上海交通大学硕士学位论文

高效时序图处理系统的设计与实现

姓　　名：石　林

学　　号：121037910049

导　　师：陈榕 教授

院　　系：电子信息与电气工程学院

学 科/专 业：电子信息

申 请 学 位：专业学位硕士

2024 年 4 月 6 日

A Dissertation Submitted to

Shanghai Jiao Tong University for Master's Degree

DESIGN AND IMPLEMENTATION OF EFFICIENT TEMPORAL GRAPH PROCESSING SYSTEM

Author: Lin Shi

Supervisor: Prof. Rong Chen

School of Electronic Information and Electrical Engineering

Shanghai Jiao Tong University

Shanghai, P.R. China

April 6th, 2024

上海交通大学

学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全知晓本声明的法律后果由本人承担。

学位论文作者签名：

日期：　　年　　月　　日

上海交通大学

学位论文使用授权书

本人同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。

本学位论文属于：

□**公开论文**

□**内部论文**，保密□1 年/□2 年/□3 年，过保密期后适用本授权书。

□**秘密论文**，保密____ 年（不超过 10 年），过保密期后适用本授权书。

□**机密论文**，保密____ 年（不超过 20 年），过保密期后适用本授权书。

（请在以上方框内选择打“✓”）

学位论文作者签名：　　指导教师签名：

日期：　　年　　月　　日　　日期：　　年　　月　　日

摘 要

越来越多的图状结构数据包含时序信息，这些时序信息描述了实体和关系的生命周期，时序图是表示包含时序信息的图状结构数据的常用模型。时序图的更新、时序图查询和时序图分析是三类最为重要的时序图处理任务。时序图查询可以是面向时序图的特定版本（包括最新版本和历史版本）的基于图拓扑的查询，也可以是基于时序图中顶点/边的生命周期信息的查询。时序图分析任务同样既可以是在时序图的特定版本上的普通图分析任务，也可以使用时序图的生命周期信息运行更为复杂的图分析算法。

时序 RDF 图、时序超图和时序属性图是定义时序图结构的三大模型，但面向时序 RDF 图和时序超图的时序图查询并不是热点研究方向，相关系统研究工作不多，且性能难以令人满意。在线图分析系统通常使用时序属性图模型作为其数据模型，与离线图分析系统相比，这类系统同时实现了图的更新和分析，且能够在图更新完成后的很短时间内对其进行分析，大大提高了图分析的时效性，但这要以大幅牺牲图分析性能为代价。

针对上述问题，本文设计并实现了一个高效的时序图处理系统 TempGraph，它由时序图查询和时序图分析两大模块组成。系统的时序图查询模块同时支持时序 RDF 图和时序超图的高效查询，它使用了基于 RDMA 的分布式键值存储辅以分布式排序数组的存储结构，键值存储使用了直接索引和间接索引相结合的设计，在实现了高效的拓扑查询和时间条件查询的同时，减少了时序数据存储的内存使用。系统的时序图分析模块实现了图的事务化更新和实时的第一类时序图分析，它使用了一个高效可更新的时序属性图存储结构 SegCSR，并使用了一种粗粒度的多版本并发控制机制来减少时序数据带来的内存使用和图分析时的额外计算开销。

本文从多个角度对 TempGraph 进行了全面的性能评测，测试结果表明，与图数据库 NebulaGraph 和 Neo4j 相比，TempGraph 的时序图查询都有一个数量级以上的性能提升，TempGraph 的图分析性能是目前最先进的在线图分析系统 LiveGraph 的 1.4~4.4 倍。

关键词：时序图，图处理，图查询，图分析

ABSTRACT

More and more graph-structured data contain temporal information, which describes the lifespan of entities and relationships. Temporal graph is a commonly used model to represent graph-structured data with temporal information. The three most important temporal graph processing tasks are temporal graph updating, temporal graph query, and temporal graph analysis. A temporal graph query can be either a topology-based query on a specific version (the latest or historical version) of the temporal graph, or a query based on the lifespan information of vertices/edges in the temporal graph. A temporal graph analysis task can be either a regular graph analysis task on a specific version of the temporal graph, or a more complex graph analysis algorithm that utilizes the lifespan information of the temporal graph.

The three major models defining the structure of a temporal graph are the temporal RDF graph, the temporal hypergraph, and the temporal property graph. Temporal graph query on the temporal RDF graph and hypergraph is not a hot research topic, with limited relevant system research and unsatisfactory system performance. Online graph analysis systems typically use temporal property graph model as their data model. Compared to offline graph analysis systems, these systems can achieve both graph updating and analysis, enabling them to analyze the graph shortly after it is updated, greatly improving the freshness of graph analysis. However, this comes at the cost of sacrificing graph analysis performance significantly.

To address the above issues, this paper designs and implements an efficient temporal graph processing system called TEMPGRAPH, which consists of two modules: temporal graph query and analysis. The temporal graph query module of the system supports efficient query for both temporal RDF graph and hypergraph. Its storage structure is a distributed key-value storage, supplemented by distributed sorted arrays based on RDMA. The key-value storage adopts a design that combines direct and indirect indexing, which achieves efficient topological and time-condition queries while reducing the memory usage for storing temporal data. The temporal graph analysis module of the system implements transactional graph updating and real-time type-1 temporal graph analysis. It utilizes an efficient updatable temporal property graph storage structure called SegCSR, and a coarse-grained multi-version concurrency control mechanism to reduce memory usage and additional computational overhead during graph analysis caused by temporal data.

Comprehensive performance evaluations of TEMPGRAPHare conducted from multiple perspectives in this paper. The evaluation results demonstrate that compared to graph databases NebulaGraph and Neo4j, TEMPGRAPH achieves temporal graph query performance improvements of over an order of magnitude, and its graph analysis performance is 1.4~4.4 times better than that of the state-of-the-art online graph analysis system LiveGraph.

Key words: Temporal Graph, Graph Processing, Graph Query, Graph Analysis

目 录

插 图

表 格

算 法

第一章 绪论

1.1 研究背景

我们正处于大数据时代，物联网、社交网络、电子商务等应用形态正不断地、越来越快地产生大量数据。大数据应用产生的数据实体之间通常是具有关联关系的，充分地利用这些数据，从中挖掘出尽量多有价值的信息是十分有意义的。

图（graph）是一种用于建模数据实体之间关系的抽象数据结构，它非常适用于表示有关联关系的数据。通常可以使用 $G = (V, E)$ 来表示一个图结构，其中 V 是图 G 中顶点的集合，E 是图 G 中边的集合。每个顶点都表示一个数据实体；每条边都将两个顶点连接起来，表示两个实体之间的关系。图可以分为有向图和无向图两类。有向图中的边是有方向的，从起始顶点指向目标顶点；无向图中的边是没有方向的。除特别说明外，本文中提到的图都是有向图。可以被抽象成图结构的数据称为图状结构数据。图论是数学的一个重要分支，研究图的性质等一切与图有关的问题。计算机科学对图的研究以图论为理论基础，以提高图状结构数据的存储、更新、查询和分析等操作的效率为主要研究目标。

图查询和图分析是两类不同的图处理任务，它们的目标都是从图状结构数据中挖掘出特定信息，但它们要解决的问题区别很大。图查询是根据查询请求指定的查询条件在图中寻找与之匹配的数据，不同查询的复杂性不尽相同，可以是获取特定顶点或特定边的相关信息等的简单查询，也可以是查询所有具有特定拓扑结构的子图等的复杂查询。例如在社交网络中，如果用无向图来建模人与人之间的朋友关系，那么查询“所有朋友数量为 5 的人”就相对比较复杂，而查询“小明的朋友列表”则比较简单。图分析任务通常是全图级别的计算，它基于某个特定的算法，对图中顶点或者边的属性以迭代的方式进行遍历、更新等操作，从而挖掘出图的某些全局或局部特征。图分析任务的复杂性较高，每个任务需要迭代几轮到几十轮，甚至更多。典型的图分析问题包括网页排序[1]（PageRank）、单源最短路径（SSSP）和强连通分量（SCC）等。Google 的 Pregel[2]是图分析领域的开山之作，它采用了“以顶点为中心”的编程模型和一种称为 BSP 的计算模型，使得大规模图上的图分析能够简洁而高效地得以实现。

图的更新、查询和分析是三类最为重要的图操作，图数据库通常实现了图的更新和查询，而图分析由于其计算的复杂性，通常需要先从图数据库中导出图快照，然后

加载到专用的图分析系统中进行离线图分析，这一过程称为 ETL（抽取-转换-加载，Extract-Transform-Load）。离线图分析系统通常具有较高的图分析性能，但它不可避免地牺牲了时效性，图的更新需要经过非常耗时的 ETL 过程才能应用到图分析系统中，为了解决这一问题，在线图分析系统应运而生。在线图分析系统同时实现了图的更新和分析，能够在图更新完成后的很短时间内对其进行分析，无需耗时的 ETL 过程，大大提高了图分析的时效性。我们将各种图数据库、图查询和图分析系统统称为图处理系统，或称图计算系统。

随着时间的推进，图数据可能是不断变化的。随着时间而变化的图叫做时序图（temporal graph），或称动态图（dynamic graph）或进化图（evolving graph）。图数据库和在线图分析系统管理的图就是典型的时序图，因为这些系统的图数据会随着图更新操作的执行而不断变化。带有生命周期信息的图状结构数据通常也可以抽象成时序图，这类数据在金融[3]、路网[4]和流行病学等领域正起着越来越重要的作用。普通图查询和图分析面向的通常是静态图或时序图的最新版本，而时序图查询不仅可以面向时序图的特定版本（包括最新版本和历史版本）进行基于图拓扑的查询，也可以基于时序图中顶点/边的生命周期信息进行查询。时序图分析同样既可以是在时序图的特定版本上的普通图分析任务（第一类时序图分析），也可以使用时序图的生命周期信息运行更为复杂的图分析算法（第二类时序图分析），例如时序单源最短路径（Temporal SSSP）、最晚出发时间（LD）[5]等。我们将支持时序图查询或时序图分析的系统统称为时序图处理系统。

1.2 国内外研究现状

1.2.1 图模型和图处理系统

图数据库和图分析系统长期以来都是学术界和工业界的研究热点，也涌现了一批代表性工作，例如图数据库 Neo4j[6]、OrientDB[7]和 NebulaGraph[8]，离线图分析系统 GraphLab[9]、GraphChi[10]、PowerGraph[11]、GraphX[12]和 Gemini[13]以及在线图分析系统 LiveGraph[14]和 GeaFlow[15]等。也有一些只支持图查询、不支持图更新的图查询系统的研究工作，例如 DEX[16]、Wukong[17]等，这类系统虽然牺牲了图数据的可变性，但通常具有较好的图查询性能。

属性图[18]、RDF[19]图和超图是定义图结构的三大模型范式，不同的图处理系统往往会根据其要解决的问题选择其中一种作为其数据模型。

属性图模型相对比较简单，但表达能力很强，大部分图数据库和图查询系统、几乎所有图分析系统都是基于属性图模型实现的。

RDF 相对比较复杂，但具有更强的表达能力和灵活性。RDF 数据集的基本组成单元是三元组，每个三元组都由主语（subject）、谓词（predicate）和宾语（object）三个部分组成。目前已经有不少面向 RDF 的图数据库和图查询系统，其中有些系统使用关系模型来存储三元组数据，例如 TriAD[20]等，这种实现最大的问题在于查询的执行需要依赖三元组的连接这种开销比较大的操作来实现，为了提高查询性能，这些系统通常需要使用提前剪枝、连接顺序选择和查询缓存等优化技术。更好的方案是使用图模型来存储 RDF 图并使用图搜索算法来实现图查询，AllegroGraph[21]、Trinity.RDF[22]和 Ontotext GraphDB[23]等系统都采用了这种方案。Wukong[17]是一个先进的分布式 RDF 图查询系统，它同样采用了该方案，它还围绕 RDMA（Remote Direct Memory Access，远端内存直接访问）这一高性能网络硬件技术，通过一系列存储和查询引擎上的优化技术，实现了 RDF 图的可扩展存储和低延迟、高并发的图查询请求处理。与 Trinity.RDF 相比，Wukong 处理单条查询的性能提高了 10 倍以上，查询吞吐量提高了上百倍。

超图（hypergraph）是在图的基础上泛化的一种数据结构，可以用来描述多元关系。虽然超图理论在上世纪就已经发展完善，但面向超图的图处理系统的研究依然鲜有人涉足。

1.2.2 时序图处理系统

在关系型数据库中引入时间维度早已成为数据库领域的研究热点[24][25][26]。数据库可能包含三类时间数据：用户自定义时间、有效时间和事务时间。用户自定义时间没有特殊语义，由用户规定其含义，例如 Jisoo 的生日是 1995 年 1 月 3 日，那么 1995-1-3 就是用户自定义时间，系统可以把用户自定义时间和普通属性同等看待，无需特别处理。有效时间指的是应用程序角度一个事件发生或一个事实成立的时间，可以是时间点、时间区间等，例如李华在 2017 到 2021 年期间是一名本科生，那么 2017-2021 年就是“李华是本科生”这一事实成立的有效时间。事务时间是数据库对数据对象进行插入、删除或修改等操作的时间，它由数据库系统的内部时钟自动生成。大多数主流关系型数据库，例如 Oracle、MySQL 和 PostgreSQL 等，仅支持对最新版本数据的查询和更新，历史版本的数据对外部是不可见的，且可能被系统的垃圾回收机制清理，这类数据库都是非时序数据库。时序数据库既可以是支持有效时间的查

询和更新的事实数据库（例如 InfluxDB[27]等），也可以是支持使用事务时间查询历史版本数据的回滚数据库，亦可以是同时支持上述两类操作的双时序数据库（例如 TimeDB[28]等）。

时序关系型数据库中的时序概念同样适用于图处理系统。与属性图、RDF 图和超图相对应，时序属性图、时序 RDF 图和时序超图构成了定义时序图结构的三大范式。

在时序属性图中，顶点、边、顶点的各属性和边的各属性都可以有对应的生命周期，生命周期既可以是有效时间，也可以是事务时间。在线图分析系统和大多数图数据库严格来说都是基于使用事务时间来表示生命周期的时序属性图模型实现的，但它们通常不能归类为时序图处理系统，因为它们面向的通常是图数据的最新稳定版本，不支持时序图查询或时序图分析。TGraph[29]是少有的支持时序图查询的图数据库之一，它是基于 Neo4j 实现的。支持第一类时序图分析的系统（例如 Chronos[30]和 SAMS[31]等）和支持第二类时序图分析的系统（例如 Graphite[32]和 Tink[33]等）通常都是基于使用有效时间来表示生命周期的时序属性图模型实现的。

时序 RDF 图模型并没有统一的标准，Gutierrez 等人[34]于 2005 年首次提出扩展 RDF 模型对有效时间的支持，并在 2007 年的另一篇文章[35]中做了进一步完善。他们提出将 RDF 三元组扩展为四元组 $(s, p, o)[t]$ 的形式，其中 t 是一个时间戳。Tappolet 等人[36]设计了 τ-SPARQL 语言来查询时序 RDF 图中的有效时间，并给出了从 τ-SPARQL 到标准 SPARQL 的转换方法。Bereta 等人[37]也提出了自己的时序 RDF 图模型 stRDF 和对应的查询语言 stSPARQL。时序 RDF 图的概念提出之后，也出现了一些面向时序 RDF 的图查询系统[38][39]，但这类系统数量不多，且在处理大规模（例如百万级以上数量三元组）的时序 RDF 图数据集上的查询时，查询响应时间通常达到毫秒级，甚至秒级，也难以有效地应对大量用户同时发起时序图查询请求的高并发场景。

时序超图方面，由于时序超图模型也没有统一的标准，加之基于超图模型的图处理系统方面的研究本身就很少，因此，针对时序超图的系统研究工作就更鲜有人涉足。

1.3 本文工作

针对时序图查询和时序图分析系统的研究现状，本文想要解决的问题包括：

1. 面向时序 RDF 图和时序超图的时序图查询并不是热点研究方向，相关系统研

究工作不多，且性能难以令人满意。

2. 在线图分析系统虽然具备良好的时效性，但这要以大幅牺牲图分析性能为代价。以 LiveGraph 为例，它在实现图的事务化更新的同时，也保证了临接列表扫描这一图分析中最常用操作的良好的数据局部性，尽管如此，与最先进的离线图分析系统 Gemini 相比，它的图分析性能依然有 40% 以上的下降。

为了解决以上问题，本文设计并实现了一个分布式时序图处理系统 TempGraph，它由时序图查询和时序图分析两大模块组成。时序图查询模块运行在多台机器上，负责时序图查询请求的处理；时序图分析模块运行在单台机器上，负责处理图的事务化更新和实时的第一类时序图分析任务。时序图查询模块同时支持时序 RDF 图和时序超图数据集的存储和查询，它采用了分布式键值存储辅以分布式排序数组的存储结构，实现了时序 RDF 图和时序超图数据集的可扩展存储；它通过一系列查询引擎的优化，实现了时序图查询的低延迟、高并发处理。时序图分析模块使用时序属性图模型作为其数据模型，它通过一个高效的动态图存储结构 SegCSR 和一种基于 epoch 的粗粒度多版本并发控制机制，实现了在保证良好的图更新效率的同时，达到接近于离线时序图分析系统的图分析性能。

1.4 论文结构

本文共分为七章，本章是第一章，主要介绍了本文的研究背景以及与本文相关的研究工作，梳理出现有时序图查询和时序图分析系统亟待解决的问题，最后概括了本文的研究内容和结构安排。

第二章介绍了与本文相关的技术背景：首先对图的三大模型范式属性图、RDF 图和超图进行了简要介绍，然后分别给出了时序属性图、时序 RDF 图和时序超图的定义和表示方法。由于系统的时序图查询模块是在 Wukong 的基础上实现的，所以该章从存储结构和查询引擎两个方面对 Wukong 进行了简要介绍。最后介绍了 CSR 和邻接列表两种图分析系统常用的图存储结构，并分析了它们各自的优劣势。

第三章首先介绍了 TempGraph 的总体架构、运行环境和基本配置，然后简要概括系统中存储层、引擎层和接口层各自的作用，最后对系统的接口层进行了详细介绍。

第四章介绍了系统时序图查询模块的设计与实现。首先介绍了本文设计的时序 RDF 图查询语言 SPARQL-T 和时序超图查询语言 HQL-T 的语法和语义，然后先后介绍了时序 RDF 图的存储结构和查询引擎、时序超图的存储结构和查询引擎。

第五章介绍了系统时序图分析模块的设计与实现。首先详细介绍了一个高效可更新的时序属性图存储结构 SegCSR/TS，然后介绍模块使用的一种粗粒度多版本并发控制机制和 SegCSR/TS 结合该机制的修改后版本 SegCSR，最后介绍了时序图分析模块的只读事务，并简要说明如何基于只读事务在图分析引擎中实现第一类时序图分析。

第六章是 TEMPGRAPH 的性能评测，通过一系列实验对系统时序图查询模块和时序图分析模块的性能进行全面的评测。

第七章对全文内容进行了总结，并展望了未来工作的方向。

第二章 背景介绍

2.1 图的三大范式

属性图、RDF 图和超图常常被并称为定义图结构的三大模型范式。

2.1.1 属性图

属性图是最为常用的建模图状结构数据的模型。属性图中的每个顶点和每条边都有一个类型，都可以包含若干属性，每个属性都是一个键值对，键是属性名，值是属性值。图2–1给出了一个简单的表示 COVID-19 传播情况的属性图示例，它由两个顶点 Alice、Bob 和一条边组成。两个顶点的类型分别是 Student 和 Worker，属性 gender 的值分别为 0 和 1。顶点 Alice 到 Bob 之间有一条类型为 infect 的边，这条边有属性 date，表示 Alice 于 2022 年 12 月 2 日把 COVID-19 传染给 Bob。

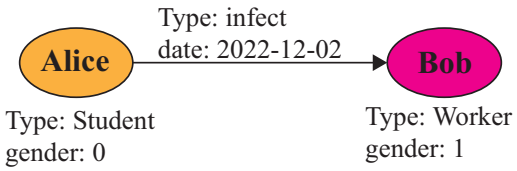


图 2–1 一个简单的属性图示例

Figure 2–1 A simple property graph example

2.1.2 RDF 图

RDF（Resource Description Framework，资源描述框架）主要用于表示万维网（World Wide Web）上资源之间的关联关系，它是随着语义网的发展而被提出并逐渐流行起来的。语义网的目标是赋予万维网上的资源以计算机可以理解的元数据，RDF 的作用就是使用一种基于图的数据模型来表示万维网上不同资源之间的关系。RDF 数据集的基本组成单元是三元组；每个三元组都由主语（subject）、谓词（predicate）和宾语（object）三个部分组成，可以用 <s, p, o> 的形式来表示。三元组中的 s 和 o 对应图中的顶点，每个三元组对应图中的一条从顶点 s 指向顶点 o 的一条类型为 p 的有向边。此外，RDF 标准还包含一些预定义的规则，例如使用谓词 `rdf:type` 来描述资源的类型。图2–2是一个简单的 RDF 数据集，它由 7 个三元组组成，其中 4 个三元

