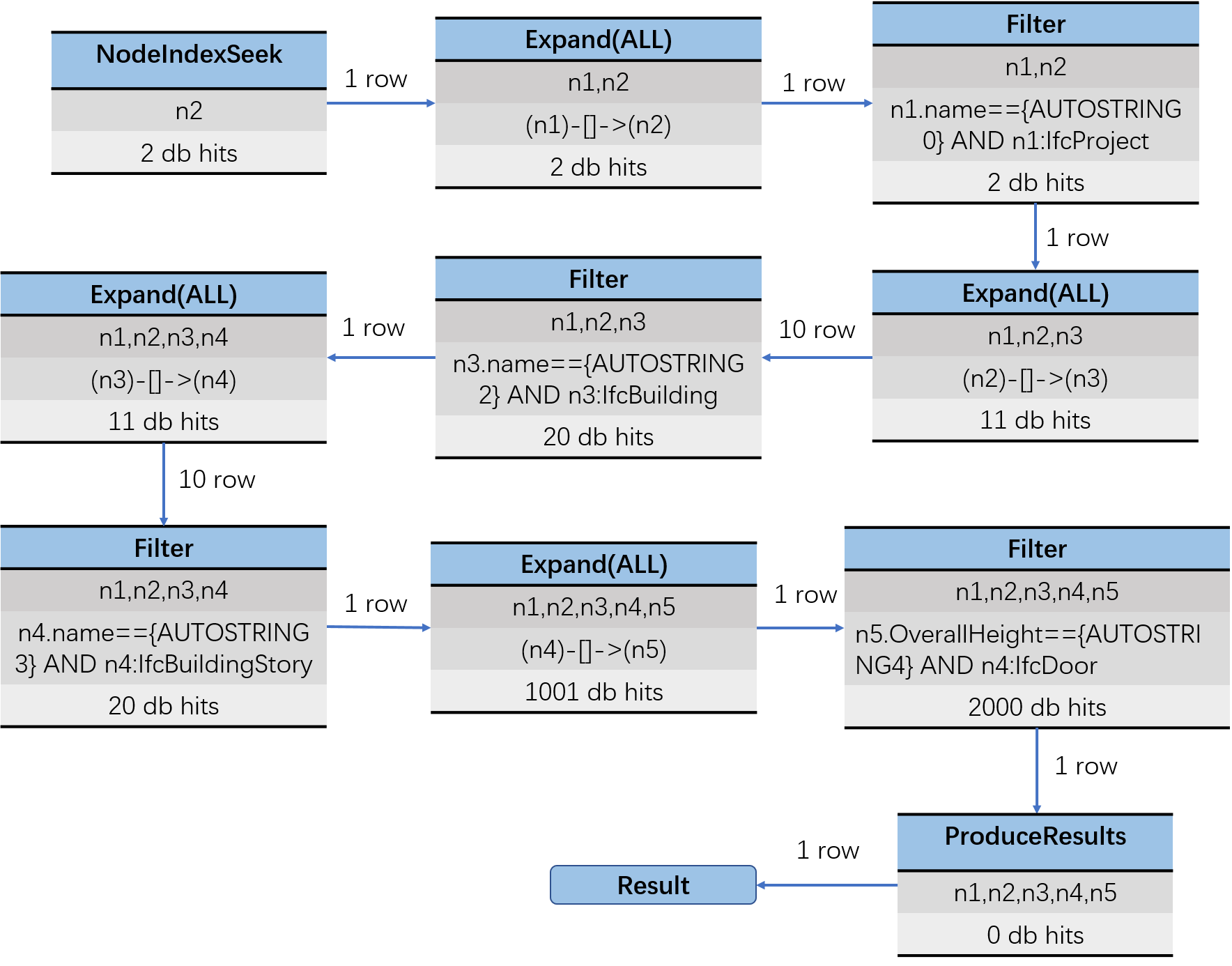
表达式2

又由IFC标准的定义可知，r1、r2、r3、r4的类型是固定的，所以可以在Cypher表达式中省略掉类型。同时，用户需要对路径中的每一个节点加以属性值的约束，本文的示例暂且以“name”属性约束作为这个路径查询中对节点的约束条件。所以Cypher表达式应为式2：



