

中国科学技术大学

University of Science and Technology of China

硕士学位论文



论文题目

高效的动态图数据存储技术

及图算法研究

作者姓名

石济帆

学科专业

计算机科学与技术-计算机系统结构

导师姓名

徐云 教授

完成时间

二〇二四年五月

中国科学技术大学

硕士学位论文



高效的动态图数据存储技术 及图算法研究

作者姓名： 石济帆

学科专业： 计算机系统结构

导师姓名： 徐云 教授

完成时间： 二〇二四年五月二十二日

University of Science and Technology of China
A dissertation for master's degree



Research on Efficient Dynamic Graph Data Storage Techniques and Graph Algorithms

Author: Jifan Shi

Speciality: Computer Systems Organization

Supervisor: Prof. Yun Xu

Finished time: May 22, 2024

摘要

在大数据时代，图数据和图算法已经成为处理复杂网络和关系数据的重要工具。它们广泛应用于各种领域，包括社交网络分析、生物信息学、交通规划、以及推荐系统等。随着应用的不断深入，对于动态图数据的存储和处理需求变得越来越普及。现有的动态图存储结构或是需要较多的空间支持动态操作（如基于哈希表的 *Stinger*），或是牺牲了动态性能来节约空间（如基于二叉树的 *Aspen*，基于 *Packed Memory Array* 的 *Tesco*），难以同时兼顾时间和空间性能。因此，如何为动态图数据设计低空间性能和高性能操作的系统便成为了当下亟待解决的重要问题。本文先利用 *vEB* 树中的位向量存储图顶点、紧凑数组存储邻接顶点，来达到低空间设计；然后，利用双对数高度的 *vEB* 树和邻接顶点的缓存等方法，来实现高性能操作。此外，本文又进一步设计了读优化的并发控制协议和异步任务分配的广度优先搜索算法，满足了实际应用中对于动态图数据的并发需求。

本文主要的研究工作与贡献如下：

(1) 动态图数据的低空间高性能存储结构：图数据的存储主要分为图顶点的存储和顶点关系（边）的存储。近年来的图存储结构多是在传统的图存储结构或关系型数据库和键值数据库的基础上进行改进以支持动态图的存储，难以同时兼顾空间性能和时间性能。本文借助 *van Emde Boas* 无节点分裂合并的特性，设计了一种既支持快速更新 ($O(\lg \lg u)$ 时间复杂度，其中 u 为数域大小) 又空间友好的图顶点索引结构，并基于邻接表更新友好和压缩稀疏行矩阵 (*CSR*) 空间友好的特性设计了平衡动态性能和空间开销的边存储结构。此外，本文还在 *ROWEX* (*Read-Optimized Write EXclusive*) 和乐观并发策略的基础上进行了优化，使其支持高效的并发图更新和图分析操作。实验结果表明，该存储结构相较于目前的先进工作，不仅在动态操作性能上具有高达 2.4 倍的加速比，还可以节约至多 38.5% 的内存空间。

(2) 广度优先搜索算法的并行优化：广度优先搜索 (*Breadth-First Search, BFS*) 是图分析中最基础的算法之一，其并行化不仅需要图数据的存储结构的支持，还对算法的并发策略设计提出了较高要求。传统的并行广度优先搜索主要针对静态图设计，在图算法的执行之前需要对图进行预处理操作，同时也具有较高的同步开销和访存代价。本文设计针对共享存储体系上方向优化的 *BFS* 算法，提出了一种异步任务分配的自上而下搜索算法和动态的自下而上的节点筛选策略，不仅可以实现动态的负载均衡，还减少了同步和访存开销。实验结果表明，相较于其它的并行 *BFS* 算法，本文的算法具有更少的执行时间和更好的可扩展性。

关键词：动态图存储，图索引，图算法，并发控制，广度优先搜索

ABSTRACT

In the era of big data, graph data and graph algorithms have become important tools for dealing with complex networks and relational data. They are widely used in various domains, including social network analysis, bioinformatics, transportation planning, and recommender systems. As the applications continue to develop, the need for storing and processing dynamic graph data becomes more and more popular. Existing dynamic graph storage structures either require more space to support dynamic operations (e.g., Stinger is based on hash tables), or sacrifice dynamic performance to save space (e.g., Aspen is based on binary trees, Teseo is based on Packed Memory Array), which makes it difficult to take into account both time and space performance.

Therefore, how to design a system with low space consumption and high performance operation for dynamic graphs has become an urgent problem to be solved nowadays. In this thesis, we use the bitvectors in the vEB tree to store graph vertices and compact arrays to store neighboring vertices to achieve space-efficient design; then, we utilize methods such as the vEB tree with double logarithmic heights and the caching of neighboring vertices to achieve high-performance operation. In addition, we further design a read-optimized concurrency control protocol and a breadth-first search algorithm for asynchronous task allocation to satisfy the concurrency demand for dynamic graph data in practical applications.

The main research work and contributions of this thesis are as follows:

(1) Fast yet space-saving structure for dynamic graph storage: The storage of graph data is mainly divided into the storage of graph vertices and the storage of vertex relations (edges). In recent years, most of the graph storage structures are based on traditional graph storage structures, relational databases, key-value databases to support dynamic graph storage, and it is difficult to take into account the space and time performance at the same time. In this thesis, we take advantage of van Emde Boas trees to design a graph vertex index structure that supports fast update ($O(\lg \lg u)$ time complexity, where u is the size of the integer universe used to represent the graph) and space-friendly storage. We design an edge storage structure that balances the dynamic performance and space overhead based on the features of the update-friendly adjacency list and space-efficient Compressed Sparse Row (CSR). In addition, we optimize it based on ROWEX (Read-Optimized Write EXclusive) and optimistic concurrency control method to support efficient concurrent graph update and graph analysis. Experiment results show that

this storage structure not only has up to 2.4 times speedup in dynamic operation performance, but also saves up to 38.5% memory space compared to current state-of-the-art work.

(2) Parallel optimization of breadth-first search algorithm: Breadth-First Search (BFS) is one of the most fundamental algorithms in graph analysis, and its parallelization not only requires the support of the storage structure of the graph data, but also puts strict requirements on the design of the concurrency strategy of the algorithm. Traditional parallel breadth-first search is mainly designed for static graphs, which requires preprocessing operations on the graph before the execution of the graph algorithm, and also has high synchronization overhead and access cost. In this thesis, we design a parallel direction-optimizing BFS optimized for the shared storage system, which propose a top-down search algorithm for asynchronous task allocation and a dynamic bottom-up node filtering strategy, which not only achieves dynamic load balancing, but also reduces the synchronization and access overhead. Experiment results show that compared with other parallel BFS algorithms, our algorithm has less execution time and better scalability.

Key Words: Dynamic Graph Storage; Graph Index; Graph Algorithms, Concurrency Control; Breadth-First Search

目 录

第 1 章 绪论.....	1
1.1 研究背景及意义.....	1
1.2 研究现状.....	3
1.2.1 图的相关概念及存储方法.....	3
1.2.2 图算法的访问模式以及并行策略.....	6
1.2.3 发展趋势及存在问题.....	7
1.3 本文研究内容.....	10
1.3.1 动态图数据的基本存储框架.....	10
1.3.2 图算法支持以及广度优先搜索的优化	11
1.4 论文组织.....	12
第 2 章 相关研究和技术	14
2.1 图数据存储结构.....	14
2.1.1 动态图数据存储结构.....	14
2.1.2 动态图数据存储结构的并发控制策略	22
2.2 图算法.....	24
2.2.1 经典图算法介绍及分析.....	24
2.2.2 并行广度优先搜索的实现方式及优化策略	28
2.3 本章小结.....	29
第 3 章 动态图数据的高效存储方法及图算法实现	30
3.1 问题分析.....	30
3.1.1 图数据的特点和动态图数据的存储要求	30
3.1.2 现存方法的解决思路及不足之处.....	31
3.1.3 vEB 树及其特点和限制.....	32
3.1.4 设计动机与思路.....	33
3.2 数据结构设计.....	34
3.2.1 顶点索引结构.....	35
3.2.2 边存储结构.....	36
3.2.3 基本操作.....	37
3.2.4 复杂度分析及讨论.....	39
3.3 并发策略设计.....	41
3.3.1 基本思想.....	41
3.3.2 并发控制协议.....	42
3.3.3 改进的 ROWEX 的正确性证明	44
3.3.4 版本控制.....	45

3.3.5 基本图算法的并行化支持.....	45
3.4 实验结果与分析.....	49
3.4.1 基本操作时间性能.....	50
3.4.2 图存储的空间性能.....	53
3.4.3 图算法的时间性能.....	54
3.4.4 动态图的并发性能.....	55
3.5 本章小结.....	57
第4章 并行广度优先搜索算法的优化	58
4.1 并行广度优先搜索的分析	58
4.1.1 并行广度优先搜索的执行流程及分析	58
4.1.2 现有并行图搜索算法的优化方法的不足	59
4.1.3 设计动机与思路.....	60
4.2 ABi-BFS:高性能的双向图搜索算法	61
4.2.1 算法概览.....	61
4.2.2 使用异步任务分配的自上而下搜索.....	63
4.2.3 使用节点筛选策略的自下而上搜索.....	65
4.3 实验结果与分析.....	67
4.3.1 消融分析.....	67
4.3.2 可扩放性实验.....	68
4.3.3 图规模对算法时间性能的影响.....	69
4.3.4 图密度对算法时间性能的影响.....	69
4.3.5 动态图存储结构上的算法性能.....	70
4.4 本章小结.....	71
第5章 总结与展望	72
5.1 本文工作.....	72
5.2 工作展望.....	73
参考文献.....	74

插图清单

图 1.1	数据库发展趋势	1
图 1.2	图结构示例	3
图 1.3	邻接矩阵	4
图 1.4	邻接表	5
图 1.5	边表	5
图 1.6	CSR	6
图 2.1	哈希表	14
图 2.2	前缀树	16
图 2.3	B 树	17
图 2.4	二叉树和 C-Tree	18
图 2.5	Packed Memory Array	20
图 2.6	跳表	21
图 3.1	vEB 树的原型结构	32
图 3.2	Spruce 的数据结构设计	34
图 3.3	BufferBlock 和 SortedBlock 的结构设计	36
图 3.4	Spruce 运行时示例	40
图 3.5	Spruce 版本控制设计	45
图 3.6	动态图随机插入性能	51
图 3.7	动态图随机删除性能	52
图 3.8	动态图顺序操作性能	52
图 3.9	动态图存储内存开销	53
图 3.10	动态图存储方法的可扩展性	56
图 3.11	并发图更新和图分析性能	57
图 4.1	广度优先搜索模式对比	58
图 4.2	ABi-BFS 执行模式示意图	61
图 4.3	滑动队列的基本结构及操作	62
图 4.4	使用不同优化方法时 PBFS 的运行性能	67
图 4.5	线程运行时间的累积分布图	68
图 4.6	不同 PBFS 方法的可扩展性	68
图 4.7	不同 PBFS 方法在不同规模的图上的性能	69

图 4.8	不同 PBFS 方法在不同密度的图上的性能 ·····	70
图 4.9	不同 PBFS 方法在动态图结构上的性能 ·····	71

表 格 清 单

表 3.1	不同动态图存储结构上基本操作时间复杂度对比·····	41
表 3.2	实验中使用的图数据集·····	50
表 3.3	不同动态图存储结构上图算法性能对比·····	55

第1章 绪 论

1.1 研究背景及意义

在如今的大数据时代,数据资料呈现出体量巨大,关系复杂的特点^[1]。这些特征对数据处理和分析带来了巨大的挑战,也对数据的表示方法提出了更高的要求。传统的数据表示方法,如关系数据库中表格数据,虽然在结构化数据的存储和查询方面表现出色,但在处理高度复杂和动态变化的数据关系时却显得力不从心。这主要是因为传统方法在设计时未能充分考虑到数据之间的内在联系和网络结构,导致在表达复杂关系和高效执行关系查询方面存在局限^[2-3]。例如,在社交网络中,用户之间的关系较为复杂,如朋友、亲人、上下级关系。传统的关系数据库在处理社交网络关系时,需要通过多表连接进行查询,这在大规模数据集上是非常低效的。为了更有效地管理这些数据, NoSQL 数据库应运而生^[4]。NoSQL 泛指非关系型数据库,包括键值数据库、图数据库、文档数据库、列数据库等。其中,图数据模型通过节点(Vertex)和边(Edge)直观地表示实体之间的关系,不仅能够高效地处理复杂的网络分析和关系查询,还能灵活适应数据结构的变化,从而有效地应对大数据时代数据表示的挑战。随着近年来社交网络、电商平台、人工智能快速发展,图数据的相关研究已然成为了当下的热门研究方向^[5-8]。根据 DB-Engine 的统计^[9],图数据库在近十年受到广泛关注、是发展趋势最迅猛的数据库类型(如图1.1所示)。