组用于描述资源的类型，3 个三元组用于描述资源间的关系。它对应的 RDF 图包含四个顶点和三条边。

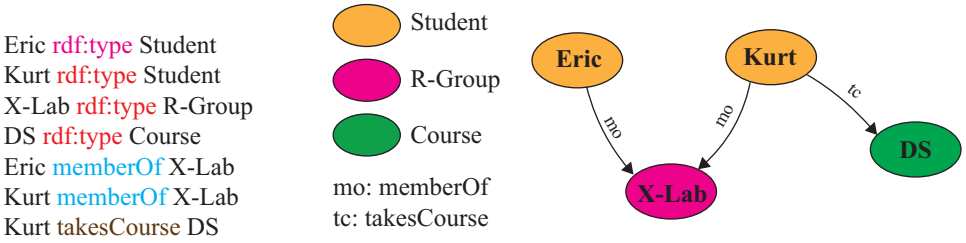


图 2–2 RDF 图示例

Figure 2–2 An example of RDF graph

SPARQL[40]是 RDF 图的标准查询语言，一条 SPARQL 查询语句通过指定一个图模式来从 RDF 图中寻找与之匹配的子图。SPARQL 查询语句最常用的形式为：

```
SELECT RD WHERE GP
```
(2–1)

其中 GP 是一组“主语-谓词-宾语”三元模式（TP），RD（Result Description，结果描述）是一组变量。三元模式的每个元素都既可以是变量，也可以是常量。除了三元模式外，GP 中还可以包含过滤器，过滤器的作用是对变量的取值按照一定条件进行过滤。RD 是 GP 中变量的非空子集，告诉 SPARQL 执行引擎应该输出哪些变量的取值。给定一个 RDF 图 G 和一个 SPARQL 查询语句 Q，SPARQL 执行引擎会在 G 上搜索与 Q 的 GP 匹配的子图，找到 RD 中所有变量的可取值。

图2–3(a) 给出了一条示例 SPARQL 查询语句，它想要找到满足如下条件的资源 Y：Y 的类型是课程且有 X-Lab 的成员选修该课程。在图2–2中寻找与图2–3(b) 所示的查询图相匹配的子图，最终可以得到如图2–3(c) 所示的查询结果。由于 RD 只包含一个变量 Y，所以查询结果仅有一列，变量 Y 仅可以取值 DS，所以查询结果只有一行。

2.1.3 超图

超图（hypergraph）是在图的基础上泛化的一种数据结构。普通图中的边只能与两个顶点相连接，而超图中的边能够连接任意数量的顶点。普通图只能描述二元关系，超图能够描述多元关系。超图同样可以使用 $G = (V, E)$ 来表示，其中 V 是超图中顶点的集合，E 是超边的集合。每条超边都是 V 的一个非空子集，它规定了这个

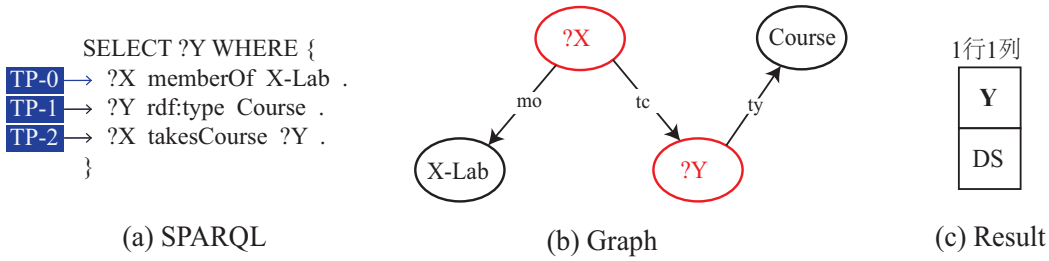


图 2–3 示例 RDF 图上的一条 SPARQL 查询语句

Figure 2–3 A SPARQL query statement on the RDF graph example

集合中包含的所有顶点之间的关系。事实上，无向图就是一种特殊的超图，这种超图的每条超边都只连接两个顶点。类似于属性图，超图中的顶点和超边都可以拥有类型和属性。

超图非常适合用来表示可以抽象成集合的数据，例如，一个微信群聊中通常包含几个到几百个微信用户，我们可以把微信群聊抽象为超边，把微信用户抽象为顶点，如果一个微信用户是某个群聊的成员，那么它对应的顶点就是该群聊对应超边连接的顶点之一。图2–4所示的超图包含 7 个顶点和 3 条超边，群聊 e_1 包含 u_1 和 u_2 两个微信用户，群聊 e_2 包含 u_2、u_3、u_4 和 u_5 四个微信用户，u_2 是这两个群聊的公共成员。

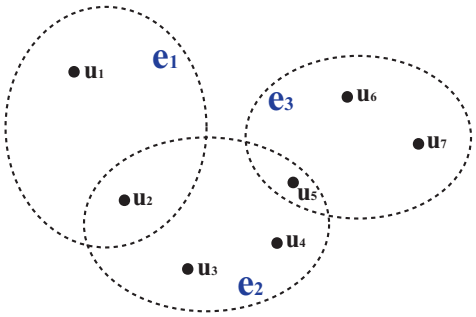


图 2–4 用超图表示的微信群聊关系

Figure 2–4 WeChat group relationship represented by hypergraph

2.2 时序图

属性图、RDF 图和超图模型都有各自的时序版本。

在时序属性图中，顶点、边、顶点的各属性和边的各属性都可以有对应的生命周期，这样，顶点和边就可以被添加和删除，顶点和边上的属性在不同的时期也可以有

不同的值。时序属性图的正式定义为：

定义 2.1 (时序属性图)　时序属性图可以表示为 $G = (V, E)$，其中 V 是顶点 $v = (vid, \tau_v)$ 的集合（vid 是其唯一 ID，$\tau_v = [t_s, t_e)$ 是其生命周期时间区间），E 是边 $e = (eid, uid, vid, \tau_e)$ 的集合（eid 是其唯一 ID，$uid, vid \in V$ 分别是其起点和终点，$\tau_e = [t'_s, t'_e)$ 是其生命周期时间区间）。

时序属性图需要满足一个完整性约束：边的生命周期不能超出它所连接的两个顶点的生命周期。图2–5给出了一个满足完整性约束的时序属性图示例，它包含五个顶点和六条边。顶点的生命周期都是 $[0, 10)$，不同边的生命周期不尽相同，例如顶点 A 和顶点 B 之间有两条边，生命周期为 $[0, 1)$ 的边的属性值为 3，生命周期为 $[1, 4)$ 的边的属性值为 7。

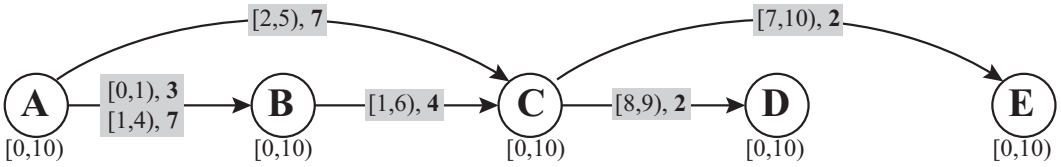


图 2–5 时序属性图示例

Figure 2–5 An example of temporal property graph

时序 RDF 图并没有统一的标准定义，结合之前工作对时序 RDF 图的定义，本文把时序 RDF 图定义为：

定义 2.2 (时序 RDF 图)　时序三元组是带有整数有效时间标签的三元组，可以表示为 $(s, p, o) : [t]$。时序 RDF 图就是时序三元组的集合。我们使用 $(s, p, o) : [t_1, t_2)$ 来简记 $\{(s, p, o) : [t] | t_1 \leq t < t_2, t \in Z\}$，它表示三元组 (s, p, o) 在 $[t_1, t_2)$ 这个时间区间上是有效的。

图2–6是一个时序 RDF 数据集示例，它是图2–2中的 RDF 图的时序版本，由 7 个时序三元组组成。它构成的时序 RDF 图包含三条边，这三条边都有各自的有效时间区间，例如 Eric 和 X-Lab 之间的边的有效时间区间是 $[1, 2)$。

时序超图同样没有标准的定义，本文把时序超图定义为：

定义 2.3 (时序超图)　顶点集合 $V = \{v_1, ..., v_{|V|}\}$ 上的时序超图 $G = (V, E)$ 是一个时序超边的集合 $E = \{e_1, ..., e_{|E|}\}$，每条时序超边 $e = \widetilde{e} : [t] \in E$ 都是一个带有整数有效时间标签 t 的顶点集合 V 的非空子集。我们使用 $\widetilde{e} : [t_1, t_2)$ 来简记 $\{\widetilde{e} : [t] | t_1 \leq t < t_2, t \in Z\}$，它表示时序超边 $\widetilde{e}$ 在 $[t_1, t_2)$ 这个时间区间上是有效的。

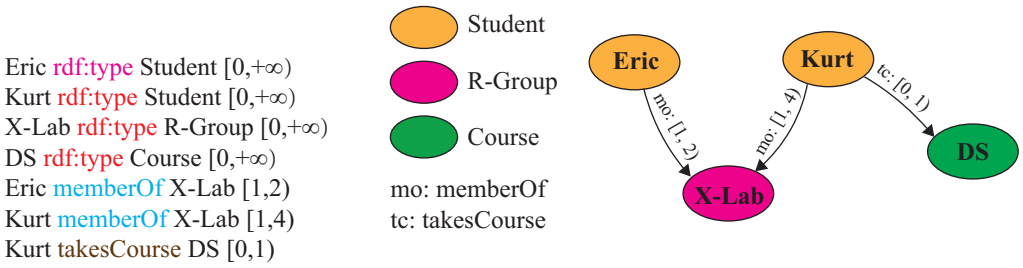


图 2–6 时序 RDF 图示例

Figure 2–6 An example of temporal RDF graph

在时序超图中，时序超边和顶点都是可以有类型的。图2–7是一个时序超图示例，它是图2–4中超图的时序版本，它包含三条时序超边，都有各自的有效时间区间，例如 e_1 表示的微信群聊状态只在时间区间 $[1,3)$ 内是成立的。

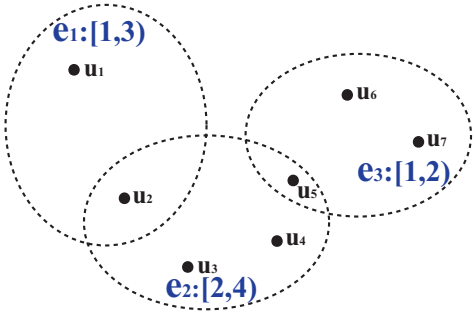


图 2–7 用时序超图表示的微信群聊关系

Figure 2–7 WeChat group relationship represented by temporal hypergraph

2.3 Wukong 介绍

TempGraph 的时序图查询模块是在 Wukong 的基础上实现的。Wukong 是一个先进的分布式 RDF 图查询系统，它基于 RDMA 这一新型硬件特性，通过一系列存储结构和查询引擎上的优化技术，实现了 SPARQL 查询的低时延、高并发执行。

RDMA（Remote Direct Memory Access，远端内存直接访问）是一种高性能的跨机器内存访问技术，能够以很低的延迟直接访问远端机器的内存。RDMA 操作主要分为两类：单边（one-sided）操作和双边（two-sided）操作。单边操作可以不经过远

端机器的 CPU 直接读、写其内存，也不需要操作系统内核的参与。双边操作类似于传统消息传递模式，即分别使用 SEND 和 RECV 两个接口进行消息的发送和接收。虽然双边操作需要服务端 CPU 的参与，但由于双边操作中的网络协议栈完全由支持 RDMA 的网卡硬件实现，因此相较于传统基于 TCP/IP 协议的消息传递，RDMA 双边操作仍然能有超过一个数量级的性能提升。

接下来，本文将从存储结构和查询引擎两个方面对 Wukong 作简要介绍。

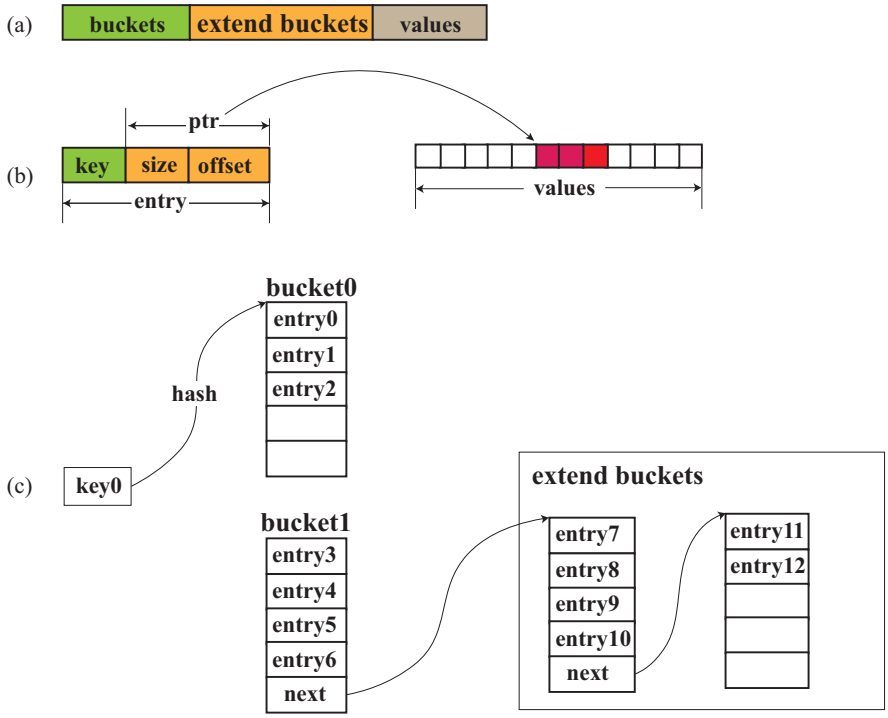


图 2–8 Wukong 的分布式键值存储结构

Figure 2–8 Wukong's distributed key-value store structure

2.3.1 存储结构

为了充分发挥 RDMA 在远端内存访问上的性能优势，Wukong 使用多机内存实现 RDF 图数据集的可扩展存储，使用单边 RDMA 操作访问远端机器的内存。Wukong 的基本存储结构是一个如图2–8的分布式键值存储结构。每台机器上的键值存储由图2–8(a) 所示的三块连续的内存区域组成，其中 buckets 和 extend buckets 是键和它对应值的位置信息的存储区域，values 是值的存储区域。一个键和它对应值的位置信息组成一个图2–8(b) 中的 entry，其中 key 是键本身，size 和 offset 分别是该键对应值的长度（值是一个列表）和位置（相对于 values 区域起始位置的偏移量）。如图2–8(c)

所示，buckets 和 extend buckets 区域由一系列桶（bucket）组成，每个桶都有 N 个槽（slot），最多可以容纳 $N-1$ 个 entry。给定一个 key，系统会使用一个哈希函数计算它对应的 entry 应该被放在 buckets 区域中的哪个桶里，当桶满（同一哈希值的 key 的数量超过 $N-1$）时，系统会从 extend buckets 里分配新桶来继续存放剩下的条目，满桶的最后一个槽会被用来存放指向新桶的指针，形成一个类似于链表的结构。

算法 2–1 根据给定键读取对应值

```
输入: key
输出: 值数组
1  server_id ← hash_server(key);
2  bucket_id ← hash_bucket(key);
3  slot ← NULL;
4  while true :
5      rdma_offset ← bucket_id ×N× sizeof(slot);
6      rdma_size ← N× sizeof(slot);
7      slots ← RdmaRead(server_id, rdma_offset, rdma_size);
8      for i in range(N) :
9          if i < N :
10             if slots[i].key = key :
11                 slot ← slots[i];
12                 goto read;
13         else:
14             if IsEmpty(slots[i]) :
15                 goto read;
16             bucket_id ← slots[i].bid;
17 read :
18 if slot = NULL :
19     return [];
20 else:
21     return RdmaRead(server_id, values_base + slot.offset × sizeof(value), slot.size ×
         sizeof(value));
```

算法2–1给出了根据给定 key 读取对应值的伪代码。系统首先会通过两个哈希函数计算该 key 会被存储在哪台机器的哪个桶里（1-2 行），然后通过 RDMA 的单边读操作从相应机器把整个桶都读取过来（5-7 行），遍历桶的前 $N-1$ 个槽，对比每个

entry 的 key 是否和给定 key 匹配，如果找不到一致的 entry，则通过第 N 个槽提供的指针继续查找 extend buckets 区域里的槽（16 行），直到寻找到匹配的 entry 或扫完最后一个桶。如果搜索到匹配的 entry，直接通过 RDMA 的单边读操作从相应机器的 values 区域的正确位置读取值即可（21 行）。

2.3.2　查询引擎

Wukong 的查询执行引擎使用图探索和全历史剪枝算法执行 SPARQL 语句。以图2–3中的查询语句为例，图2–9展示了该语句的执行过程。查询引擎依次处理语句中的三元模式（❶），对于每个三元模式，查询引擎会通过构造合适的 key 从系统的分布式键值存储中查找与该三元模式匹配的子图，这一过程称为图探索。每个子图都对应着查询语句中变量的一组取值，执行完每个三元模式后探索到的所有子图对应的变量的取值构成中间结果表。所有三元模式都执行完毕后，中间结果表中 RD 对应的列就作为最终的查询结果返回给用户。例如，变量 ?*X* 和 ?*Y* 的所有可取值在 TP-1 执行完毕后已经被找到，三元模式 TP-2 的目的是从中筛选出存在 takesCourse 关系的变量 ?*X* 和 ?*Y* 的取值。在执行三元模式 TP-2 时，查询引擎会将中间结果表第一列中的每个值（变量 ?*X* 的取值，❷）分别与三元模式中的谓词常量 takesCourse 构成 key（❸），从分布式键值存储中读取对应的值（❹），当且仅当第二列中的值出现在❹中，该行才会被保留在中间结果表中。这种通过之前步骤的历史信息精确地修剪掉无用中间结果的的方法就是全历史剪枝。

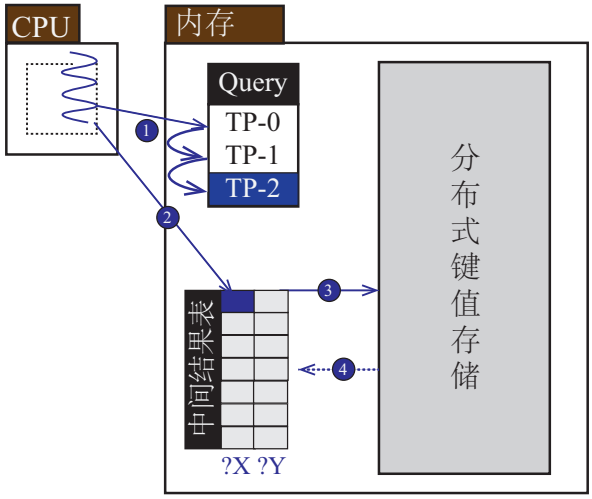
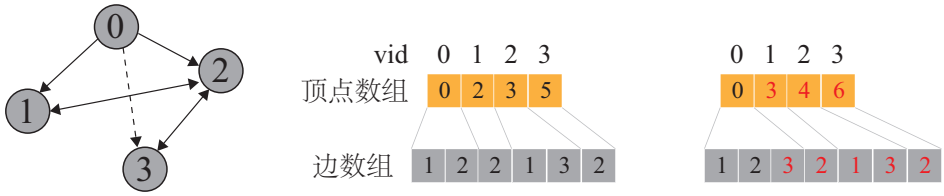