表达式3

这个Cypher查询语句的完整的执行计划是：



1. 执行NodeIndexSeek。访问索引“index on :IfcSite(name)”，返回节点n2。在这里假设只有一个IfcSite节点满足条件而返回。
2. 执行Expand(ALL)，访问指向节点n2的关系链表。因为一个IfcSite只有一个IfcProject指向它，也就是说指向n2的关系只有一个。所以访问db一次。
3. 执行Filter。过滤2）中关系所关联的节点的类型和属性，返回满足条件的IfcProject节点n1。
4. 然后从节点n2开始，依次执行Expand(ALL)和Filter过程。知道最后一个关系所关联的节点完成filter，找到满足条件的节点为止。
5. 返回结果。

分析这个过程，可以得出这样的结论：只有在整个执行计划的第一步使用了index，先找到一个切入点作为起始节点。而之后是通过访问关系链表数据找到关联的节点，而后对这些节点进行过滤。

这里有几个值得讨论的问题：1）没有充分利用index 2）当节点所关联的关系数量增长时，链表结构的访问效率会成为影响查询效率的不容忽视的因素。3）由于是从起始节点依次开始访问关系链表和节点，很有可能出现当执行到最后一个filter环节时才发现没有满足条件的节点而返回空值，此时前面的查询操作已消耗了大量时间。

对此，本文提出用查询分解的方式，先将原始的查询分解为多个针对节点的查询，找出所有的满足条件的ID之后，然后将ID进行拼接，利用BloomFitler做结果过滤。

下面小节的安排是，先介绍关键的BloomFilter数据结构，然后介绍如何构建存储用于本小节所介绍的特定路径查询的BloomFilter，最后会介绍如何基于BloomFilter进行查询。

### 5.6.2 BloomFilter原理

BloomFilter于1970年由Burton H. Bloom提出, 利用位数组标识一个集合，可以以较低的误判率来判断一个元素是否属于这个集合。因此这种数据结构适合应用在能容忍低错误率的场合。相较于传统的哈希函数映射和存储元素的方式，BloomFilter更加节省空间，从而能够满足数据量更大的应用场景。

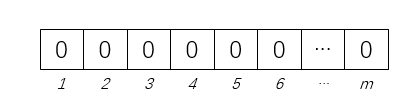
对于4.6.1小节中提到的特定的路径查询来说，如果我们能够把符合空间组成关系的IfcProject、IfcSite、IfcBuilding、IfcBuildingStory、IfcDoor（以及其它建筑构件）的ID按照顺序拼接起来，维护在BloomFilter当中。当查询来到时，将满足条件的各个节点的ID拼接起来，查看是否在BloomFilter当中，就可以知道，这个结果集是否存在。BloomFilter存在一定的误判。也就是说，如果这条路径并不在neo4j存储的图中，但是也有可能会判定为存在，此时用户得到的结果将会是由原始的Cypher查询得到的正确结果。同时，BloomFilter不会漏掉判断。这对我们是相当有利的，因为不会漏判意味着，只要判定这个字符串在BloomFilter当中，那么它一定存在。此时，我们只需要返回这个字符串就可以了。另外，BloomFilter用很少的内存就可以支撑上亿条记录，基本可以满足我们的存储和查询需求。

BloomFilter的核心思想就是利用多个不同的Hash函数来解决单一Hash带来的“冲突”。传统的Hash表利用同一个Hash，计算不同的字符串得到的值可能相同时，这就是“冲突”。如果要减少冲突，必须要将位数组的长度扩大。这样会消耗大量内存。为了解决这个问题，BloomFilter支持选择k个不同的哈希函数，这样就可以大大地减少冲突，同时保证低的内存消耗。