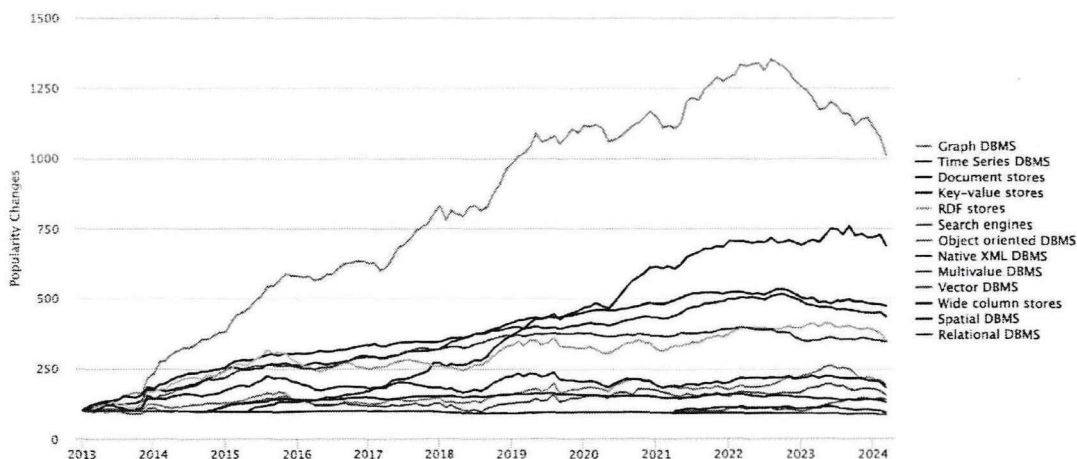


图 1.1 DB-Engine 统计的各类数据库自 2013 年以来的发展趋势^[9]

图数据的存储和分析是图数据库研究领域的基础问题。图数据的存储旨在有效地保存和管理图形结构数据,即由节点和边组成的数据模型。它不仅包括对图数据结构的物理存储,也涵盖了为高效访问、查询、分析和更新这些数据所

需的索引、优化策略和管理工具。图分析算法通过在图存储结构上进行如图遍历查询、路径查找、模式匹配等基础操作来满足不同应用场景的需求,如社交网络分析、推荐算法、知识图谱挖掘和网络威胁检测等^[10-11]。现在的图数据往往都处于快速的动态变化之中^[12],由于传统的图存储结构修改步骤繁琐,并不能很好的满足动态图的存储,所以需要支持动态数据的系统来存储这些数据。具体而言,这些系统应当支持:(1)图规模的动态缩扩;(2)图数据的快速更新;(3)对高并发场景的支持;(4)对图算法的支持和优化。一般的关系型数据库(如SQLite)虽然也能支持增删查改等操作,但是其表格式的存储方式并不擅长处理复杂图关系的查询和分析^[13]。因此,如何为动态图设计专门的结构来实现图数据的高效存储,以及设计与之匹配的高效图分析算法是亟待解决的重要问题。

现在的动态图系统一般是在传统的图结构(邻接表, Compressed Sparse Row (CSR) 等)上进行修改或添加一些结构来支持动态操作。邻接表对动态性支持的主要问题在于,存储顶点(即链表头)的数组的大小在一开始就要根据图的规模(即最大的顶点编号)确定好,不方便进行改变,因此典型的方法是将图的实际顶点编号映射为逻辑顶点编号,然后使用一个动态结构进行索引^[14],如哈希表^[15], B+ 树^[16]等等。目前很多图顶点的索引结构虽然支持动态操作,但是在可扩充性,操作性能等方面仍然不是很好。同时,链表的结构虽然方便进行修改,但是指针会占用相当多的空间。CSR 由于完全使用静态数组存储图的数据,任何插入操作都需要移动数组中的元素。当数据规模较大时,时延会变得不可忍受。为了克服这一缺点,常用的方法是使用快照或者使用 Packed Memory Array^[17]代替静态数组。但是在现在频繁的图更新的场景下,快照往往需要大量的空间开销。而 Packed Memory Array 由于采用预留空位的做法,在空位较少时需要重新调整数据位置,计算也较为复杂,在图的顶点度数较高时,表现不佳。因此,需要一个高效的专用结构来存储动态的图数据,实现空间、操作性能、动态性的均衡。此外,图的各种应用场景还要求图系统在高并发的环境下保证数据的一致性。常见的保证方案有多版本并发控制(Multiversion concurrency control, MVCC)和乐观锁^[18]等。

目前,在动态图系统的研究中,索引的设计往往较为简单,很多是移植键值数据库中的索引设计方案,有相当大的进步空间。同时由于较为繁琐的存储格式和并发策略,系统运行时的内存开销往往很大。因此,如何选择合适的索引方案,设计一个既空间节约,又具有高效并发性能的图系统是一个重要的挑战。需要指出的是尽管现实生活中的大图都十分的稀疏,但是其顶点的编号往往是相对稠密的^[19]。大图的更新也往往呈现出多点、散发的特征。因此可以尝试而利用这些图的特殊性质,有针对性的设计动态图的存储方式,从而实现动态图的高效存储。

图分析使用基于图的方法来分析连接的数据。图算法是图分析的关键工具之一，常通过对图数据的查询和计算来发现一些定性或定量的结论。宽度优先搜索 (Breadth First Search, BFS) 作为一种基础的图算法，是最短路径，连通分量，最大流等图算法的基石，并在模型检查，图像处理，社会和语义网络发现等方面有着广泛的应用^[20-21]。在如今的大数据时代，并行 BFS 的性能已经成为了制约相关系统运行性能的重要因素。但由于其不规则的访存行为模式，它在一定程度上难以被有效地并行化。

自上个世纪以来，学术界针对 BFS 的并行优化已有了广泛的研究。这些研究主要是考虑如何进行负载平衡、减小访存开销、优化算法流程等以提高 BFS 的并行效率^[22-24]。在这其中，相当多的优化方法主要针对静态图进行设计（如需要获取图的完整静态结构信息^[25]），并不能很好地适配动态的图数据结构。因此，设计对静态图和动态图通用的有效并行 BFS 的优化方案对于数据频繁变动下的图分析问题具有重要意义。

1.2 研究现状

本节将介绍图数据存储和分析相关的背景知识和该领域的发展趋势。首先介绍的是图数据的表示和存储方法，然后介绍针对图数据的图算法的并行策略以及典型的优化方法，最后介绍该领域最近的研究现状，发展趋势以及在发展过程中存在的不足之处。

1.2.1 图的相关概念及存储方法

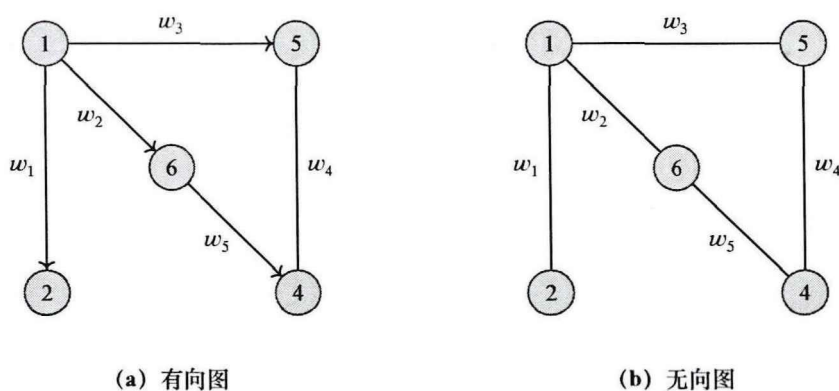


图 1.2 图结构示例

图 (Graph) 是一种数据的抽象表示方法，通常用来表示一组对象及其之间的关系。在图论中，一个图被定义为一组节点（顶点）以及连接这些节点的边的集合，如图1.2所示。该定义的形式化表述为：一张图 G 由 $G = (V, E)$ 定义，其中

V 表示图中所有顶点的集合, E 表示图中所有边的集合。按照边的性质, 图可以进一步分类为有向图和无向图。在有向图中, 边存在方向, 用于表示顶点间的单向关系。一条连接顶点 u 和顶点 v 的边被定义为一个三元组 $e = \langle u, v, p \rangle$, 其中 p 为边携带的属性信息 (如边权)。此时我们称 v 是 u 的邻居, e 是 u 的出边, v 的入边。对于顶点 $u \in V$, 该顶点的邻居集合 $N(u)$ 包含了其所有出边指向的顶点。 u 的入度是指所有以 u 为邻居的顶点的数目, u 的出度是指 $N(u)$ 的大小。有向图中的一个顶点的出度与入度之和称为该顶点的度。在无向图中, 所有的边都没有方向, 均表示顶点间的双向关系。一条连接顶点 u 和顶点 v 的边被定义为一个三元组 $e = (u, v, p)$, 此时我们称 u 和 v 互为邻居。在无向图中, 顶点的度等于出度等于入度。

图数据可以通过多种方式表示, 以适应不同的应用场景和计算需求。常见的图表示方法有邻接矩阵、邻接表和边表等。邻接矩阵 (Adjacency Matrix) 是一个二维矩阵, 其中的元素用于表示图中节点之间是否存在边^[26]。如果矩阵中第 i 行第 j 列的元素非空, 则表示图中存在从顶点 i 指向顶点 j 的边。对于无权图, 矩阵中的元素是二元的 (0 或 1), 表示节点间是否相连; 对于有权图, 矩阵中存储的是边的权重。对于图1.2(a) 的无向图, 其邻接矩阵表示如图1.3所示。基于二维数组存储的邻接矩阵, 其所需要的空间复杂度为 $O(|V|^2)$, 访问一个顶点的邻居所需要的时间复杂度为 $O(|V|)$ 。邻接矩阵具有良好的点查询和修改性能, 可以在 $O(1)$ 时间内完成顶点和边的添加、删除和查询。但是由于其空间复杂度较高, 并且对邻居的查询需要遍历矩阵的某一行/一列, 因此一般仅适用于稠密图或者小规模图的存储和计算。

	1	2	3	4	5	6
1		w_1			w_3	w_2
2	w_1					
3						
4					w_4	w_5
5	w_3			w_4		
6	w_2			w_5		

图 1.3 邻接矩阵

邻接表 (Adjacency List) 为图中每个节点维护一个链表, 列出与之直接相连的所有邻居节点^[27]。通常, 各个节点的入口 (即链表的头节点) 一般被存储在一个静态的顶点表 (Vertex Table) 中, 而每个节点的所有邻居及边属性被维护在

一个链表 (Edge List) 中。对于图1.2(a) 的无向图, 其邻接矩阵表示如图1.4所示。当图使用该种表示方法时, 所需要的存储空间为 $O(|V| + |E|)$, 访问一个顶点的邻域所需要的时间为 $O(d)$, 其中 d 为该顶点的度数。邻接表较低的空间需求和较高效的邻居访问能力使其成为存储和分析大图的理想结构。

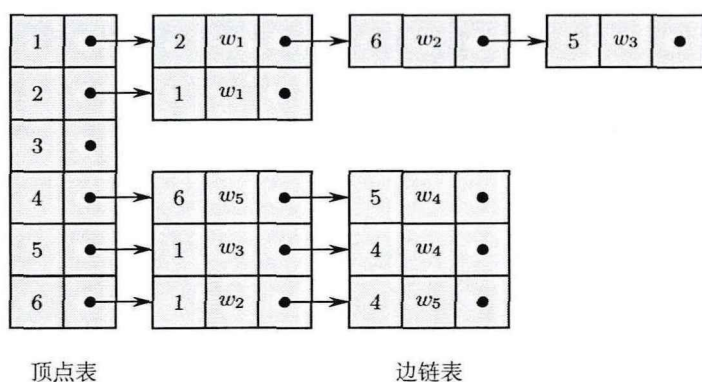


图 1.4 邻接表

边表是图的另一种简单表示方法, 由所有边组成的列表构成, 每条边是一个包含两个节点以及边的属性的元组^[28]。对于图1.2(a) 的无向图, 其边表表示如图1.5所示。基于数组表示的边表所需要的存储空间为 $O(|E|)$ 。由于在边表中, 边通常是无序存储的, 因此访问一个顶点的邻居需要遍历整个边表, 其时间为 $O(|E|)$ 。由于其较低的查询性能, 边表在大多数情况下只用于存储图, 而不被用于图的相关分析计算。

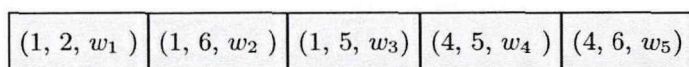


图 1.5 边表

压缩稀疏行矩阵 (Compressed Sparse Row, CSR)^[29] 是目前应用最为广泛的表示静态图的存储结构。如图1.6所示, CSR 仅包含几个静态数组, 即偏移数组、边数组和边属性数组。偏移数组为每个顶点保留在边数组中边的开始和结束的偏移量。边数组连续存储顶点的出边, 而边属性数组维护这些边的属性 (权重、名称等)。和邻接表类似, CSR 所需要的存储空间为 $O(|V| + |E|)$, 访问某顶点的邻域所需要的时间为 $O(d)$ 。由于在 CSR 中, 所有的数据都被紧凑地存储在静态数组中, 在进行数据的插入或删除时, 相关联的数组都需要进行重构, 所以 CSR 并不适合于存储动态变化的图数据。

随着图数据规模的增大, 相当一部分的研究专注于采用更先进的压缩算法来节约图数据存储的空间。Boldi P. 和 Vigna S. 提出了使用公共序列和差分编码

偏移数组

0	3	4	4	6	8
---	---	---	---	---	---

边数组

2	6	5	1	6	5	1	4	1	4
---	---	---	---	---	---	---	---	---	---

边属性数组

w_1	w_2	w_3	w_2	w_5	w_4	w_3	w_4	w_2	w_5
-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

图 1.6 压缩稀疏行矩阵 (Compressed Sparse Row, CSR)

来压缩邻接表的方法^[30]。Shun J. 等人在差分编码的基础上进行了扩展, 引入 Byte Code 和 K-bit Code 来进一步提高压缩的效率^[31]。Karypis G. 和 Kumar V. 采用了图分割的方法来改进稀疏矩阵的空间效率^[32]。Krishnapuram B. 等人提出了基于二分的图顶点编号重新排序的方法使得图中顶点 ID 编号更稠密, 从而提高数据编码时的压缩率^[33]。遗憾的是, 由于压缩算法往往针对静态的数据, 现存的图压缩方法均无法支持动态图数据。