图 2–9　SPARQL 查询执行过程

Figure 2–9　SPARQL query execution process

2.4 图分析系统的图存储结构

CSR (Compressed Sparse Row) 是静态图分析系统广泛使用的图存储结构，它是稀疏矩阵的紧凑化表示，图2–10(a) 是一个简单的图拓扑，图2–10(b) 是它的 CSR 表示，CSR 由一个顶点数组和一个边数组组成，其中边数组按照边的起点 ID 的顺序来存储每条边的终点 ID，顶点数组通过顶点的 ID 来索引，它存储各个顶点的出边在边数组中起始位置的偏移量。在在线图分析场景中，图拓扑会不断更新，而 CSR 是一种紧凑的结构，在 CSR 上做图拓扑更新的效率很低，例如，如果在2–10(a) 的图结构中添加一条从顶点 0 指向顶点 3 的边，它的 CSR 表示就需要对顶点数组和边数组做大量更新（如图2–10(c)）



(a) 图拓扑更新 (b) CSR更新前 (c) CSR更新后

图 2–10 图拓扑的 CSR 表示

Figure 2–10 CSR representation of graph topology

在线图分析系统（例如 RisGraph[41]、GraphOne[42]和 LiveGraph）和动态图存储（例如 CSR++[43]和 Sortledton[44]）通常使用邻接列表作为图拓扑存储的基本数据结构。如图2–11(a)，使用邻接列表表示图拓扑时，每个顶点的所有出边的终点 ID 都会被存放在一个单独的邻接列表中，邻接列表指针数组存放指向每个邻接列表的指针。如图2–11(b)，在添加一条从顶点 0 指向顶点 3 的边时，只需要往顶点 0 的邻接列表中添加一个元素 3 即可。逐顶点的邻边扫描（以下简称扫边）是图分析中最常用的操作。图拓扑的邻接列表表示虽然具有更高的图更新效率，但它的结构较为松散，在扫边过程中的数据局部性较差，扫边性能远不及图拓扑的 CSR 表示。

2.5 本章小结

本章介绍了与本工作相关的背景知识。属性图、RDF 图和超图是定义图结构的三大范式。属性图是最为常用的建模图状结构数据的模型；RDF 的作用是使用一种基于图的数据模型来表示万维网上不同资源之间的关系，SPARQL 是 RDF 图的标准查询语言；超图是在图的基础上泛化的一种数据结构，超图中的边能够连接任意数量

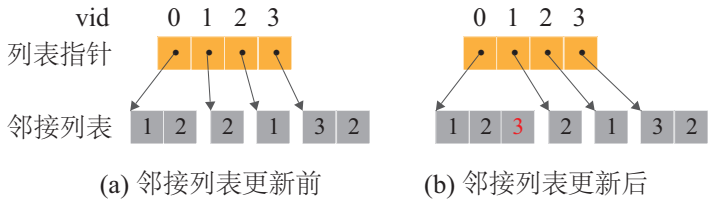


图 2–11 图拓扑的邻接列表表示

Figure 2–11 Adjacency list representation of graph topology

的顶点。随着时间而变化的图叫做时序图，属性图、RDF 图和超图都有各自的时序版本。本文给出了时序属性图的正式定义，然后结合之前工作给出了时序 RDF 图和时序超图的定义。TEMPGRAPH 的时序图查询模块是在 Wukong 的基础上实现的，本章从存储结构和查询引擎两个方面对 Wukong 进行了简要介绍。CSR 和邻接列表是图分析系统常用的两种图存储结构，它们分别适用于离线和在线图分析系统。

第三章 TempGraph 的总体架构

TempGraph 是一个同时支持时序图查询和第一类时序图分析的图处理系统，它由时序图查询和时序图分析两大模块组成。传统的图处理系统通常是基于单一图模型实现的，例如 Neo4j 是一个基于属性图模型的图数据库，而 TempGraph 将时序属性图、时序 RDF 图和时序超图三种时序图模型集成到同一系统中。具体来说，TempGraph 的时序图查询模块同时支持时序 RDF 图和时序超图两种时序图模型，而时序图分析模块则是基于时序属性图模型实现的。本章将首先介绍系统的总体架构，然后详述系统接口层的实现。

3.1 架构概述

TempGraph 的系统架构如图3–1，系统运行在一个由 RDMA 网络相互连接的集群环境中。在一个机器数量为 $N+2$ 的集群中，每台机器运行一个服务节点。其中，一个字符串服务器节点用于处理字符串和整型 ID 之间的转换，一个节点负责处理图的事务化更新和执行系统管理员发起的时序图分析任务（下称分析节点），其余 N 个节点负责处理来自用户的时序 RDF 图查询请求和时序超图查询请求（下称查询节点）。

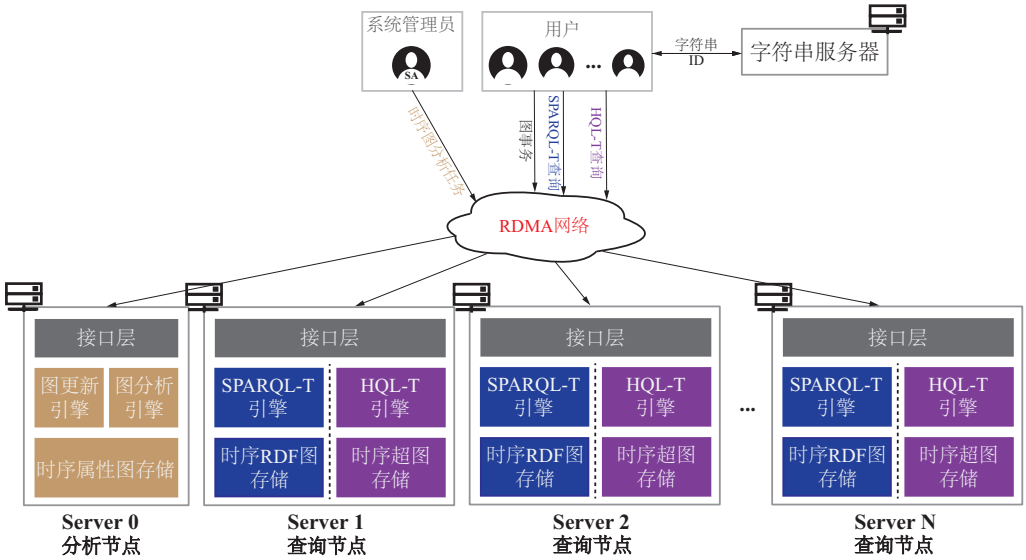


图 3–1 TempGraph 的系统架构

Figure 3–1 The architecture of TempGraph

由于时序 RDF 图和时序超图都没有标准的查询语言，所以本文设计了系统支持的时序 RDF 图查询语言和时序超图查询语言，分别命名为 SPARQL-T 和 HQL-T。在时序 RDF 图数据集中，时序三元组的主语、谓词和宾语都是字符串，时序超图数据集中的超边名和顶点名也都是字符串。TempGraph 在存储时序数据集时，并不直接存储字符串，而是将字符串转化成整型 ID 进行存储，字符串服务器节点存储了字符串和整型 ID 之间的映射并负责转换，这一方面减少了系统的内存使用量，另一方面将字符串匹配转换成整型的匹配，加快了匹配速度。用户在发起 SPARQL-T 和 HQL-T 查询请求时，会先和字符串服务器节点通信，将查询语句中的字符串转换成整型 ID，然后再把它交给系统执行；用户在收到系统返回的查询执行结果时，同样会和字符串服务器节点通信，将查询结果中的 ID 转回字符串。

TempGraph 的时序图查询模块采用了去中心化的分布式架构，它采用边割[45]的方式将时序图数据分区存储到 N 个查询节点的内存中以实现水平扩展（章节 §4.2.1和 §4.3.1）。N 个查询节点是对等的，都能接收和执行来自用户的时序图查询请求，并可以通过 RDMA 网络实现跨节点的协作，进而提高查询执行的局部性和并行性，提高查询的执行效率。系统的时序图分析模块是单机的，来自用户的图事务请求和来自系统管理员的时序图分析任务都会被转发到分析节点，分别由分析节点的事务处理线程和时序图分析线程来完成。

分析节点和查询节点存储的图数据是相互独立的，查询节点中的时序 RDF 图和时序超图存储也是相互独立的，图事务请求和时序图分析任务需要读写的是分析节点中的时序属性图存储，SPARQL-T 和 HQL-T 查询请求查询的分别是时序 RDF 图存储和时序超图存储中的图数据。由于分析节点需要同时支持图事务和时序图分析，所以时序属性图存储是可读可写的。查询节点只需要支持对时序图数据的检索，时序图数据被加载完毕之后便不会再被更新，因此时序 RDF 图存储和时序超图存储都是只读的。

除了字符串服务器节点外，系统中的每个节点都由存储层、引擎层和接口层三层组成。存储层负责图数据的存储，并提供访问和更新接口（仅分析节点的存储层提供更新接口）供引擎层使用；引擎层负责图更新、查询和分析任务的执行，它需要使用存储层提供的接口来读写图数据；接口层负责接收来自于用户或系统管理员的请求，将它们解析为系统的内部表示，然后分配给特定节点的引擎层执行。

3.2 接口层

如 §3.1所述，接口层负责请求的接收、解析和分配。接口层由分析节点和查询节点上的若干代理线程和为每个代理线程准备的专用缓冲区来实现。每个节点的接口层都是对等的，每个代理线程都可以接收和解析来自用户或系统管理员的请求。节点上的线程分为代理线程和工作线程两种，代理线程位于接口层；工作线程位于引擎层，负责任务的实际执行。对于分析节点，工作线程又可以分成图事务处理线程和时序图分析处理线程两类；对于查询节点，工作线程负责执行时序图查询任务。每个线程都有一个专用的缓冲区，线程按照 FIFO 的顺序从缓冲区中读取请求或结果，本地节点的线程可以直接写缓冲区，远程节点的线程通过单边 RDMA Write 操作写缓冲区。代理线程的缓冲区用于接收工作线程的执行结果，工作线程的缓冲区用于接收代理线程或其他工作线程发来的请求。每个缓冲区被分为 $N+1$ 块，编号为 $0, 1, ..., N$，编号为 $i(0 \le i \le N)$ 的节点只能写缓冲区的第 i 块，这样可以保证不同节点的线程对缓冲区的写不会相互干扰，不需要锁的参与，从而实现节点内和跨节点的高效线程间通信。

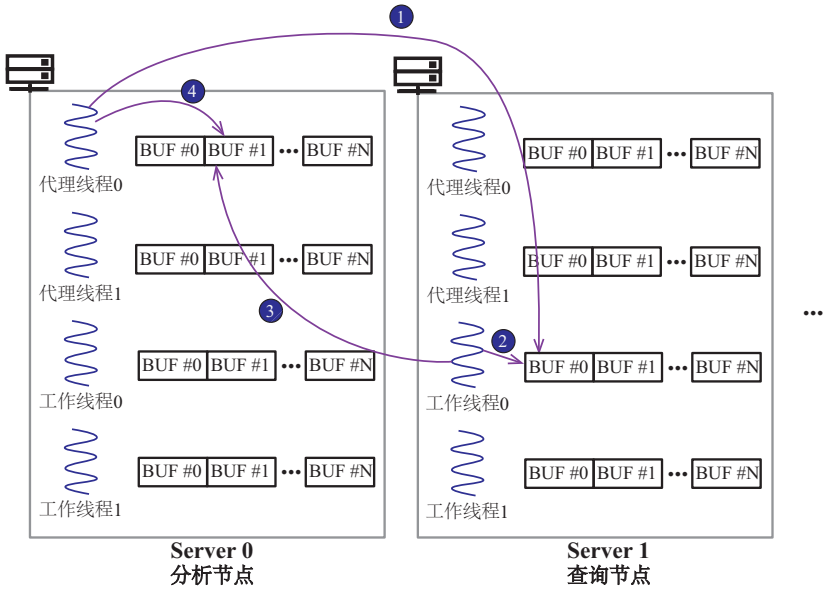


图 3–2 线程间的通信流程

Figure 3–2 The communication flow between threads

图3–2演示了线程间的通信流程。代理线程接收并解析完成请求后，会将请求发送给适当节点的工作线程执行（❶），方法是使用单边 RDMA Write 操作写或直接本地写工作线程对应缓冲区中供代理线程所在节点使用的块，然后代理线程会轮询其

缓冲区，等待执行结果。工作线程会轮询其缓冲区，从中读取请求（❷）开始执行。执行完成后，工作线程会将结果发回代理线程（❸），方法同样是使用单边 RDMA Write 操作写或直接本地写代理线程对应缓冲区中供工作线程所在节点使用的块。此时代理线程就可以读取到请求结果（❹）并返回给用户。

步骤❶中应该将请求发送给哪个节点的哪个工作线程由以下两个准则决定：

- 图事务和时序图分析请求分别会被发送给分析节点的随机图事务工作线程和时序图分析工作线程处理；
- 时序图查询请求会被发送给能够使得数据局部性最好的查询节点的随机工作线程执行。虽然 RDMA 能够高效地实现对远端内存的读取，但它的性能还是落后于对本地内存的读，我们希望查询能够在使得单边 RDMA Read 操作尽量少、本地内存读尽量多的节点上被处理。

3.3 本章小结

本章介绍了 TempGraph 的系统架构。系统运行在一个由 RDMA 网络相互连接的集群环境中，集群包含查询节点、分析节点和字符串服务器节点三类节点。查询节点用于执行来自用户的 SPARQL-T 和 HQL-T 查询语句，分析节点用于处理图事务和时序图分析请求，字符串服务器节点用于处理字符串和整型 ID 之间的转换。TempGraph 的时序图查询模块采用了去中心化的分布式架构，而时序图分析模块则是单机的。每个查询节点和分析节点都由存储层、引擎层和接口层三层组成。存储层负责图数据的存储，引擎层负责图更新、查询和分析任务的执行，接口层负责请求的接收、解析和分配。本章还对系统的接口层作了详细介绍。接口层由分析节点和查询节点上的若干代理线程和为每个代理线程准备的专用缓冲区来实现，代理线程和工作线程之间可以通过单边 RDMA 操作实现高效的线程间通信，从而实现高效的请求的分配和执行结果的获取。

第四章 时序图查询模块的设计与实现

时序 RDF 图和时序超图查询系统是受关注相对较少的研究方向，即使已经出现了一些面向时序 RDF 图的图查询系统，但它们的性能较差，查询响应时间和并发处理能力都难以令人满意，时序超图查询系统的研究更是少有人问津。本章将介绍 TEMPGRAPH 的时序图查询模块，它在 Wukong 的基础上对时序图查询进行了针对性的优化，实现了时序 RDF 图和时序超图的可扩展存储和高效查询。

4.1 时序图查询语言

RDF 的标准查询语言 SPARQL 并没有与时序相关的语法，所以需要在 SPARQL 原有语法的基础上进行时序扩展，以支持时序数据的查询。超图并没有其标准的查询语言，时序超图当然也没有，所以我们同样需要设计时序超图的查询语言。

4.1.1 时序 RDF 图查询语言 SPARQL-T

本文设计的时序 RDF 图查询语言 SPARQL-T 从两方面对 SPARQL 的语法进行了扩展：

- 将 GP 中的“主语-谓词-宾语”三元模式扩展为“主语-谓词-宾语-有效时间区间的开始时间-有效时间区间的截止时间”五元模式（QP），新加入的两个元素要求是变量，用来获取时序三元组的有效时间数据。为了兼容标准的 SPARQL 语法，我们规定新加入的两个元素是可选的，但它们要么同时存在，要么同时不存在。
- GP 中的过滤器可以对时间变量的取值按照一定条件进行过滤，例如与变量或常量的大小关系比较等。

图4–1给出了一条图2–6中的时序 RDF 图上的 SPARQL-T 语句示例。QP-0 包含两个时间变量 `ts` 和 `te`，它们应该分别取值为匹配到的时序三元组的有效时间区间的开始时间和截止时间。GP 还包含一个过滤器，它会过滤出变量 `ts` 的取值为 1 的查询结果条目。GP 中的三个模式构成了图4–1(b) 所示的查询图，可以在图2–6中的时序 RDF 图上找到一个与之匹配的子图，过滤后得到的查询结果如图4–1(c) 所示，共一行二列。

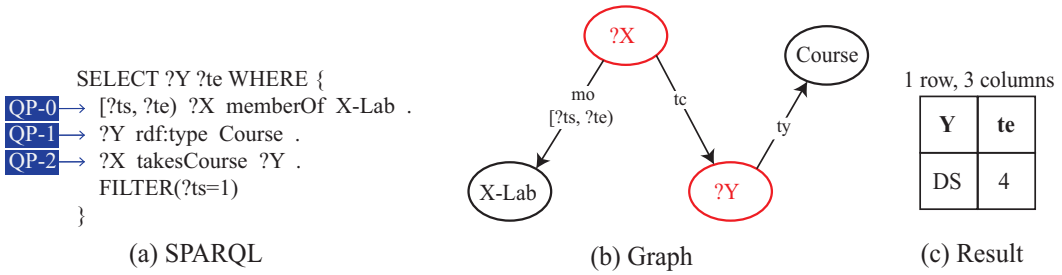


图 4–1 示例时序 RDF 图上的一条 SPARQL-T 查询语句

Figure 4–1 A SPARQL-T query statement on the temporal RDF graph example

4.1.2 时序超图查询语言 HQL-T

时序超图是一种表示集合关系的数据模型，所以时序超图查询语言应该着重查询**集合间的关系**（即时序超边间的关系）、**元素和集合间的关系**（即顶点和时序超边间的关系）以及**元素间的关系**（即顶点间的关系）。此外，时序超图查询语言也应当支持**对时序超边的有效时间的查询**以及**基于有效时间的筛选**。

HQL-T 查询语句的基础形式依然是

$$\texttt{SELECT RD WHERE GP} \tag{4–1}$$

其中 GP 由一组基础关系模式（RP）组成，也可以包含过滤器，过滤器和 RD 的作用与 SPARQL-T 中的相同。给定一个时序超图 G 和一个 HQL-T 查询语句 Q，HQL-T 执行引擎会根据 GP 指定的模式在 G 中搜索与之匹配的子结构，找到 RD 中所有变量的可取值。RP 的基本结构为：

$$\texttt{input builtin:type(args) output interval} \tag{4–2}$$

其中 `input` 是输入元素列表，`builtin` 是内置关键字，`type` 指定了该模式的类型，`args` 是该模式的参数，`output` 是输出元素列表。`interval` 是可选的，它是一个由两个变量组成的区间，格式为 $[?\mathrm{var}_1, ?\mathrm{var}_2)$，当它出现在 RP 中时，HQL-T 执行引擎应当将变量 var_1 和 var_2 分别取值为使得此 RP 成立的时间区间的开始时间和截止时间。根据 `type` 的值，RP 可分为以下几类：

- **GE 模式：**该类型 RP 的 `type` 为 `etype`，输入元素列表要求只包含一个表示时序超边类型的常量；输出元素列表要求只包含一个变量；没有参数。这类 RP 的作用是查询所有指定类型的时序超边。如果包含 `interval`，那么变量 var_1 和 var_2 会分别取值为查询到的时序超边的有效时间区间的开始时间和

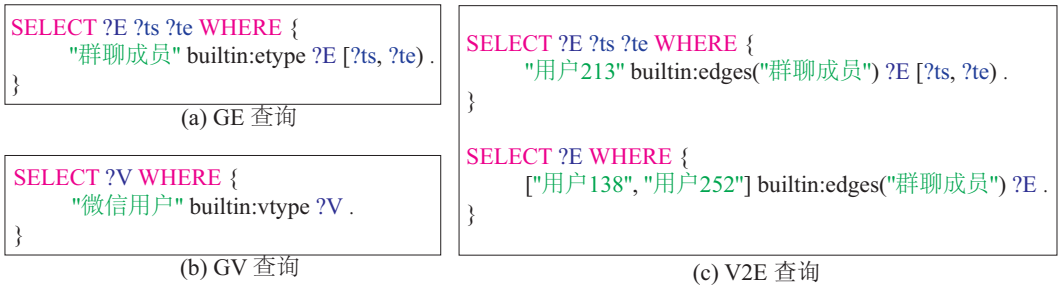


图 4–2 HQL-T 查询语句示例（1）

Figure 4–2 HQL-T query statement example (1)

