

復旦大學

碩 士 学 位 论 文
(专业学位)

BIM 数据存储研究

Research on BIM Data Storage

院 系： 计算机科学与技术学院

专 业： 计算机技术

姓 名： 张亚迪

指 导 教 师： 王 鹏 副教授

完 成 日 期： 2017 年 9 月 8 日

目录

摘要.....	IV
Abstract.....	V
第 1 章 绪 论.....	1
1.1 研究背景及意义.....	1
1.2 国内外研究现状.....	2
1.3 研究内容和本文贡献.....	3
1.4 组织结构.....	4
第 2 章 相关技术基础.....	5
2.1 IFC 相关技术.....	5
2.1.1 IFC 标准.....	5
2.1.2 IFC 文件格式.....	7
2.2 全文检索技术.....	8
2.2.1 全文检索基础理论.....	8
2.2.2 Lucence 概述.....	9
2.2.3 ElasticSearch 技术.....	11
2.3 图数据库技术.....	12
2.3.1 图数据与图数据库.....	12
2.3.2 Neo4j 概述.....	13
第 3 章 系统总体架构.....	17
3.1 总体架构设计.....	17
3.2 子模块设计要点.....	18
第 4 章 基于 ElasticSearch 的 IFC 对象属性数据的存储和检索研究.....	19
4.1 IFC 对象的属性数据模型概述.....	19
4.1.1 属性.....	19
4.1.2 属性集.....	21
4.1.3 属性集与 IFC 对象关联.....	22
4.2 数据提取技术.....	23
4.2.1 IFC 文件解析.....	23
4.2.2 JSON 数据映射.....	24
4.3 数据索引技术.....	28
4.3.1 数据模式映射.....	28
4.3.2 索引过程.....	29
4.3.3 索引过程优化.....	31
4.4 数据检索技术.....	33
第 5 章 基于 Neo4j 的 IFC 对象空间关系数据的存储和查询研究.....	36
5.1 IFC 实体空间关系数据模型.....	36
5.2 图数据模型设计.....	38
5.3 图数据提取技术.....	40
5.4 图数据存储设计.....	42
5.5 图数据查询设计.....	43

5.6	基于 BloomFilter 的路径查询优化.....	45
5.6.1	特定路径查询场景	45
5.6.2	BloomFilter 原理	47
5.6.3	构建 BloomFilter	49
5.6.4	基于 BloomFilter 的查询	49
第 6 章	系统实验.....	51
6.1	属性数据的索引与查询.....	51
6.1.1	实验环境.....	51
6.1.2	数据集	51
6.1.3	索引测试.....	53
6.1.4	查询测试.....	55
6.2	空间关系数据的索引与查询.....	55
6.2.1	实验环境.....	55
6.2.2	数据集	56
6.2.3	索引测试.....	57
6.2.4	查询测试.....	58
6.2.5	基于 BloomFilter 的特定路径查询测试.....	58
第 7 章	总结与展望.....	60
7.1	总结.....	60
7.2	展望.....	60
参考文献	62

摘要

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。BIM 全称为 Building Information Model，即建造信息模型。BIM 系统要求将建筑全生命周期中的所有信息整合在统一的数据库中。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 BIM 数据库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。

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，通过增加冗余的方式，满足用户对于属性数据的高效查询，避免了多表连接的问题。在试验中，着重解决了 IFC 文件解析后的数据到 JSON 数据的映射问题，和 ElasticSearch 的索引优化等问题，最终实现了 IFC 实体的属性数据的高效存储和查询。

其次，本文还深入研究了 IFC 数据标准，提出了 IFC 实体空间关系数据模型的存储设计。本文基于开源的 Neo4j 图数据库实现了 IFC 实体空间关系数据模型的存储，从而除了属性值查询之外，为用户提供了另外一种维度的查询方式，即空间关系的查询方式。

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。

关键词：建筑信息模型（BIM），IFC 标准，ElasticSearch，Neo4j，BloomFilter

Abstract

BIM system is a new information management system, and now more and more used in the construction industry. It is called Building Information Model. The BIM system requires that all information in the entire lifecycle of the building be integrated into a unified database.

However, with the development of the times, the wisdom of the city concept, BIM data has grown by leaps and bounds, the previous BIM database storage and query means has gradually become increasingly unable to cope with the data greatly improved. So it is necessary and meaningful to provide a fast and efficient means of storage and query for mass BIM data.

Combining the development direction of BIM database research both at home and abroad, it can be seen that the goal of BIM database research is to support the storage and query of large data volume and support more flexible query mode.

Based on this goal, this paper chooses the open resilience search framework ElasticSearch and studies it. It can solve the problem of multi-table join and satisfy the efficient query of attribute data by increasing the redundancy. In the experiment, we focus on solving the problem of mapping parsed IFC data to JSON data and ElasticSearch index optimization, and finally realize the efficient storage and querying of attribute data of IFC entity.

Secondly, this paper also studies the IFC data standard in depth, and puts forward the storage design of IFC entity spatial relation data model. Based on the open source Neo4j graph database, this paper realizes the storage of the IFC entity spatial relation data model, and provides the query method of the other dimension, that is, the query method of the spatial relation, other than the attribute value query.

In addition, this paper further focuses on a specific path query, and proposed a solution based on BloomFilter for this query, so that the query efficiency has been improved, especially when it is null value query it can fail in advance, and response to the user.

Keywords: BIM, IFC, ElasticSearch, Neo4j, BloomFilter

第1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 1 章，描述了本文的研究背景与研究意义。

第 2 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 3 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 4 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 5 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 6 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2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 EXPRESS 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 2-1 是 IFC 模型总体结构图。

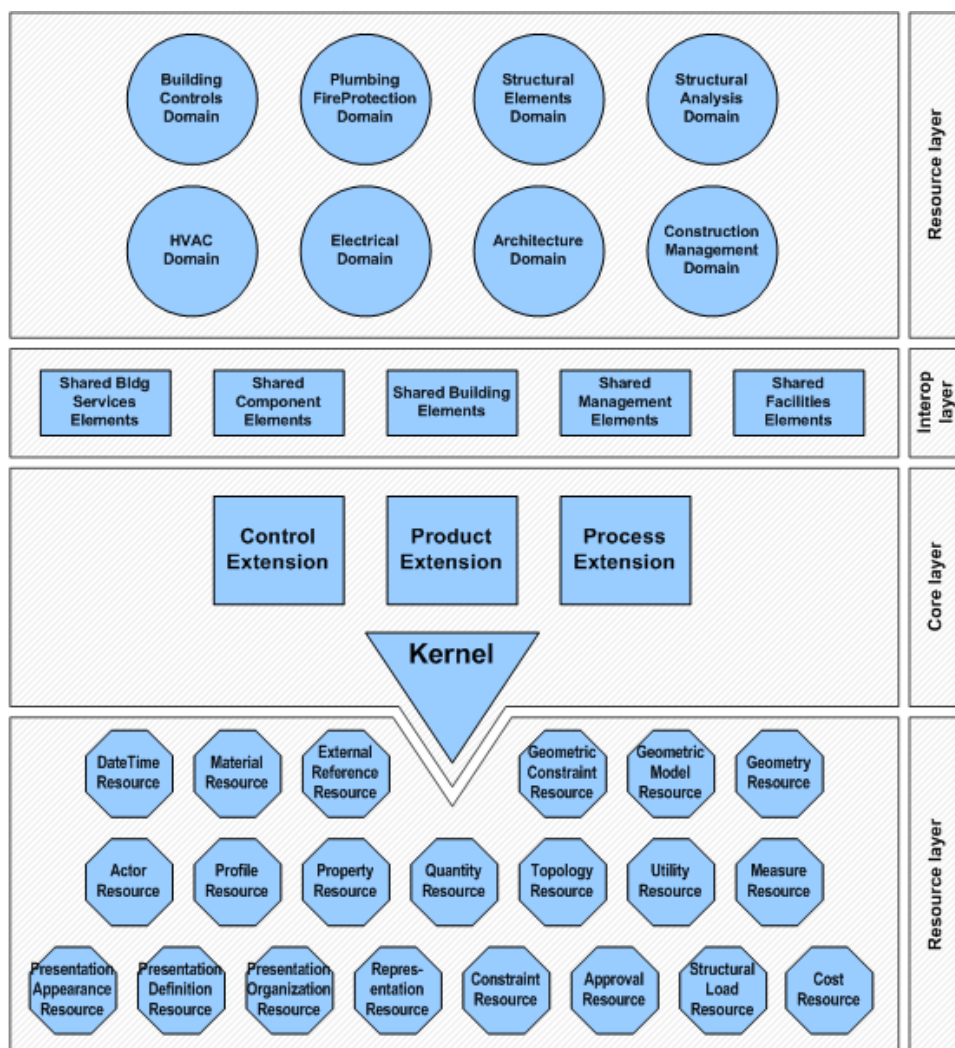


图 2-1 IFC 模型总体结构图

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。下面图 2-2 中展示的是 IfcDoor 实体类型的 EXPRESS 定义。

```
ENTITY IfcDoor
  SUBTYPE OF (IfcBuildingElement);
    OverallHeight : OPTIONAL IfcPositiveLengthMeasure;
    OverallWidth : OPTIONAL IfcPositiveLengthMeasure;
END_ENTITY;
```

图 2-2

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 IfcBuildingElement 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。

```

1 ISO-10303-21;
2 HEADER;
3 FILE_DESCRIPTION (('ViewDefinition [CoordinationView]'), '2;1');
4 FILE_NAME ('example.ifc', '2009-01-26T11:26:35', ('Architect'), ('Building De
5 FILE_SCHEMA (('IFC2X3')));
6 ENDSEC;
7 DATA;
8 #1 = IFCPROJECT('2o1ykWxGT4ZxPjHNe4gayR', #2, 'Default Project', 'Description
9 #2 = IFCOWNERHISTORY(#3, #6, $, .ADDED., $, $, $, 1232965595);
10 #3 = IFCPERSONANDORGANIZATION(#4, #5, $);
11 #4 = IFCPERSON('ID001', 'Bonsma', 'Peter', $, $, $, $, $);

91 ENDSEC;
92 END-ISO-10303-21;