1.2.2 图算法的访问模式以及并行策略

图算法涉及在图数据结构上执行的一系列操作, 可以解决如最短路径寻找、网络流量优化、社区发现等问题。目前的大部分图算法都是访存密集型而非计算密集型的, 主要的性能瓶颈在于内存访问^[34]。图算法对图数据的访问模式可以分成以下几类: (1) 对顶点的随机访问, (2) 对图算法中相关记录数据的随机访问 (如记录路径长度的数组), (3) 对顶点的顺序访问, (4) 对顶点邻居的顺序访问。大部的图算法的数据访问模式较为复杂, 是以上几类数据访问模式的结合。如在深度优先搜索 (Breadth-First Search, BFS) 和单源最短路径 (Single Source Shortest Path, SSSP), 既有对顶点和算法特定属性的随机访问, 也有对邻居的顺序访问。由于图算法数据访问模式的复杂性和大规模图数据的处理需求, 开发有效的并行策略成为了提高图算法性能的关键。

基于顶点的并行策略是图算法中使用最为广泛的并行策略^[35]。其基本思想是将图算法中的访问和计算任务按照顶点进行划分给各个计算节点或线程。该方法在编程时易于实现, 由于图中的顶点度数往往分布不均匀, 所以容易出现各个计算节点之间工作量不均衡的问题。为了解决这一问题, Zhang Y. 和 Hansen E. 从全局任务调度的角度出发提出和实现了图算法运行过程中的动态调度策略来改善负载均衡问题^[36]。其后, Cong G. 等人从计算线各个线程自身的角度出发, 提出了局部的任务窃取策略以平衡线程之间的任务量^[37]。需要指出的是, 尽管这些方法一定程度上平衡了不同计算节点之间的工作负载, 但是也在图算法

的执行过程中引入了额外的通信开销。

近十年的图数据规模的快速增长和分布式系统的广泛应用催生了基于图分割的图计算并行策略^[38]。其基本思想是在执行图算法前,先将整个大图进行切割并分配各个计算节点。在执行计算任务时,各个节点分别进行计算,并进行全局的同步。该类并行策略的分割效果相当依赖于图切割方法的好坏。按照切割方法的不同,图的切割可以分为边割(Edge Cut)和顶点割(Vertex Cut)。边割尽量保持顶点的完整性,将连接不同子图的边进行切割。其目标是尽量减少子图之间的边的数量,以便在分布式计算过程中降低通信开销。为了提高图算法的并行性能,好的边割策略会在最小化跨子图边的数量的同时,保持子图的大小相对平衡。采用边割进行图计算系统有 Pregel^[39]和 GraphLab^[40]等。顶点割尽量保持边的完整性,将连接不同子图的顶点进行切割。其目标是在保持子图之间的边的数量不变的情况下,尽量减少切割顶点的数量。这种方法特别适用于处理具有高度连接性的图,因为它可以有效地减少分布式计算过程中的不平衡工作负载。采用顶点割进行图计算系统有 PowerGraph^[41]和 GraphX^[42]等。边割尝试将顶点进行均匀的划分,但是可能会在高度数节点带来不平衡的计算开销。而顶点割尝试将边平均分配到各个计算节点上,但是这样可能会带来很高的通信开销。在后续的工作中,PowerLyra^[25]结合了二者的优点,通过对大小顶点采用不同的分割策略,并采用了异步的通信方法在尽可能保持负载均衡的同时降低通信开销。

1.2.3 发展趋势及存在问题

在当今的数据密集型应用中,动态图常被用于表示随时间变化的复杂系统中实体间的关系,如社交网络中的人际关系、交通网络的变化,或互联网结构的演进。与静态图不同,动态图的结构和/或权重会随时间而变化,这包括节点的添加和删除、边的添加和删除,以及边权重的变化等。这些变化不仅增加了对存储系统的要求,也对数据的查询、更新和分析提出了新的挑战。因此,设计一个能够高效管理动态图变化的存储系统,对于支持实时分析、网络监控、社交网络服务等应用至关重要。由于传统图表示以及存储方法无法很好地支持图数据动态变化的需求,近年来有许多相关工作专注于为动态图存储和分析设计新的结构。

2012年,最早在这方面做出尝试的 Stinger^[43]采用了哈希表(HashTable)和邻接表相结合的方法支持图的高效插入操作。在传统的邻接表中,顶点表使用顶点ID(Vertex Identifier)进行索引,其大小由图中最大的顶点号决定,当图的规模增加时,整个顶点表需要重新生成。在 Stinger 中,顶点表中并不存储实际的顶点号,而使用逻辑顶点号。其使用一个哈希表来实现从实际的顶点号到逻辑的顶点号进行映射。当实际的顶点号发生插入、删除或更新等操作时,在大多数情况下只需要修改哈希表映射关系,而不需要重构整个数组。Stinger 中对于邻居

节点的存储设计较为平凡，仍然保持着传统邻接表的形式，仍然存在着较多的指针开销问题和访存时较差数据局部性的问题。

2015 年，哈佛大学提出的 LLAMA^[44] 采用了快照技术使得 CSR 支持了增量更新。当图数据出现更新时，LLAMA 将会针对更新的部分新建一个新的 CSR 快照。当进行查询、删除等操作时，LLAMA 需要同时访问存在的所有快照。因此，当更新的量较少时，LLAMA 得益于 CSR 紧凑存储格式的优势，具有较好的查询和图分析性能，但面对大量更新时，其不仅空间占用急剧增长，而且更新速度也会随快照数量的增加而下降。尽管 LLAMA 为了解决空间问题进一步提出了内存-外存相结合的存储模式，但是仍难以满足如今很多场景下图数据流式更新的需求。

2018 年，Wheatman B. 和 Xu H. 提出了 Packed CSR，使用 Packed Memory Array (PMA) 替代 CSR 中的普通数组来解决 CSR 面对流式更新的更新效率低下的问题。PMA 的核心思想是在数组中保持元素部分有序，并且在数组中预留空间以应对插入和删除操作，以避免每次插入或删除操作都重新组织整个数组。当数组中某一部分的元素过于密集时，PMA 会对数组进行重新平衡 (Rebalance) 操作来使得空隙相对均匀地分布在数组中。在数据量较少时，Packed CSR 具有较好的更新性能，但是当图的规模变得很大时，进行重平衡操作时需要在很大的数组中计算空隙位置并搬动数据，其更新性能会快速下滑^[45]。同时，数组中的空隙也会削弱数据的局部性，降低了在其上进行图分析的性能。

2019 年，来自乔治华盛顿大学的团队设计了 GraphOne^[46]，一个支持图更新和图分析的动态图存储系统。为了减少邻接表中链表在图更新时的频繁访存操作，提高更新效率，GraphOne 在邻接表的基础上引入了边表作为图更新的缓冲区：对图数据的更新会先以追加的方式写入到边表中，然后通过归档排序的方法，将对一定范围内顶点的更新操作进行聚合，最后再以非原子的方式以批量更新的方式写入到邻接表中。除此之外，为了支持图分析对图数据的快照需求，GraphOne 为每条边引入了时间戳来构建视图。GraphOne 尽管通过批量更新的方式提高了更新效率，但是缺乏对邻接表中顶点有效的索引定位方式，在应对大规模图数据时效率仍然不是很理想。

同年，来自卡耐基梅隆大学的团队基于 Purely Functional Trees 的思想设计了 Aspen^[47]，一种使用路径压缩的二叉树来支持动态图的存储。Aspen 通过在树上引入 Hash 算法来改善树的插入性能。为了节约空间，其使用了差分编码来编码一个顶点的所有邻居节点。为了解决传统二叉树中并发更新时容易产生的数据冲突问题，其提出了类似于写复制 (Copy-On-Write, COW) 的路径复制策略支持并发的更新操作。Aspen 是当时空间效率最好的动态图存储结构，但是由于差分编码的复杂性和路径复制的代价较高，Aspen 还无法支持边权的存储以及高效的

流式更新。

2020 年起,有越来越多的工作不仅关注图数据单纯的存储需求,开始尝试同时兼顾动态图数据的存储和分析性能。由清华大学领衔的团队提出了 LiveGraph^[48],一个支持事务操作的图系统。此前的图数据存储结构均不支持事务操作,图分析需要等待图更新完成之后执行以确保正确性。相较于之前的图存储结构,LiveGraph 最大的改进是引入了基于时间戳的边日志记录和多版本并发控制来支持事务操作,允许图更新和图分析并发执行。LiveGraph 使用了平铺式的 Vector 作为图数据的存储结构,因此更新效率相对较低。

2021 年,Oracle 团队提出了 CSR++^[49],采用将 CSR 中数组分段的策略减小数组重构的代价。对于顶点数组,其采用直接分段存储的方式,并使用额外的 map 进行从实际顶点 ID 到逻辑顶点 ID 的映射。对每一个顶点的边,其组织成类似于分块数组的方式。由于 CSR++ 中仍然存在着一定的重构开销,所以其仅在图数据批量更新时效果较好,而无法很好的支持即时的图数据更新。

同年,Leo D. 和 Boncz P. 提出了基于 CSR 的支持事务的动态图数据存储结构, Teseo^[50]。为了解决 CSR 顶点的更新效率问题,其使用了 Adaptive Radix Tree (ART)^[51]替代原来的静态数组作为顶点索引。和 Packed CSR 一样其同样使用了 PMA 作为边的存储结构,为了减少 PMA 的重平衡代价, Teseo 将 PMA 进行了分块处理,将不同块分散到 ART 的不同叶子结点中。为了更好地支持图分析的需求, Teseo 还使用了哈希表作为额外的辅助索引,并采用了顶点重编号技术以提高图分析时数据的局部性。

该年,来自清华大学的团队为实时图分析的需求专门设计了 RisGraph^[12]。RisGraph 的数据存储设计和 Stinger 较为类似,主要的贡献在于其创新性地提出了专门针对更新操作的图分析优化方法。RisGraph 使用稀疏数组来跟踪频繁被修改的顶点及其更新结果以提高数据访问的效率,并针对动态图数据上不同更新操作设计了不同的基于顶点和边的并行方法和并发控制策略。

2022 年,卡耐基梅大学的研究团队在 Aspen 的基础上进一步扩展,设计了 PaC-Trees^[52],一种支持动态图数据存储和倒排索引的动态数据结构。相较于 Aspen, PaC-Trees 进一步将 Aspen 的叶子结点也以紧凑二叉树的方式进行组织以提高数据的更新效率。同时, PaC-Trees 还设计了全新的节点分裂、合并和旋转策略以替代 Aspen 中的 Hash 计算在降低树高的同时提高压缩率。

2023 年, Fuchs P. 等人设计了基于邻接表的支持事务的动态图数据存储结构, Sortledton^[53]。Sortledton 使用了多级的 Vector 作为索引替代邻接表中的 VertexTable 支持顶点的修改,并使用 Concurrent Unrolled Skiplist 替代邻接表中原有的链表结构支持边的快速更新。同时, Sortledton 也采用了 Teseo 类似的顶点映射技术来加速图数据分析。

纵观近十年来动态图的相关研究,不难发现动态图的相关工作或是追求更快的更新速度,或是追求更低的空间占用,或是追求更高效的图分析性能。在这一进程中,鲜有工作能够同时兼顾动态数据存储的空间性能和时间性能:Aspen、PaC-Trees 等作为近年来空间最节约的动态图数据存储结构,在流式更新下更新效率低下;Sortledton 和 Tesco 作为近年来动态性能最好的工作,具有着较高的空间开销。同时,在图分析方面,RisGraph 等工作主要着眼于如何设计算法的执行框架来支持图算法的运行,还有一些工作着眼于改进算法本身使之适用于动态图数据^[54-55]。目前使用最为广泛的图算法 BFS 在动态图数据上仍有较大的改进和优化空间。这些已存在的工作及其不足为本文提供了良好的研究基础。

1.3 本文研究内容

本文的主要研究工作主要是动态图数据的存储和图算法的优化。首先,本文针对已有的动态图数据存储的解决方案进行深入对比研究和分析,发现目前图数据的顶点索引在面对图数据的大量更新时有着较高的更新开销,并且在边数据的存储上不易兼顾空间和时间效率。基于该分析,本文提出了一种高效的图存储结构,Spruce。Spruce 借助 van Emde Boas (vEB) 树的思想概念设计了多级的结合 BitVector 和哈希表的高效图顶点索引,并结合 CSR 和邻接表的优势设计了兼顾空间性能和更新速度的边存储结构。其后,本文观察到面对大规模并发操作时,传统的并发控制方案由于采用写时复制和较粗粒度的锁等策略,在多线程执行更新操作时加速比并不理想。因此,本文借助于 Spruce 的结构优势进一步设计了一种基于 Read Optimized Write EXclusion (ROWEX)^[56]和改进了 MVCC 的高效并发控制方案。最后,本文实现了动态图上图算法的支持,并提出了一种在静态图上和动态图上均具有良好效果的并行广度优先搜索算法。

1.3.1 动态图数据的基本存储框架

优秀的动态图存储技术应当做到空间性能和动态性能二者皆优。在空间方面,现有动态图系统的空间开销可以分为三部分:索引空间占用、信息存储空间占用、一致性保障的空间占用(即为了保障一致性所设计的辅助结构的开销)。考虑到图数据的顶点编号相对密集,呈现相似度较高的特性,可以利用图数据中的冗余信息,合理设计索引和存储结构以减少空间占用。在动态性能方面,图系统的动态性能与索引结构、数据组织格式和并发设计均息息相关,需要设计合适的辅助结构以及并发策略来提高图系统动态性能。根据以上分析,本文在该部分的研究分为三点:

1. 支持动态数据的低空间索引方法。现有的工作中,图索引结构的设计很多

都是搬运 Key-Value 数据库中的索引方案,如 B+ 树、前缀树、二叉树等,这些索引结构在动态更新时,往往都会产生节点的分裂、合并等操作。同时,很多方案在进行数据的索引时,还需要进行图中实际顶点编号到逻辑顶点编号的转换。vEB 树是一种多叉树结构,树的层级规模自下而上以平方根的规模递减,并且支持在 $O(\lg \lg u)$ 内完成树中的增删查改等操作。由于该种树形结构的规模仅仅和数据条目的长度有关,且树中的修改操作均通过对位向量 (Bitvector) 的置位来实现,所以并不存在节点的分裂和合并操作,是理想的高效数据存储兼索引的结构。因此,本文选择借助 vEB 的思想设计了一种低层数、空间自适应的图顶点索引结构。

2. **基于邻接表的边存储结构设计。**传统的邻接表中的边链表虽然支持 $O(1)$ 时间的插入操作,但是链表中的指针占用了相当大的空间开销,访问链表元素时,指针在内存中的频繁跳转也严重制约了数据的访存性能。CSR 虽然通过将所有的边放在一个数组中,在节约空间的同时利用数据局部性提高了其访存性能,但是其本身无法支持数据的动态变更操作。因此,在边的存储上,本文对二者取其精华,去其糟粕,设计了二者相结合的高效边存储方案。本文设计的边存储结构在一般数组的基础上引入了缓冲区块 (BufferBlock) 来降低数组重构的代价,并使用了位向量进行快速的数据定位,以实现空间性能和时间性能的平衡。
3. **基于 ROWEX 和乐观锁的并发机制设计。**ROWEX (Read Optimized Write Exclusive) 是为改进这一操作而诞生的新的锁策略。其基本概念是写者之间使用互斥锁来保证一致性,而读者在读取数据时不需要任何锁。为了保证读操作的正确性,需要对相关的数据结构进行修改,使之支持原子化操作。由于操作系统底层逻辑的限制,通常只能最多在 8 字节的数据上进行原子化操作,因此应用 ROWEX 的数据结构往往使用 Read-Copy-Update (RCU) 策略,通过原子地修改指针来一次性改变一整个节点的内容。vEB 树由于没有节点的分裂和合并,大多数的操作仅仅需要在现有的结构上进行修改即可,是使用 ROWEX 的理想结构。需要指出的是,ROWEX 本身无法保证原子化的范围扫描,因此无法在并发读写的情况下保证一些图操作(如:获取某一节点的邻居)的正确性,所以本文将之和乐观锁相结合来保证这些操作的正确性。

1.3.2 图算法支持以及广度优先搜索的优化

在设计图存储框架上,本文先是实现了常见的并行图算法,然后针对广度优先算法设计了不少一般性的高效并行化方案。一般而言,并行 BFS (Parallel Breadth-First Search, PBFS) 的执行是一个迭代过程,它按层次遍历整个图,并且

每次迭代可以分为两个阶段：节点扩展和任务分配。目前针对 PBFS 的优化方法主要有两个方向：(1) 减少边的访问次数和增强数据局部性来优化 BFS 的内存访问。(2) 实现负载均衡的任务分配，以减少同步开销。近年来，越来越多基于图分析的应用程序被构建在不断变化的图上。许多针对 BFS 的优化方法（如图分割和顶点排序）无法很好地在动态图上执行。因此，亟需设计一种不依赖于图预先计算特征的高性能 PBFS 算法的新需求。根据以上分析，本文在该部分的研究分为两点：

1. **异步任务分配的 PBFS 迭代模式。**传统的 PBFS 算法中，在节点扩展和任务分配之间存在着大量的同步操作。本文设计了一种异步的任务执行模式，借助于滑动队列（Sliding Queue）的数据结构将节点扩展和任务分配统合到一起以减少同步开销。同时，本文还设计了不依赖于图数据的全局信息、自适应的任务分配方法，以实现 PBFS 执行过程中仅此的负载均衡。
2. **动态的节点扩展选择策略。**Beamer S. 等人设计了双向 BFS，在这种算法中引入了自底向上的搜索模式，以减少在扩展阶段检查的边的数量^[24]。随着 BFS 的进行和更多节点的访问，自底向上搜索的性能由于需要检查所有顶点是否已经被访问，性能受到其较差的数据局部性的限制。本文提出一种动态的节点选择策略，能够在多次迭代中筛选出有可能被扩展的节点，从而在迭代中减少检查的顶点数量，降低访存开销，提高双向 BFS 的性能。

1.4 论文组织

本文首先概述了针对动态数据存储和分析研究的背景和重要性，并对现有研究进行了回顾，概括了其发展动态和面临的挑战。随后，本文介绍了相关研究和技术背景。文章详细阐述了两个研究主题，包括设计依据、方法论以及实施方案，并通过实验结果证明了所提出设计的效果。文末，本文总结了文章的关键贡献与创新之处，并规划了未来的研究方向。本文分为五个章节，每个章节的内容组织如下：第一章是绪论。本文首先介绍了动态图的研究背景及意义，接着介绍了图的相关概念及存储方法、图算法的访问模式以及并行策略、动态图数据存储和算法领域相关研究的发展趋势以及存在问题。基于目前研究的不足，本文提出了两项研究内容，并介绍了文章的组织结构。

第二章是相关研究和技术。本文在这一章首先介绍了动态图数据存储结构的相关研究，包括动态图数据存储的数据结构设计以及与之相关的并发控制策略。然后，本文介绍了图算法的相关知识，包括常见图算法的介绍、分析，以及目前针对广度优先搜索的优化策略。

第三章是动态图数据的高效存储方法以及图算法实现。该章节首先对动态

图数据的需求以及现有的工作进行了分析,得出现存工作的不足之处和需要解决的问题,然后就此引出研究工作的动机和思路。之后,该章节介绍了用于存储的动态图数据结构和基本操作的设计,并就其空间和时间性能进行了复杂度分析和讨论。在此基础上,该章节进一步介绍了该结构上的并发策略设计以及图分析算法的并行化设计。最后本文设计了对比实验,比较本文的设计和目前先进工作的性能优劣。

第四章是广度优先搜索算法的并行优化。首先,该章节分析了并行广度优先搜索算法的执行流程,并指出传统并行图搜索算法的不足,借此引出本文的设计动机和思路。然后该章节详细介绍了设计的高性能双向图搜索算法。最后,该章节设计实验就算法的效果进行了验证,并与其它典型的并行广度优先搜索算法在静态图和动态图上进行了性能对比。

第五章为总结与展望。本章对本文的工作进行了总结,并就文章的不足之处以及下一步打算进行的工作作了讨论。

第2章 相关研究和技术

本章概要：本章主要对与本文工作内容相关的研究进行介绍。本章分为三个部分，第一部分介绍图数据存储结构的相关工作，包括动态图的顶点索引、动态图的边存储结构，然后在进一步介绍在这些存储结构上的并发设计策略。第二部分首先介绍图算法的相关知识，包括一些经典的图算法以及访问模式。然后，本章再专门针对广度优先搜索的实现方式和优化方案进行相关的介绍。

2.1 图数据存储结构

2.1.1 动态图数据存储结构

图主要是由顶点和边构成的。据此划分，现在常见的高效的动态图数据存储结构可以解离为两个部分：动态的顶点存储结构以及动态的边存储结构。下面小节将依次对它们进行介绍。

1. 动态图的顶点索引

设计动态图顶点索引的目的是支持顶点快速的增删查改需求。目前主流的动态图数据索引主要有哈希表、前缀树、B树、二叉树等。哈希表（Hash Table）也称为散列表，是一种利用哈希函数组织数据，以支持快速插入和搜索的数据结构，被广泛应用于键值（Key-Value）类型数据的存储，在近年来也被一些工作用作图数据的顶点索引（如 Stinger^[43]）。哈希表的核心思想是使用一个数组来存储数据，通过哈希函数将要存储的键值转换成数组的索引，从而实现平均时间 $O(1)$ 内对数据的快速访问。如图2.1所示，当插入一个新键值对时，哈希表使用哈希函数计算key的哈希值，将其转换成数组的索引，并将键值存储在对应的位置上。对于修改和删除操作也是类似，先通过哈希函数计算出key的哈希值并转换为索引，再在索引的位置上进行修改或删除操作。

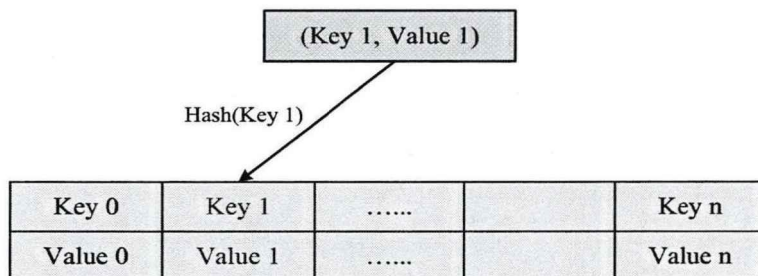


图 2.1 哈希表

在哈希表的实际使用过程中，由于哈希表的大小限制和哈希函数的选择等

相关问题,会出现两个不同的键经过哈希函数计算后得到同一个索引时发生的情况,即哈希冲突(Collision)。处理哈希冲突的方法有很多,典型的方法有链地址法^[57]和开放寻址法^[15]。链地址法的核心思想是将具有相同哈希值的所有元素存储在同一个位置,但不是直接存储在数组的这个槽位中,而是在该槽位上挂接一个链表。所有映射到同一个索引的元素都会被加入到这个索引位置的链表中,查找、插入和删除操作都需要在链表上进行。链地址法较为简单直观,但是链表的引入一方面带来了指针的额外开销,另一方面也降低了数据的局部性,增加了增删查改的代价。开放寻址法是另一种解决哈希冲突的方法,它不需要额外的数据结构(如链表)。当发生冲突时,开放寻址法会根据一定的探测序列,在哈希表数组内寻找下一个空闲位置。根据探测序列的不同,开放寻址法又可细分为线性探测(遇到冲突时,顺序查找表中的下一个空闲位置)、二次探测(遇到冲突时,使用一个二次方程确定下一个探测的索引位置)和双重哈希(使用第二个哈希函数确定探测的步长)等。相比于链地址法,开放定址法能更好地借助数据局部性提高查找效率,但是当哈希表中元素过多时,需要探测的次数会显著增加,同时删除时还需要对数据进行特殊标记。在上述两大类方法的基础上,为了提高数据查询的效率,本世纪以来学术界出现了诸多更为先进的冲突解决方法,如通过搬运其它 key 来处理冲突的布谷鸟哈希(Cuckoo Hashing)^[58]以及将冲突探测地址限定在一定范围内的 Hopscotch Hashing^[59]。

在哈希表中一般通过负载因子(Load Factor)衡量哈希表的满载程度,它定义为哈希表中已存储的元素数量与底层数组容量的比率,表示为 $\alpha = \frac{n}{k}$,其中 n 是元素数量, k 是数组容量。无论是何种哈希表,在负载因子较高时,都会无可避免地出现性能的显著下滑。因此,大多数系统均会将哈希表的负载因子维持在一个特定的阈值左右(如0.75)来保证哈希表的性能。为此,当哈希表的负载因子超出阈值时,需要将哈希表扩容(通常是扩容为2倍以保证基本操作的平均时间为 $O(1)$),并将其中所有的元素进行重新哈希(Rehash)并迁移到新的扩容后的哈希表中。因此,尽管哈希表的空间复杂度为 $O(n)$,其实际的占用空间通常数倍于原始数据的大小。

前缀树(Trie)^[60],又称字典树,是另一种适用于图顶点索引的数据结构。前缀树一种特殊形式的树形结构,最早被用于存储一系列字符串中的字符。大部分前缀树按照字节编码,每个节点代表一个字符,一条从根节点到某一节点的路径代表一个字符串,如图2.2所示。除此之外,也有按照比特编码的前缀树(Binary Trie),每个节点表示一个比特,但是由于其高度较高(如对于4byte的int类型数据,需要32层),访存次数多,性能不理想,所以较少被使用。

前缀树上的增删查改操作都是基于对节点的修改而实现。一般而言,前缀树的节点需要包含如下的必要信息:字符值,子节点指针(每个节点都可能有多

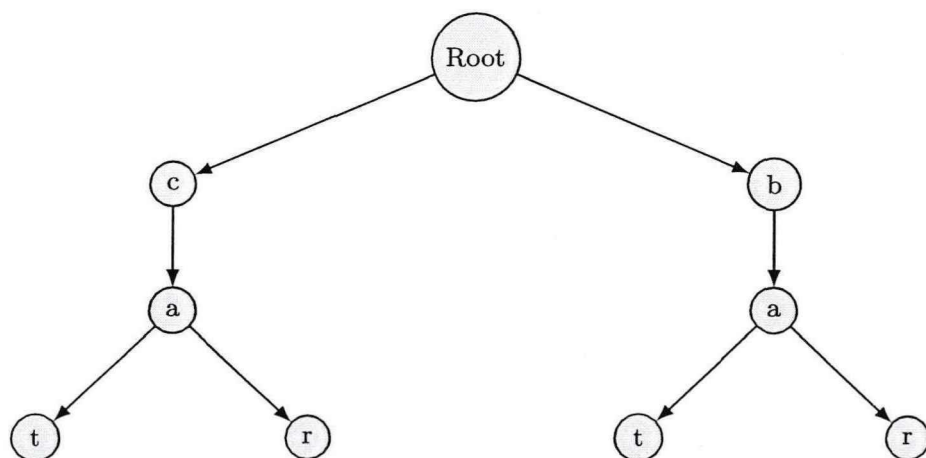


图 2.2 一棵按照字节编码，包含“cat”，“car”，“bat”，“bar”四个 key 的前缀树

子节点指针，对应于该节点字符后可能出现的所有字符）和终止标志（每个节点都有一个标志位，用来指示从根节点到当前节点的路径是否构成了完整的 key）。将 key 看作是一个字符串在前缀树中进行插入时，需要从根节点按照 key 的字符顺序沿着树向下遍历。对于当前字符，需要检查当前节点的子节点中是否已经存在该字符对应的节点。如果存在，移动到该子节点，继续下一个字符的插入。如果不存在，创建一个新的节点，代表这个字符，然后将其作为当前节点的子节点，继续下一个字符的插入。插入最后一个字符的节点后，将这个节点标记为终止节点，表示一个完整的 key 被插入。在前缀树上的查找过程和插入类似。从根节点开始，对于 key 中的每一个字符，从当前节点的子节点中查找匹配该字符的节点。如果存在，则继续沿该节点向下遍历下一个字符。如果在任何时刻找不到匹配的节点，说明该 key 不在前缀树中，查找失败。如果所有字符都成功找到，并且最后一个字符对应的节点终止标记为真，则查找成功。前缀树中的删除操作依赖于查找操作。删除时首先执行查找操作以确定 key 是否存在。如果 key 存在，则从字符串的最后一个字符对应的节点开始，逆向遍历。如果一个节点除了终止标志外没有其他子节点，那么删除这个节点，并继续向上逆向遍历。如果节点有其他子节点或成为另一个 key 的一部分（即非终止节点），则只移除终止标志，停止删除操作。