BloomFilter的原理要点：位数组、k个独立的Hash函数。

1. 位数组：

初始状态时，BloomFilter是一个包含m位的bit数组。每个bit都为0，如下图：



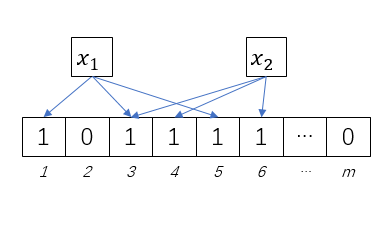
1. 基于K个独立Hash函数，添加元素：

待添加的元素设为 。当加入任何一个元素x的时候，BloomFilter使用k个互相独立的Hash函数，得到k个Hash值，对应的范围内的k个位置。将这个m bit数组的对应k个位置的值置为1。如果这个位置被多次置为1，那么只有第一次会起作用。

例如，k=3，,,,

,,.

加入，则bit数组为：



1. 判断元素是否存在：

在判断元素y是否属于这个集合时，只需要对y使用k个Hash函数得到k个hash值，对应bit数组中k个位置。如果这k个位置都是1，那么就认为y是集合中的元素，否则不是。显然这不是100%正确的。如果y事实上并不在集合里，但映射到的k个位置刚好恰好已经置为1了，那么就会出现False Positive，也就是误判。

1. BloomFilter的参数：

误判率与插入BloomFilter的元素个数n、bit数组大小m以及Hash函数个数k有关。研究表明，当Hash函数个数k时错误率最小。在错误率不大于E的情况下，m至少要等于才能表示任意n个元素的集合。

对于在本文IFC数据存储应用中的BloomFilter的参数选择将在接下来的构件BloomFilter中介绍。

### 5.6.3 构建BloomFilter

根据4.6.1小节对于改进算法的描述可知，如果要想实现这样的查询，必须先基于IfcProject、IfcSite、IfcBuilding、IfcBuildingStory、IfcDoor（等）的空间组成关系，将所有的这些ID拼接字符串存入到BloomFilter当中。这需要在内存中解析IFC对象时就把它们的关系对应起来。为了达到这个目的，需要借助于以下几个Map结构。

1. Map<Long, Long> siteToPro;

存储IfcSite的ID与其所在的IfcProject的ID的映射。

1. Map<Long, Long> bdgToSite;

存储IfcBuilding的ID与其所在的IfcSite的ID的映射。

1. Map<Long, Long> storyToBdg;

存储IfcBuildingStory的ID与其所在的IfcBuilding的ID的映射。

1. Map<Long, Long> elmtToStory

存储IfcElement的ID与其所在的IfcBuildingStory的ID的映射。

基于这4个数据结构，可以构建一个IFC对象getStoryID()、getBdgID()、getSiteID()、getProID()方法，得出IFC对象所对应的IfcBuildingStory、IfcBuilding、IfcSite、IfcProject的ID。然后按照从Ifcproject到IfcElement的空间组成顺序拼接ID为长字符串，插入到BloomFilter中。

关于上面的步骤，会有人有疑问：通过几个get方法不是已经可以找到上文所介绍的路径查询的结果了吗？的确是这样。但是问题就在于，Map结构非常占用内存。这也是本文为什么需要BloomFilter数据结构的原因，它能大大地缩小内存占用，从而可以常驻内存。在后面的试验中，本文会对内存占用做进一步的比较分析。

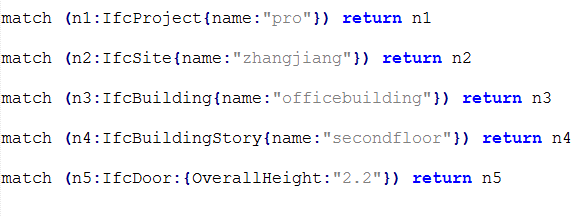
### 5.6.4 基于BloomFilter的查询

回顾4.6.1节中表达式2的Cypher查询所存在的问题，本文改进的想法和目标是：

1. 将查询分解为多个对节点的查询，从而可以充分利用index；
2. 由于1)步骤改变了查询计划执行的顺序，从而使得当节点本身不满足属性要求时可以快速失败，返回结果；
3. 使用BloomFilter的过滤机制，而不是链表的顺序访问机制，可以有效提升查询速度。