```

图 2-3

图 2-3 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。

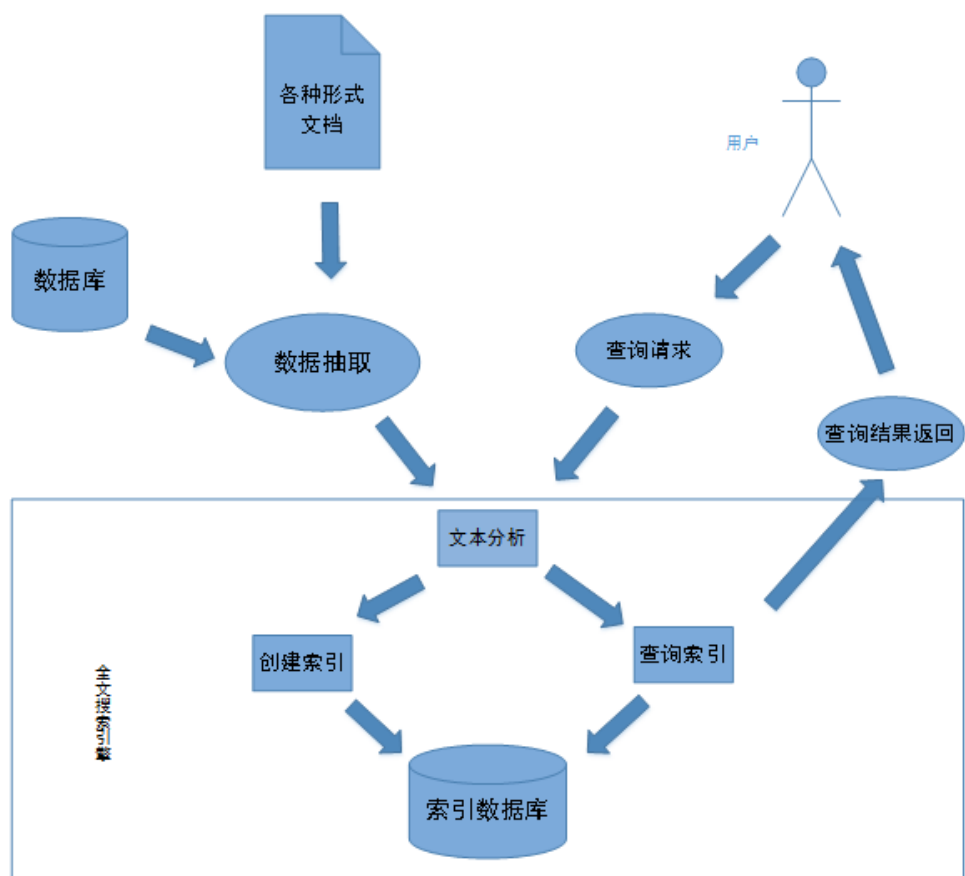


图 2-4 全文搜索一般流程

全文搜索的一般流程如图 2-4 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 2-4 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 and 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 以及其一系列的 ElasticSearch 与 Solr。

2.2.2 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。

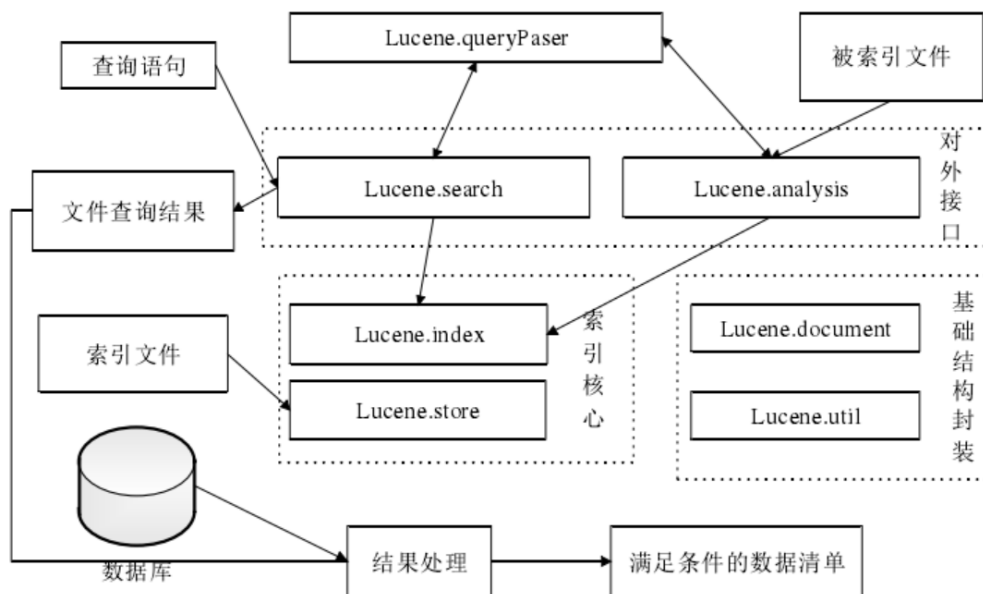


图 2-5 lucence 代码结构

Lucene 的代码结构大致如图 2-5 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。

2.2.3 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。如图 2-6 所示。

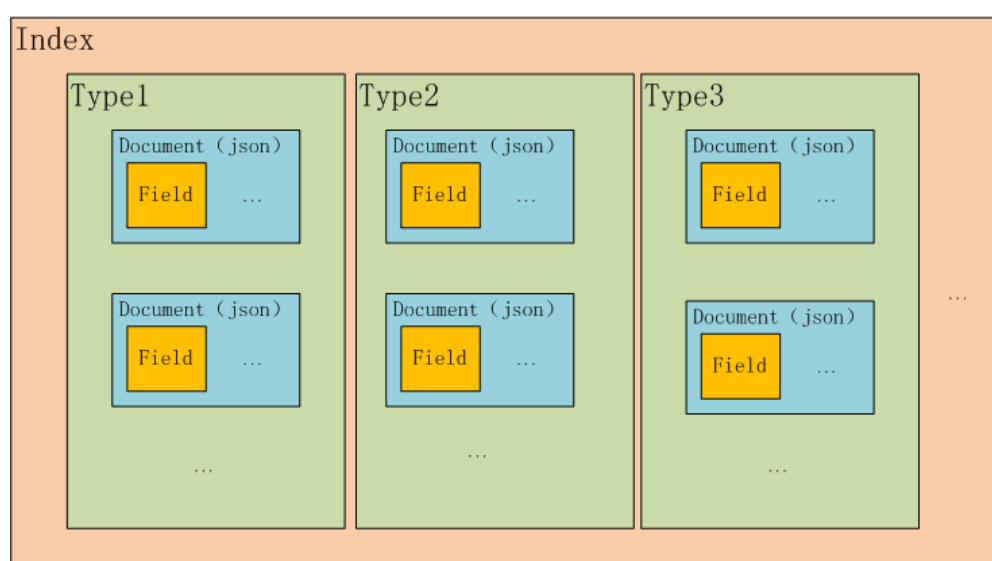


图 2-6 Elasticsearch 存储逻辑

- 4) 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 2-7 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, 其语法格式如图 2-8:

```
curl -X<REST Verb> <Node>:<Port>/<Index>/<Type>/<ID>
```

图 2-8

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：

$$G = (V, E)$$

$$V = \text{vertex(节点)}$$

$$E = \text{edge(边)}$$

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，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：

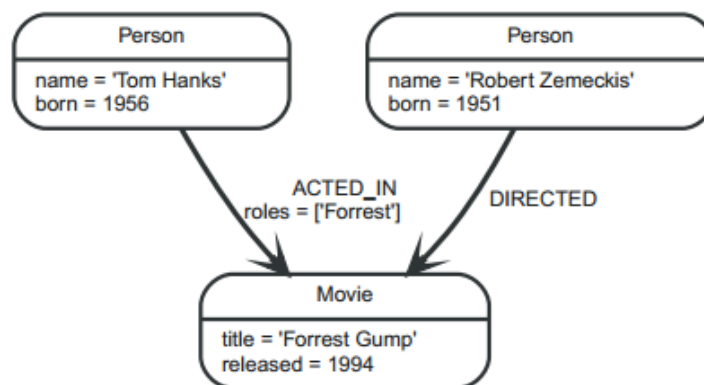


图 2-9 简单图模型

图 2-9 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。

2) 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 2-9 的模型，如果要查找 Tom Hanks 演过的电影，Cypher 语句可以写为如图 2-10 形式：

```
match (n:{name:"Tom Hanks"}) - [:ACTED_IN] -> (m)
return m
```

图 2-10

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。

3) 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 语法如图 2-11：

```
create index on :Label(Property)
```

图 2-11

其中的 Label 和 Property 分别对应具体的标签和属性。

第3章 系统总体架构

本文在上一章中已经介绍了 IFC 标准和 IFC 文件数据的存储格式，以及 ElasticSearch 和 Neo4j 的技术基础。本文在深入研究了 IFC 标准的基础上，提出基于 ElasticSearch 弹性搜索框架来实现 IFC 属性数据的存储和查询，同时，基于 Neo4j 图数据库来实现 IFC 实体空间关系信息的存储和查询。

接下来的章节会详细地介绍两个系统在设计 and 实现时涉及到的问题和方案。但在这之前，需要先介绍一下本实验系统的总体架构。

3.1 总体架构设计

按照数据处理过程，该系统的模块可以分为：IFC 文件解析模块、数据解析、数据索引、数据查询等模块。结合 ElasticSearch 搜索框架和 Neo4j 图数据库，其总体架构如下图 3-1：