通过对前缀树基本操作的分析可知，假定 key 的全域大小为 $|U|$ ，则在前缀树的基本高度为 $\lg(|U|)$ ，在树上进行操作的时间复杂度为 $O(\lg|U|)$ 。尽管其时间复杂度较低，但是在实际的使用过程中，节点中用于存储指向其孩子的指针的指针数组往往不能被充分利用（如图 2.2，节点“c”的指针数组只存储了一个指针，而其本身设计的大小可以存储 $2^8 = 256$ 个指针），因此无法充分发挥共享前缀的优势，空间开销较大。近年来有诸多工作着眼于减少前缀树索引的空间开销，在这之中，最具有代表性的两项工作为 Adaptive Radix Tree (ART)^[51]和 Height

Optimized Trie (HOT)^[61]。ART 是一种自适应的、压缩的前缀树结构，它通过动态调整节点的内部表示来优化空间利用率和查询性能。其核心思想是根据子节点的数量使用不同大小的节点表示，从而减少内存消耗，并保持高效的查找速度。它使用多种类型的节点（如 4 位、16 位、48 位、256 位），根据前缀树节点中孩子的数量动态选择最合适的节点类型，从而显著减少了不必要的空间浪费。同时 ART 还采用路径压缩技术，减少了访问深度，提高了查询效率。HOT 是一种专为减少树的高度而设计的前缀树变种。其核心思想是在路径压缩的基础上，对于 trie 树中的聚合节点根据数据/字符分布设置一个适应性的节点大小，从而有效地调整树高。与其他 Trie 树相比，HOT 的中节点容纳的字符跨度 (span) 会随着数据分布而调整，而保持节点大小 (fanout) 的稳定来避免数据稀疏的问题，从而在减小树的高度的同时提高了数据的访存效率。同时，HOT 还引入了 SIMD (单指令流多数据流) 并行指令来进一步提高操作的速度。需要指出的是，为了节约空间 ART 和 HOT 均采用了有损的压缩技术，需要使用额外的空间保存原始的 key 以避免查询时假阳性（即查找成功但实际上 key 并不存在）的情况。

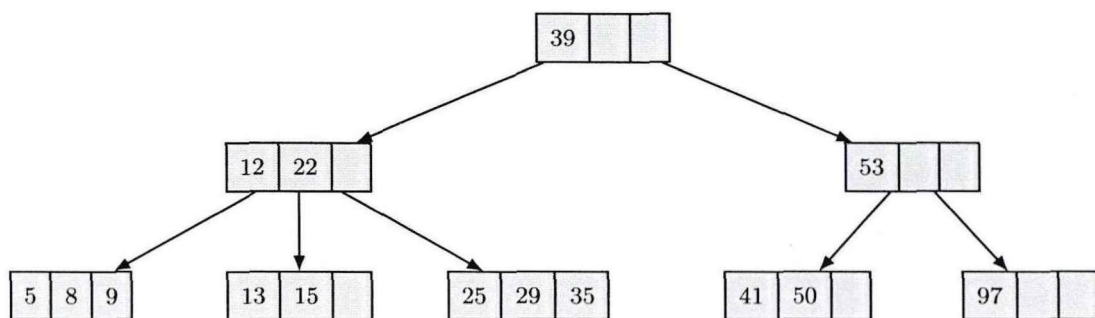


图 2.3 一棵三阶 B 树

B 树最早被 Dhulipala L. 等人提出作为图顶点索引的可行结构^[47]。B 树是一种多路自平衡的树数据结构，树中的数据保持有序，支持在对数时间内数据的插入、删除、搜索等操作。B 树的“阶 (Order)”是指节点中子指针的最大数量，通常记为 m ，一棵三阶的 B 树如图 2.3 所示。和前缀树类似，B 树的节点通常也会包括 keys、指向孩子的子指针数组。为了操作的方便，一些 B 树的节点中还会包括指向父节点的指针和用于判断节点满载情况的计数器。B 树中的任意非根节点和非叶子结点至少包含 $\lceil \frac{m}{2} \rceil$ 个孩子节点，非叶子结点的根节点至少有两个孩子。并且所有的叶子结点都在同一层，非叶子结点若包含了 k 个孩子，则包含 $k - 1$ 个 key。节点内的 key 将子指针分割成对应于 key 值范围的子树。例如，对于节点中的第 i 个 key，它左侧的子指针指向的子树包含的所有 key 都小于该 key，而右侧子指针指向的子树包含的所有 key 都大于该 key。上述种种性质保证了 B 树的平衡性，同时也决定了 B 树的高度范围为 $\lceil \log_m(n + 1) \rceil - 1 \sim \left\lceil \log_d \frac{n+1}{2} \right\rceil$ ^[62]，

其中 $d = \lceil m/2 \rceil$, n 为树中存储的 key 的数量。从 B 树衍生而来的 B+ 树进一步减少了树高, 并针对磁盘访问做了优化: 在 B+ 树中, 内部节点不存储任何指向记录的指针, 所有的 Key 都存储在叶子节点中。B+ 树中每个叶子节点包括一个指向下一个叶子节点的指针, 以加速顺序访问。相比于 B 树, B+ 树的内部节点拥有较少的指针, 每个叶子节点可以容纳更多的 Key (通常优化为系统中一页的大小), 树的高度更低。

在 B 树和 B+ 树中, 为了维护树高的平衡, 在插入和删除操作时需要对节点进行分裂/合并操作。以 B 树中的插入操作为例, 在 B 树中添加一个 key 时, 首先需要通过树上的二分查找找到待插入的位置。如果节点在插入新 key 后仍满足 B 树的性质, 则插入操作完成。如果插入导致节点中的 key 数量超过最大值 ($m-1$), 则需要分裂节点。在分裂节点时, 需要将节点中的所有 key (包括新插入的 key) 排序, 然后选择中间的 key 作为分裂点。这个 key 将上移至父节点, 作为分裂后的两个新节点的分隔键。然后根据分裂结果, 创建两个新节点, 一个包含分裂点左侧的 key (较小), 另一个包含右侧的 key (较大)。此后 B 树将分裂点的键上移至父节点中。如果父节点也因此而超过了其键的最大数量, 就递归地对父节点进行分裂。最后调整父节点的子指针, 使其指向两个新创建的节点。类似地, 当 B 树中发生删除操作时, 当节点中存储的元素数量过小时, 也需要合并兄弟节点甚至父节点。由于 B 树和 B+ 树最初主要是针对存放在外存中的索引结构而设计的, 在内存中的节点分裂、合并操作相对于外存的 I/O 操作时间代价较小。但随着近年来内存索引的流行, B 树和 B+ 树的设计中复杂的维护操作由于对内存并不友好, 所以往往成为制约其性能的关键因素。

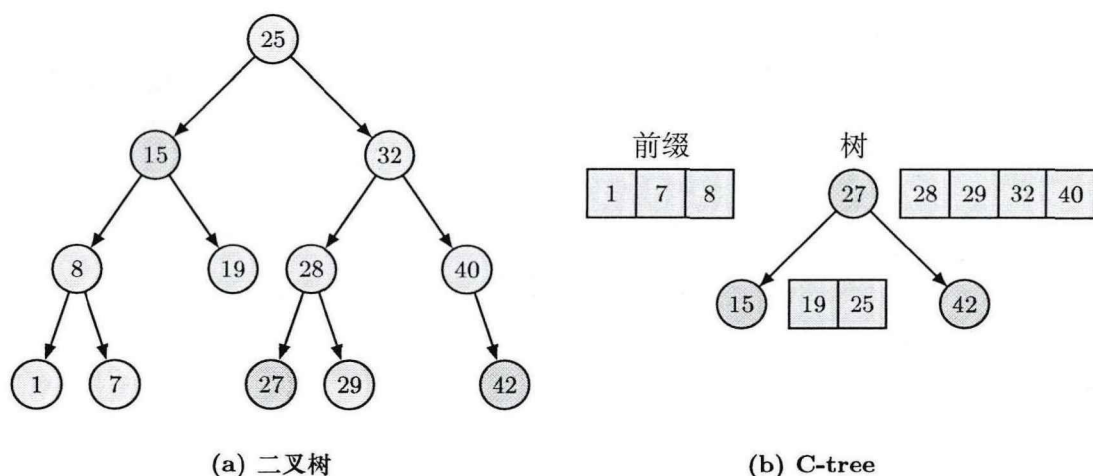


图 2.4 (a) 二叉树和从该二叉树演化而来的 (b) C-Tree, 二叉树中被选作 C-tree 中头节点的节点被标记为浅橙色。

二叉树 (binary tree) 是指树中节点的度不大于 2 的有序树, 如图 2.4 (a) 所示。

相较于B树和B+树,二叉树的树高虽然稍高,但是对节点的维护操作相对简单,可以有效减少在树更新时由于节点维护所导致的内存拷贝的开销(不需要拷贝大量的指针数组),因此被一些工作用作内存中的图顶点索引结构。Blleloch G. E.等人发展了基于二叉树的适用于图顶点的数据结构——C-tree^[52],一种压缩的二叉树的结构,在尽可能减少对树的更新开销的影响下节约树的内存占用的压缩二叉树结构。C-tree在二叉树中引入块处理(chunking)的概念,通过将多个元素连续存储在树节点内的数组中来提高缓存性能,减少所需的树节点数量以降低空间开销。

C-tree通过使用一个哈希方案来决定哪些元素作为“头”(head)存储在树节点中,哪些元素作为“尾”(tail)连续存储在数组中。首先,C-tree选择一个合适的哈希函数 h 用于将原二叉树中的元素较均匀地映射到一个较大的整数空间中,对于集合中的每个元素 e ,计算其哈希值 $h(e)$ 。如果 $h(e)$ 满足特定条件($h(e) \bmod b = 0$,其中 b 是预先定义的块大小),则该元素被选为头元素。非头元素根据其在原集合中的顺序连续存储,形成尾部数组。每个头元素及其对应的尾元素数组构成了一个块,剩下的元素则作为前缀存放在前缀数组中,如图2.4(b)所示。C-tree巧妙地借助了哈希的方法保持了树中节点大小的近似均衡的同时减小了分块处理的代价。C-tree最早是被用于存储图中的边,^[47]后来被进一步发展为PaC-Trees作为顶点的索引方案。相较于C-tree,PaC-Trees设计了全新的节点分裂、合并和旋转等算法来取代哈希以获得更高的压缩率。

2. 动态图的边存储结构

动态图的边存储结构在本质上也是一个支持增删查改的动态集合。按照数据结构的不同,主要可以划分为数组类结构和链表类结构。在数组类结构中,大部分的存储结构相对简单,如LiveGraph^[48]和CSR++^[49]均为每个节点使用数组来存储其所有的邻居。这样存储可以较好地节省空间,但其缺点也显而易见:如果需要维护数组的有序性,在向数组中插入一个元素时,可能涉及到大量的元素搬运,降低了操作的效率(CSR++) (时间复杂度为 $O(n)$, n 为元素数量);而如果不考虑数组的有序性,采用在尾部附加的方式进行插入(时间复杂度为 $O(1)$),则会由于无序性而增加更新和删除操作的开销(因为无法进行二分查找,需要通过遍历方式查找到待删除或更新的节点)。

目前的一种改良方案是使用Packed Memory Array (PMA)^[63]作为动态图的边存储结构,以一定的空间性能换取时间性能。PMA是一种维护动态元素集合的数组,其核心思想是在元素之间散布多个空位或间隙(Empty Slots),这样在插入或删除操作时只需要移动少量的元素。给定PMA的总容量 n ,它被分割成多个连续的段(Segment),每个段有预定的大小 s 。PMA保持元素排序的同时,通过合理分配空位(间隙),来优化插入和删除操作的效率。插入和删除操作以