截止时间。图4–2(a) 给出了一条只包含 GE 模式的 HQL-T 查询语句示例，该查询是要查找所有类型为“群聊成员”的时序超边。

- **GV 模式：**该类型 RP 的 `type` 为 `vtype`，输入元素列表要求只包含一个表示顶点类型的常量；输出元素列表要求只包含一个变量；没有参数。这类 RP 的作用是查询所有指定类型的顶点。由于顶点没有时序数据，所以 `interval` 不能出现在这类 RP 中。图4–2(b) 给出了一条只包含 GV 模式的 HQL-T 查询语句示例，该查询是要查找所有类型为“微信用户”的顶点。
- **V2E 模式：**该类型 RP 的 `type` 为 `edges`，输入元素列表可以包含若干表示顶点的常量或变量，输出元素列表要求只包含一个表示时序超边的变量，参数必须是一个表示时序超边类型的常量。这类 RP 的作用是查找所有指定类型且同时包含各输入顶点的时序超边。如果包含 `interval`，那么变量 $\texttt{var}_1$ 和 $\texttt{var}_2$ 会分别取值为查询到的时序超边的有效时间区间的开始时间和截止时间。图4–2(c) 给出了两条只包含 V2E 模式的 HQL-T 查询语句示例，第一个查询是要查找所有类型为“群聊成员”且包含顶点“用户 213”的时序超边，第二个查询是要查找所有类型为“群聊成员”且同时包含顶点“用户 138”和“用户 252”的时序超边。
- **E2V 模式：**该类型 RP 的 `type` 为 `vertices`，输入元素列表可以包含若干表示时序超边的常量或变量，输出元素要求只包含一个表示顶点的变量，没有参数。这类 RP 的作用是查找各输入时序超边包含的所有公共顶点。如果包含 `interval`，那么变量 $\texttt{var}_1$ 和 $\texttt{var}_2$ 会分别取值为能够使得各输入时序超边都有效的最大时间区间的开始时间和截止时间。图4–3(a) 给出了两条只包含 E2V 模式的 HQL-T 查询语句示例，第一个查询是要查找时序超边“群聊 7”的有效时间区间和包含的所有顶点，第二个查询是要查找时序超边“群聊 1”和

```
SELECT ?V ?ts ?te WHERE {
        "群聊7" builtin:vertices ?V [?ts, ?te) .
}

SELECT ?V WHERE {
        ["群聊1", "群聊6"] builtin:vertices ?V .
}
```

(a) E2V 查询

```
SELECT ?V WHERE {
        "用户168" builtin:intersectVertices(
                        "群聊成员", >,  2) ?V .
}
```

(b) V2V 查询

```
SELECT ?E ?ts ?te WHERE {
        "群聊7" builtin:intersectEdges(
                "群聊成员", >=,  2) ?E [?ts, ?te) .
}

SELECT ?E WHERE {
        "群聊12" builtin:containEdges("群聊成员") ?E .
}

SELECT ?E WHERE {
        "群聊3" builtin:inEdges("群聊成员") ?E .
}
```

(c) E2E 查询

图 4–3 HQL-T 查询语句示例（2）

Figure 4–3 HQL-T query statement example (2)

“群聊 6”包含的公共顶点。

- **V2V 模式：**该类型 RP 的 `type` 为 `intersectVertices`，输入元素列表可以包含若干表示顶点的常量或变量，输出元素要求只包含一个表示顶点的变量。要求有三个参数，第一个参数是时序超边的类型，第二个参数是比较运算符，可以是 >、<、=、!=、>= 和 <= 其中之一，该参数可以省略，省略时默认是 >=，第三个参数是一个整数。这类 RP 的作用是查找同时与各输入顶点出现在相同时序超边的顶点，且时序超边的类型和数量要满足参数的要求。这类 RP 不能包含 `interval`。图4–3(b) 给出了一条只包含 V2V 模式的 HQL-T 查询语句示例，该查询是要查找与顶点“用户 168”同时加入超过 2 个共同群聊的顶点。
- **E2E 模式：**该类型 RP 的 `type` 为 `intersectEdges`、`containEdges` 或 `inEdges`，输入元素列表可以包含若干表示时序超边的常量或变量，输出元素要求只包含一个表示时序超边的变量。当 `type` 为 `intersectEdges` 时，要求有三个参数，第一个参数是时序超边的类型，第二个参数是比较运算符，可以是 >、<、=、!=、>= 和 <= 其中之一，该参数可以省略，省略时默认是 >=，第三个参数是一个整数，这类 RP 的作用是查找与各输入时序超边的交集的基数满足参数要求的时序超边；当 `type` 为 `containEdges` 或 `inEdges` 时，参数必须是一个表示时序超边类型的常量，这类 RP 的作用是查找指定类型且是各输入时序超边子集（当 `type` 为 `containEdges` 时）或超集（当 `type` 为 `inEdges` 时）的时序超边。如果包含 `interval`，那么变量 $\texttt{var}_1$ 和 $\texttt{var}_2$ 会分别取值为能够使得各输入、输出时序超边同时有效的最大时间区间的开始时

间和截止时间。图4–3(c) 给出了三个只包含 E2E 模式的 HQL-T 查询语句示例，第一个查询是要查找与时序超边“群聊 7”拥有不少于 2 个公共顶点的类型为“群聊成员”的时序超边，第二个查询是要查找所有顶点都在时序超边“群聊 12”中且类型为“群聊成员”的时序超边，第三个查询是要查找包含时序超边“群聊 3”的所有顶点且类型为“群聊成员”的时序超边。

以上六类 RP 的组合可以实现很强的查询语义，基本可以满足对时序超图的查询需求。对于包含多 RP 的 HQL-T 查询语句，查询引擎需要按序逐条执行各 RP。图4–4是一条包含两个 RP 的查询语句，在执行该语句时，需要先找到所有类型为“群聊成员”的时序超边，然后将找到的所有时序超边依次作为输入元素，查找其包含的所有顶点，最后过滤出符合条件的查询结果。

```
SELECT ?E ?V ?ts ?te WHERE {
    "群聊成员" builtin:etype ?E [?ts, ?te) .
    ?E builtin:vertices ?V .
    FILTER(?ts >= 2023-10-01 && ?ts <= 2023-10-07 && ?te > now())
}
```

图 4–4 普通 HQL-T 查询语句示例

Figure 4–4 Common HQL-T query statement example

4.2 时序 RDF 图存储结构和查询引擎

时序 RDF 图的存储主要是基于 §2.3.1中介绍的分布式键值存储实现的，在此基础上，它还使用分布式排序数组结构加速时间条件查询。时序 RDF 图查询引擎使用存储层提供的接口、按照用户指定的查询计划执行 SPARQL-T 查询语句中 GP 的各模式。本节将详细介绍时序 RDF 图的存储结构和查询引擎。

4.2.1 存储结构

如 §3.1所述，TempGraph 并不直接存储时序 RDF 图数据集中的字符串，而是将其转化成整型 ID 进行存储，并使用字符串服务器管理字符串和整型 ID 之间的对应关系。时序 RDF 图数据集中的字符串分为两类：

1. 时序三元组中表示谓词的字符串及表示类型名的字符串，例如时序三元组 `Eric memberOf X-Lab [1,2)` 中的 `memberOf` 和时序三元组 `Eric rdf:type Student [0,+∞)` 中的 `rdf:type`、`Student`。这类字符串对应的 ID 范围

为 $[1, 2^{17})$，特别地，`rdf:type` 对应的 ID 为 1。本文将表示谓词的字符串对应的 ID 称为 `pid`，将表示类型名的字符串对应的 ID 称为 `tid`。

2. 时序三元组中表示主语和宾语的字符串（表示类型名的字符串除外），例如时序三元组 `Eric memberOf X-Lab [1,2)` 中的 `Eric` 和 `X-Lab`。这类字符串对应的 ID 范围为 $[2^{17}, 2^{46})$。本文将这类字符串对应的 ID 称为 `vid`。

系统默认使用 32 位 UINT 表示各种 ID，对于规模比较庞大的数据集，可以在系统编译时指定使用 64 位 UINT。

时序 RDF 图存储内存布局如图4–5，分为键值存储区和时序三元组存储区两部分。时序三元组存储区本质上是内存数组，它分为两部分，在“时序三元组存储 1”中，时序三元组按照有效时间区间的开始时间升序排布；在“时序三元组存储 2”中，时序三元组按照有效时间区间的截止时间升序排布。键值存储区中的键值对分为直接索引和间接索引两类，在直接索引中，值是实际数据；在间接索引中，值由指向“时序三元组存储 1”中时序三元组的指针组成。

图 4–5 时序 RDF 图存储内存布局

Figure 4–5 Temporal RDF graph storage memory layout

在键值存储区中，各种键值对及其类型如表4–1所示。键的数据类型是 64 位 UINT，由三部分组成，第一部分使用其高 46 位，第二部分使用其中间 17 位，第三部分使用其最低位。第三部分只能是 OUT（1，表示出方向）和 IN（0，表示入方向）其中之一。值的基本数据类型与 ID 的数据类型相同，可以是 32 或 64 位 UINT，这取决于系统编译时的配置。

图4–6给出了系统会为图2–6中的数据集生成的一部分键值对。为了便于理解，图中使用字符串表示主语、谓词和宾语，系统实际使用的是这些字符串对应的 ID。在间接索引中，值是一个由时序三元组在“时序三元组存储 1”中的偏移量组成的列表，例如，时序三元组 `Kurt memberOf X-Lab [1,4)` 在“时序三元组存储 1”中的偏移量为 5，那么间接索引就使用 5 来表示该时序三元组。

为了实现 RDF 数据集的分布式可扩展存储，系统使用如下规则将键值对和时序三元组划分到各查询节点中存储：

1. 每个查询节点负责一部分 `vid` 的管理，`vid` 会被分配到的查询节点（查询节

表 4–1 时序 RDF 图存储使用的键值对

Table 4–1 Key-value pairs used in temporal RDF graph storage

序号	键	值	类型
1	`[0|0|OUT]`	所有 `pid`	直接索引
2	`[0|1|OUT]`	所有 `tid`	
3	`[0|1|IN]`	所有 `vid`	
4	`[0|tid|IN]`	所有类型为 `tid` 的 `vid`	
5	`[0|pid|OUT]`	谓词为 `pid` 的所有时序三元组中所有表示宾语的 `vid`	
6	`[0|pid|IN]`	谓词为 `pid` 的所有时序三元组中所有表示主语的 `vid`	
7	`[vid|0|OUT]`	主语为 `vid` 的所有时序三元组中的所有 `pid`	
8	`[vid|0|IN]`	宾语为 `vid` 的所有时序三元组中的所有 `pid`	
9	`[vid|1|OUT]`	描述 `vid` 类型的时序三元组	间接索引
10	`[vid|pid|OUT]`	主语为 `vid`、谓词为 `pid` 的所有时序三元组	
11	`[vid|pid|IN]`	宾语为 `vid`、谓词为 `pid` 的所有时序三元组	

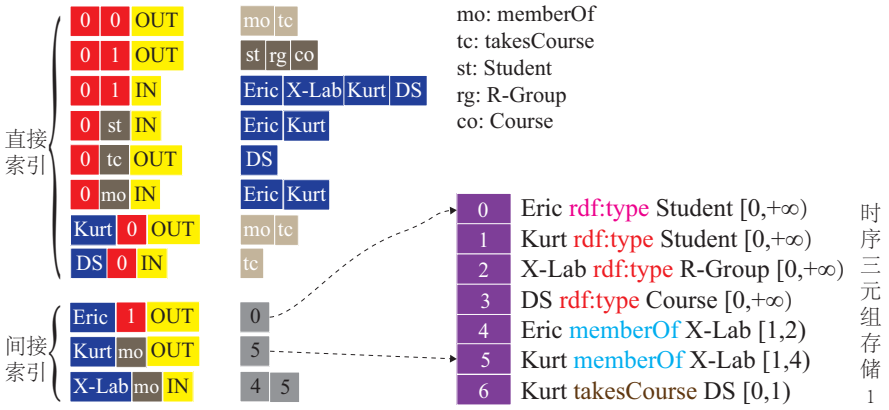


图 4–6 时序 RDF 图存储使用的键值对示例

Figure 4–6 Example of key-value pairs used in temporal RDF graph storage

点从 0 开始编号）为：

$$\texttt{vid} \,\%\, \texttt{num\_qnodes} \tag{4–3}$$

其中 `num_qnodes` 是查询节点数。

2. 对于键值对 1-6，将值按照 1 中的规则划分到各查询节点存储，同一键会出现在每个查询节点的键值存储中；对于键值对 7-11，将键依据其第一部分按照 1

中的规则划分到不同查询节点存储，每个键只会出现在一个查询节点的键值存储中。

3. 将所有时序三元组分别按有效时间区间的开始时间和截止时间排序后，分别按块划分到各查询节点的“时序三元组存储 1”和“时序三元组存储 2”中存储，偏移量为 `off` 的时序三元组会被分配到的查询节点（从 0 开始编号）为：

$$\texttt{off} \,/\, \lceil \texttt{num\_ttriples} \,/\, \texttt{num\_qnodes} \rceil \tag{4–4}$$

其中 `num_ttriples` 是时序三元组总数。

分布式键值存储结构可以实现高效的拓扑查询，两个“时序三元组存储”可以实现高效的时间条件查询。对于以时序三元组有效时间区间开始时间为条件的查询，可以在“时序三元组存储 1”上通过二分法快速找到目标时序三元组，同理可以使用“时序三元组存储 2”快速处理以时序三元组有效时间区间截止时间为条件的查询。直接索引和间接索引相结合的键值存储结构可以在实现高效的拓扑查询和时间条件查询的同时，减少时序数据存储占用的空间。

表4–2列出了存储层向上提供的接口，接口❶只涉及对本节点上键值存储结构的读，接口❷可能涉及对各节点的键值存储结构和“时序三元组存储 1”的读，接口❸和❹可以分别筛选出本节点的“时序三元组存储 1”和“时序三元组存储 2”中符合时间条件的时序三元组。

4.2.2　查询引擎

SPARQL-T 查询引擎沿用了 Wukong 使用的图探索和全历史剪枝算法。默认情况下，查询引擎会使用存储层提供的接口❶和❷通过键值存储查找所需的数据，但对于一些以时间点或时间范围为条件的查询，通过接口❸和❹直接从“时序三元组存储”中寻找符合条件的时序三元组可能更加高效。例如图4–1中的查询语句想要找到满足如下条件的资源 Y：Y 的类型是课程且有加入 X-Lab 时间为 1 的 X-Lab 成员选修该课程。在执行该语句时，可以先在“时序三元组存储 1”中快速找到有效时间区间开始时间为 1 的时序三元组，然后再对这些时序三元组做进一步筛选。如果查询引擎不直接使用“时序三元组存储”，而是先使用键值存储做图探索，然后再筛选出变量 `ts` 的取值为 1 的查询结果，那么就会带来更大的数据访问开销。用户可以通过在 GP 中使用**时序三元组时间范围模式**显式地要求查询引擎使用“时序三元组存储”进行时间条件查询，语法为：

$$\texttt{interval s p o STRAT/END}(\texttt{const}_1, \texttt{const}_2) \tag{4–5}$$

表 4–2 时序 RDF 图存储提供的接口

Table 4–2 Interfaces provided by temporal RDF graph storage

序号	接口	作用
❶	`list<value_t> get_values(key)`	获取本节点上键 `key` 对应的值列表，适用于键值对 1-6
❷	`list<triple_t> get_ttriples(key)`	获取键 `key` 对应的时序三元组列表，适用于键值对 7-11
❸	`list<triple_t> get_ttriples_ts(start, end)`	从本节点的“时序三元组存储 1”中获取所有有效时间区间开始时间范围为 `[start,end]` 的时序三元组
❹	`list<triple_t> get_ttriples_te(start, end)`	从本节点的“时序三元组存储 2”中获取所有有效时间区间截止时间范围为 `[start,end]` 的时序三元组

其中 `s`、`p` 和 `o` 分别表示时序三元组的主语、谓词和宾语，它们都可以是常量或变量；`interval` 是可选的，它是一个由两个变量组成的区间，格式为 $[?var_1, ?var_2)$，当它存在时，两个变量会分别取值为查询到的时序三元组的有效时间区间的开始时间和截止时间；`START` 和 `END` 二选一，`START` 指定依据有效时间区间开始时间进行查找，`END` 指定依据有效时间区间截止时间进行查找；$const_1$ 和 $const_2$ 是两个时间常量且 $const_1 \leq const_2$，用于指定要查找的时间范围。值得注意的是，用户需要自行估算使用时序三元组时间范围模式是否能够提高查询的执行效率，避免出现负优化。图4–7中的查询语句与图4–1中的查询语句含义相同，但它显式地指定使用“时序三元组存储 1”进行时间条件查询。

```
    SELECT ?Y ?te WHERE {
P-0 → [?ts, ?te) ?X memberOf X-Lab START(1, 1) .
P-1 → ?Y type Course .
P-2 → ?X takesCourse ?Y .
    }
```

图 4–7 显式指定使用“时序三元组存储”的查询语句示例

Figure 4–7 Example of a query that explicitly specifies the use of ”temporal triple storage”

由于一条 SPARQL-T 查询语句可能包含多个五元模式或时序三元组时间范围模式，使用不同的顺序执行这些模式的效率可能不同，例如图4–7中的查询语句可以以

以下两种不同的顺序执行：

- 顺序一：
  - 利用“时序三元组存储 1”寻找变量 X 和 te 的所有可取值（P-0）
  - 对于每行中间结果，从变量 X 的取值 val(X) 出发使用键 [val(X)|takesCourse|OUT] 找到变量 Y 的所有可取值（P-2）
  - 对于每行中间结果，从变量 Y 的取值 val(Y) 出发使用键 [val(Y)|1|OUT] 获取变量 Y 的类型，如果是 Course，则保留该行中间结果（P-1）
- 顺序二：
  - 利用“时序三元组存储 1”寻找变量 X 和 te 的所有可取值（P-0）
  - 使用键 [0|Course|IN] 找到变量 Y 的所有可取值（P-1）
  - 对于每行中间结果，从变量 X 的取值 val(X) 出发使用键 [val(X)|takesCourse|OUT] 找到变量 Y 的可取值，如果其中包含变量 Y 的取值，则保留该行中间结果（P-2）

由于系统并未实现 SPARQL-T 的执行优化器，无法自动确定最优的执行方案，所以需要用户在发起查询请求时通过自定义的查询计划指定查询引擎执行各模式的顺序。另外，如果查询语句中包含时序三元组时间范围模式，查询引擎会先执行时序三元组时间范围模式，再执行五元模式。

SPARQL-T 查询语句可分为大查询和小查询两类，大查询通常从大量顶点出发，探索庞大数量的路径，这一过程往往会访问时序 RDF 图中的很大一部分数据；小查询通常从一个固定的顶点开始，访问少量的路径和顶点。具体来说，大查询包含以下两类：

1. 第一个查询模式需要用到键值对 1-6 的查询语句
2. 包含时序三元组时间范围模式的查询语句

其他的查询语句都属于小查询。

查询引擎使用 fork-join 机制来处理大查询。工作线程在收到代理线程发来的大查询时，会将其分成 num_qnodes × parallel_factor 个子查询，然后分别发送给各个查询节点的 parallel_factor 个工作线程共同执行，最后原工作线程负责子查询结果的合并。parallel_factor 是并行因子，其值越大，查询执行的并发度越高。

图4–8展示了一个大查询的 fork-join 过程，该查询的第一个查询模式需要使用键

值对 6。由于键值对 6 的值会被划分到所有查询节点存储，所以不同查询节点上读取到的键 `[0|memberOf|IN]` 对应的值是无共享的。同一查询节点上的不同工作线程读取到的值是相同的，工作线程会根据其线程号使用值列表中的一部分元素。