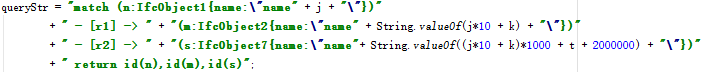
具体的算法逻辑是：

1. 将上面的Cypher查询分解为四个对节点的Cypher查询：



1. 分别执行以上四个查询，也就是四个NodeIndexSeek。如果结果集为空，则可以直接结束整个查询，返回给用户空值；
2. 如果结果集不为空，则将节点的ID字段拼接为一个字符串。
3. 已创建的BloomFilter包含了所有的可能的结果字符串。查看3）中的字符串是否包含在BloomFilter中。如果在，则返回ID对应的节点。如果不在，则说明查询结果为空。

原始的查询格式比较固定，会是如下情况：



而分解后的查询如下：







之后进行ID拼接：



最后基于BloomFilter过滤：



# 第六章 系统实验

## 6.1 属性数据的索引与查询

本文提出了基于ElasticSearch的IFC属性数据存储和查询方案。接下来将针对该方案进行测试，分以下内容进行介绍：

1. 实验环境

本文搭建了具有4个节点的ElasticSearch集群，另有一台主机来访问集群。集群的服务器配置如下图：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| IP | CPU | MEM | HDD | OS |
| 10.141.211.150 | Intel(R)Xeon(R)  CPU E5649 @2.53GHz | 10G | 194G | CentOS 6.7 |
| 10.141.211.151 | Intel(R)Xeon(R)  CPU E5649 @2.53GHz | 14G | 220G | CentOS 6.7 |
| 10.141.211.152 | Intel(R)Xeon(R)  CPU E5649 @2.53GHz | 13G | 198G | CentOS 6.7 |
| 10.141.211.153 | Intel(R)Xeon(R)  CPU E5649 @2.53GHz | 12G | 195G | CentOS 6.7 |

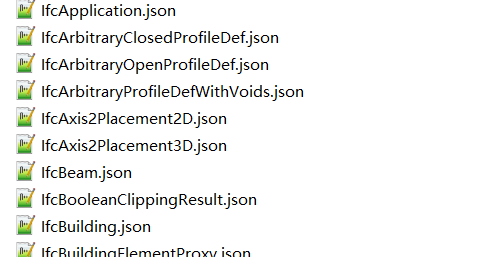
本次试验部署的ElasticSearch的版本为5.4.2,是近期更新的版本。

1. 数据集

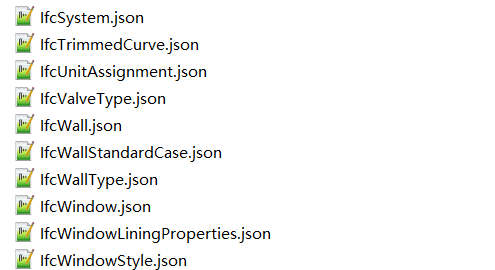
受限于IFC文件来源并不多，本文将基于现有IFC文件生成测试数据。现有IFC文件大小为94M，是重庆研究院大楼的设计文件。其用IFC文件查看工具IFC Viewer呈现出来的效果如下：



IFC文件解析后生成JSON文件160个，JSON数据总量为126M，部分文件截图如下：



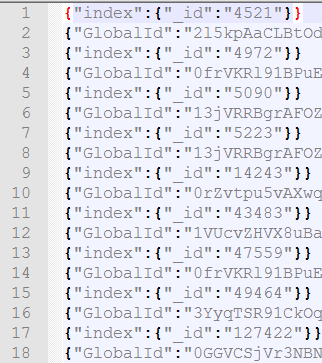
…



下图是解析后的IfcWallStandard.json的示例：



下图是解析后的IfcWallType.json的示例：

本文在解析IFC文件的每一行数据时为IFC对象分配新的ID，代替原来的行号。IFC对象之间存在互相引用，引用字段的值是被引用对象的ID。为了保持这种引用，需要引入map结构，来维护旧ID与新ID的映射。