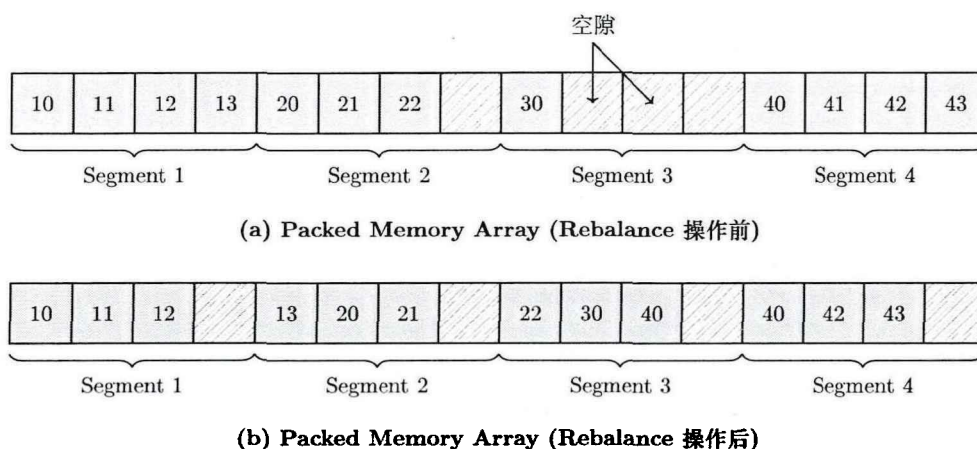


图 2.5 (a) 大小为 12, 每个 Segment 大小为 4 的 Packed Memory Array, (b) 进行 Rebalance 操作后的 Packed Memory Array。

段为单位执行。当插入/删除操作使得段变为全满或半满时, 需要将执行重平衡 (Rebalance) 操作来平衡整个序列中的空隙, 如图 2.5 所示。其直觉是访问相邻的段并分享它们的元素。参与重平衡的段序列被称为窗口。重平衡的基本思想是将元素或者空位分散到相邻的段中去。参与重平衡的段序列被称为窗口 W , 其选择基于以下方式: 给定一个段集合 W , 其包含的段数为 $|W|$, 窗口 W 中所有段的元素总数为 $s(W)$ 。为了决定是否进行重平衡以及如何进行重平衡, 需要计算 W 的最小可能元素数 $L(W)$ 和最大可能元素数 $U(W)$ 。这两个值分别表示了窗口 W 在进行重平衡后, 可以接受的元素数量的下限和上限。其中:

$$L(W) = \lfloor \alpha \cdot S \cdot |W| \rfloor$$

$$U(W) = \lceil \beta \cdot S \cdot |W| \rceil$$

α 和 β 是事先定义的密度阈值, 用于确定段的最低和最高允许密度。 s 是每个段的预分配大小。如果 $s(W) < L(W)$, 则表示窗口 W 内的元素过于稀疏, 需要通过合并其他级来增加密度。如果 $s(W) > U(W)$, 则表示窗口 W 内的元素过于密集, 可能需要分割级或在相邻级之间重新分配元素和空位。一般而言, 重平衡操作从一个不平衡的段开始创建一个窗口, 并迭代地添加它的相邻段, 直到满足上述条件为止。一旦找到一个窗口, 就将所有包含的元素在其段中平均分布。如果即使对于最大的窗口也无法满足公式, 则需要对 PMA 重新分配空间 (将容量加倍或减半)。考虑到平衡的开销, 在 PMA 上进行插入的时间复杂度为 $O(\log_2^2 n)^{[64]}$ 。需要指出的是, 一旦数组变得很大, 重平衡的代价也会变得很大, 因此直接使用 PMA 替代原来 CSR 中数组的 Packed CSR 在存储大规模的动态图时操作时延会变得很大^[65]。Teseo 使用 Trie 树将 PMA 分散到不同的叶子结点中, 控制 PMA 的大小从而维持图中边更新速度的相对稳定^[50]。同时, Teseo 还在此基础上使用了批处理技术进一步将插入的时间复杂度减小到对数时间。

另一类用作动态图边存储的结构为链表类结构，传统的单向链表尽管有利于插入（头插法），但具有访问随机、指针开销大等问题，因此最直观的做法是直接将链表中的多个节点进行合并，即使用块链表（Blocked Linked List）。在传统的链表中，每个节点通常包含数据部分和一个指向下一个节点的指针。而在块链表中，每个节点（或称为“块”）存储一组数据项而不是单个数据项，以及一个指向下一个块的指针。Stinger^[43]即使用了块链表进行数据的存储。但需要指出的是，块链表本质上还是一种单向链表，无法很好地维护数据的有序性，Stinger 和 GraphOne 均选择不维护链表中元素的有序性^[46]，因此其对边的更新和删除均需要线性时间 $O(n)$ 才能实现。除此之外块的大小需要根据具体应用场景事先设定，块的大小会影响数据存储和访问的效率。太大或太小的块大小都可能不利于性能优化。如果引入动态变化的块，则会引入额外处理块合并分裂的开销。

为了解决链表结构删除和更新效率低的痛点，Sortledton 引入了跳表（Skip List）来解决这一问题^[34,66]。跳表包含多层链表，每一层都是下一层的一个子集。最底层（Level 1）包含所有的元素。如果一个元素出现在 Level i 的链表中，它也必须出现在 Level $i - 1$ 的链表中，如图2.6所示。跳表每层的头节点通常表示负无穷，而尾节点表示正无穷。当进行搜索操作时，跳表从顶层开始，如果下一个元素大于要搜索的元素，则下降一层继续搜索。这个过程重复进行，直到到达第一层。因为在搜索中的每一层都跳过了下一层的一些元素，所以搜索效率比普通链表要高。在进行插入时，先搜索元素应该插入的位置，然后在最底层插入该元素。接着，通过抛硬币（随机过程）决定是否将元素提升至上一层。这个随机过程决定了元素在跳表中的最终高度。在删除时，先搜索要删除的元素，然后从找到的最高层开始，逐层删除该元素。在平均情况下，跳表的搜索、插入和删除操作的时间复杂度都是 $O(\lg n)$ 。在最坏情况下（所有元素都在最高层），这些操作的时间复杂度退化为 $O(n)$ 。

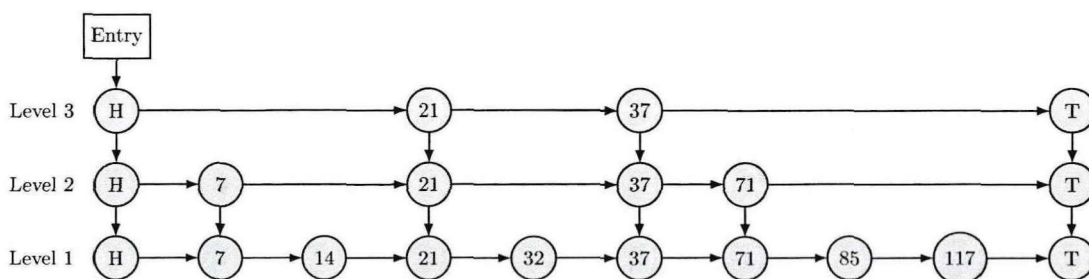


图 2.6 一个三层跳表，其中“H”代表头节点，“T”代表尾结点。

跳表通过增加层级的方式减少了在链表结构上维护有序数据的代价，但同时也带来了显著的额外空间开销，因此，Sortledton 使用了 Concurrent Unrolled Skiplist 作为实际存储节点的边的结构^[67]。Concurrent Unrolled Skiplist 通过在每

个节点中存储多个键值对来减少跳表的深度和提高空间局部性，从而减少了指针跳转次数，以便在查找给定元素时提高效率。同时，它还使用基于组互斥的高级锁定原语允许某些操作并发地在同一节点上工作，从而减少了争用窗口并增加了并发性。这意味着可以允许多个相同类型的操作（例如，都是插入或都是删除）同时在表中同一节点上执行，而不会相互冲突。

2.1.2 动态图数据存储结构的并发控制策略

早期的动态图数据存储结构或是没有涉及特殊的并发控制策略，仅提供简单的并行化支持（如 LLAMA），或是采用较简单的互斥锁策略，将读写进行分离（如 Stinger）。近年来数据中心的快速发展对图数据的并发性能提出了更高的要求，所以近年图存储结构的设计中并发性能也是被着重考虑的一个方面。就近年的工作而言，由于图存储结构的并发控制策略和它的索引结构、边存储结构息息相关，所以并发控制方案也源于与它们所用到的经典索引和存储结构。目前各类系统中常用到的并发控制策略主要包括乐观锁、Read Optimized Write EXclusive (ROWEX)、多版本并发控制 (Multi-Version Concurrency Control, MVCC) 和路径复制技术。

乐观锁^[18]是一种基于乐观假设的并发控制方法，它假设对同一资源的并发修改会很少。乐观锁通常由一个独占锁和一个时间戳 (Timestamp) 组成。每个写入者独占地修改数据并在每次写入时更新时间戳。每个读取者以时间戳作为参数，并在锁空闲时读取数据。读取后，它会重新检查时间戳以确保在读取过程中没有发生变化。如果检查发现数据过时，读取结果无效，相应的操作需要重新开始。乐观锁定的优点是它可以防止死锁并避免频繁在读取数据时锁定数据的开销。

当写入者之间的冲突较少时，乐观锁定表现良好，但在写入密集的工作负载上，重启策略可能导致读取性能显著下降。为了解决这个问题，提出了一种名为 Read Optimized Write EXclusive (ROWEX) 的策略^[56]，并首次在 Adaptive Radix Tree (ART) 中使用。ROWEX 的主要组成部分是写锁。这种锁只在写入者之间提供排他性，而读取者可以忽略锁进行数据读取。为了在确保读取正确性的同时避免重启，任何读取或写入操作必须原子执行。此外，使用 ROWEX 作为并发策略的数据结构内部通常还需要特殊设计以与原子操作协调。例如，为了应用 ROWEX，ART 为字典树中的每个节点添加了一个锁。因为在单一原子操作内无法确保整个节点结构的一致性（原子操作只能一次性操作 8 字节的数据），它采用读取-复制更新 (RCU) 来进行更新。在更新节点时，ART 首先锁定该节点及其父节点。之后，ART 会创建该节点的一个副本，并在副本上进行更新。然后，它使用原子操作将指向旧节点的指针更改为指向新副本。最后，它将旧节点标记

为过时，并释放旧节点及其父节点的锁。

多版本并发控制 (Multi-Version Concurrency Control, MVCC) 是一种广泛应用于数据库管理系统中的技术，用于提高数据库并发性能和实现对事务的支持。MVCC 的工作原理是为每个修改的数据项保留多个版本。这些版本通过事务在修改数据前的快照来实现，每个事务看到系统在特定时间点的一致性视图。这样，读取操作可以访问到事务开始时数据库的状态，不会看到正在进行的写入操作的中间状态，这样就避免了读-写冲突。当写入操作进行时，它们创建数据的新版本，而不是覆盖旧数据。不同事务可以看到同一数据的不同版本，因此 MVCC 允许多个读取和写入事务同时进行，而不会相互干扰。在图系统中，MVCC 通常是借助时间戳服务来实现的。以 LiveGraph^[48] 为例，每个顶点的邻接边被存储在一个称为事务边日志 (TEL) 的日志中，它记录了该顶点所有边的修改历史。TEL 中的每一项都包含边的目标顶点、创建时间戳、失效时间戳和边的属性。TEL 利用创建时间戳和失效时间戳来管理数据的版本，每当边被修改时，原有条目的失效时间戳会被更新，而新的修改会作为一个新条目被追加到 TEL 的末尾。读事务可以根据自己的时间戳，顺序扫描 TEL 中的条目，忽略那些在其时间戳之后创建或失效的条目。写事务则通过追加新的日志条目来记录修改，这些修改在事务提交后才对其他事务可见。

路径复制技术主要被用在 C-tree 类的数据结构上进行并发控制^[47]。C-tree 实际上是一种纯函数式树 (Purely-Functional Trees)，其设计宗旨是保证数据的不可变性和函数式编程原则^[68]。在这种树结构中，任何对树的修改操作（如插入或删除节点）都不会直接改变原有树的状态，而是返回一个新的树实例，这个新实例反映了修改后的状态。同时原有的树结构保持不变，这保证了数据的不可变性。当需要修改树中的某个元素时，系统会从根节点沿着到达目标节点的路径，复制路径上的每一个节点，直到目标节点。这样，原始树的结构保持不变，而修改是在路径复制产生的新节点上进行。当更新完成时，系统仅需要将树的根引用更新为新创建的树的根节点。这一过程可以原子性地完成，从而确保了写操作的原子性，不会因为并发读写操作导致数据不一致。读者操作通过获取 C-tree 的一个不变快照来执行查询，这意味着查询操作可以在数据结构的读开始时的版本上执行，而不会看到并发写操作所做的修改。（因为每次写入操作实际上是在创建一个新的 C-tree 版本，而旧版本保持不变）。在 C-tree 中，每个版本的树中都会有读者的引用计数，当非最新版本的数据的引用计数归零时，旧版本的路径将被删除。需要指出的是，由于路径复制的代价较大，本身并不适用于数据需要频繁更新的场景。路径复制技术在实现时往往与批处理操作进行结合，在一次复制中完成对多个数据条目的更新来平摊数据复制的开销。

2.2 图算法

2.2.1 经典图算法介绍及分析

在图分析中,常用的图算法有广度优先搜索 (Breadth-First Search, BFS), PageRank, 单源最短路径 (Single-Source Shortest Paths, SSSP), 弱连通分量 (Connected Components), 聚类系数 (Clustering Coefficient, CC) 等, 下面将依次对其进行介绍。

算法 2.1 Breadth-First Search (BFS)

Input: A graph $G = (V, E)$ and a starting vertex s
Output: The bfs tree *parent* with the source vertex s as root

```

1  $Q \leftarrow$  empty queue
2  $parent[s] \leftarrow -1$ 
3 Enqueue  $Q, s$ 
4 while  $Q$  is not empty do
5    $v \leftarrow$  Dequeue  $Q$ 
6   foreach neighbor  $u$  of  $v$  do
7     if  $parent[u]$  is undefined then
8        $parent[u] \leftarrow v$ 
9       Enqueue  $Q, u$ 
10    end
11  end
12 end
13 return parent
```