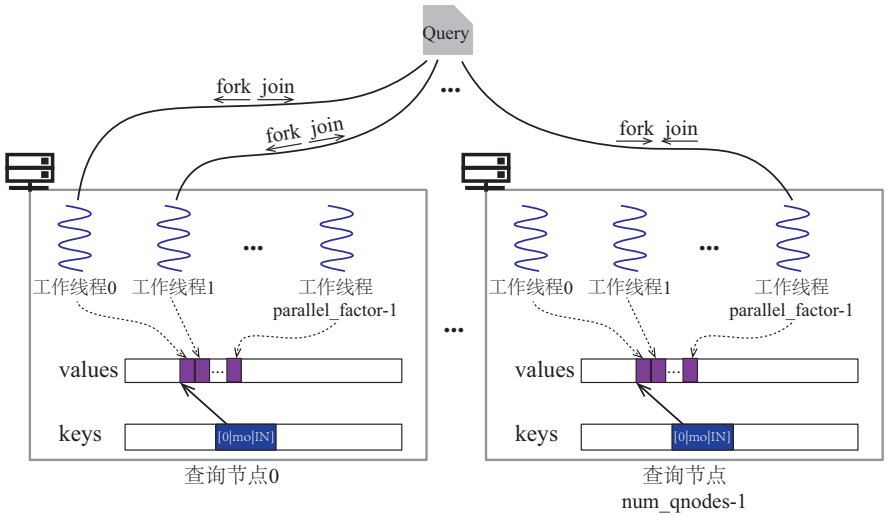


图 4–8 大查询的 fork-join 过程

Figure 4–8 Fork-join process of a heavy query

查询引擎在处理小查询时也可能会使用 fork-join 机制：当工作线程在准备开始执行一个五元模式时，如果中间结果条目数达到系统预设的阈值（例如 100），且工作线程需要通过较多跨节点的单边 RDMA Read 操作来执行该模式，那么工作线程会将其分成 `num_qnodes` 个子查询，然后分别发送给各个查询节点的一个工作线程共同执行。查询的划分实质上是中间结果的划分，中间结果是依据执行该五元模式需要使用的变量的取值划分的，如果某行中间结果中该变量的取值为 `val`，那么该行中间结果会被划分到的子查询（从 0 开始编号）为：

$$\texttt{val} \ \% \ \texttt{num\_qnodes} \tag{4–6}$$

查询划分完成后，子查询 i 会被发送到查询节点 i 执行。

只要遵守以上规则，子查询可以被进一步地划分，这样就可能形成一颗以代理线程发来的查询为根节点的查询树。

查询语句执行过程中，一个变量可能处于 Unknown 和 Known 两种状态，它们的含义分别为：

- Unknown：该变量没有在之前的模式中出现过，当前中间结果表中没有该变量对应的列；

- Known：该变量已经在之前的模式中出现过，当前中间结果表中有该变量对应的列。

查询引擎在执行时序三元组时间范围模式时，会先从“时序超边存储”中查找所有符合模式指定的时间范围的时序三元组，然后逐一判断每个时序三元组是否与模式的前半部分（即 `interval s p o`）匹配。匹配一个时序三元组的方法为：先从 `s` 开始，如果它是变量且状态是 Unknown，那么该时序三元组的主语就是它的一个可取值；如果它是变量且状态是 Known，那么就需要逐一验证中间结果表每一行中该变量对应的列是否与该时序三元组的主语相同，若是则保留该行，否则舍弃该行；如果它是常量，那么只有在该时序三元组的主语与之相同时才可继续匹配。接着以类似过程匹配 `p`、`o`、`interval` 的$?var_1$和$?var_2$。

查询引擎在执行五元模式时，通常会先从其中的常量开始，通过构造合适的键从分布式键值存储中为模式中的变量寻找可取值，从而将变量的状态从 Unknown 转移到 Known。然后可能会使用中间结果表每一行中 Known 变量对应的列的值继续构造合适的键进行进一步的图探索，直到所有五元模式都执行完毕。

4.3　时序超图存储结构和查询引擎

时序超图存储的设计思路与时序 RDF 图存储类似，都使用了分布式键值存储辅以分布式排序数组的存储结构。时序超图查询引擎使用存储层提供的接口逐一执行 HQL-T 查询语句中的各 RP。本节将详细介绍时序超图的存储结构和查询引擎。

4.3.1　存储结构

系统支持的时序超图数据集格式为`时序超边类型，时序超边名称，有效时间区间的开始时间，有效时间区间的截止时间,{顶点名称 1，顶点名称 2,...}`。TempGraph 同样会将时序超图数据集中的字符串转化成整型 ID 进行存储。系统默认使用 32 位 UINT 表示顶点字符串对应的 ID（下称 `vid`），取值范围为 $[2^{16}, 2^{32})$，对于顶点数量比较庞大的数据集，可以在系统编译时指定使用 64 位 UINT 表示 `vid`，取值范围就扩充到 $[2^{16}, 2^{48})$。系统使用 64 位 UINT 表示时序超边 ID（下称 `heid`），取值范围为 $[2^{16}, 2^{48})$。值得注意的是，可能有多条时序超边的名称相同，例如字符串“群聊 1”可能对应多条时序超边，表示“群聊 1”在不同时间区间下的成员情况。系统会给予相同名称的不同时序超边不同的 `heid`，在将表示时序超边名称的字符串转换成 `heid` 时，字符串服务器返回的是 `heid` 的列表。表示时序超边和顶点的类

型的字符串对应的 ID（下分别称 `htid` 和 `tid`）的范围为 $[2, 2^{16})$。

时序超图存储内存布局如图4–9，与时序 RDF 图存储类似，它分为键值存储区和时序超边存储区两部分。时序超边存储区存储的是每个超边的 ID、类型及其有效时间，即由 `[超边 ID，超边类型，有效时间区间的开始时间，有效时间区间的截止时间]` 四元组组成，它本质上也是内存数组。时序超边存储区分为两部分，在“时序超边存储 1”中，四元组按照有效时间区间的开始时间升序排布；在“时序超边存储 2”中，四元组按照有效时间区间的截止时间升序排布。时序超边的 `heid` 与它对应的四元组在“时序超边存储 1”中的偏移量 `off` 的关系为：

$$\texttt{heid} = \texttt{off} + 2^{16} \tag{4–7}$$

键值存储区中的键值对分为直接索引和间接索引两类，在直接索引中，值是实际数据；在间接索引中，值由指向“时序超边存储 1”中四元组的指针组成。

图 4–9 时序超图存储内存布局

Figure 4–9 Temporal hypergraph storage memory layout

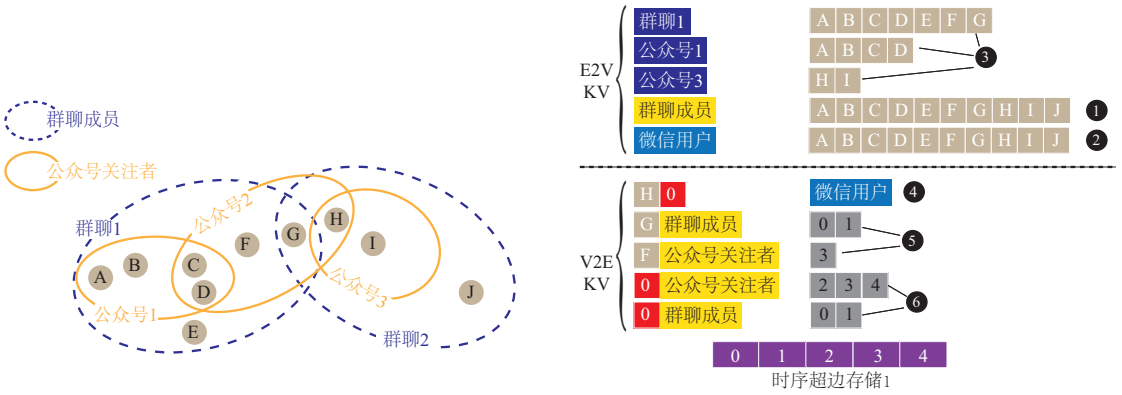
在键值存储区中，各种键值对及其类型如表4–3所示，系统使用两个分布式键值存储结构 V2E KV 和 E2V KV 来存储这些键值对。键值对❶~❸位于 E2V KV，键的数据类型是 64 位 UINT，值的基本数据类型和 `vid` 的数据类型相同；键值对❹~❼位于 V2E KV，键的数据类型也是 64 位 UINT，由高 48 位和低 16 位两部分组成，值的基本数据类型是 64 位 UINT。E2V KV 中的键值对都是直接索引，V2E KV 中的键值对除了❹外都是间接索引。

表 4–3 时序超图存储使用的键值对

Table 4–3 Key-value pairs used in temporal hypergraph storage

序号	键	值	类型
❶	`[htid]`	类型为 `htid` 的所有时序超边包含的所有 `vid`	直接索引
❷	`[tid]`	类型为 `tid` 的所有 `vid`	
❸	`[heid]`	时序超边 `heid` 包含的所有 `vid`	
❹	`[vid|0]`	顶点 `vid` 的 `tid`	
❺	`[vid|htid]`	包含顶点 `vid` 且类型为 `htid` 的所有时序超边	间接索引
❻	`[0|htid]`	类型为 `htid` 的所有时序超边	

图4–10(a) 展示了一个简单的时序超图拓扑，它从微信中的群聊及公众号和微信用户之间的关系抽象而来，它包含两类时序超边：群聊成员和公众号关注者。有两条类型为群聊成员的时序超边：“群聊 1”、“群聊 2”，有三条类型为公众号关注者的时序超边：“公众号 1”、“公众号 2”、“公众号 3”，有 10 个名称分别为 A~J 的类型为微信用户的顶点。图4–10(b) 给出了系统会为其生成的一部分键值对。为了便于理解，图中使用字符串表示时序超边、顶点及其类型，系统实际使用的是这些字符串对应的 ID。在间接索引中，值是一个由时序超边对应的四元组在“时序超边存储 1”中的偏移量组成的列表，例如，时序超边“群聊 2”对应的四元组在“时序超边存储 1”中的偏移量为 1，那么间接索引就使用 1 来表示“群聊 2”。



(a) 时序超图拓扑　　(b) 各种键值对示例

图 4–10 时序超图存储使用的键值对示例

Figure 4–10 Example of key-value pairs used in temporal hypergraph storage

为了实现分布式可扩展存储，系统使用如下规则将键值对和时序超边划分到各查询节点中存储：

1. 每个查询节点负责一块连续 ID 的时序超边和顶点的管理，时序超边 `heid` 会被分配到的查询节点（查询节点从 0 开始编号）为：

$$(\texttt{heid} - 2^{16}) \,/\, \lceil \texttt{num\_thes} \,/\, \texttt{num\_qnodes} \rceil \tag{4–8}$$

   其中 `num_thes` 是时序超边总数，`num_qnodes` 是查询节点数。顶点的分配方法同理。

2. 对于键值对❶❷❻，将值按照1中的规则划分到各查询节点存储，同一键会出现在每个查询节点的键值存储中；对于键值对❸❹❺，将键依据其高 48 位按照1中

的规则划分到不同查询节点存储，每个键只会出现在一个查询节点的键值存储中。

3. 将所有四元组分别按有效时间区间的开始时间和截止时间排序后，分别按块划分到各查询节点的“时序超边存储 1”和“时序超边存储 2”中存储，偏移量为 `off` 的四元组会被分配到的查询节点（从 0 开始编号）为：

$$\texttt{off} \,/\, \lceil \texttt{num\_thes} \,/\, \texttt{num\_qnodes} \rceil \tag{4–9}$$

4.3.2 查询引擎

HQL-T 查询引擎同样沿用了 Wukong 使用的图探索和全历史剪枝算法。默认情况下，查询引擎会使用存储层提供的接口通过键值存储查找所需的数据，但对于一些以时间点或时间范围为条件的查询，直接在“时序超边存储”上通过二分法查找符合条件的时序超边可能更加高效。例如图4–4中的查询语句是要查找最后一次成员变动发生于 2023 年 10 月 1 日到 2023 年 10 月 7 日期间的所有群聊及其成员，在执行该语句时，可以使用“时序超边存储 1”快速找到有效时间区间开始时间在 2023 年 10 月 1 日到 2023 年 10 月 7 日期间的所有时序超边，然后从中筛选出类型为“群聊成员”且有效时间区间截止时间晚于当前时间的时序超边即可。如果查询引擎不使用“时序超边存储”，而是先从键值存储中找到所有类型为“群聊成员”的时序超边，再筛选出有效时间区间符合条件的时序超边，那么就会带来更大的数据访问开销。目前系统尚未实现 HQL-T 的查询优化器，无法自动判断使用键值存储和使用“时序超边存储”哪个效率更高。系统的替代方案是将选择权交给用户，由用户在查询语句中使用**时序超边时间范围模式**显式地指定使用“时序超边存储”进行时间条件查询，语法为：

$$?\texttt{var}\ \texttt{STRAT/END}(\texttt{const}_1, \texttt{const}_2)\ \texttt{interval} \tag{4–10}$$

其中变量`?var` 会取值为查询到的时序超边的 `heid`；`START` 和 `END` 二选一，`START` 指定依据有效时间区间开始时间进行查找，`END` 指定依据有效时间区间截止时间进行查找；$\texttt{const}_1$ 和 $\texttt{const}_2$ 是两个时间常量且 $\texttt{const}_1 \leq \texttt{const}_2$，用于指定要查找的时间范围；`interval` 是可选的，它是一个由两个变量组成的区间，格式为 $[?\texttt{var}_1, ?\texttt{var}_2)$，当它存在时，两个变量会分别取值为查询到的时序超边的有效时间区间的开始时间和截止时间。我们规定，时序超边时间范围模式只能出现在 GP 中最开始的位置，它们会先于 RP 被处理。值得注意的是，用户需要自行估算使用时序超边时间范围模式是否能够提高查询的执行效率，避免出现负优化。图4–11中的查

询语句与图4–4中的查询语句含义相同，但它显式地指定使用“时序超边存储 1”进行时间范围查询。

```
SELECT ?E ?V ?ts ?te WHERE {
    ?E START(2023-10-01, 2023-10-07) [?ts, ?te) .
    FILTER(?te > now())
    "群聊成员" builtin:etype ?E .
    ?E builtin:vertices ?V .
}
```

图 4–11 显式指定使用“时序超边存储”的查询语句示例

Figure 4–11 Example of a query that explicitly specifies the use of ”temporal hyperedge storage”

与 SPARQL-T 类似，HQL-T 查询语句同样可分为大查询和小查询两类，大查询通常以大量时序超边或顶点作为起始，然后探索更多数量的时序超边和顶点；小查询通常从一个固定的时序超边或顶点开始，访问少量的时序超边和顶点。具体来说，大查询包含以下两类：

1. 第一个查询模式是 GE 或 GV 模式的查询语句；
2. 包含时序超边时间范围模式的查询语句。

其他的查询语句都属于小查询。查询引擎同样使用 fork-join 机制来处理大查询，其过程与 SPARQL-T 查询引擎完全相同，在此不作赘述。

查询引擎在处理小查询时同样可能会使用 fork-join 机制：当工作线程在准备开始执行一个 RP 时，如果中间结果条目数达到系统预设的阈值（例如 100），且工作线程需要通过较多跨节点的单边 RDMA Read 操作来执行该 RP，那么工作线程会将其分成 `num_qnodes` 个子查询，然后分别发送给各个查询节点的一个工作线程共同执行。在划分查询时，中间结果是依据输入元素列表中的第一个变量的取值划分的，如果某行中间结果中该变量的取值为 `val`，那么该行中间结果会被划分到的子查询（从 0 开始编号）为：

$$(\texttt{val} - 2^{16}) \,/\, \lceil \texttt{num\_thes} \,/\, \texttt{num\_qnodes} \rceil \tag{4–11}$$

V2E、E2V、V2V 和 E2E 这四种 RP 的执行比较复杂，执行这些模式时输入元素列表中各变量的状态一定是 Known，输出元素列表中变量的状态是 Known 或 Unknown。接下来我们分别详细介绍这四种 RP 的执行逻辑，我们只讨论输出元素列表中变量的状态是 Unknown 的情况；对于 V2E、E2V 和 E2E 模式，我们只讨论包含 `interval` 的情况。

- **V2E 模式：**该模式的输入元素列表可以包含若干表示顶点的常量或变量。执行该模式时，首先需要使用键值对❺分别获取输入元素列表中的各常量对应的 `heid` 集合，然后求取这些集合的交集 s_1。接下来对于每行中间结果，再使用键值对❺分别获取输入元素列表中各变量取值对应的 `heid` 集合，求取这些集合的交集 s_2，然后求取 s_1 和 s_2 的交集 s，s 即是该变量可取值的集合。最后根据 `heid` 计算出偏移量，在“时序超边存储 1”中读取对应四元组中的生命周期信息即可。
- **E2V 模式：**该模式的输入元素列表可以包含若干表示时序超边的常量或变量。输入元素列表中的一个常量可能对应多条时序超边，查询引擎需要首先计算能够使得每个常量对应的时序超边都有且仅有一个有效的所有时间区间，记为 $itv_1, itv_2, ..., itv_n$（在图4–12中是红色虚线标出的三个时间区间），然后对于每一个时间区间，使用键值对❸分别获取每条有效的时序超边对应的 `vid` 集合并求交集，记得到的集合为 $s_1, s_2, ..., s_n$。接下来对于每行中间结果，计算能够使得输入元素列表中各变量取值表示的时序超边都有效的最大时间区间，记为 itv（在图4–12中是黑色虚线标出的时间区间，由于每个变量的取值只能表示一条时序超边，所以此处最多计算得到一个时间区间），然后使用键值对❸分别获取这些时序超边对应的 `vid` 集合，求取这些集合的交集 s，最后将时间区间 itv 分别与时间区间 $itv_1, itv_2, ..., itv_n$ 求交，得到新的时间区间 $itv'_1, itv'_2, ..., itv'_m$（在图4–12中是黑色双向箭头标出的两个时间区间）并计算每个新时间区间上 s_i 和 s 的交集 s'_i。最终，一行中间结果能够生成 $|s'_1| + |s'_2| + ... + |s'_m|$ 行新的中间结果。

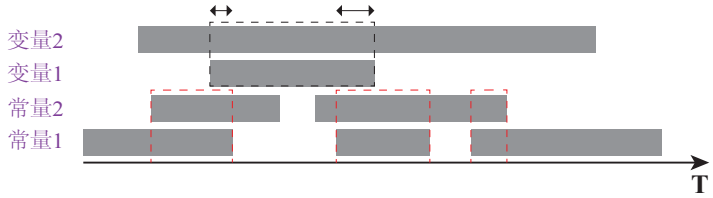


图 4–12 执行 E2V 模式时对时间区间的处理过程

Figure 4–12 Processing of time intervals when executing E2V pattern

- **V2V 模式：**该模式的输入元素列表可以包含若干表示顶点的常量或变量。执行该模式时，首先要为输出元素列表中的变量寻找候选取值，为此，查询引擎将包含输入元素列表中第一个常量（如果输入元素列表没有常量，则是第一个变量的取值）所表示的顶点的所有时序超边包含的所有顶点作为候选顶点集合。

确定候选顶点集合后，查询引擎会逐一验证每个候选顶点是否同时与各输入顶点出现在相同时序超边且时序超边的数量满足参数的要求，验证通过的候选顶点才会作为输出元素列表中变量的取值。

- **E2E 模式：**该模式的执行逻辑与 E2V 模式的执行逻辑相似，此外，执行此模式同样需要为输出元素列表中的变量寻找候选取值，然后验证候选取值是否能够使得参数指定的条件成立。该模式的详细执行过程不再赘述。

4.4　本章小结

本章介绍了 TempGraph 时序图查询模块的设计与实现。时序 RDF 图和时序超图都没有标准的查询语言，所以本文在 RDF 图查询语言 SPARQL 的基础上设计了时序 RDF 图查询语言 SPARQL-T，从零开始设计了时序超图查询语言 HQL-T。SPARQL-T 将 SPARQL 的 GP 中的三元模式扩展到五元模式，HQL-T 的 GP 则由基础关系模式（RP）组成。时序 RDF 图存储和时序超图存储的设计思路类似，都使用了分布式键值存储辅以分布式排序数组的存储结构，键值存储都使用了直接索引和间接索引相结合的设计，在实现了高效的拓扑查询和时间条件查询的同时，减少了时序数据存储占用的空间。SPARQL-T 的时序三元组时间范围模式和 HQL-T 的时序超边时间范围模式可以用来在特定条件下加速时间条件查询。SPARQL-T 和 HQL-T 查询引擎都沿用了 Wukong 使用的图探索和全历史剪枝算法，都使用了 fork-join 机制来加速复杂查询的执行。

第五章　时序图分析模块的设计与实现

TempGraph 的时序图分析模块运行在一个单独的分析节点上，同时处理图事务请求和第一类时序图分析任务，它包含的时间数据属于事务时间。本章将详细介绍一个高效可更新的、基于时序属性图模型的图存储结构 SegCSR/TS，然后介绍时序图分析模块使用的基于 epoch 的粗粒度 MVCC 机制和 SegCSR/TS 结合该机制的修改后版本 SegCSR，分析这些优化是如何带来读写效率的提升的。最后将介绍时序图分析模块的只读事务，以及如何基于只读事务在图分析引擎中实现时序图分析算法。

5.1　动态图存储结构 SegCSR/TS

图拓扑的 CSR 表示具有非常好的读性能，但它更新起来非常困难；邻接列表表示的更新效率更高，但它会牺牲读的性能。为了在实现时序图分析模块的高效图分析性能的同时，保证其良好的图更新效率，本文设计了一个高效的动态图存储 SegCSR/TS，它实现了与 CSR 接近的良好的数据局部性，同时也支持高效的图数据更新。