新分配的ID对应ElasticSearch的\_id，从而索引为不同的文档。每解析一次，产生一倍的数据。试验共生成了约220万行文档数据，5个索引分片分别配置了一个副本，数据分布在集群中的4台服务器上。

1. 索引测试

本文采用bulk机制对于大数据量的索引进行批量索引。对于ElasticSearch的配置按照3.3.3小节的方案进行优化，其关键配置参数如下：

|  |  |  |
| --- | --- | --- |
| 参数 | 默认值 | 配置值 |
| ES\_HEAP\_SIZE | 2g | 5g |
| index.number\_of\_shards | 5 | 5 |
| index.number\_of\_replicas | 1 | 1 |
| http.max\_content\_length | 100mb | 1024mb |
| index.refresh\_interval | 1s | -1 |
| Index.translog.flush\_threshhold\_ops | 5000 | 100000 |
| indices.memory.index\_buffer\_size | 10% | 50% |

试验结果显示，同样的实验环境，默认配置下的ElasticSearch的索引效率是1076 doc/s。而经过优化配置后索引的效率为2012 doc/s，可以看出性能有了很大提升。

索引后的集群可以通过head插件来访问：

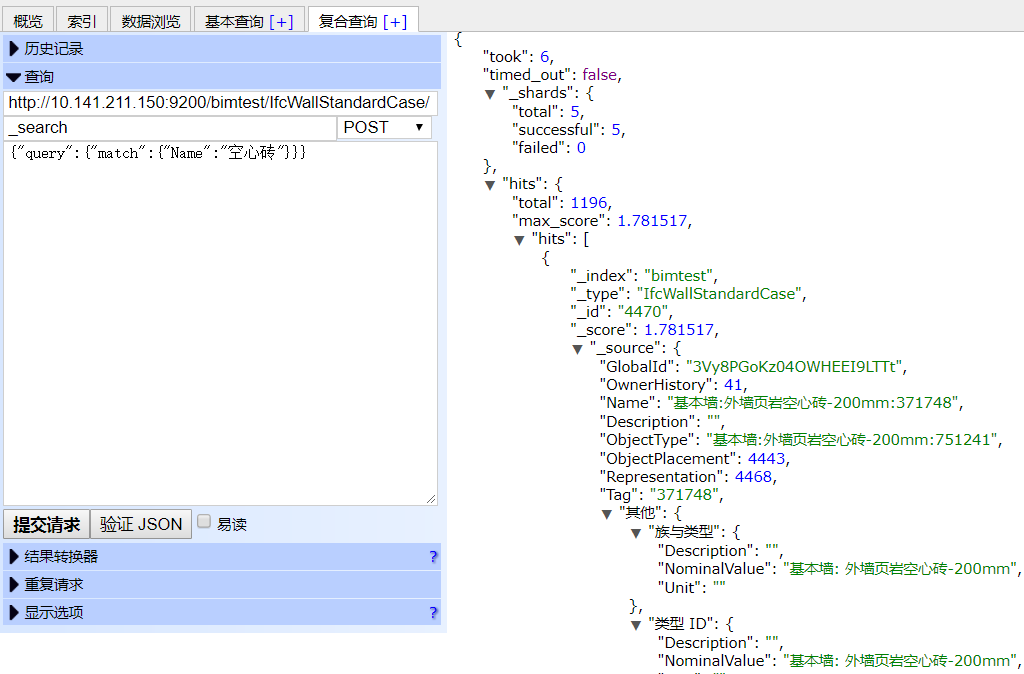


ElasticSearch自动mapping的部分结果如下：



1. 查询测试

在ElasticSearch的head插件界面进行查询测试如下, 可以看到，集群能够以ms级别的速度来响应查询。



## 6.2 空间关系数据的索引与查询

本文采用Neo4j来存储IFC对象的空间关系数据。下面来介绍基于Neo4j的索引和查询效率。从以下几个方面介绍：

1. 实验环境

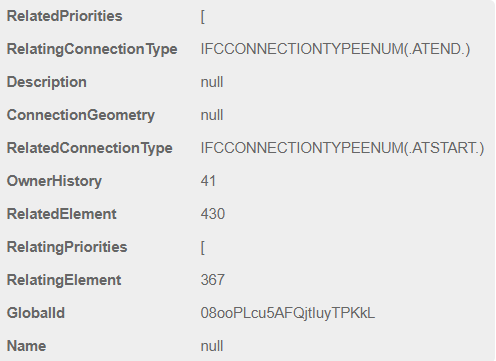
本文基于嵌入式Neo4j 3.0.6版本的开发包来进行空间关系数据存储的实验。实验主机为一台Windows 10笔记本，8GB DDR3内存。