广度优先搜索 (Breadth-First Search, BFS) 是一种用于图的查找算法, 适用于无权图或者树的层次遍历。它从图的某一顶点开始, 访问起始顶点的所有邻接点, 然后再按这些邻接点被访问的顺序去访问它们各自的邻接点, 依此类推, 直到所有顶点都被访问, 如算法2.1所示。从图中的某个顶点 v 开始遍历。访问顶点 v 的所有未被访问的邻接点, 标记它们为已访问, 并将它们加入到一个先进先出队列中。当一个顶点的所有邻接点都被访问后, 从队列中取出下一个顶进行访问, 直到队列为空。当队列为空, 表示图中所有可达的顶点都被访问过。在遍历过程中, 存在着对顶点的随机访问 (第 5 行)、对图算法中相关记录数据的随机访问 (第 8 行对 *parent* 数组的访问) 和对顶点邻居的顺序访问 (第 6 行)。

PageRank 是由 Page L. 和 Brin S. 提出的算法, 用于衡量网页的重要性或质量^[69], 其伪代码如图2.2所示。它最初被用于帮助谷歌搜索引擎确定搜索结果中网页的排名。其核心思想是基于一个假设: 更重要的网页会有更多的网页链接到它, 而每个链接都可以视为对目标网页的一种投票。算法开始时会给予网络中每个页面一个初始的 PageRank 值, 通常是均等分配, 即每个页面的初始 PageRank 值为 $1/|V|$, 其中 $|V|$ 是网络中页面的总数。在每次迭代中, 每个页面 v 的 PageRank 值是基于指向它的页面的 PageRank 值计算得出的。计算公式考虑了从页面 u 到 v 的链接对 v 的 PageRank 值的贡献, 以及一个阻尼因子 d , 这个因

算法 2.2 PageRank Algorithm**Input:** A web graph $G = (V, E)$, damping factor d , convergence threshold ϵ **Output:** A PageRank score $PR(v)$ for each page $v \in V$

```

1 Initialize  $PR(v) \leftarrow 1/|V|$  for all  $v \in V$ 
2 repeat
3    $PR_{old}(v) \leftarrow PR(v)$  for all  $v \in V$ 
4   foreach page  $v \in V$  do
5      $PR(v) \leftarrow (1-d)/|V| + d \cdot \sum_{u \in B(v)} \frac{PR_{old}(u)}{L(u)}$ 
6   end
7   convergence  $\leftarrow$  true if  $\forall v \in V, |PR(v) - PR_{old}(v)| < \epsilon$ 
8 until convergence
9 return  $PR$ 

```

子模拟了浏览者不通过链接而是随机跳转到某个页面的行为。在每轮迭代后，算法会检查所有页面的 PageRank 值是否已经收敛，即这些值的变化是否小于某个预设的阈值 ϵ 。如果所有页面的 PageRank 值变化都小于 ϵ ，算法结束；否则，进行下一轮迭代，直至收敛为止。需要指出的是，现在多数的 PageRank 实现中会设置最大迭代次数限制以避免在无限循环迭代的情况，在 PageRank 算法中，存在着对顶点的顺序访问（第 4 行）以及对顶点邻居的顺序访问（第 5 行）。

算法 2.3 Dijkstra's Algorithm**Input:** A weighted graph $G = (V, E)$, source vertex s **Output:** Shortest path distance from s to every other vertex

```

1 for each vertex  $v \in V$  do
2    $dist[v] \leftarrow$  infinity
3    $Q[v] \leftarrow$  true // all vertices are initially in Q
4 end
5  $dist[s] \leftarrow 0$ 
6 while  $Q$  is not empty do
7    $u \leftarrow \text{ExtractMin}(Q, dist)$ 
8    $Q[u] \leftarrow$  false
9   for each neighbor  $v$  of  $u$  do
10    if  $dist[u] + \text{weight}(u, v) < dist[v]$  then
11       $dist[v] \leftarrow dist[u] + \text{weight}(u, v)$ 
12    end
13  end
14 end
15 return  $dist[]$ 

```

单源最短路径（Single Source Shortest Path, SSSP）问题是在加权图中找到从一个源顶点到所有其他顶点的最短路径的问题。常见的 SSSP 算法有广度优先搜索、Dijkstra 算法，以及 Bellman-Ford 算法^[70]。依据 SSSP 算法的不同，算法的访存模式也有所差异。基于广度优先搜索的 SSSP 只适用于有向图或者边权权相等的图，其执行流程和访问模式和 BFS 基本相同，唯一不同的是需要在搜索过程中为每个访问到的顶点标记它到源顶点的距离，即访问的层次。Dijkstra 算法（算法 2.3）基于贪心策略，每次找到距离源点最近的未被访问过的顶点，更新其邻接顶点的距离。由于 Dijkstra 算法通常需要借助堆实现优先队列获取最

算法 2.4 Bellman-Ford Algorithm**Input:** A weighted graph $G = (V, E)$, source vertex s **Output:** Shortest path distance from s to every other vertex, or detection of a negative weight cycle

```

1 Initialize  $dist[v] \leftarrow \infty$  for all  $v \in V$  except  $dist[s] \leftarrow 0$ 
2 for  $i \leftarrow 1$  to  $|V| - 1$  do
3   foreach  $edge(u, v) \in E$  do
4     if  $dist[u] + weight(u, v) < dist[v]$  then
5        $dist[v] \leftarrow dist[u] + weight(u, v)$ 
6     end
7   end
8 end
9 return  $dist[]$ 

```

近未被访问过的顶点，运行时为了维护堆结构，会产生较多的对堆的随机访问。Bellman-Ford 算法通过反复访问途中每一个顶点的所有邻接边进行松弛（检查是否可以通过该边得到从源顶点到其它顶点的更短的路径）操作，逐步更新顶点间的最短距离，直到达到最短路径或发现负权环，存在着较强的顺序访存模式，其伪代码如算法2.4所示。Delta-Stepping 算法将顶点分成多个桶（bucket），每个桶内的顶点按距离源点的距离分组。每次松弛处理一个桶中的所有顶点，更新其邻接顶点的距离，并根据这些距离将邻接顶点分配到合适的桶中^[71]，既存在着对顶点点邻居的顺序访问，也存在着对桶中元素以及图中顶点的随机访问。

算法 2.5 Weakly Connected Components Algorithm**Input:** A directed graph $G = (V, E)$ **Output:** Components array where $Components[v]$ is the component id to which vertex v belongs

```

1 Function DFS( $v, componentId$ ):
2    $Components[v] \leftarrow componentId$ 
3   foreach  $u \in neighbors(v)$  do
4     if  $Components[u] = -1$  then
5       DFS( $u, componentId$ )
6     end
7   end
8  $Components \leftarrow$  array of size  $|V|$  initialized with  $-1$ 
9  $componentId \leftarrow 0$ 
10 foreach  $v \in V$  do
11   if  $Components[v] = -1$  then
12     DFS( $v, componentId$ )
13      $componentId \leftarrow componentId + 1$ 
14   end
15 end
16 return  $Components$ 

```

在图论中，一个无向图的弱连通分量（Weakly Connected Component, WCC）是图中的一个最大子图，其中的每一对顶点都是连通的，即在这个子图中的任意两个顶点之间都存在一条路径。对于有向图，当忽略边的方向时，如果能找到这样的路径，则称这个有向图是弱连通的。对于无向图，其通常简称为连通分量。

识别一个图中的所有连通分量是图论中的一个基本问题,对于数据挖掘、社交网络分析等领域均有使用。最简单的 WCC 计算是通过深度优先搜索 (Depth-First Search, DFS) 实现的,如算法2.5所示^[72]。其基本思想是对于每个未被访问的顶点 v ,使用深度优先搜索遍历所有可达的顶点,并将这些顶点标记为属于同一个连通分量。目前效率较高的一种 WCC 算法为 Shiloach-Vishkin 算法^[73]。Shiloach-Vishkin 算法的核心在于并行地更新图中每个顶点所属的连通分量标识符。初始时,每个顶点的连通分量标识符设置为其自身的 ID。算法在迭代过程中不断地“钩连”操作(即如果两个顶点通过一条边直接相连,它们应属于同一连通分量,因此它们的连通分量标识符应该一致,并行地检查和更新边的两端顶点的连通分量标识符,直到所有顶点的连通分量标识符不再发生变化,算法结束。与 PageRank 算法类似, WCC 中也在着对顶点的顺序访问(第 10 行)以及对顶点邻居的顺序访问(第 3 行)。

算法 2.6 Local Clustering Coefficient

Input: A graph $G = (V, E)$ and a target vertex v
Output: The local clustering coefficient $C(v)$ of vertex v

```

1  $neighbors \leftarrow$  Get all neighbors of  $v$ 
2  $\Delta \leftarrow 0$ 
3 foreach  $u \in neighbors$  do
4   foreach  $w \in neighbors$  do
5     if  $u < w$  and  $(u, w) \in E$  then
6        $\Delta \leftarrow \Delta + 1$ 
7     end
8   end
9 end
10  $N \leftarrow$  size of  $neighbors$ 
11  $C(v) \leftarrow \frac{2\Delta}{N(N-1)}$ 
12 return  $C(v)$ 

```

聚类系数 (Clustering Coefficient, CC) 是图论中用来衡量一个顶点的邻居之间互相连接的程度的指标,经常被用于图模式匹配 (Graph Pattern Matching)。对于一个给定的顶点,它的局部聚类系数定义为该顶点的邻居顶点之间实际连接的边数与这些邻居顶点之间可能形成的最大边数之比^[74]。目前常用的计算局部聚类系数的方法是通过统计三角形的数量。在对于给定的顶点 v ,其局部聚类系数可以通过比较 v 参与的三角形数量与 v 可能参与的最大三角形数量之比来计算。顶点 v 的局部聚类系数 $C(v)$ 计算公式为 $C(v) = \frac{2\Delta(v)}{N(v)(N(v)-1)}$, $N(v)$ 是 v 的邻居数, $N(v)(N(v)-1)/2$ 是 v 的邻居间可能形成的最大边数(即最大可能的三角形数),其伪代码如算法2.6所示。全局聚类系数则用来衡量整个图中闭合三角形的比例,反映了网络整体的聚集程度。最常见的是计算方法基于三角形闭合的比例,即 $C = \frac{3 \times \text{图中三角形数量}}{\text{图中的连线构成的三元组的数量}}$ 。在计算整个图的聚类系数时,CC 也会呈现出较强的顺序访问特征。

2.2.2 并行广度优先搜索的实现方式及优化策略

并行的广度优先搜索大致可以分为基于容器的方法和基于顶点的方法。通常来说,基于容器的方法会使用两种相同的容器来进行层级同步。其中一级为当前层中访问的节点,另一级为下一次迭代中各个处理器将要访问的顶点。基于容器的方法分别使用两种容器来存储这些顶点,这两个容器在遍历中扮演的角色(即 **current/next**) 在每一次迭代中会进行交换。容器一般被所有的计算节点共享^[75],因此将不可避免地产生并发读写访问的问题。简单的列表(List)即可以作为一种容器。为了克服并发访问的问题,一种方法是采用各个处理器或线程维护自己私有的列表,并让各个处理器将自身所拥有的列表分为两类:一种是自己的本地列表,用来保存需要访问或扩展的本地节点;另一种是用于通信的非本地列表,用来保存不在本地的顶点编号^[76]。具体而言,每一个顶点都被静态的分配到一个固定的线程当中,如果一个线程访问到了其它线程所拥有的顶点,那么它就将该顶点添加到通信列表中。当到达 **Barrier** 后,所有的线程会进行一次通信操作来同步在下一轮中需要访问的顶点。经过该步骤后,每一个线程在下一轮中需要扩展的源结点都是在本地的。需要指出的是,虽然该方法通过静态分配保证了良好的局部性。由于在共享存储体系下进行全局,最坏情况下为 **all-to-all** 的通信操作,随着线程数量的增加,通信所需要的时间和空间代价也可能迅速增长。因此其受制于总线带宽、内存大小等条件,可扩充性有限。由于采取了静态分配策略,该方法的负载平衡也只能依赖于划分方法,而这通常是难以保证的。

与之相对应的,有以局部性为代价,专门专注于负载平衡的方法^[77-78]。和容器的私有方法不同,这类方法中通常使用全局的、允许并发访问的容器。所有的线程共同在这些容器上工作,并采取粗略的负载平衡方法。例如,采用基于原子化的 **CAS (Compare and Swap)** 操作来支持并发访问的无锁队列。需要指出的是,由于使用全局的队列来保障基础的负载平衡,该类方法完全忽略了数据的局部性,因此该类方法的性能严重受制于访存效率。

除此之外,还有在非均匀内存访问架构 (**Non-Uniform Memory Access, NUMA**) 下将上述两种方法结合,各取所长的方法^[79-81]。该类方法在 **NUMA** 架构下,将各个顶点划分到对应的 **Memory Bank** 当中,形成了基于 **NUMA** 节点的顶点划分。对于计算节点中的 **socket** 其设置了对应的通信缓冲区。在每个 **socket** 的内部仅有 **socket** 内全局的共享列表用于 **BFS**, **socket** 下所有的线程可以并发的读写该 **socket** 中的内容。

近些年来,随着图形处理单元 (**Graphics Processing Unit, GPU**) 的发展,越来越多的 **GPU** 上基于容器的并行 **BFS** 算法也被开发出来^[21,82]。不同于使用 **CPU** 的 **BFS** 算法, **GPU** 上以 **Warp** (本质上是一组线程) 而不是线程为实际单位进行

编程。GPU 的 BFS 分为 SISD（单指令流单数据流）和 SIMD（单指令流多数据流）两个阶段。在 SIMD 阶段中，Warp 当中的每一个线程执行相同的代码把全局存储中的顶点信息拷贝到本地，通常该操作可以在硬件层面快速完成。在 SIMD 阶段中，每个线程并行地访问分配到顶点的邻接顶点。