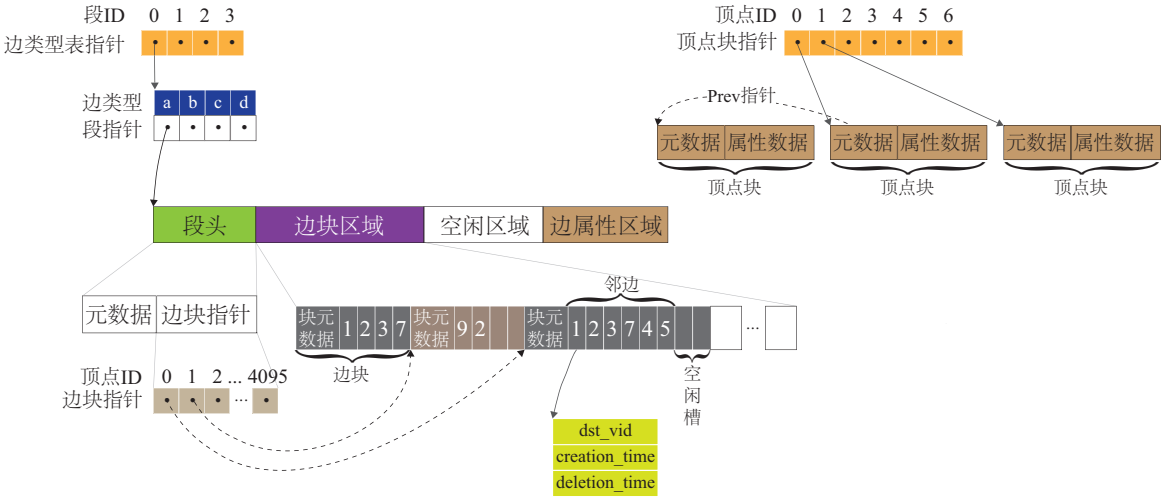


图 5–1　动态图存储 SegCSR/TS 的结构

Figure 5–1　The structure of dynamic graph storage SegCSR/TS

SegCSR/TS 的结构如图5–1。SegCSR/TS 使用段来管理固定数量（默认 4096）顶点的所有特定类型的邻边，如果使用 `Seg(sid,type)` 表示 ID 为 `sid`、管理的邻边类型为 `type` 的段，那么段 `Seg(0,a)` 管理的是 ID 为 0 – 4095 的顶点的所有类型为 `a` 的邻边，段 `Seg(1,b)` 管理的是 ID 为 4096 – 8191 的顶点的所有类型为 `b`

的邻边。SegCSR/TS 使用边类型表 `ETT(sid)` 来维护段 ID 为 `sid` 的管理不同邻边类型的段的地址，边类型表的每个表项都是一个边类型到段地址的映射。SegCSR/TS 使用一个边类型表地址数组来维护各边类型表的地址。

一个段由四部分组成，分别是段头、边块区域、空闲区域和边属性区域。段的初始大小是固定的（例如 1MB）。段头由两部分组成：段的元数据和指向各顶点边块的指针。段的元数据包括段 ID、段大小和已使用段空间大小等。在一个段中，每个顶点都对应一个最新的边块。边块是一块连续的内存区域，由块元数据和边数据区两部分组成。块的元数据包括顶点 ID、块大小和已使用块空间大小等。边数据区由描述边信息的结构体组成，该结构体包含三个字段，dst_vid 是边的终点 ID，creation_time 是该边的创建时间，deletion_time 是该边的删除时间，它的默认值是 INT64_MAX，表示它还未被删除。边块的初始大小固定，最初被分配出来时会有若干空闲槽，供后续边的插入。如果在插边时发现目标边块的空闲槽已经被用尽，分配器会从空闲区域里分配一个大小为目标边块 2 倍的新边块，然后把原边块里的数据都迁移到这个新边块里，之后相应顶点的邻边会插入这个新边块中。段头中每个顶点的边块指针指向的是其最新的边块。边块是自段头开始从左往右分配的，而边属性区域是自段尾开始从右往左增长的，边块区域里的边和边属性区域里的边属性数据是对称排布的（如图5–2），块元数据中包含该边块里的边的属性数据在边属性区域中的起始位置。我们要求一个段中各边的属性数据大小相同，以便能够为边块中的空闲槽预留边属性存储空间。

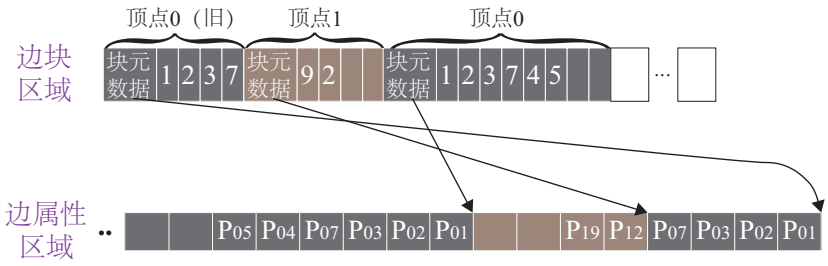


图 5–2 边和边属性数据的对称排布

Figure 5–2 Symmetrical layout of edge and edge property data SegCSR/TS

如果段的空闲区域不足以分配出新边块和边属性存储空间时，分配器会启动段的迁移过程：分配一个大小为原段二倍的新段，然后按照顶点 ID 的顺序把原段中每个顶点最新的边块和对应的边属性数据复制到新段里并更新每个顶点的边块指针，最后修改边类型表里的相应地址为新段的地址。

SegCSR/TS 使用顶点块来存储顶点的属性数据，顶点块由元数据区和属性数据区两部分组成，元数据区包含顶点 ID、属性数据大小、顶点创建时间和旧顶点块指针等。顶点属性数据更新时，新的顶点块会被创建，其元数据中的旧顶点块指针会指向较旧版本的顶点块，形成一个由不同版本的顶点块组成的链表结构。SegCSR/TS 使用一个顶点块地址数组来维护每个顶点的最新版本顶点块的地址。

SegCSR/TS 使用一个段来管理一定数量的顶点的所有特定类型的邻边，这使得同属于一个段的顶点的边块之间和邻边的属性数据之间都不会离得太远，大大提高了扫边时的数据局部性，从而显著提高扫边性能。段的空闲空间不足时的段迁移操作会按照顶点 ID 的顺序把原段中每个顶点的边块和邻边的属性数据复制到新段里，使得新段达到近似于 CSR 的瞬时数据局部性。

绝大多数的并发写问题都可以通过原子操作来解决，例如边块的并发分配可以通过对段头元数据中的已使用段空间大小字段的原子写来实现，对同一边块的并发插入可以通过对块元数据中的已使用块空间大小字段的原子写来实现。但以下两种并发问题需要使用锁来解决：

1. 段迁移过程中的并发问题。考虑以下场景：事务 A 和事务 B 在同时往段 S 中插边，事务 A 的目标边块有空闲槽，事务 B 的目标边块没有空闲槽，且段 S 的空闲区域不足以分配出一个新边块，这时分配器就要为事务 B 启动段 S 的迁移，如果段迁移时事务 A 还未提交（假设系统保证了快照隔离），那么迁移后的段就会丢失事务 A 插入的边。为了解决这种并发冲突，系统引入了段级别的读写锁：普通的边更新操作只需要尝试获取段的共享锁（读锁），而当该边更新操作会触发段的迁移时，它就会尝试获取段的互斥锁（写锁）。
2. 边块迁移过程中的并发问题。该问题与1类似，系统使用顶点级别的互斥锁来解决这种并发冲突：边更新操作需要尝试获取起点的互斥锁。

5.2 粗粒度的 MVCC 机制

MVCC (multi-version concurrency control，多版本并发控制) 是一种实现数据库的并发读写的常用方法，它能够避免读操作造成的写操作的阻塞，当数据库中的某个数据值被更新时，MVCC 会为该值创建一个新版本，使得并发的读操作可以读到一个稳定的版本。SegCSR/TS 使用两个时间戳 creation_time 和 deletion_time 分别表示边的创建时间和删除时间，使用时间戳 creation_time 表示顶点的创建时间并使用链表结构将不同版本的顶点块组织起来，以此来实现 MVCC。实时图处理系统通常使用

细粒度的 MVCC 机制，例如 LiveGraph 使用了组提交策略[46]，每次提交都会将全局写版本号加 1，事务的本地写版本号是事务开始时的全局写版本号，它会被用于标识事务所操作数据的版本号。在图分析场景中，细粒度的 MVCC 机制是不必要的：图分析是一类相对繁重的计算任务，其执行时间要远远超过一个图事务的执行时间，无需为每次图事务的提交都准备一个不同的版本号，因为图分析任务对细粒度的版本号并不敏感。

为了减少版本数据带来的内存使用和额外计算开销，系统使用了一种基于 epoch 的粗粒度 MVCC 机制。系统会维护一个全局的 epoch ID，它从 0 开始，每隔几十微秒递增一次。如图5–3，两个相邻的 epoch 之间的屏障会使得上一个 epoch 中的所有事务都完成时，才会开始下一个 epoch 中事务的执行。事务的本地写版本号就是其所在 epoch 的 ID，它会被用于标识事务所更新数据的版本号。同一 epoch 中的事务可以并行执行，图更新引擎会保证这些事务不会有依赖关系（有依赖关系的事务会按照串行顺序被分配到不同的 epoch 中执行）。系统管理员在发起时序图分析任务时需要指定读版本号，时序图分析能够使用的最新稳定版本是 `ceid-1`，其中 `ceid` 是系统当前的 epoch ID。

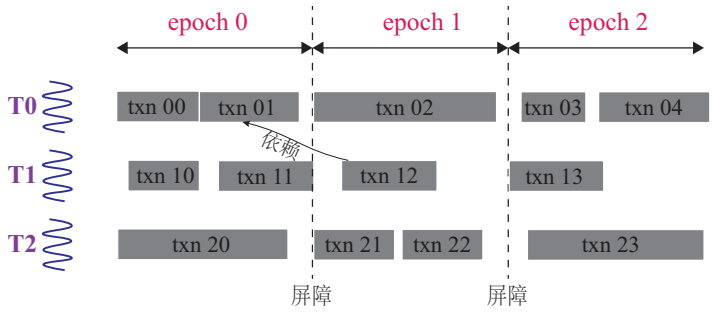


图 5–3　epoch 屏障示例

Figure 5–3　epoch barrier example

基于上述 MVCC 机制，我们对 SegCSR/TS 的结构进行了调整，使其充分从新的 MVCC 机制受益，调整后的存储结构称为 SegCSR。SegCSR 的结构如图5–4，与 SegCSR/TS 相比，它将边块中描述边信息的结构体简化为只包含表示边终点 ID 的 dst_vid，边块中各边的版本信息（即添加各边的事务的本地写版本号）被统一存储在一个 epoch 表中。系统会为段管理的每个顶点都分配一块连续内存作为 epoch 表，epoch 表负责维护顶点的最新边块里的所有边的版本信息，段头会存储每个顶点的 epoch 表的地址。边块中的边是只追加的（append-only），相同版本号的边会连续存储，

epoch 表存储的是每个 epoch 里添加的第一条边在边块里的逻辑偏移量，例如图5–4中顶点 0 的 epoch 表的含义就是偏移量为 0~1 的边是在 epoch 0 被插入的，偏移量为 2~4 的边是在 epoch 3 被插入的，偏移量为 5 的边是在 epoch 4 被插入的。

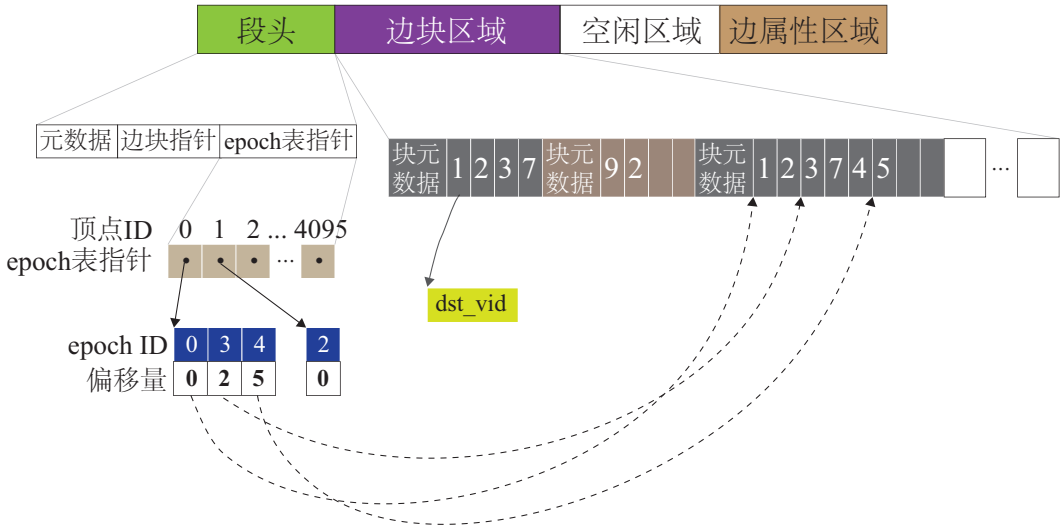


图 5–4 SegCSR 的结构

Figure 5–4 The structure of SegCSR

边的插入。假设要插入的边是 (u, v)，其类型为 ty。首先通过边类型表地址数组找到边类型表，然后在边类型表中查找类型 ty 对应的表项，即为目标段的地址。接下来通过目标段的段头中顶点 u 的边块地址即可定位到目标边块，如果目标边块还有空闲槽，则直接把 v 插入第一个空闲槽即可，否则启动边块的迁移，然后再把 v 插入新边块的第一个空闲槽。边的属性数据要插入边属性区域中的对称位置，相关的元数据（例如块元数据中已使用块空间大小等）也要更新。如果事务的 epoch ID 已经存在于顶点 u 对应的 epoch 表中，则 epoch 表无需改变，否则需要在 epoch 表中增加一个表项 `<teid,off>`，其中 `teid` 就是事务的 epoch ID，`off` 是 v 在边块中的偏移量。

边的删除。由于边块中的边数据是只追加的，所以边的删除需要通过边数据的添加来实现。使用同样的方法定位到目标边块后，通过在边块中添加一个删除标记来记录被删除的边在边块中的偏移量。如果在图分析的扫边过程中遇到删除标记，被删除的边会被直接跳过。

垃圾回收。后台的垃圾回收线程负责过时数据的清理，SegCSR 的过时数据包括：

- 旧段。段迁移完成后，新段的地址会被注册到边类型表，后续段寻址就会被转

移到新段。由于旧段上可能还有未完成的读，所以旧段不能在段迁移完成后就被立即回收。垃圾回收线程会监控旧段的使用，旧段上的所有读都完成后，垃圾回收线程就会将其回收。

- 被删除的边。为了实现边块的只追加性，系统使用删除标记来记录边的删除。后台的垃圾回收线程会定期对边块中的删除标记进行整理，将删除标记和对应的被删除的边物理地从边块移除。

5.3　时序图分析算法的实现

时序图分析模块提供了只读事务 RO_TXN，时序图分析算法都是基于 RO_TXN 实现的。表5–1给出了与 RO_TXN 相关的接口。

表 5–1 RO_TXN 相关接口

Table 5–1 RO_TXN related interfaces

接口	作用	类型
`ROTxn begin_ROTxn(reid)`	创建一个读 epoch ID 为 `reid` 的只读事务	SegCSR 的成员函数
`Seg* locate_seg(sid,etype)`	定位段 `Seg(sid,etype)`	ROTxn 的成员函数
`EdgeIter get_edges(segptr,vid)`	获取地址为 `segptr` 的段中顶点 `vid` 的邻边的迭代器	
`sid_t max_sid get_max_sid()`	获取最大的段 ID	
`bool valid()`	迭代器是否迭代完毕	EdgeIter 的成员函数
`void next()`	迭代到下一条边	
`vertex_t dst_id()`	当前边的终点 ID	
`string edge_data()`	当前边的属性数据	
`size_t size()`	剩余未迭代到的边的数量	

算法5–1使用这些接口实现了一个简单的时序图分析算法，它扫描版本（epoch ID）`reid` 下所有顶点的所有有效邻边，同时把边的属性数据打印出来。首先需要通过 begin_ROTxn 接口创建一个读 epoch ID 为 `reid` 的只读事务（第 1 行），然后使用 RO_Txn 提供的 get_max_sid 接口获取该版本下最大的段 ID（第 2 行），接着逐个遍历每个段（3-12 行）。在遍历一个段的过程中，需要逐一扫描该段管理的每个顶点的边块：通过 RO_Txn 提供的 get_edges 接口获取顶点邻边的迭代器（第 7 行），依次打印迭代到的每条边的属性数据即可（8-12 行）。

算法 5–1 一个简单的时序图分析算法的实现

```
1  txn = graph.begin_ROTxn(reid);
2  max_sid = txn.get_max_sid();
3  for sid = 0 to max_sid :
4      for etype in etypes :
5          segptr = txn.locate_seg(sid,etype);
6          for vid = sid * NUM to (sid + 1) * NUM - 1 :
7              iter = txn.get_edges(segptr,vid);
8              while iter.valid() :
9                  dst_id = iter.dst_id();
10                 data = iter.edge_data();
11                 print(data);
12                 iter.next();
```

目前，时序图分析模块已经实现了多种常用的图分析算法，包括广度优先搜索（BFS）、PageRank、单源最短路径（SSSP）和强连通分量（SCC）等。

5.4 本章小结

本章介绍了 TEMPGRAPH 时序图分析模块的设计与实现。系统的时序图分析模块使用了一个高效可更新的、基于时序属性图模型的图存储结构 SegCSR，它使用段（一块连续的内存空间）来管理固定数量顶点的所有特定类型的邻边，这使得同属于一个段的顶点的边块之间和邻边的属性数据之间都不会离得太远，大大提高了扫边时的数据局部性，从而显著提高扫边性能。为了减少时序数据带来的内存使用和额外计算开销，SegCSR 使用了一种基于 epoch 的粗粒度 MVCC 机制。最后，时序图分析模块提供了只读事务 RO_TXN，时序图分析算法都是基于 RO_TXN 实现的。

第六章 实验与评测

TempGraph 是一个高效的时序图处理系统，为了检验系统的性能，本文设计了一系列实验从多个角度对系统的性能进行全面的评测。本章将对各实验的设计和结果进行详细的介绍，并给出各实验数据的详细分析。

6.1 实验配置

实验环境 系统运行在一个由 8 台机器组成的集群中，其中一台机器用作字符串服务器节点，一台机器运行分析节点，其余 6 台机器运行查询节点。每台机器都配备了两个 Intel Xeon E5-2650 CPU，每个 CPU 都有 8 个核心、16 个超线程。每台机器的内存都是 128GB，都配备一张 Mellanox ConnectX-5 100Gbps InfiniBand 网卡。8 台机器通过一台 Mellanox 100Gbps InfiniBand 交换机互联。所有机器运行的操作系统都是 Ubuntu 16.04 (4.15.0-46-generic)，InfiniBand 网卡驱动为 MLNX_OFED_LINUX-4.9-3.1.5.0。其他软件版本为 gcc/g++ v7.4.0、cmake v3.22.0 和 Python v3.8。

使用的数据集 实验使用的 RDF 数据集是开源项目 LUBM[47]生成的数据集。LUBM 是里海大学发布的用于评测 RDF 数据集管理系统的查询性能的基准测试项目，LUBM 数据集描述了大学里各种实体之间的关系，例如学生、教师、学院和课程等。生成数据集时，可以通过指定数据集包含的大学的数量来设置生成的数据集的规模，实验使用的数据集是 LUBM 10240，它包含 337M 个实体，1410M 个三元组。LUBM 数据集本身不包含时序数据，我们为每个三元组都随机生成了两个 64 位 INT 型时间戳 t_1 和 t_2（$t_1 < t_2$，时间戳是从 1970-01-01T00:00:00Z 开始经过的秒数），分别表示三元组有效时间区间的开始时间和截止时间，得到的时序数据集称为 tLUBM 10240。

实验使用的时序超图数据集是一个现实的金融数据集 FIN，它描述了金融领域中的上市公司、基金、客户和供应商等之间的关系，包含 661K 条时序超边和 360K 个顶点。

图存储结构 SegCSR/TS 和 SegCSR 的微观性能评测使用了 LDBC SNB[48]、Wiki[49]、R-MAT[50]、UK-2005[51]和 Twitter-2010[52]5 个图数据集。LDBC SNB 是一个图分析系统的基准测试项目，它包含一个社交网络图数据集生成器和多种图分析工作负载。LDBC SNB 数据集是一个描述社交网络中人、论坛、帖子和评论等实体之间关系的图，在生成数据集时，可以通过指定规模因子 (scale factor, SF) 来控制生成的数据集