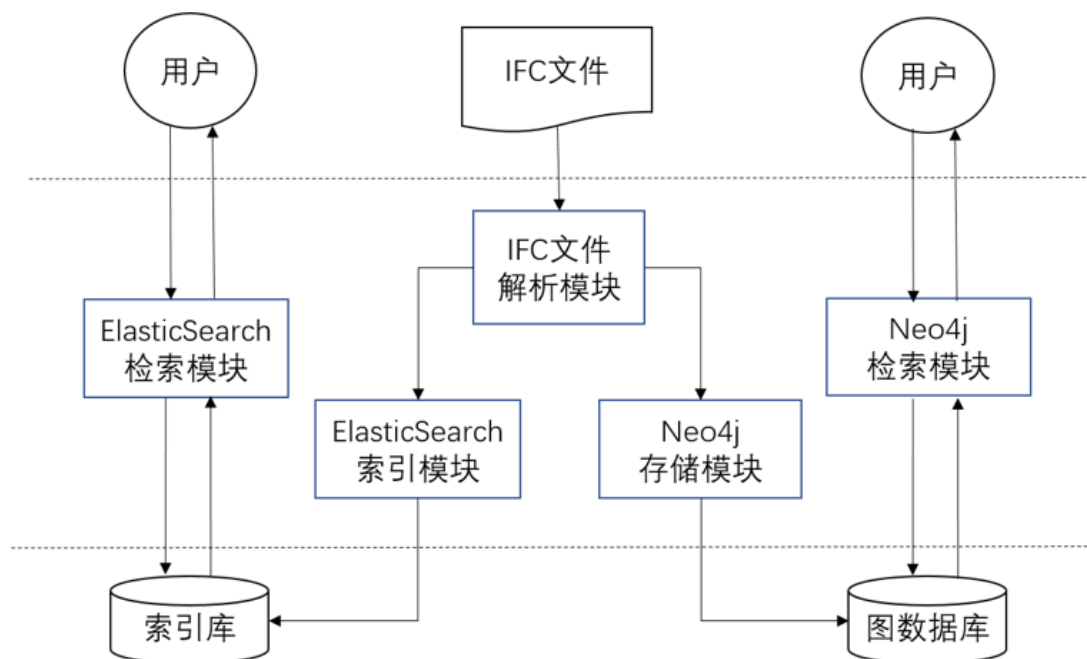


图 3-1 系统总体架构

系统中各个模块的执行流程是：

- 1) 使用 IFC 文件作为数据来源，上传 IFC 文件到系统。
- 2) 文件解析模块将解析 IFC 文件为内存中的 IFC 对象数据集合。数据包括属性信息和空间关系信息。
- 3) ElasticSearch 索引模块将获取到的属性信息索引到集群中的索引库中去。
- 4) Neo4j 存储模块将获取到的空间关系数据存储到图数据库当中去。

- 5) Elasticsearch 和 Neo4j 的检索模块各自接受用户的查询请求,返回结果。

3.2 子模块设计要点

- 1) IFC 文件解析模块需要将 IFC 文件的每行数据解析成对应的 Java 类对象。根据 Java 对象的数据获取属性数据和空间关系数据。由于数据需要转换为 JSON 文档的格式,所以需要定义和实现 Java 对象数据到 JSON 数据的映射。同样对于 Neo4j 来讲,需要定义和实现 Java 对象数据到 Node 节点数据的映射。
- 2) Elasticsearch 索引模块需要支持批量插入的方式以及使用优化配置,以提高索引效率。
- 3) Neo4j 存储模块需要支持批量插入的方式,以提高存储效率,同时需要为所有的节点创建索引,以提高查询效率。
- 4) 用户需要通过可视化前端页面来访问 Elasticsearch 和 Neo4j,安装现有的插件可以使用户通过前端页面的交互访问 Elasticsearch 集群和 Neo4j 图数据库。
- 5) Neo4j 的查询方面,要实现基于 BloomFilter 的特定路径查询优化。这涉及到 BloomFilter 的构建、优化算法的实现。

第4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，以及如何将基于 Elasticsearch 来实现属性数据的索引和查询。

4.1 IFC 对象的属性数据模型概述

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` 的定义如图 4-1

```
ENTITY IfcProperty;  
  ENTITY IfcProperty;  
    Name : IfcIdentifier;  
    Description : OPTIONAL IfcText;  
  INVERSE  
    PropertyForDependance : SET OF IfcPropertyDependencyRelationship FOR DependingProperty;  
    PropertyDependsOn : SET OF IfcPropertyDependencyRelationship FOR DependantProperty;  
    PartOfComplex : SET [0:1] OF IfcComplexProperty FOR HasProperties;  
END_ENTITY;
```

图 4-1 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，它的派生子类如图 4-2

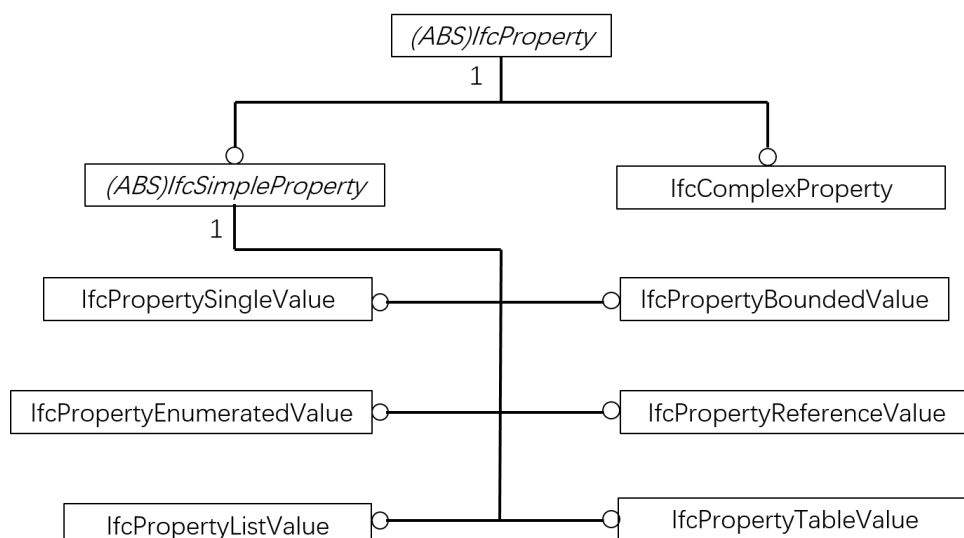


图 4-2Property 派生子类图

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，它的定义如图 4-3 所示

```
ENTITY IfcPropertySingleValue;  
  ENTITY IfcProperty;  
    Name : IfcIdentifier;  
    Description : OPTIONAL IfcText;  
  INVERSE  
    PropertyForDependence : SET OF IfcPropertyDependencyRelationship FOR DependingProperty;  
    PropertyDependsOn : SET OF IfcPropertyDependencyRelationship FOR DependantProperty;  
    PartOfComplex : SET [0:1] OF IfcComplexProperty FOR HasProperties;  
  ENTITY IfcSimpleProperty;  
  ENTITY IfcPropertySingleValue;  
    NominalValue : OPTIONAL IfcValue;  
    Unit : OPTIONAL IfcUnit;  
END_ENTITY;
```

图 4-3

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，它的定义如图 4-4

```
ENTITY IfcPropertyListValue;  
  ENTITY IfcProperty;  
    Name : IfcIdentifier;  
    Description : OPTIONAL IfcText;  
  INVERSE  
    PropertyForDependence : SET OF IfcPropertyDependencyRelationship FOR DependingProperty;  
    PropertyDependsOn : SET OF IfcPropertyDependencyRelationship FOR DependantProperty;  
    PartOfComplex : SET [0:1] OF IfcComplexProperty FOR HasProperties;  
  ENTITY IfcSimpleProperty;  
  ENTITY IfcPropertyListValue;  
    ListValues : LIST [1:?] OF IfcValue;  
    Unit : OPTIONAL IfcUnit;  
END_ENTITY;
```

图 4-4

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，在此不一一介绍。

相对于 IfcSimpleProperty 的简单属性类型而言，IfcComplexProperty 是复杂属性。它的定义如图 4-5 所示

```
ENTITY IfcComplexProperty;  
  ENTITY IfcProperty;  
    Name : IfcIdentifier;  
    Description : OPTIONAL IfcText;  
  INVERSE  
    PropertyForDependence : SET OF IfcPropertyDependencyRelationship FOR DependingProperty;  
    PropertyDependsOn : SET OF IfcPropertyDependencyRelationship FOR DependantProperty;  
    PartOfComplex : SET [0:1] OF IfcComplexProperty FOR HasProperties;  
  ENTITY IfcComplexProperty;  
    UsageName : IfcIdentifier;  
    HasProperties : SET [1:?] OF IfcProperty;  
END_ENTITY;
```

图 4-5

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`。它的派生子类图如图 4-6:

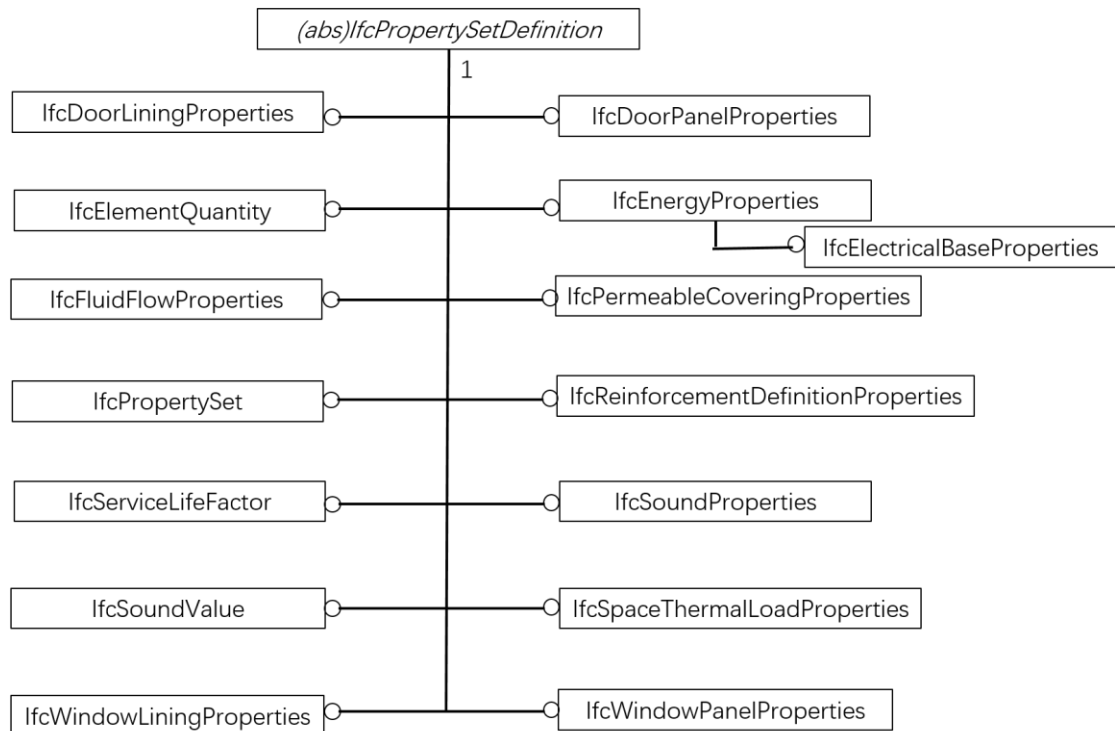


图 4-6 `IfcPropertySetDefinition` 派生类图

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` 的定义如图 4-7:

```

ENTITY IfcDoorLiningProperties;
  ENTITY IfcRoot;
    GlobalId                : IfcGloballyUniqueId;
    OwnerHistory             : IfcOwnerHistory;
    Name                     : OPTIONAL IfcLabel;
    Description              : OPTIONAL IfcText;
  ENTITY IfcPropertyDefinition;
  INVERSE
    HasAssociations         : SET OF IfcRelAssociates FOR RelatedObjects;
  ENTITY IfcPropertySetDefinition;
  INVERSE
    PropertyDefinitionOf    : SET [0:1] OF IfcRelDefinesByProperties FOR RelatingPropertyDefinition;
    DefinesType             : SET [0:1] OF IfcTypeObject FOR HasPropertySets;
  ENTITY IfcDoorLiningProperties;
    LiningDepth             : OPTIONAL IfcPositiveLengthMeasure;
    LiningThickness         : OPTIONAL IfcPositiveLengthMeasure;
    ThresholdDepth          : OPTIONAL IfcPositiveLengthMeasure;
    ThresholdThickness      : OPTIONAL IfcPositiveLengthMeasure;
    TransomThickness        : OPTIONAL IfcPositiveLengthMeasure;
    TransomOffset           : OPTIONAL IfcLengthMeasure;
    LiningOffset            : OPTIONAL IfcLengthMeasure;
    ThresholdOffset         : OPTIONAL IfcLengthMeasure;
    CasingThickness         : OPTIONAL IfcPositiveLengthMeasure;
    CasingDepth             : OPTIONAL IfcPositiveLengthMeasure;
    ShapeAspectStyle        : OPTIONAL IfcShapeAspect;
END_ENTITY;
  
```

图 4-7

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。它的定义如图 4-8：

```

ENTITY IfcPropertySet;
  ENTITY IfcRoot;
    GlobalId                : IfcGloballyUniqueId;
    OwnerHistory             : IfcOwnerHistory;
    Name                    : OPTIONAL IfcLabel;
    Description              : OPTIONAL IfcText;
  ENTITY IfcPropertyDefinition;
  INVERSE
    HasAssociations         : SET OF IfcRelAssociates FOR RelatedObjects;
  ENTITY IfcPropertySetDefinition;
  INVERSE
    PropertyDefinitionOf    : SET [0:1] OF IfcRelDefinesByProperties FOR RelatingPropertyDefinition;
    DefinesType             : SET [0:1] OF IfcTypeObject FOR HasPropertySets;
  ENTITY IfcPropertySet;
    HasProperties            : SET [1:?] OF IfcProperty;
END_ENTITY;

```

图 4-8

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 4-9 所示：

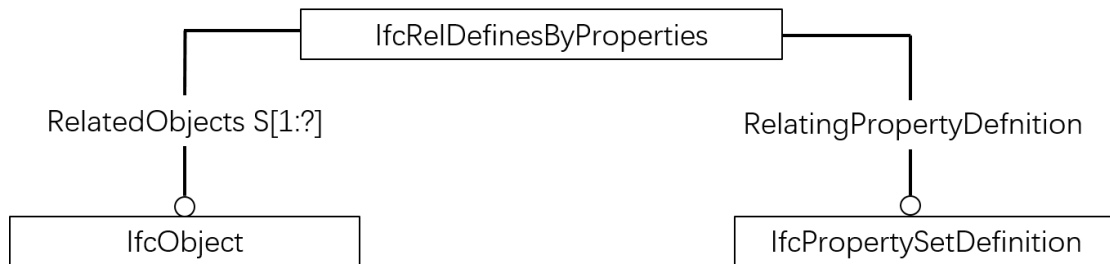


图 4-9 IfcRelDefinesByProperties 关联机制

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；

举例 IfcWallStandardCase 实例如下：

```
#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。

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 的定义如图 4-10:

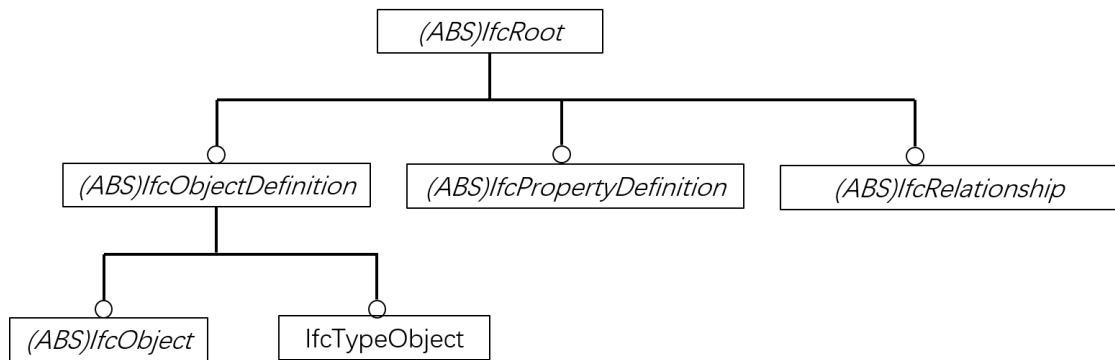
```
ENTITY IfcRoot;  
  ENTITY IfcRoot;  
    GlobalId                : IfcGloballyUniqueId;  
    OwnerHistory             : IfcOwnerHistory;  
    Name                    : OPTIONAL IfcLabel;  
    Description              : OPTIONAL IfcText;  
  END_ENTITY;
```

图 4-10

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 4-11:

图 4-11 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，如图 4-12:

```
1  {
2
3      ...
4      "PropertySets": [
5          {
6              "GlobalId": "",
7              "OwnerHistory": "",
8              "Name": "",
9              "Description": "",
10             "HasProperties": {
11                 "IfcPropertySingleValue_1": {},
12                 "IfcPropertySingleValue_2": {},
13                 "IfcPropertyBoundedValue_3": {}
14             }
15         }
16     ]
17 }
```

图 4-12

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。

- 2) 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。如图 4-13:



```
1 {
2   ...
3   "标识数据": {
4     "GlobalId": "",
5     "Description": "",
6     "OwnerHistory": "",
7     "标高": {
8       "Description": "", "NominalValue": "", "Unit": ""
9     },
10    "偏移量": {
11      "Description": "", "NominalValue": "", "Unit": ""
12    }
13  }
14 }
```

图 4-13

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属性。

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 格式如图 4-14:

```

1  {
2      ...
3      "标识数据":{
4          "标高":{
5              "Description":"","NominalValue":"","Unit":""
6          },
7          "偏移量":{
8              "Description":"","NominalValue":"","Unit":""
9          },
10     },
11     "Pset_XXXXCommon":{
12         "CO2SensorRange":{
13             "Description":"","UpperBound":"","LowerBoundValue":"","Unit":""
14         },
15         "AgreementType":{
16             "Description":"","EnumerationValues":[],"EnumerationReference":""
17         },
18         "PreventiveMeasures":{
19             "Description":"","ListValues":"","Unit":""
20         },
21         "WallMaterial":{
22             "Description":"","UsageName":"","PropertyReference":""
23         },
24         "Spectrum":{
25             "Description":"","DefiningValues":[],"DefinedValues":[],
26             "Expression":"","DefiningUnit":"","DefinedUnit":""
27         }
28     },
29     "Others":{
30         "完工时间":""
31     }
32 }

```

图 4-14 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。

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。
- 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。

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。图 4-15 是一个 mapping 示例：

```
{
  "settings": {
    "number_of_shards": 5,
    "number_of_replicas": 1,
  },
  "mappings": {
    "IfcWall": {
      "_all": {
        "analyzer": "ike"
      },
      "_source": {
        "compress": true
      },
      "properties": {
        "GlobalId": {
          "type": "keyword",
          "index": true
        },
      },
    },
  },
}
```

图 4-15

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 会为你创建图 4-16 的映射：

```
{
  "foo": {
    "type": "text",
    "fields": {
      "keyword": {
        "type": "keyword",
        "ignore_above": 256
      }
    }
  }
}
```

图 4-16

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。

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 的语法格式如图 4-17：

```
{action:{metadata}}\n
{request body}\n
{action:{metadata}}\n
{request body}\n
...
```

图 4-17

需要注意两点：每行一定要以换行符（\n）结尾，包括最后一行；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。Metadata 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：

```
{"index":{"_index":"bim","_id":"123"}}
{"GlobalId":"1Az$ukFWP4dw7vzE_21JL3","OwnerHistory":"2"}
```

使用 bulk 的 REST API 的命令形式如下：

```
curl -XPOST localhost:9200/bim/IfcWall/_bulk?pretty --data-binary @IfcWall.json
```

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，如图 4-18：

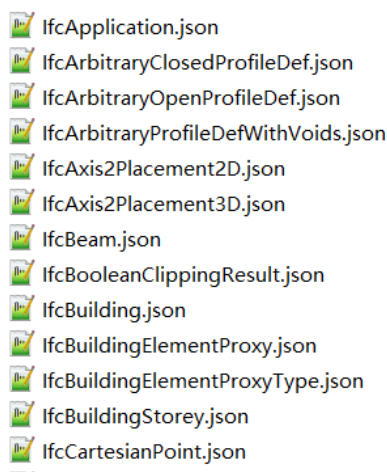
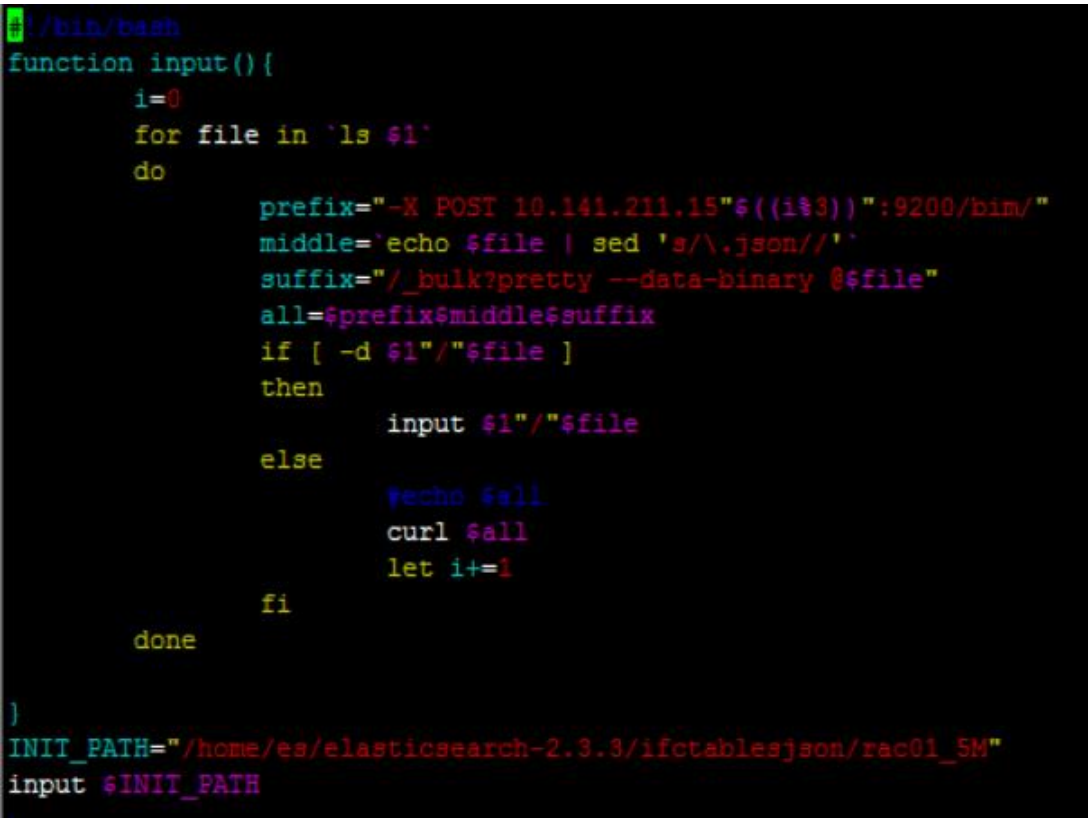


图 4-18

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 如图 4-19:



```
#!/bin/bash
function input() {
    i=0
    for file in `ls $1`
    do
        prefix="-X POST 10.141.211.15"`${i%3}`":9200/bim/"
        middle=`echo $file | sed 's/\.json//`
        suffix="/_bulk?pretty --data-binary @$file"
        all=$prefix$middle$suffix
        if [ -d $1/"$file ]
        then
            input $1/"$file
        else
            echo $all
            curl $all
            let i+=1
        fi
    done
}
INIT_PATH="/home/es/elasticsearch-2.3.3/ifctablesjson/rac01_5M"
input $INIT_PATH
```

图 4-19

4.3.3 索引过程优化

在上小节已经对 bulk 对索引性能带来的影响进行了讨论。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：

```
export ES_HEAP_SIZE=10g
```

- 2) 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：

```
curl -XPUT localhost:9200/bim/_settings -d '{
  "index":{
    "refresh_interval":"60s"
  }
}'
```

- 3) 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 在索引时会将会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：

```
curl -XPUT localhost:9200/bim/_mapping -d '{
  "settings":{
    "index":{
      "translog":{
        "flush_threshold_ops":100000
      }
    }
  }
}'
```

- 4) 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：

```
curl -XPUT localhost:9200/bim/_mapping -d '{
  "mappings":{
    "_all":{
      "enabled":"false"
    }
  }
}'
```

- 5) 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：

```
indices.memory.index_buffer_size: 50%
```

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。

举例如图 4-20:

```
curl -XPOST localhost:9200/company/employee/_search?pretty -d '{
  "query":{
    "constant_score":{
      "filter":{
        "term":{
          "age":25
        }
      }
    }
  }
}'
```

图 4-20

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图 4-21 所示：

```
"hits": [
  {
    "_index" : "company",
    "_type" : "employee",
    "_id" : "2",
    "_score" : 1.0,
    "_source" : {
      "name" : "Lily Clarkson",
      "age" : 25
    }
  }
]
```

图 4-21

在应用当中，应尽量使用 filter 查询。

2) 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)

举例如图 4-22 所示：

```
curl -XPOST localhost:9200/company/employee/_search?pretty -d '{
  "query":{
    "match":{
      "name":"Lily"
    }
  }
}'
```

图 4-22

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图 4-23：

```
"hits": [
  {
    "_id": "1",
    "_score": 0.8,
    "_source": {
      "name": "Lily Clarkson"
    }
  },
  {
    "_id": "3",
    "_score": 0.44194174,
    "_source": {
      "name": "Lily Allen"
    }
  },
  {
    "_id": "2",
    "_score": 0.1025,
    "_source": {
      "name": "lilyah"
    }
  }
]
```

图 4-23

第5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 实体空间关系数据模型

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 实体模型的整体框架，不仅包括 IFC 实体的位置和几何形状等，同时也定义了 IFC 实体之间的关系。这些关系表现为空间结构上的组成、包含等多种关系，与 IFC 实体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。

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 实体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 实体之间的空间结构关系。它们的定义如下：

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。

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。

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。

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。

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 实体的空间关系，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$93HI7',#41,'\X2\57FA672C5899\X0\CW 102-50-215p:309591',$,'\X2\57FA672C5899\X0\CW 102-50-215p:412',#124,#151,'309591');
```

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

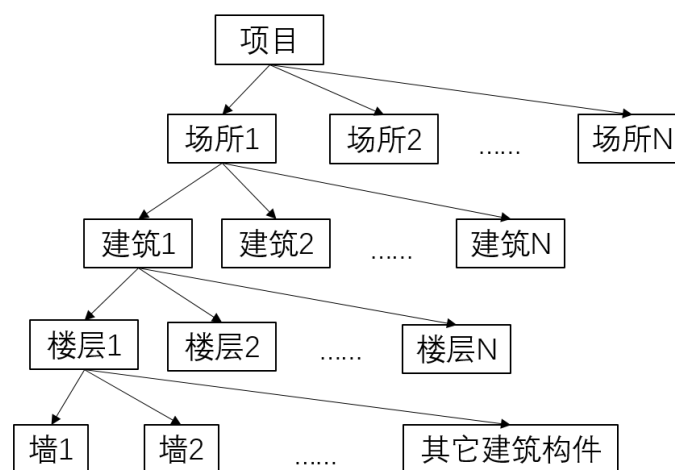
#94 代表项目，#592 代表场地，#679 代表项目（#94）“包含”场地（#592）。#104 代表建筑，#683 代表场地（#592）包含建筑（#104）。#113 代表楼层，#687 代表建筑（#104）包含楼层（#113）。（#155）代表墙，#367、#430、#550 代表其它建筑构件，#672 代表楼层（#113）包含这四个建筑构件。

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 实体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 5-1。

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，则对应的是图 5-2。

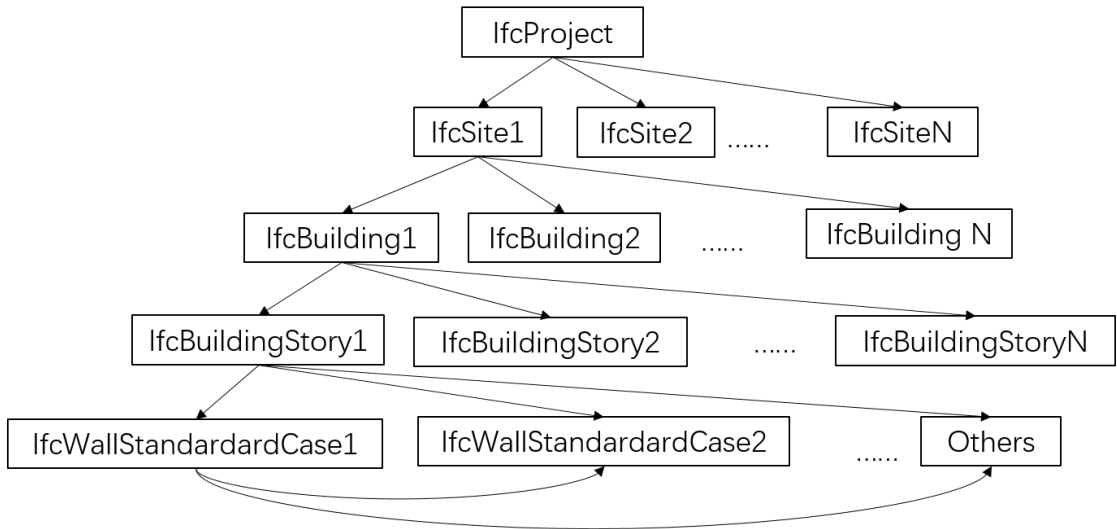
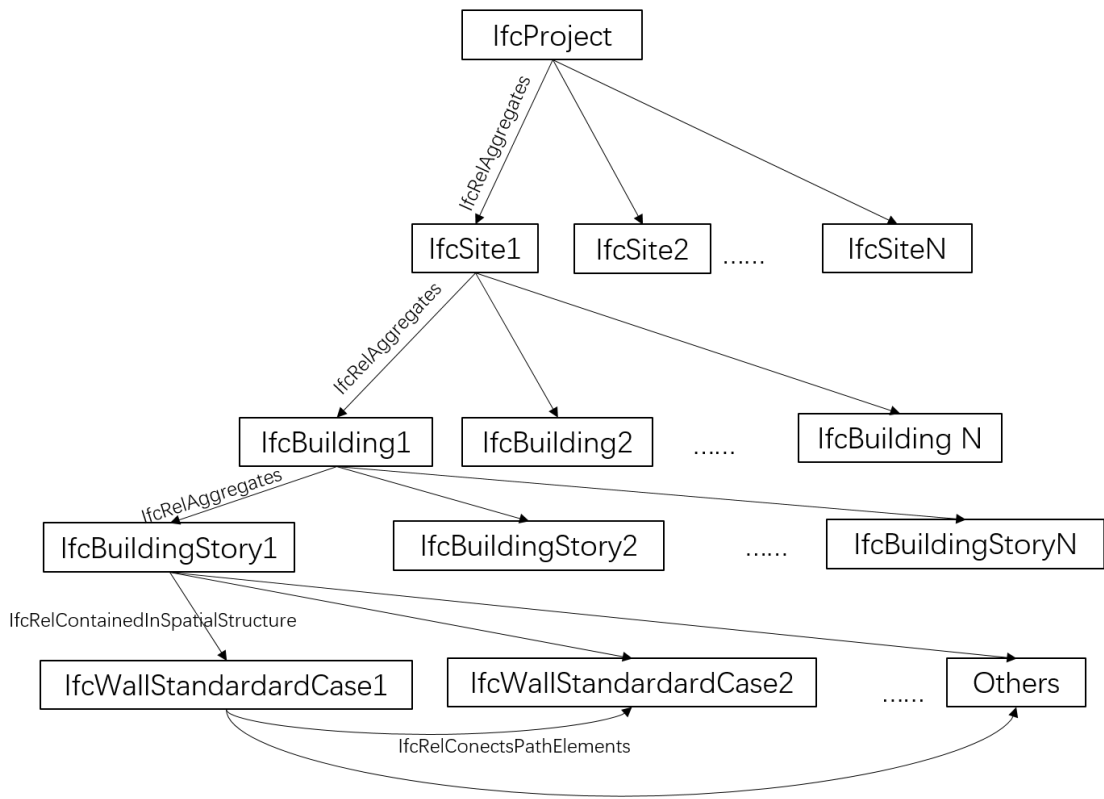