1. 数据集

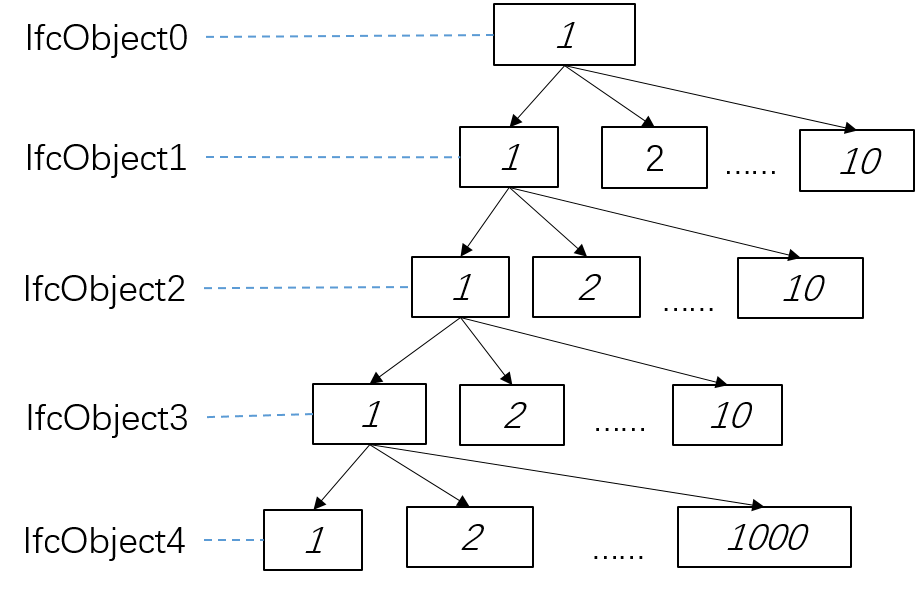
实验原始数据为重庆研究院的设计文件。本文基于此数据集进行系统有效性试验。经过解析，形成了8222个Node，11834个Relationship。每一个节点存储了自身所定义的属性数据，如下面的一个IfcBuildingStorey类型的节点的数据：