的大小，实验使用的数据集 SF=10。Wiki、R-MAT、UK-2005 和 Twitter-2010 都是现实的图数据集。5 个图数据集的统计信息见表6–1。

时序图分析模块的事务处理性能和图分析性能评测使用的是 LDBC SNB (SF=10) 基准测试，数据集会通过数据生成器提前生成，所有顶点和 50% 的边会被批量加载到系统中，剩余 50% 的边会通过客户端发起的图更新事务插入到系统中。

表 6–1 LDBC SNB (SB)、Wiki (WK)、R-MAT (RM)、UK-2005 (UK) 和 Twitter-2010 (TW) 5 个图数据集的统计信息

Table 6–1 Statistics of 5 graph datasets: LDBC SNB (SB), Wiki (WK), R-MAT (RM), UK-2005 (UK) and Twitter-2010 (TW)

图数据集	SB	WK	RM	UK	TW
|V|	7.5M	5.7M	5.0M	39.5M	41.7M
|E|	8.8M	130M	300M	936M	1.47B

实验基准 为了评测系统处理时序 RDF 图查询的性能，我们将 TempGraph 与 NebulaGraph 和 Neo4j 两个系统进行了对比。NebulaGraph 和 Neo4j 都是基于属性图模型的图数据库，NebulaGraph 支持分布式的存储和图查询处理，而 Neo4j 是单机图数据库。为了实现 NebulaGraph 和 Neo4j 对 tLUBM 10240 数据集的存储，我们将 tLUBM 10240 通过表6–2中列出的映射转换为属性图数据集。NebulaGraph 和 Neo4j 支持的图查询语言分别是 nGQL 和 Cypher，SPARQL-T 查询语句都可以转换成这两种查询语言。

表 6–2 时序 RDF 图到属性图的映射

Table 6–2 Mapping from temporal RDF graph to property graph

时序 RDF 图	属性图
时序三元组中的主语和宾语（表示类型名的宾语除外）	顶点名
表示类型名的宾语	顶点类型
时序三元组中的主语、谓词和宾语（描述主语类型的时序三元组除外）	主语对应的顶点指向宾语对应的顶点的有向边，边的类型是谓词
时序三元组中的有效时间区间（描述主语类型的时序三元组除外）	对应边的两个属性
描述主语类型的时序三元组的有效时间区间	主语对应的顶点的两个属性

由于超图查询系统相关研究工作很少，所以我们并没有找到合适的基准时序超图

查询系统来与 TEMPGRAPH 作对比，我们的方案是将与时序超图数据集 FIN 等价的时序 RDF 图数据集加载到 TEMPGRAPH 的时序 RDF 图存储中，然后通过等价的 HQL-T 和 SPARQL-T 查询语句来对比在查询等价数据时，TEMPGRAPH 的时序超图查询的性能和时序 RDF 图查询的性能。将时序超图数据集转换为时序 RDF 图数据集的方法为：将类型为 `ty`、名称为 `name`、连接 n 个顶点 $v_1, v_2, ..., v_n$、有效时间区间为 $[t_1, t_2)$ 的时序超边转换为 n 个时序三元组 $\{(\texttt{name ty } v_i\ [t_1, t_2)) \mid 1 \le i \le n, i \in Z\}$，将类型为 `ty`、名称为 `name` 的顶点转化为时序三元组 `name rdf:type ty` $[-\infty, +\infty)$。

为了对图存储结构 SegCSR/TS 和 SegCSR 进行微观性能评测，我们选取了两种基准图存储结构 CSR 和 LiveGraph 的存储结构来进行比较。LiveGraph 的存储结构使用了高度优化过的邻接列表格式来存储图拓扑和属性数据，同时它使用的是一种细粒度的 MVCC 机制。我们使用 GraphScope[53]、Neo4j 和 LiveGraph 三个支持图分析的基准系统来与 TEMPGRAPH 的图事务处理性能和图分析性能进行对比。GraphScope 是一个先进的高性能离线图分析系统，由于它不支持图更新，所以我们会将图事务数据定期加载到其中。Neo4j 虽然是一个图数据库，但它的图数据科学 (Graph Data Science, GDS) 库实现了在图的最新快照上的图分析。这些系统上运行了三类图分析工作负载：

- 三个常用图分析算法：网页排序 (PageRank, PR)、强连通分量 (Strong Connected Components, SCC) 和单源最短路径 (Single Source Shortest Path, SSSP)；
- LDBC SNB 提供的一个交互式查询 IS-3 和两个商业智能查询 BI-2 和 BI-3；
- 三个图神经网络模型：GCN[54](Graph Convolution Network)、GSG[55] (GraphSage) 和 SGC[56] (Simple Graph Convolution)。

6.2 时序 RDF 图查询性能

本文首先使用 tLUBM 10240 数据集对 TEMPGRAPH 和各基准系统（NebulaGraph 和 Neo4j）的单个时序 RDF 图查询处理性能进行对比测试。NebulaGraph 运行在 6 台机器上，而由于 Neo4j 是单机系统，受存储空间限制，它无法完全加载 tLUBM 10240 数据集，所以我们仅让它们加载全部数据的 4%。实验使用了 6 条 SPARQL-T 查询语句 Q1-Q6 作为工作负载，其中 Q1 和 Q2 是小查询、Q3 和 Q4 是大查询、Q5 和 Q6 包含时序三元组时间范围模式。表6–3给出了各系统执行这 6 条查询的时延情况。与 NebulaGraph 相比，TEMPGRAPH 处理各查询都有一个数量级以上的性能优势，尤其是对于包含时序三元组时间范围模式的查询（Q5 和 Q6），TEMPGRAPH 的性能领先更加

明显。TEMPGRAPH 的性能提升主要得益于 TEMPGRAPH 围绕 RDMA 的一系列存储结构和查询引擎上的设计和对时间条件查询的特殊优化。即使 Neo4j 存储的数据规模只有 TEMPGRAPH 的 4%，它也需要消耗数倍以上的时间来处理相同的图查询请求。

表 6–3 Q1-Q8 执行时延（毫秒）对比

Table 6–3 Q1-Q8 execution latency (msec) comparison

	TEMPGRAPH	NebulaGraph	Neo4j
Q1	**0.26**	12.00	13.25
Q2	**0.31**	26.86	4.79
Q3	**172.02**	3729	272
Q4	**153.87**	3910	198
Q5	**3.70**	2791	245
Q6	**19.59**	1977	203

本文又测试了 TEMPGRAPH 的并发查询处理性能，在本实验中，为了模拟高并发场景，系统为每个查询节点都创建 4 个代理线程和 16 个工作线程，每个代理线程都会不断地将 Q1 交给工作线程执行，代理线程会统计 Q1 的执行时延分布。图6–1是 Q1 的执行时延的累积分布函数（CDF），在高并发场景下，Q1 的执行时延平均是执行单个 Q1 的 8 倍左右，P99 尾时延也在 10 毫秒以内，这说明 TEMPGRAPH 的 FIFO 查询请求执行模式可以使得其在高并发的场景下仍能保证每条查询请求的执行时延不至于过大。

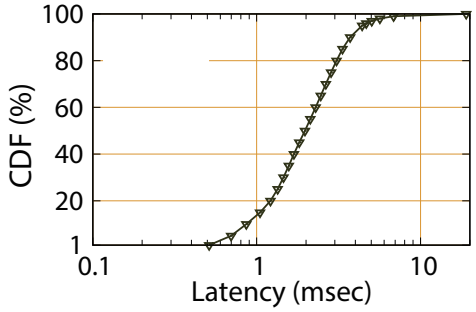


图 6–1 Q1 执行时延的 CDF

Figure 6–1 The CDF of latency for Q1

最后，本文从以下两个角度来测评 TEMPGRAPH 处理时序 RDF 图查询的可扩展性：

- 通过设置每个（共 6 个）查询节点上不同的工作线程数量，来研究系统的多线程可扩展性；
- 通过设置不同的查询节点数量，来研究系统的多机可扩展性。

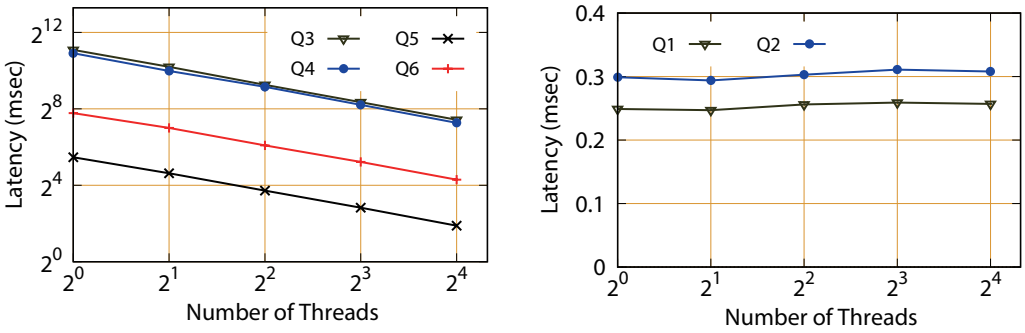


图 6–2 不同工作线程时 Q1-Q6 的执行时延

Figure 6–2 The latency of Q1-Q6 with different worker threads

图6–2展示了 Q1-Q6 的执行时延与工作线程数量的关系，工作线程数量的增加能够给 Q3-Q6 带来接近线性的性能提升，而对 Q1 和 Q2 的执行时延几乎没有影响。这是因为 Q3-Q6 属于大查询，查询引擎会使用 fork-join 机制来处理这些查询，随着工作线程数量的增加，增加的工作线程会参与请求的执行；而 Q1 和 Q2 是小查询，系统只会使用一个工作线程来处理这两条查询，所以它们的执行时延和工作线程数量几乎是无关的。

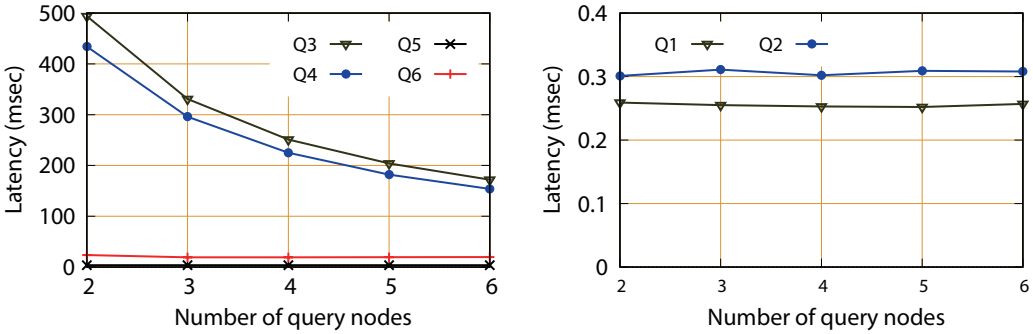


图 6–3 不同查询节点时 Q1-Q6 的执行时延

Figure 6–3 The latency of Q1-Q6 with different query nodes

图6–3是 Q1-Q6 在不同查询节点配置下的执行时延，Q3 和 Q4 都是不包含时序三元组时间范围模式的大查询，查询节点数量的增加能够给这类查询带来接近线性的性能提升。查询节点数量的增加给 Q4 和 Q5 带来的性能提升不明显，原因是“时序

三元组存储 1”是按块划分到各查询节点上的，只有少数查询节点会存储符合查询指定时间条件的时序三元组。查询节点数量对 Q1 和 Q2 这类小查询的性能几乎没有影响。

6.3 时序超图查询性能

为了评测 TempGraph 的时序超图查询的处理性能，本文首先构造了只包含单个 SP 的基础 HQL-T 查询 Q1-Q7，由于 FIN 数据集中的顶点没有类型，所以我们没有构造 GV 查询。表6–4给出了这些查询的基本信息及 TempGraph 处理这些基础查询和与它们等价的 SPARQL-T 查询的时延对比。对于这些基础查询，TempGraph 都能达到亚毫秒级的执行时延。对于有些类型的 SP（例如 containEdges 和 inEdges 等），HQL-T 的查询效率更高，而有些类型的 SP 更适合转换为等价的 SPARQL-T 语句来执行，这些性能上的差异与数据集的特征有关。与时序 RDF 图模型相比，时序超图模型更适合用来解决与集合关系有关的查询，因为 HQL-T 主要是为集合关系查询设计的一种查询语言，而 SPARQL-T 则需要使用较为复杂的查询语句才能实现集合关系查询。

表 6–4 TempGraph 处理 HQL-T 查询 Q1-Q7 和与它们等价的 SPARQL-T 查询的时延（微秒）对比

Table 6–4 Comparison of the latency (usec) of processing HQL-T queries Q1-Q7 and their equivalent SPARQL-T queries

	类型	HQL-T	SPARQL-T
Q1	etype	1215	35
Q2	edges	45	35
Q3	vertices	46	34
Q4	intersectVertices	93	182
Q5	intersectEdges	237	150
Q6	containEdges	81	285
Q7	inEdges	65	148

最后，本文构造了一些包含多条模式的复杂 HQL-T 查询（Q8-Q14），表6–5给出了这些查询语句的执行时延对比。

表 6–5 TEMPGRAPH 处理 HQL-T 查询 Q8-Q14 和与它们等价的 SPARQL-T 查询的时延（毫秒）对比

Table 6–5 Comparison of the latency (msec) of processing HQL-T queries Q8-Q14 and their equivalent SPARQL-T queries

	HQL-T	SPARQL-T
Q8	277.0	1266.8
Q9	17.9	7.7
Q10	8.7	1.0
Q11	1147.1	1037.8
Q12	268.7	499.9

6.4 SegCSR/TS 和 SegCSR 微观性能

本节将对 SegCSR/TS、SegCSR、CSR 和 LiveGraph 这四种图存储结构在内存使用、写吞吐量和读吞吐量等方面的微观性能进行比较。为了模拟不同的真实工作负载，我们分别使用以下两种模式来将相同的图数据加载到各存储结构里：

- 顺序插入：按照边的起点的 ID 的顺序插入各边，这模拟的是批量加载图数据集的场景；
- 随机插入：以随机的顺序插入各边，这模拟的是图的事务化更新场景。

本文首先测试了各存储结构上的单版本扫边性能，单版本扫边不需要版本检查，只涉及对图拓扑的扫描。实验使用每秒扫描到的边的数量来作为各种图存储结构的扫边吞吐量。如图6–4，CSR 作为最紧凑的、数据局部性最好的存储结构，是理论上扫边性能最好的。顺序插入时，各种图存储结构在不同数据集上表现出的相对性能基本一致。例如，与 CSR 相比，SegCSR/TS 扫描 Wiki 数据集的性能下降约为 68%，而 LiveGraph 的性能下降超过 80%，这主要得益于 SegCSR/TS 的段迁移过程能够使得各顶点的边块位置更加接近，实现更好的数据局部性，从而可以更好地从 CPU 的预取机制受益。SegCSR 的扫边吞吐量超过 SegCSR/TS 的 2 倍，主要原因是 SegCSR 简化了边块中的边的数据结构（移除了两个时间戳）。LiveGraph 中各顶点的邻接列表在内存中的位置是互不相关的，而 SegCSR/TS 会将 ID 相近的顶点的邻接列表通过一块段内存空间关联起来，这种设计带来的性能提升在随机插入场景下尤为明显：SegCSR/TS 的扫边吞吐可达 LiveGraph 的 2~5.9 倍，原因是随机插入使得 LiveGraph 中各顶点的邻接列表在内存中的分布更加混乱，而 SegCSR/TS 仍能实现良好的数据局部性。

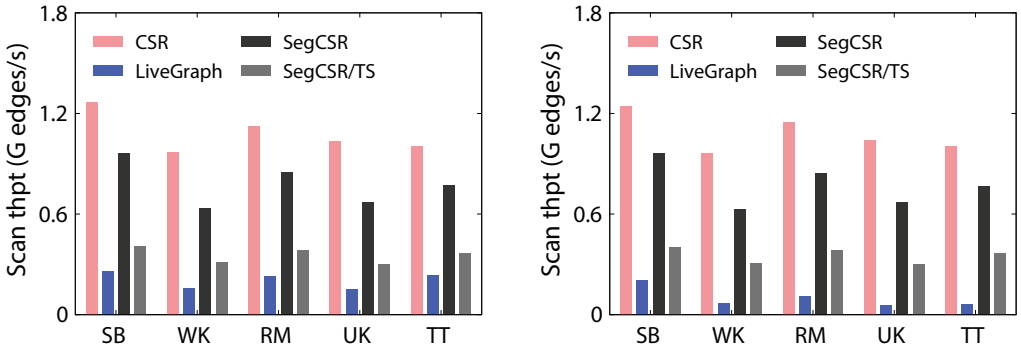


图 6–4 顺序插入（左）和随机插入（右）时不同图存储结构扫边吞吐量的对比

Figure 6–4 Comparison of edge scan throughput of different graph storage structures when sequential insertion (left) and random insertion (right)

图6–5给出了各种图存储结构的内存使用对比。为了实现高效的图更新，SegCSR 需要使用大约 3 倍于 CSR 的内存，但这比 SegCSR/TS 和 LiveGraph 的要少得多。粗粒度的 MVCC 机制使得 SegCSR 不用像 SegCSR/TS 和 LiveGraph 一样为每条边都维护两个表示其生命周期的时间戳，只需要为每个顶点维护一个 epoch 表即可，这大大减少了时序数据的内存使用。由于段中存在空闲空间，SegCSR/TS 的内存使用量比 LiveGraph 稍高，但通常多出不超过 20%。

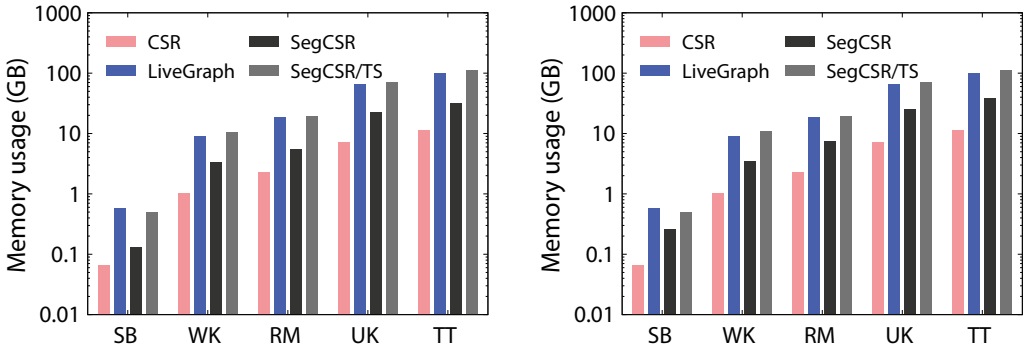


图 6–5 顺序插入（左）和随机插入（右）时不同图存储结构内存使用量的对比

Figure 6–5 Comparison of memory usage of different graph storage structures when sequential insertion (left) and random insertion (right)

图6–6给出了各种图存储结构的写吞吐量对比，实验使用每秒插入边的数量来作为各种图存储结构的写吞吐量，由于 CSR 不支持更新，所以图中没有 CSR 的写吞吐量数据。在顺序插入和随机插入两种情况下，SegCSR 的写吞吐量分别为 LiveGraph 的 2.2 倍和 2 倍。结合 SegCSR/TS 的性能表现，SegCSR 相对于 LiveGraph 的性能提

升主要来源于粗粒度的 MVCC 机制：不需要为插入的每条边都写入版本号，只有在有新的版本（epoch）产生时才需要往边的起始顶点的 epoch 表中插入一个值。性能提升的另一个来源是二者不同的垃圾回收机制：SegCSR 的垃圾回收是由后台线程完成的，不在图更新操作的关键路径上，而 LiveGraph 的图更新操作则有可能触发在关键路径上的垃圾回收操作。SegCSR 的段迁移过程比较耗时，平均时延大约是普通插边操作的 7000 倍，这会导致图更新具有较高的尾延迟，但段迁移触发的频率较低（小于 0.01%）。