图 5-2 IFC 对象的空间关系示意图

在上图中我们用一条带箭头的连线来代表 IFC 实体之间的空间关系。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

图 5-3 IFC 实体的空间关系数据模型

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 实体的空间关系模型，如图 5-3。为了简洁，只标出了部分关系。

图 5-3 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 5-3 中所展示的 IFC 实体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 实体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 实体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，如图 5-4。

```

public IfcProject(IfcGloballyUniqueId GlobalId, IfcOwnerHistory OwnerHistory, IfcLabel Name, IfcText Des:
{
    this.GlobalId = GlobalId;
    this.OwnerHistory = OwnerHistory;
    this.Name = Name;
    this.Description = Description;
    this.ObjectType = ObjectType;
    this.LongName = LongName;
    this.Phase = Phase;
    this.RepresentationContexts = RepresentationContexts;
    this.UnitsInContext = UnitsInContext;
    resolveInverses();
}

public IfcSite(IfcGloballyUniqueId GlobalId, IfcOwnerHistory OwnerHistory, IfcLabel Name, IfcText Description,
{
    this.GlobalId = GlobalId;
    this.OwnerHistory = OwnerHistory;
    this.Name = Name;
    this.Description = Description;
    this.ObjectType = ObjectType;
    this.ObjectPlacement = ObjectPlacement;
    this.Representation = Representation;
    this.LongName = LongName;
    this.CompositionType = CompositionType;
    this.Reflatitude = RefLatitude;
    this.Reflongitude = RefLongitude;
    this.Refelevation = RefElevation;
    this.LandTitleNumber = LandTitleNumber;
    this.SiteAddress = SiteAddress;
    resolveInverses();
}

public IfcRelAggregates(IfcGloballyUniqueId GlobalId, IfcOwnerHistory OwnerHistory, IfcLabel Name,
{
    this.GlobalId = GlobalId;
    this.OwnerHistory = OwnerHistory;
    this.Name = Name;
    this.Description = Description;
    this.RelatingObject = RelatingObject;
    this.RelatedObjects = RelatedObjects;
    resolveInverses();
}

```

图 5-4

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，如图 5-5:

```

private static final String[] attributes = new String[]{"GlobalId", "OwnerHistory", "Name", "Description", "RelatingObject", "RelatedObjects"};

String[] getAttributes() {
    return IfcRelAggregates.attributes;
}

```

图 5-5

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键值对。

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 Relationship 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 relatingObject 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。

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：

```

BatchInserter inserter = BatchInserters.inserter(new File(dbPath));

for(ClassInterface ifcObject : ifcObjects){
    InternalAccessClass object = (InternalAccessClass) ifcObject;
    if (object instanceof IfcRelAggregates){
        //处理IfcRelAggregates的RelatingObject和RelatedObjects
        ArrayList<CloneableObject> parameters = InternalAccess.getStepParameter((IfcRelAggregates)object);
        IfcObject relatingObject = (IfcObject) parameters.get(4);
        Set<IfcObject> relatedObjects = (Set<IfcObject>) parameters.get(5);
        //检查是否已插入, 若没有, 获取relatingObject的属性数据, 放入Map<String, Object> nodeMap中, 省略代码
        //使用BatchInserter插入relatingObject node
        inserter.createNode(relatingObjectId, nodeMap, label);
        //遍历relatedObjects, 检查是否已插入, 若没有, 获取其属性数据, 放入Map<String, Object> nodeMap中, 省略代码
        //每次遍历, 插入relatedObject node, 并且创建一个relatingObject->relatedObject的关系
        inserter.createNode(id, nodeMap, label);
        inserter.createRelationship(relatingObjectId, relatedObjectsObjectId, RelTypes.IfRelAggregates, relMap);
        //
    } else if(object instanceof IfcRelNests) {
        //处理IfcRelNests的RelatingObject和RelatedObjects, 省略代码
    }
}

```

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 attributes 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:

```

for(ClassInterface ifcObject : ifcObjects){
    InternalAccessClass object = (InternalAccessClass) ifcObject;
    if(object instanceof IfcRelConnects || object instanceof IfcRelDecomposes){
        ArrayList<CloneableObject> parameters = InternalAccess.getStepParameter(object);
        int relatingIndex, relatedIndex, found;
        String[] attributes = object.getAttributes();
        //遍历attributes, 获取对象的RelatingObject和RelatedObject字段的index, 省略代码
        IfcObject relatingObject = (IfcObject) parameters.get(relatingIndex);
        Set<IfcObject> relatedObjects = (Set<IfcObject>) parameters.get(relatedIndex);
        //以relatingObject为例. 利用反射机制, 基于字段名反射出方法名, 获取属性值
        String[] relatingAttributes = relatingObject.getAttributes();
        for(String attribute : relatingAttributes){
            Class c = relatingObject.getClass();
            Method m = c.getMethod("get" + attribute + "()", null);
            Object value = m.invoke(relatingObject, null);
            //将<attribute,value>c存入nodeMap中
            //创建node, 省略代码。
        }
        //省略其它代码
    }
}

```

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。

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, 房间与门的关系如图 5-6:



图 5-6

于是其查询逻辑使用 Cypher 如下表达式:

```
match (n:IfcSpace) - [r1:IfcRelContainedInSpatialStructure] -> (m:IfcDoor{OverallHeight:"2.2"})
return n
```

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:

```
create index on :IfcDoor(OverallHeight)
```

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。
- 4) 对于 Cypher 语言的执行, 使用嵌入式 Neo4j 的 GraphDatabaseService

API, 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。

```
public Result cypherQuery(GraphDatabaseService graphDatabaseService, String query){
    //...
    result = graphDatabaseService.execute(query8);
    return result;
}
```

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 语句的简单形式如图 5-7:

```
match (n1:IfcSite) - [r1:IfcRelAggregates] -> (n2:IfcBuilding)
      - [r2:IfcRelAggregates] -> (n3:IfcBuildingStory)
      - [r3:IfcRelContainedInSpatialStructure] -> (n4:IfcDoor)
return n1,n2,n3,n4
```

图 5-7

又由 IFC 标准的定义可知, r1、r2、r3 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 表达式应为图 5-8:

```
match (n1:IfcSite{name:"zhangjiang"})
      - [] -> (n2:IfcBuilding{name:"officebuilding"})
      - [] -> (n2:IfcBuildingStory{name:"secondfloor"})
      - [] -> (n4:IfcDoor:{OverallHeight:"2.2"})
return n1,n2,n3,n4
```

图 5-8

这个 Cypher 查询语句的完整的执行计划如图 5-9:

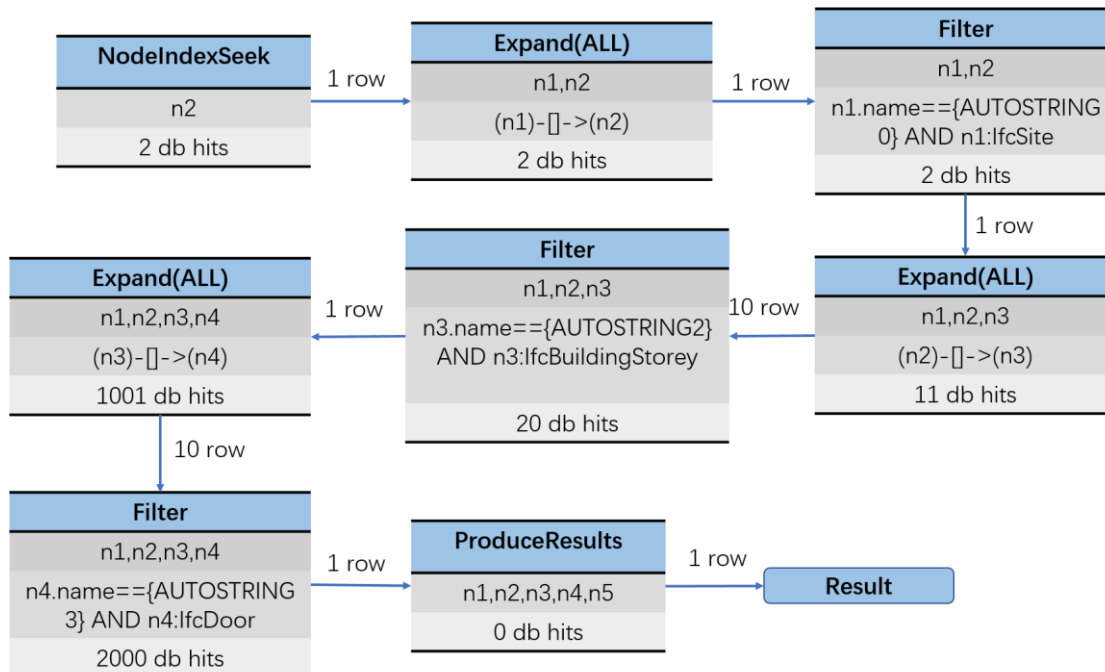


图 5-9

- 1) 执行 NodeIndexSeek。访问索引 “index on :IfcBuilding(name)”，返回节点 n2。在这里假设只有一个 IfcBuilding 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 IfcBuilding 只有一个 IfcSite 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 IfcSite 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 BloomFilter 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。

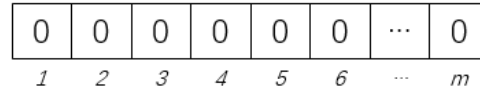
对于 4.6.1 小节中提到的特定的路径查询来说，如果我们能够把符合空间组成关系的 IfcSite、IfcBuilding、IfcBuildingStory、IfcDoor（以及其它建筑构件）的 ID 按照顺序拼接起来，维护在 BloomFilter 当中。当查询来到时，将满足条件的各个节点的 ID 拼接起来，查看是否在 BloomFilter 当中，就可以知道，这个结果集是否存在。BloomFilter 存在一定的误判。也就是说，如果这条路径并不在 neo4j 存储的图中，但是也有可能判定为存在，此时用户得到的结果将会是由原始的 Cypher 查询得到的正确结果。同时，BloomFilter 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：

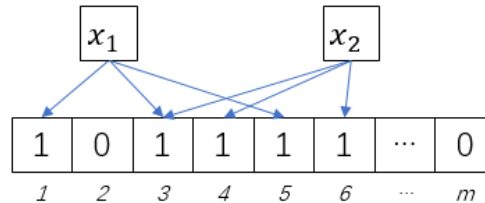


2) 基于 K 个独立 Hash 函数, 添加元素:

待添加的元素设为 $S = \{x_1, x_2, \dots, x_n\}$ 。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 $\{1, 2, \dots, m\}$ 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, $k=3$, $hash_1(x_1) = 1$, $hash_2(x_1) = 3$, $hash_3(x_1) = 5$,
 $hash_1(x_2) = 3$, $hash_2(x_2) = 4$, $hash_3(x_2) = 6$.