如下是一个IfcRelConnectsElements关系类型的数据：

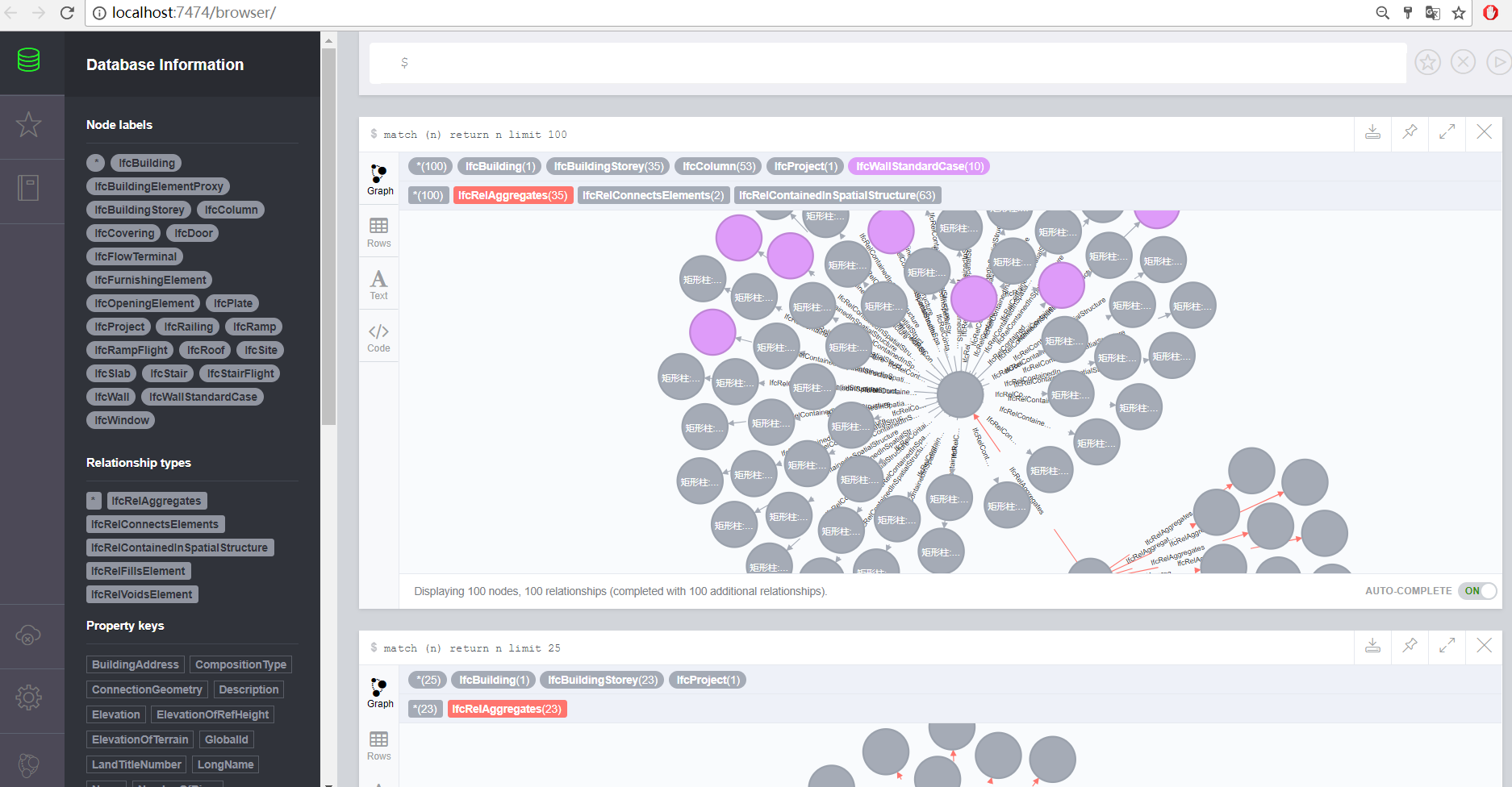


对于大数据量下特殊路径查询的效率的实验，单单通过分配新ID的方法，只能实现一栋楼变成孤立的十栋楼的效果，无法实现图数据关联复杂度的增长。所以本文另外生成了测试数据，参照IfcSite、IfcBuilding、IfcBuildingStorey、IfcWall的关联方式，生成IfcObject1、IfcObject2、ifcObject3、IfcObject4四种类型数据，依次以10倍、10倍、10倍、1000倍的比例增长。总计共有约100万个节点。它们的数据量和关联关系为：



1. 索引测试

本实验使用BatchInsert方式批量索引节点和关系，经试验，插入的效率是平均每秒9708个Node，接近1万，是相当高效的。索引数据后，使用Neo4j社区版本的客户端连接数据库，并可以通过7474端口从浏览器访问数据库。其界面如下：