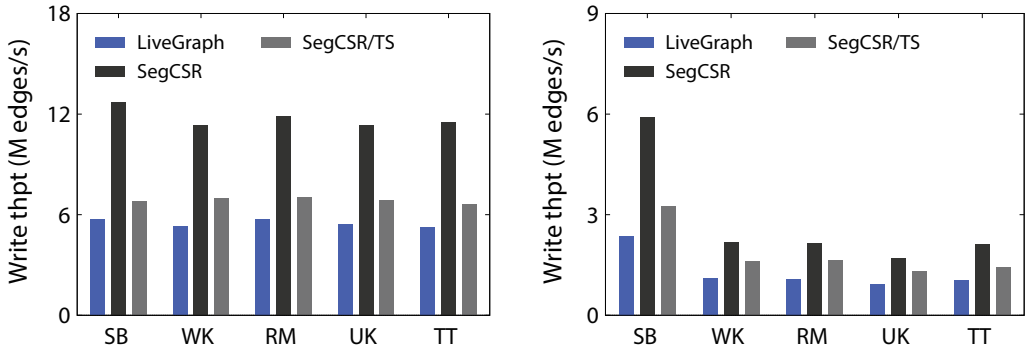


图 6–6 顺序插入（左）和随机插入（右）时不同图存储结构写吞吐量的对比

Figure 6–6 Comparison of write throughput of different graph storage structures when sequential insertion (left) and random insertion (right)

6.5 图事务处理性能和图分析性能

表6–6列出了 GraphScope、Neo4j 和 LiveGraph 三个支持图分析的基准系统与 TempGraph 的图事务处理吞吐量和图分析时延的对比测试结果。

图事务处理方面，TempGraph 的性能领先 LiveGraph 约 31%，性能提升主要来源于 TempGraph 使用的粗粒度的 MVCC 机制，它减少了 TempGraph 在处理图事务时对时序数据的写入。Neo4j 的图事务处理性能则是远远落后于 TempGraph 和 LiveGraph，原因可能是 Neo4j 对图的更新需要涉及复杂的索引更新操作。

图分析处理方面，GraphScope 的图分析性能是最好的（除了 IS-3，它是一个非常简单的图分析任务，16.9 毫秒主要是 GraphScope 系统层面的开销），原因是它是一个离线图分析系统，使用了一种紧凑的图存储结构，且不需要版本数据的管理。TempGraph 的图分析性能是 LiveGraph 的 1.4~4.4 倍，这些性能提升的主要来源是 SegCSR 基于段的边块管理策略和粗粒度的 MVCC 机制带来的数据访问量的下降和

数据局部性的提升。Neo4j 的图分析性能要远远落后于 TemporalGraph，尤其是对于三个图分析算法 PR、SCC 和 SSSP，二者的性能差距达到了一个数量级左右，原因是 Neo4j 的 GDS 库在执行这三个图分析算法时，会先把图的最新快照投影到内存中，然后在投影上执行图分析计算，投影过程消耗了大量时间。

表 6–6 TemporalGraph、GraphScope、Neo4j 和 LiveGraph 的图事务处理吞吐量和图分析时延对比
Table 6–6 Comparison of graph transaction processing throughput and graph analysis latency of TemporalGraph, GraphScope, Neo4j and LiveGraph

	TemporalGraph	GraphScope	Neo4j	LiveGraph
图事务	279K	×	3.5K	213K
PR	377	309	5323	1276
SCC	362	312	4726	1137
SSSP	513	433	4668	1381
IS-3	0.0012	16.9	2.0	0.0039
BI-2	235	201	568	828
BI-3	292	266	573	1278
GCN	1097	940	×	1834
GSG	1774	1443	×	2502
SGC	779	717	×	1237

6.6　本章小结

本章对 TemporalGraph 的性能进行了全面的评测。时序 RDF 图查询方面，系统的查询执行时延和基准系统 NebulaGraph 和 Neo4j 相比都能取得大幅度的性能领先；对于大查询，系统能够实现优秀的多线程可扩展性和多机可扩展性；在高并发场景下，系统也能够保证较低的平均执行时延和尾时延。时序超图查询方面，系统能够实现基础查询的亚毫秒级执行时延；系统处理时序超图和时序 RDF 图查询的性能接近，具体为图数据选择何种图模型取决于实际工作负载。图分析方面，微观性能评测实验表明，SegCSR 能够通过更少的内存使用量，实现更高的读、写吞吐量（与 LiveGraph 相比）；TemporalGraph 具有优秀的图事务处理性能和图分析性能。

第七章 总结和展望

我们正处于大数据时代，各种应用形态正不断地、越来越快地产生大量相互关联的数据。图是一种用于建模数据实体之间关系的抽象数据结构，可以被抽象成图结构的数据称为图状结构数据。

图的更新、查询和分析是三类最为重要的图操作，图数据库通常实现了图的更新和查询，而图分析通常由离线的图分析系统完成。离线图分析系统通常具有较高的图分析性能，但它不可避免地牺牲了时效性，在线图分析系统同时实现了图的更新和分析，能够在图更新完成后的很短时间内对其进行分析，大大提高了图分析的时效性，但这要以大幅牺牲图分析性能为代价。

随着时间的推进，图数据可能是不断变化的。随着时间而变化的图叫做时序图。普通图查询和图分析面向的通常是静态图或时序图的最新版本，而时序图查询不仅可以面向时序图的特定版本（包括最新版本和历史版本）进行基于图拓扑的查询，也可以基于时序图中顶点/边的生命周期信息进行查询。时序图分析同样既可以是在时序图的特定版本上的普通图分析任务（第一类时序图分析），也可以使用时序图的生命周期信息运行更为复杂的图分析算法（第二类时序图分析）。属性图、RDF 图和超图是定义图结构的三大模型范式，它们都有各自的时序版本。面向时序 RDF 图和时序超图的时序图查询并不是热点研究方向，相关系统研究工作不多，且性能难以令人满意。

本文基于上述背景，针对现有时序图处理系统的不足，设计并实现了一个高效的时序图处理系统 TempGraph。

7.1 全文总结

本文首先介绍了时序图处理系统的研究现状和相关的技术背景，然后详细介绍了 TempGraph 的设计和实现，最后通过一系列实验对系统的性能进行全面的评测。

TempGraph 是一个同时支持时序图查询和第一类时序图分析的图处理系统，它由时序图查询和时序图分析两大模块组成。TempGraph 将时序属性图、时序 RDF 图和时序超图三种时序图模型集成到同一系统中。具体来说，TempGraph 的时序图查询模块同时支持时序 RDF 图和时序超图两种时序图模型，而时序图分析模块则是基于时序属性图模型实现的。

TempGraph 系统运行在一个由 RDMA 网络相互连接的集群环境中，集群包含查询节点、分析节点和字符串服务器节点三类节点。查询节点用于执行来自用户的时序 RDF 图查询请求和时序超图查询请求，分析节点用于处理图事务和时序图分析请求，字符串服务器节点用于处理字符串和整型 ID 之间的转换。TempGraph 的时序图查询模块采用了去中心化的分布式架构，而时序图分析模块则是单机的。每个查询节点和分析节点都由存储层、引擎层和接口层三层组成。存储层负责图数据的存储，引擎层负责图更新、查询和分析任务的执行，接口层负责接收来自于用户或系统管理员的请求，将它们解析为系统的内部表示，然后分配给特定节点的引擎层执行。接口层由分析节点和查询节点上的若干代理线程和为每个代理线程准备的专用缓冲区来实现，代理线程和工作线程之间可以通过单边 RDMA 操作实现高效的线程间通信，从而实现请求的分配和结果的获取。

时序图查询方面，本文在 RDF 图查询语言 SPARQL 的基础上设计了时序 RDF 图查询语言 SPARQL-T，从零开始设计了时序超图查询语言 HQL-T。时序 RDF 图存储和时序超图存储的设计思路类似，都使用了分布式键值存储辅以分布式排序数组的存储结构，键值存储都使用了直接索引和间接索引相结合的设计，在实现了高效的拓扑查询和时间条件查询的同时，减少了时序数据存储占用的空间。SPARQL-T 的时序三元组时间范围模式和 HQL-T 的时序超边时间范围模式可以用来在特定条件下加速时间条件查询。SPARQL-T 和 HQL-T 查询引擎都沿用了 Wukong 使用的图探索和全历史剪枝算法，都使用了 fork-join 机制来加速复杂查询的执行。

时序图分析方面，系统使用了一个高效可更新的、基于时序属性图模型的图存储结构 SegCSR，它使用段（一块连续的内存空间）来管理固定数量顶点的所有特定类型的邻边，这种设计能够显著提高扫边性能。为了减少时序数据带来的内存使用和额外计算开销，SegCSR 使用了一种基于 epoch 的粗粒度 MVCC 机制。系统的时序图分析模块提供了只读事务 RO_TXN，时序图分析算法都是基于 RO_TXN 实现的。

本文从多个角度对 TempGraph 的性能进行全面的评测，实验结果表明，系统能够实现时序 RDF 图查询和时序超图查询的高效处理，也能够在保证良好图事务处理性能的同时，实现第一类时序图分析的高效执行。

7.2　未来工作展望

大数据时代的背景下，时序图数据在产业界正得到越来越重要的应用，针对时序图数据的查询和分析是一个非常值得关注的研究领域。

TempGraph 虽然是一个高效的时序图处理系统，但它依然有其局限性。时序图查询方面，时序 RDF 图和时序超图存储都是从预先准备好的数据集中加载得到的，系统并不支持对时序 RDF 图和时序超图存储的更新，让时序图查询模块同样支持图的事务化更新，使其具备图数据库的功能是未来的一个工作方向。时序图分析方面，系统的时序图分析模块是单机的，要想支持更大规模图数据的存储和分析，系统就必须对其进行多机扩展，这同样是未来的一个工作方向。

参考文献

[1] BRIN S, PAGE L. The Anatomy of a Large-Scale Hypertextual Web Search Engine [C]//Proceedings of the Seventh International Conference on World Wide Web 7. Brisbane, Australia: Elsevier Science Publishers B. V., 1998: 107-117.

[2] MALEWICZ G, AUSTERN M H, BIK A J, et al. Pregel: A System for Large-Scale Graph Processing[C]//Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010: 135-146.

[3] HASLHOFER B, KARL R, FILTZ E. O Bitcoin Where Art Thou? Insight into Large-Scale Transaction Graphs[C]//International Conference on Semantic Systems. 2016.

[4] COOKE K L, HALSEY E. The shortest route through a network with time-dependent internodal transit times[J]. Journal of Mathematical Analysis and Applications, 1966, 14(3): 493-498.

[5] WU H, CHENG J, HUANG S, et al. Path Problems in Temporal Graphs[J]. Proc. VLDB Endow., 2014, 7(9): 721-732.

[6] Neo4j the graph database[Z]. https://neo4j.com/.

[7] Orientdb - distributed graph/document multi-model database[Z]. http://orientdb.com/.

[8] NebulaGraph[Z]. https://www.nebula-graph.io/.

[9] LOW Y, GONZALEZ J, KYROLA A, et al. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud[J]. Proc. VLDB Endow., 2012, 5(8): 716-727.

[10] KYROLA A, BLELLOCH G, GUESTRIN C. GraphChi: Large-Scale Graph Computation on Just a PC[C]//Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. Hollywood, CA, USA: USENIX Association, 2012: 31-46.

[11] GONZALEZ J E, LOW Y, GU H, et al. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs[C]//Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. Hollywood, CA, USA: USENIX Association, 2012: 17-30.

[12] GONZALEZ J E, XIN R S, DAVE A, et al. GraphX: Graph Processing in a Distributed

Dataflow Framework[C]//Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. Broomfield, CO: USENIX Association, 2014: 599-613.

[13] ZHU X, CHEN W, ZHENG W, et al. Gemini: A Computation-Centric Distributed Graph Processing System[C]//Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. Savannah, GA, USA: USENIX Association, 2016: 301-316.

[14] ZHU X, FENG G, SERAFINI M, et al. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans[J]. Proc. VLDB Endow., 2020, 13(7): 1020-1034.

[15] PAN Z, WU T, ZHAO Q, et al. GeaFlow: A Graph Extended and Accelerated Dataflow System[J]. Proc. ACM Manag. Data, 2023, 1(2).

[16] MARTÍNEZ-BAZAN N, ÁGUILA-LORENTE M Á, MUNTÉS-MULERO V, et al. Efficient Graph Management Based on Bitmap Indices[C]//Proceedings of the 16th International Database Engineering & Applications Sysmposium. Prague, Czech Republic: Association for Computing Machinery, 2012: 110-119.

[17] SHI J, YAO Y, CHEN R, et al. Fast and Concurrent RDF Queries with RDMA-Based Distributed Graph Exploration[C]//Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. Savannah, GA, USA: USENIX Association, 2016: 317-332.

[18] ANGLES R. The Property Graph Database Model[C]//Alberto Mendelzon Workshop on Foundations of Data Management. 2018.

[19] Resource Description Framework (RDF)[Z]. https://www.w3.org/RDF/.

[20] GURAJADA S, SEUFERT S, MILIARAKI I, et al. TriAD: A Distributed Shared-Nothing RDF Engine Based on Asynchronous Message Passing[C]//Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. Snowbird, Utah, USA: Association for Computing Machinery, 2014: 289-300.

[21] Franz Inc. AllegroGraph[Z]. http://www.franz.com/agraph/allegrograph/.

[22] ZENG K, YANG J, WANG H, et al. A Distributed Graph Engine for Web Scale RDF Data[J]. Proc. VLDB Endow., 2013, 6(4): 265-276.

[23] Ontotext GraphDB[Z]. https://www.ontotext.com/products/graphdb/.

[24] TANSEL A. Temporal relational data model[J]. IEEE Transactions on Knowledge and Data Engineering, 1997, 9(3): 464-479.

[25] TANSEL A U, CLIFFORD J, GADIA S, et al. Temporal Databases: Theory, Design, and Implementation[M]. USA: Benjamin-Cummings Publishing Co., Inc., 1993.

[26] 汤庸. 时态数据库导论[M]. 北京大学出版社, 2004.

[27] InfluxDB Open Source[Z]. https://www.influxdata.com/products/influxdb/.

[28] TimeDB - A Temporal Relational DBMS[Z]. https://www.timeconsult.com/Software/Software.html.

[29] HUANG H, SONG J, LIN X, et al. TGraph: A Temporal Graph Data Management System[C]//Proceedings of the 25th ACM International on Conference on Information and Knowledge Management. Indianapolis, Indiana, USA: Association for Computing Machinery, 2016: 2469-2472.

[30] HAN W, MIAO Y, LI K, et al. Chronos: A Graph Engine for Temporal Graph Analysis [C]//Proceedings of the Ninth European Conference on Computer Systems. Amsterdam, The Netherlands: Association for Computing Machinery, 2014.

[31] THEN M, KERSTEN T, GÜNNEMANN S, et al. Automatic Algorithm Transformation for Efficient Multi-Snapshot Analytics on Temporal Graphs[J]. Proc. VLDB Endow., 2017, 10(8): 877-888.

[32] GANDHI S, SIMMHAN Y. An Interval-centric Model for Distributed Computing over Temporal Graphs[C]//2020 IEEE 36th International Conference on Data Engineering (ICDE). 2020: 1129-1140.

[33] LIGHTENBERG W, PEI Y, FLETCHER G, et al. Tink: A Temporal Graph Analytics Library for Apache Flink[C]//Companion Proceedings of the The Web Conference 2018. Lyon, France: International World Wide Web Conferences Steering Committee, 2018: 71-72.

[34] GUTIERREZ C, HURTADO C, VAISMAN A. Temporal RDF[C]//Proceedings of the Second European Conference on The Semantic Web: Research and Applications. Heraklion, Greece: Springer-Verlag, 2005: 93-107.

[35] GUTIERREZ C, HURTADO C A, VAISMAN A. Introducing Time into RDF[J]. IEEE Transactions on Knowledge and Data Engineering, 2007, 19(2): 207-218.

[36] TAPPOLET J, BERNSTEIN A. Applied temporal RDF: efficient temporal query-

ing of RDF data with SPARQL[C]//The Semantic Web: Research and Applications: vol. 5554. Springer Berlin Heidelberg, 2009: 308-322.

[37] BERETA K, SMEROS P, KOUBARAKIS M. Representation and Querying of Valid Time of Triples in Linked Geospatial Data[C]//The Semantic Web: Semantics and Big Data. Springer Berlin Heidelberg, 2013: 259-274.

[38] ZANG S, HAN S, YUAN P, et al. HyperBit: A Temporal Graph Store for Fast Answering Queries[J]. Data Knowl. Eng., 2023, 144(C).

[39] YAN L, ZHANG Z, YANG D. Temporal RDF(S) Data Storage and Query with HBase [J]. J. Comput. Inf. Technol., 2020, 27: 17-30.

[40] SPARQL Query Language for RDF[Z]. http://www.w3.org/TR/rdf-sparql-query/.

[41] FENG G, MA Z, LI D, et al. RisGraph: A Real-Time Streaming System for Evolving Graphs to Support Sub-Millisecond Per-Update Analysis at Millions Ops/s[C]// Proceedings of the 2021 International Conference on Management of Data. Virtual Event, China: Association for Computing Machinery, 2021: 513-527.

[42] KUMAR P, HUANG H H. GraphOne: A Data Store for Real-Time Analytics on Evolving Graphs[J]. ACM Trans. Storage, 2020, 15(4).

[43] FIRMLI S, TRIGONAKIS V, LOZI J P, et al. CSR++: A fast, scalable, update friendly graph data structure[C]//24th International Conference on Principles of Distributed Systems. 2020.

[44] FUCHS P, MARGAN D, GICEVA J. Sortledton: A Universal, Transactional Graph Data Structure[J]. Proc. VLDB Endow., 2022, 15(6): 1173-1186.

[45] KARYPIS G, KUMAR V. Parallel Multilevel K-Way Partitioning Scheme for Irregular Graphs[C]//Proceedings of the 1996 ACM/IEEE Conference on Supercomputing. Pittsburgh, Pennsylvania, USA: IEEE Computer Society, 1996: 35-es.

[46] BERNSTEIN P A, HADZILACOS V, GOODMAN N. Concurrency Control and Recovery in Database Systems[M]. Addison-Wesley, 1987.

[47] LUBM[Z]. http://swat.cse.lehigh.edu/projects/lubm/.

[48] LDBC SNB - LDBC Social Network Benchmark[Z]. https://ldbcouncil.org/benchmarks/snb/.

[49] Wikipedia page-to-page link database[Z]. http://haselgrove.id.au/wikipedia.htm.

[50] CHAKRABARTI D, ZHAN Y, FALOUTSOS C. R-MAT: A recursive model for

graph mining[C]//Proceedings of the 2004 SIAM International Conference on Data Mining. 2004: 442-446.

[51] BOLDI P, CODENOTTI B, SANTINI M, et al. UbiCrawler: A scalable fully distributed web crawler[J]. Software: Practice and Experience, 2004, 34(8): 711-726.

[52] KWAK H, LEE C, PARK H, et al. What is Twitter, a social network or a news media? [C]//Proceedings of the 19th international conference on World wide web. 2010: 591-600.

[53] FAN W, HE T, LAI L, et al. GraphScope: A Unified Engine for Big Graph Processing [J]. Proc. VLDB Endow., 2021, 14(12): 2879-2892.

[54] KIPF T N, WELLING M. Semi-Supervised Classification with Graph Convolutional Networks[C]//ICLR '17: International Conference on Learning Representations. 2017.

[55] HAMILTON W L, YING R, LESKOVEC J. Inductive Representation Learning on Large Graphs[C]//Proceedings of the 31st International Conference on Neural Information Processing Systems. Long Beach, California, USA: Curran Associates Inc., 2017: 1025-1035.

[56] SCHLICHTKRULL M, KIPF T N, BLOEM P, et al. Modeling relational data with graph convolutional networks[C]//European semantic web conference. 2018: 593-607.

致 谢

进入实验室学习和科研已三年有余，在这三年多的时间里，我在知识储备、技术能力和科研水平等各方面都收获颇丰。三年多的成长，我最想感谢的是我的导师陈榕老师，感谢他的伯乐之恩让我有机会进入 IPADS 学习，感谢他三年多来对我科研工作上的指导，感谢他严谨认真的治学、科研态度对我的言传身教。实验室的两位学长姚子航和沈斯杰是我科研道路上的同行者、引路人，我和他们一起完成了多个科研项目，在这期间他们也给予了我很多帮助和指导，让我大为受益。我还要感谢组里的魏星达老师和同学们，很开心和你们一起学习和科研。感谢 IPADS 的各位老师和同学，能成为 IPADS 这样一个优秀的实验室的成员我深感幸运，祝愿实验室能够越来越好。

最后，感谢父母多年来对我的付出，感谢爷爷奶奶一直以来对我最朴实的关心，愿家人们身体健康，每一天都能享受自由和快乐。

学术论文和科研成果目录

专利

[1] 陈榕, 石林, 夏虞斌, 陈海波, 臧斌宇. “面向大规模时序 RDF 图数据的查询方法及系统”, 专利申请号 CN202111678455.5.