加入 x_1, x_2 , 则 bit 数组为:



3) 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。

4) 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 $k = (\ln 2) * (m/n)$ 时错误率最小。在错误率不大于 E 的情况下， m 至少要等于 $n * \lg(1/E)$ 才能表示任意 n 个元素的集合。

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 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> bdgToSite;
存储 IfcBuilding 的 ID 与其所在的 IfcSite 的 ID 的映射。
- 2) Map<Long, Long> storyToBdg;
存储 IfcBuildingStory 的 ID 与其所在的 IfcBuilding 的 ID 的映射。
- 3) Map<Long, Long> elmtToStory
存储 IFC 实体的 ID 与其所在的 IfcBuildingStory 的 ID 的映射。

基于这 3 个数据结构，可以构建一个 IFC 对象 getStoryID()、getBdgID()、getSiteID()，得出 IFC 对象所对应的 IfcBuildingStory、IfcBuilding、IfcSite 的 ID。然后按照从 Ifcproject 到 IFC 实体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。

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 查询分解为针对节点的 Cypher 查询。原始的查询格式比较固定，会是如下这种情况：

```
queryStr = "match (n:IfcObject1{name:\\"name\" + j + \"\"})"
+ " - [r1] -> " + "(m:IfcObject2{name:\\"name\" + String.valueOf(j*10 + k) + \"\"})"
+ " - [r2] -> " + "(s:IfcObject7{name:\\"name\" + String.valueOf((j*10 + k)*1000 + t + 2000000) + \"\"})"
+ " return id(n),id(m),id(s)";
```

而分解后的查询会是如下这种情况：

```
bfQueryStr = "match (n:IfcObject1{name:\\"name\" + j + \"\"}) return id(n)";
bfQueryStr = "match (m:IfcObject2{name:\\"name\" + String.valueOf(j*10 + k) + \"\"}) return id(m)";
bfQueryStr = "match (s:IfcObject7{name:\\"name\" + String.valueOf((j*10 + k)*1000 + t + 2000000) + \"\"}) return id(s)";
```

- 2) 分别执行以上单个针对节点的查询，也就是 3 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；
- 3) 如果结果集不为空，则将节点的 ID 字段拼接为一个字符串。

```
if (myResult.hasNext()){
    composedId.append(myResult.next().get(keyList.get(0)));
}
```

- 4) 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。

```
if (bloomFilter.mightContain(composedId.toString())){
```

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。

以上便是基于 BloomFilter 进行特定路径查询的算法及相关实现。

第6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：

6.1.1 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) E5649 2.53GHz	10G	194G	CentOS 6.7
10.141.211.151	Intel(R)Xeon(R) E5649 2.53GHz	14G	220G	CentOS 6.7
10.141.211.152	Intel(R)Xeon(R) E5649 2.53GHz	13G	198G	CentOS 6.7
10.141.211.153	Intel(R)Xeon(R) 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。











6.1.2 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：

 IfcApplication.json
 IfcArbitraryClosedProfileDef.json
 IfcArbitraryOpenProfileDef.json
 IfcArbitraryProfileDefWithVoids.json
 IfcAxis2Placement2D.json
 IfcAxis2Placement3D.json
 IfcBeam.json
 IfcBooleanClippingResult.json
 IfcBuilding.json
 IfcBuildingElementProxy.json

.....

 IfcSystem.json
 IfcTrimmedCurve.json
 IfcUnitAssignment.json
 IfcValveType.json
 IfcWall.json
 IfcWallStandardCase.json
 IfcWallType.json
 IfcWindow.json
 IfcWindowLiningProperties.json
 IfcWindowStyle.json

下图是解析后的 IfcWallStandard.json 的示例：

```

1  {"index":{"_id":"4470"}}
2  {"GlobalId":"3Vy8PGoKz04OWHEEI9LTtT","OwnerHistory":41,"Name":"基本墙:外墙页岩空
3  {"update":{"_id":"4470"}}
4  {"doc":{"其他":{"族":{"Description":"","NominalValue":"基本墙: 外墙页岩空心砖-2
5  {"update":{"_id":"4470"}}
6  {"doc":{"尺寸标注":{"长度":{"Description":"","NominalValue":11399.9999999697,"U
7  {"update":{"_id":"4470"}}
8  {"doc":{"结构":{"结构用途":{"Description":"","NominalValue":"非承重","Unit":""}
9  {"update":{"_id":"4470"}}
10 {"doc":{"阶段化":{"创建的阶段":{"Description":"","NominalValue":"阶段2-改造","U
11 {"update":{"_id":"4470"}}
12 {"doc":{"限制条件":{"无连接高度":{"Description":"","NominalValue":2700.0,"Unit"
13 {"update":{"_id":"4470"}}
14 {"doc":{"Pset_WallCommon":{"LoadBearing":{"Description":"","NominalValue":false
15 {"index":{"_id":"4631"}}
16 {"GlobalId":"3Vy8PGoKz04OWHEEI9LTP0","OwnerHistory":41,"Name":"基本墙:外墙页岩空
17 {"update":{"_id":"4631"}}
18 {"doc":{"其他":{"族":{"Description":"","NominalValue":"基本墙: 外墙页岩空心砖-2
  
```

下图是解析后的 IfcWallType.json 的示例：



图 6-1

ElasticSearch 自动 mapping 的部分结果如图 6-2:



图 6-2

6.1.4 查询测试

在 ElasticSearch 的 head 插件界面进行查询测试如图 6-3, 可以看到, 集群能够以 ms 级别的速度来响应查询。

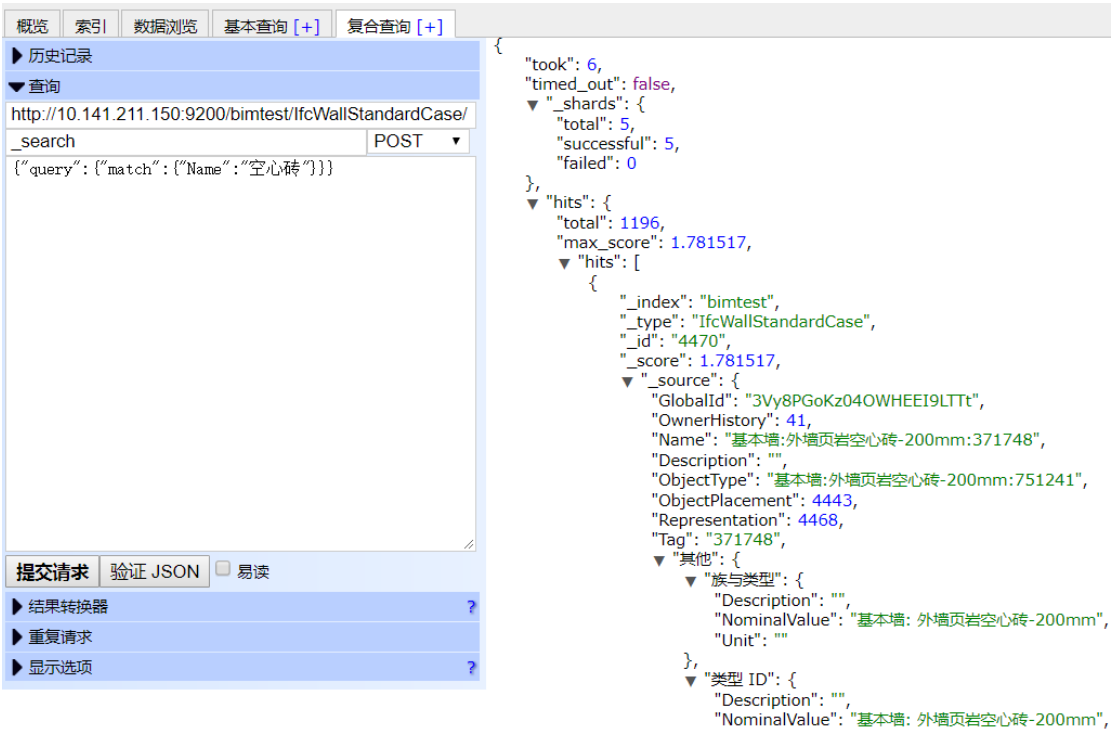


图 6-3

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。

6.2.1 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。实验环境如下：

运行环境	CPU	MEM	OS
个人本地 PC	Intel(R)CORE i5	8G	Win10

6.2.2 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：

ObjectType	null
Elevation	0
Description	null
LongName	标高 1
ObjectPlacement	111
Representation	null
OwnerHistory	41
CompositionType	IFCELEMENTCOMPOSITIONENUM(.ELEMENT.)
GlobalId	0DVPyye2z0deMtQa2CjRCZ
Name	标高 1

如下是一个 IfcRelConnectsElements 关系类型的数据：

RelatedPriorities	[
RelatingConnectionType	IFCCONNECTIONTYPEENUM(.ATEND.)
Description	null
ConnectionGeometry	null
RelatedConnectionType	IFCCONNECTIONTYPEENUM(.ATSTART.)
OwnerHistory	41
RelatedElement	430
RelatingPriorities	[
RelatingElement	367
GlobalId	08ooPLcu5AFQjtluyTPKkL
Name	null