其中，左侧显示了节点的标签、关系的类型、属性等信息，右边可以使用Cypher语句进行查询。

1. 查询测试

基于Cypher语句可以方便地访问Neo4j数据库。不同的Cypher查询类型的效率会大不相同。对于100万个节点的Cypher查询效率测试如下：

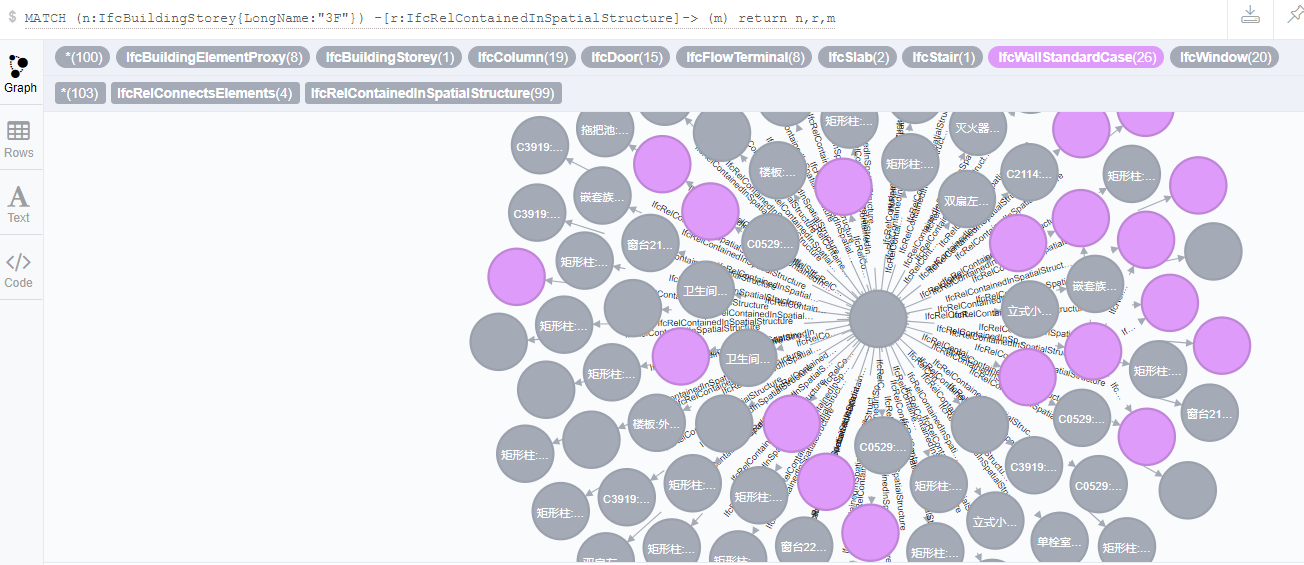
|  |  |  |
| --- | --- | --- |
|  | 无索引 | 有索引 |
| 基于property查询：  match (n:{name:’TopFloor’}) | 1170ms | 1091ms |
| 基于Label、property查询  Match (n:IfcObject{name:’TopFloor’}) | 265ms | 45ms |

关系查询的效率与数据复杂度有关，也与查询的类型有关，难以给出标准评判，在此做了可行性测试的举例。上图是一个关系查询实例：

MATCH (n:IfcBuildingStorey{LongName:"3F"})

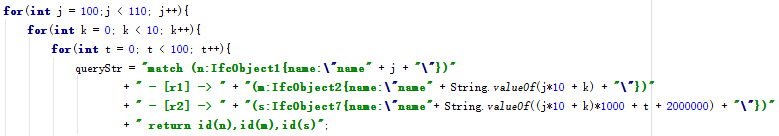
-[r:IfcRelContainedInSpatialStructure]-> (m) return n,r,m

表示返回包含（IfcRelContainedInSpatialStructure）在“3F”这层楼的所有的建筑构件。共返回了331条建筑构件的数据，只用了230ms。



1. 基于BloomFilter的特定路径查询测试

再次试验中本文使用的数据集是参照了IFC空间关系而生成的约100万节点的测试数据。由于是特定路径查询，所以它的查询具有较为固定的格式。本文抽取了其中的10000条数据进行普通的Cypher查询。



同时，基于BloomFilter进行相同的查询，即分解查询①

、拼接ID②、BloomFilter过滤③的方式，进行这10000次查询，如下：

1. 
2. 
3. 

对比时间消耗：

三次试验，原始的Cypher查询的耗时是33986ms、24051ms、24991ms，而使用基于BloomFilter的查询方式耗时是11757ms、11972ms、11058ms，是耗费了前者最快的时间的一半，相当地高效。

# 第七章 总结与展望