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如图 6-4：

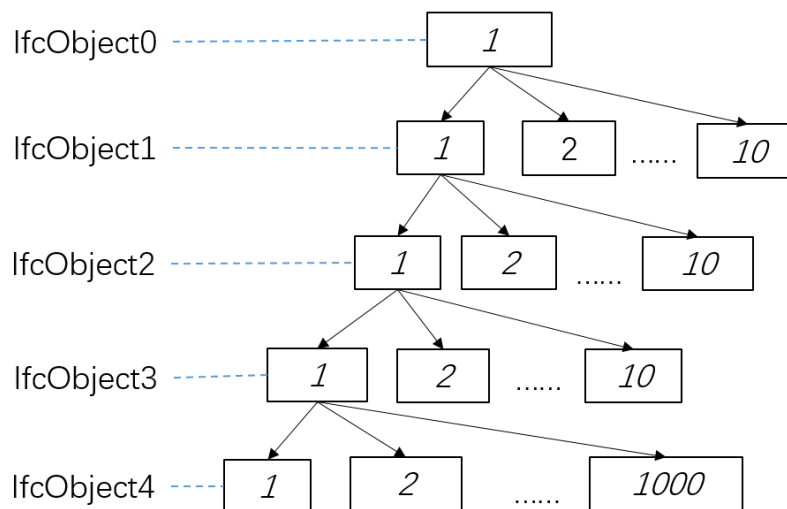


图 6-4

6.2.3 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。其界面如图 6-5：

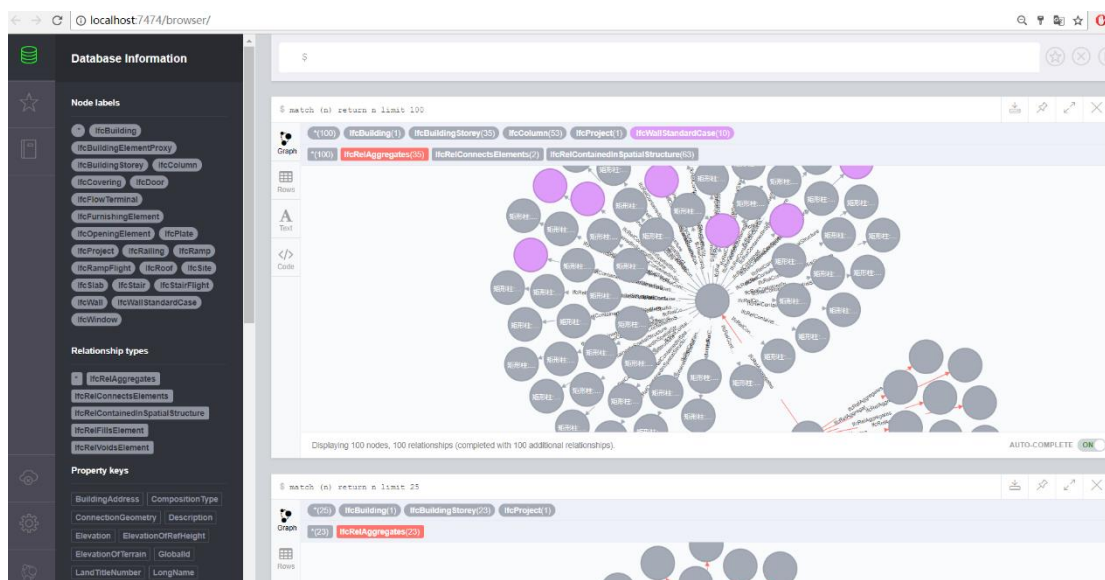


图 6-5

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。

6.2.4 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。图 6-6 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。

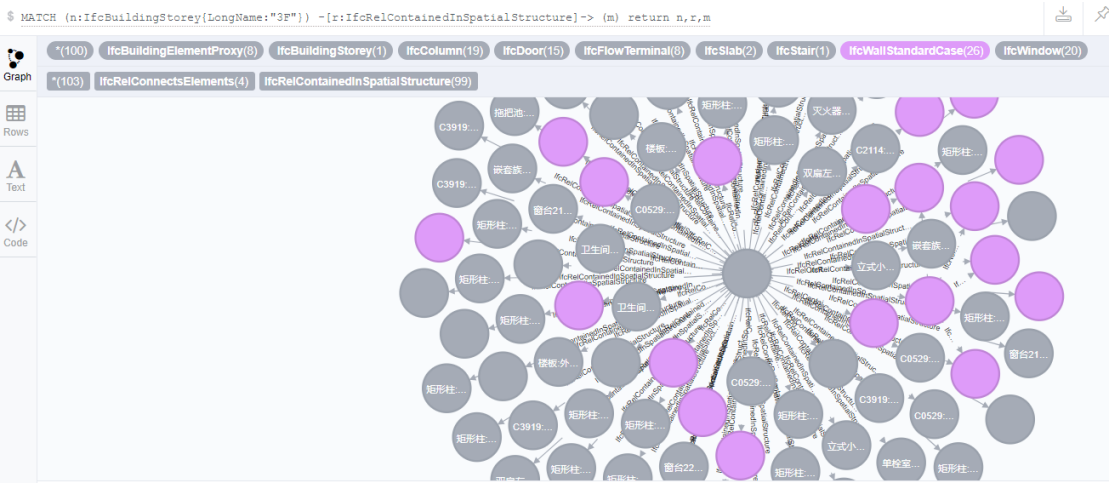


图 6-6

6.2.5 基于 BloomFilter 的特定路径查询测试

在此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、拼接 ID、BloomFilter 过滤③的方式，进行这 10000 次查询。对比时间消耗：

查询方式	第 1 次 (ms)	第 2 次 (ms)	第 3 次 (ms)
Cypher (原生)	33986	24051	24991
BloomFilter	11757	11972	11058

由以上三次试验，试验中原生 Cypher 最快时间 24051ms 较基于 BloomFilter 最慢时间 11972ms 还要多耗费 10000ms。可知基于 BloomFilter 的查询速度明显快于原生 Cypher 查询方式。

第7章 总结与展望

7.1 总结

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。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。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。

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，通过增加冗余的方式，满足用户对于属性数据的高效查询，避免了多表连接的问题。在试验中，着重解决了 IFC 文件解析后的数据到 JSON 数据的映射问题，和 ElasticSearch 的索引优化等问题，最终实现了 IFC 实体的属性数据的高效存储和查询。

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。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。

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。

7.2 展望

本系统基于弹性搜索框架 ElasticSearch 和图数据库 Neo4j，实现了 IFC 实体属性数据和空间关系数据的存储和查询。但是目前，两个子系统在查询模块是独立的，用户只能通过各自的 web 界面来访问数据库，这带来了一定的不便。未来将考虑提供统一的访问界面。

另外，本系统中 Neo4j 的节点只存储了 IFC 实体类型本身的自有属性，这也

是因为本文将研究的重心放在了 IFC 实体的空间关系信息的存储和查询上。如果用户需要满足一定空间关系的节点的某些热点属性(即用户关注的属性)的话, 需要根据 Neo4j 结果集中的 ID 到 ElasticSearch 中查询数据。未来, 可以考虑给 IFC 实体节点数据增加冗余的设计, 将部分热点属性数据集成到节点数据当中去, 这将使得用户可以直接在 Neo4j 找到热点属性数据, 而不必在 Neo4j 做完关系查询之后再 到 ElasticSearch 中查询。

参考文献

- [1] Jorgensen K A,Skauge J,Christiansson P, Use of IFC Model Servers:Aalborg University and Aarhus School of Architecture, 2008
- [2] 张洋, 基于 BIM 的建筑工程信息集成与管理研究, 清华大学, 2009
- [3] Sciences N I O B National Building Information Modeling Standard Verion1-Part1:Overview,Principles,and Methodologies,2011-9-10 ed 2007
- [4] Fu C;Aouad G;Lee A IFC model viewer to support nD model application[外文期刊] 2006(2)
- [5] Mell P;Grance T,The NIST Definition of Clouding Computing: National Institute of Standards and Technology
- [6] Faraj I;Alshawhi M;Aouad G.An Industry foundation classes Web-based collaborative construction computer environment:WISPER 2000(1)
- [7] 陆宁, 马智亮, 利用面向对象数据库与关系数据库管理 IFC 数据的比较, 清华大学学报(自然科学版), 2012
- [8] Kang H;Lee G,Development of an Object-Relational IFC Server 2009
- [9] 刘强, 张建平, 胡振中, 基于键-值缓存的 IFC 模型 Web 应用技术, 清华大学学报(自然科学版), 2016
- [10] 余芳强, 张建平, 刘强, 赵文忠, 基于云计算的半结构化 BIM 数据库研究, 土木建筑工程信息技术, 2013(6)
- [11] 张建平, BIM 技术的研究与应用, 2011
- [12] 周雅莉, 胡珉, 喻钢, 基于 IFC 标准的隧道工程信息传递研究, 土木建筑工程信息技术, 2015(12)
- [13] 佐佐木达也, NoSql 数据库入门, 2012
- [14] 岳莉, 基于 Lucene 的全文检索系统的研究与应用, 西安电子科技大学, 2010
- [15] 徐财应, 基于 Lucene 的搜索引擎技术的研究与改进[D].中国科学技术大学, 2014
- [16] 陈亚杰,王锋,邓辉,刘应波, ElasticSearch 分布式搜索引擎在天文大数据检索中的应用研究,天文学报, 2016(3).
- [17] Rafa l K, Marek R. ElasticSearch 可扩展的开源弹性搜索解决方案. 时金桥, 柳厅文, 徐菲, 等译.电子工业出版社, 2015: 25
- [18] 龙慧芬, 移动社交网络中的数据库应用[J].山西青年管理干部学院学报, 2013,26(3):106-108
- [19] 王余蓝, 图形数据库 Neo4j 的内嵌式应用研究[J].现代电子技术, 2013,35(22):36-38
- [20] 王余蓝, 图形数据库 Neo4j 与关系数据库的比较研究[J].现代电子技术, 2012,35(20):77-79.
- [21] Huang H,Dong Z. Research on architecture and query performance based on

distributed graph database Neo4j[C]. Consumer Electronics, Communications and Networks(CECNet), 2013 3rd International Conference on .IEEE, 2013, 533-536

[22] Robin Hecht, Stefan Jablonski. NoSql evaluation: A use case oriented survey [A]. Proceedings of 2011 International Conference on Cloud and Service Computing [C]. 2011. 336-341

[23] Sacco G M. Inverted index and inverted list process for storing and retrieving information, U.S. Patent 8,738,631[P]. 2014-5-27.

[24] See R, Karshoej J, Davis D. An Integrated Process for Delivering IFC Based Data Exchange[J]. 2012-09-16