在基于顶点的 PBFS 方法中，每一个并行实体被分配到图中的顶点中，然后所有的算法在这些顶点上并行地被执行，每个顶点的邻居被并行地访问，部分被标记为下一层将要扩展的节点^[83]。在最坏情况下，该方法的时间复杂度为 $O(n^2)$ （比如线性表，每一层只有一个节点， n 为节点数量）。不同于基于容器的方法，该方法不需要任何的容器，只需要一个用于标记节点深度的层次数组。不是待扩展的节点被分配到各个并行实体上，而是所有顶点都会自发地检查自身在接下的过程中是否需要被访问，因此该方法也不需要额外的通信等同步步骤。尽管该方法简单高效，但是没有考虑负载平衡问题，也不适用于深度较深的图。

由于在该类方法中，大量节点的工作非常相似，同样也非常适合使用 GPU 来执行^[22]。借助于 GPU 的高度并行化，每个节点被分配到一个线程来进行遍历。由于在大数据时代，可能会出现图较大，远超 GPU 能同时并发执行的线程数的情况，因此有人提出使用层次性的线程管理来适应计算机的内存的层次结构，避免由于过载导致的频繁的 kernel 重启。在本类方法中，对于顶点的访问通常是线性的，因此在分布式机器上也具有较好的局部性。当采取合适的划分方式时，该类方法在分布式的体系结构中也可以以较低的通讯代价高效执行。

在各类 PBFS 方法中，有一些通用的方法被用于优化访存和负载性能。使用 Bitmap 来替代数组进行顶点访问信息的存储能够有效地减少内存占用，改善 cache 的局部性^[79]。在并行中，细粒度的优化如内存预取，多线程同时扩展同一顶点等方法可以减少延迟，有效克服 GPU 的性能瓶颈^[84]。在为了减少由高度数顶点扩展而引起的等待现象，Hong S. 等人提出设置专门的数据结构存储顶点的度数，当遇到高度数顶点时，将其分配到额外的容器中，交由专门的线程处理^[85]。Zhang H. 等人提出了非精确的负载分配方案：一个并行实体关注其邻居实体的工作状况，并且可以在自身空闲时接取邻居的工作^[36]。

2.3 本章小结

本章内容聚焦于介绍与本文研究密切相关的各项研究工作。在第一部分中，着重介绍了图数据存储结构的相关研究，具体包括动态图的顶点索引和边存储结构，并对这些存储结构上的并发设计策略进行了详细讨论。第二部分深入到图算法的相关知识，回顾了一些经典图算法，分析了其执行流程和内存访问模式，并特别对并行广度优先搜索的实现及其优化方案进行了相关介绍。

第3章 动态图数据的高效存储方法及图算法实现

本章概要：本章节主要介绍 Spruce, 一种高效的内存动态图数据存储方法。本章节首先从需要解决的问题入手, 分析现存方法针对该问题的解决思路以及不足之处, 引出 Spruce 设计的动机和思路。然后, 本章介绍 Spruce 的基础数据结构以及在该数据结构上的基本操作方法, 分析其操作的时间复杂度并和目前主流的工作进行对比。在此基础上, 本章进一步提出针对该类数据结构上设计的并发控制策略, 以及并行图算法的实现。最后通过设计实验和目前主流的工作进行对比, 验证了 Spruce 的操作性能、空间开销、图算法性能、并发性能的图存储方法等关键指标。

3.1 问题分析

3.1.1 图数据的特点和动态图数据的存储要求

图数据是对现实生活中实体以及实体关系的抽象表达方式, 其特征和作为数据源头的实体息息相关。通常而言, 现实生活中的大图往往具有如下的多种特征^[86]:

1. **稀疏性:** 对于一张图 $G = (V, E)$ 而言, 其边的数量与顶点数量之比 $\frac{|E|}{|V|}$ 总是小于一个较小的常数。或者说, 图 G 的中边的数量远远小于其可容纳的最大的边的数量 (不考虑重边)。
2. **小世界特性 (Small World Phenomenon):** 图的直径较小, 即任意两个顶点之间往往存在一条较短的路径。同时, 拥有公共邻居的两个顶点很有可能互为邻居。
3. **幂律特征:** 图中所有顶点的度数分布往往遵循幂律分布。

由于图数据的存储需要依赖于具体的数据存储格式, 除去上述图本身的性质外, 图数据在计算机系统上的表达形式也有明显的特征。图的顶点一般适用整数进行表示。在图从现实实体抽象到整数表示的过程中, Graph Labeling 技术发挥了重要作用。在图论中, Graph Labeling 是指将整数 (标签) 分配给各个顶点或边的过程^[87]。由于边可以由用一组顶点对构成的二元组进行表示, 在现代的图数据中, Graph Labeling 主要被用于对图中的顶点进行编号。考虑到数据的动态增长需求以及存储的便利性, 现在的图数据往往采用顺序的顶点编号方法, 如表示用户的 User ID, 表示商品的 Object ID 等。这类特征为压缩存储数据提供了便利性, 如静态图框架 Ligra+ 中就采用了差分编码技术对图中的顶点进行压

缩^[31]。

动态图存储结构即是在存储图数据的同时高效地支持如下一系列基本操作：边的插入、删除、属性更新；顶点的插入、删除；边、顶点的属性查询；某个顶点的邻居节点的获取等等。一般将上述的基本操作简称为图数据的更新和查询。在上述基本操作的基础之上，现在的各种应用场景还对其提出了诸如高并发、图模式匹配等更多的功能需求。

3.1.2 现存方法的解决思路及不足之处

动态图存储的关键在于以较低的延迟实现图数据的更新和查询（插入、删除、属性更改等）。现有的大多数动态图系统都是在邻接表或 Comparesed Sparse Row(CSR) 的基础上进行改进以存储不断变化的图。

基于邻接表的动态图系统主要集中在两个方面：(1) 使顶点表支持动态操作；(2) 提高边链表的访问和更新速度。对于 (1)，大多数系统使用索引将原有的顶点标识符（顶点 ID）映射为逻辑顶点标识符（类似于重新编号），并将逻辑顶点存储在数组中。例如，哈希表、Adaptive Radix Tree 和多级向量数组都不失为用来索引逻辑顶点的有效方法。然而哈希表在处理碰撞时比其他索引占用更多空间，而且现有的用于图系统的树形索引在执行图更新时无法避免合并、拆分或调整节点大小，从而降低了效率。对于 (2)，现存的方法迥乎不同。Stinger^[43] 将边链表中的节点聚合为块以提高读取效率，并使用哈希表作为二级索引以加速点查询。Graphone^[46] 新颖地将邻接表与边表（Edge Array）结合在一起。图上的任何更新都会首先追加到边表的尾部，然后批量合并到邻接表中。Sortledton^[34] 采用 unrolled skip list^[67] 来替换边链表。Unrolled skip list 可被视为聚合节点的跳表结构，它在一个节点中存储原链表节点中的多条边，并借用跳表的概念实现对数时间复杂度内的查找、插入和删除操作。Aspen^[12] 和 Pac-trees^[52] 开发了一种新的压缩纯功能搜索树数据结构，它可以以较低的空间占用来存储压缩边数据，更新速度会较慢。遗憾的是，这些方法都无法既减少边链表的空间占用的同时获得较高的图更新操作的吞吐量。

使 CSR 支持图更新的典型方法有三种。第一种是将 CSR 分成若干段，以避免重建整个数组^[49]。顶点存储在以全局数组为索引的分段数组中，每个顶点的边存储在一个边数组中。插入时只需编辑单个段并移动相应边数组中的值。虽然这样的设计可以实现 CSR 更新，但每次插入时移动数组中元素的成本在一定程度上限制了系统的吞吐量。第二种是使用快照进行图更新^[44]。初始图上的任何批量更改都会被写入新的快照，对图的查询可能会遍历多个快照。当图频繁更新时，快照会迅速增长并占用大量空间。因此，这种方法只适用于很少变化的图形。第三种方法是使用 Packed Memory Array (PMA) 来存储边^[65]。PMA 中的元

素之间安排了空槽或间隙,因此在插入新元素时无需移动现有元素。然而,当空隙较少或分布很不均匀时,这些空隙需要重新平衡使之均匀分布在各个元素中,这导致数组较大时会出现明显的更新延迟。此外,间隙还会占用额外的空间,无法很好地体现 CSR 的空间优势。

除去存储结构对动态图操作性能的影响外,并发协议设计对其性能也至关重要。早期的工作只提供了并行化支持^[88-90]。为了提高并发效率,这些系统通常依次执行更新和查询,这意味着更新要等所有查询完成后才能更新图,而查询要等更新完成后才能访问图。最近的研究中,图更新与查询的并发执行是一个新的关注点^[34,48,50,52]。大多数研究使用乐观锁策略和多版本并发控制(MVCC)在不断演化的图上运行图算法。例如,LiveGraph 对每条边使用了两个时间戳,以支持有序、事务性的图扫描,而 Teseo 则对事务使用了乐观锁的变体。本文注意到,这种方法可以使用 ROWEX 进行优化,从而提高读取效率。

3.1.3 vEB 树及其特点和限制

位向量(Bitvector)是一种用于紧凑地存储比特的数组,是 vEB 树的基础结构。该数组中的每一位元素为 0 或者 1,它可以使用更少的空间有效地将一系列整数映射到集合 $\{0,1\}$ 中的值(例如,5 个字节的集合 $A = \{0,2,5,6,7\}$ 可以表示为 8 位位向量 $B[8] = \{1,0,1,0,0,1,1\}$)。Van Emde Boas 树(vEB 树)是一种支持在最坏情况下以 $O(\lg \lg(u))$ 的时间对从数域 $U = \{0,1,2,\dots,u-1\}$ 中选择的 n 个键的集合 S 进行动态操作的树形结构。为了方便读者理解,此处仅讨论 vEB 树的原型结构。vEB 树的原型是一棵由位向量组成的多叉树,需要 $O(u)$ 空间存储 n 个元素,如图 3.1 所示。其底部的位向量总共包含 u 位,其中每个位表示 U 中相应整数的存在或不存在。上面的节点被视为一个位向量 $summary[0,\dots,\sqrt{u}-1]$ 。如果 $summary[i]$ 为 0,则下层大小为 $\sqrt{u}$ 的位向量中的位都为 0。

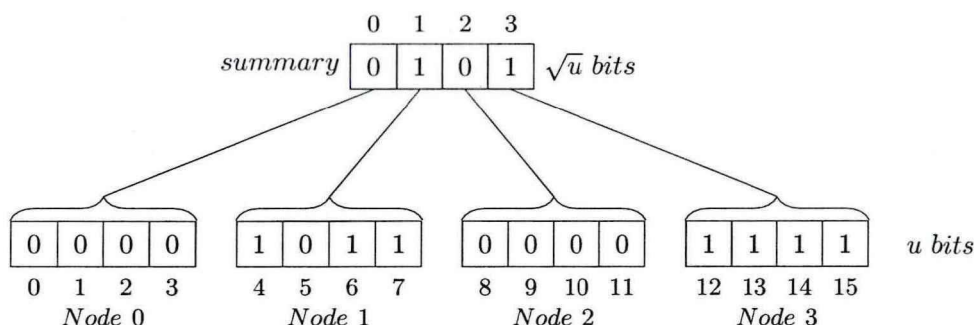


图 3.1 vEB 树的原型结构

vEB 树的主要优势特点是具有良好的动态操作性能。以图 3.1 中的 vEB 树为例。为了插入 2, 只需要将 Node 0 中的相应位设置为 1, 同时将 $summary[0]$ 设置为 1。要删除 15, 需要首先将 Node 3 中的第 4 位设置为 0, 然后将 $summary[3]$

设置为 *Node 3* 中的所有位的逻辑或的结果, 即 1 这些操作在 *vEB* 树上只需要 $O(\lg \lg(u))$ 的时间, 因为位向量的总长度在每一级都以平方根的规模缩减。与 *B* 树^[91]、*B⁺* 树^[62]、前缀树^[60]以及它们的许多变体^[61,92-93]不同, *vEB* 树的操作时间仅为 $O(\lg \lg(u))$, 这些操作不会导致节点的大小调整、拆分或合并, 为高效的动态操作提供了充分的支持。

vEB 树的主要限制是其规模和数域 U 有关, 空间性能往往不理想。一棵标准的 *vEB* 树需要 $O(u)$ 的空间来存储数域 U 中大小为 n 的子集。当 $n \ll u$ 时, 会产生大量的空间浪费。如果使用哈希表来代替位向量, 并只为非空节点 (即至少有一个值的节点) 分配空间, 这个上限可以减少到 $O(n \lg \lg(u))$ ^[94]: 每层最多占用 $O(n)$ 空间, 共有 $O(\lg \lg u)$ 层。因此, 总空间约束为 $O(n \lg \lg(u))$ 。然而, 这样的表示法需要大量不同的哈希表, 仍然耗费相当多的空间, 并降低了 *vEB* 树的性能。

3.1.4 设计动机与思路

基于现有顶点图存储工作的局限性以及对近年来相关工作的分析, 本文提出以下设计动机与思路:

1. 用类似 *vEB* 树的结构取代顶点表以支持顶点的动态操作。与动态图存储中常用的其他索引相比, *vEB* 树有具有节点在更新时不会合并、拆分或复制的显著优势。此外, 由于图数据通常使用 *Graph Labeling* 技术来提取, 其中唯一、密集的整数被用作顶点 ID^[50,95-96]以提高存储和计算效率, 所以位向量非常适用于存储这些数据。然而, 原始 *vEB* 树中位向量的规模仅与给定整数的数域大小有关。对于 64 位整数, 有 6 层位向量, 最底层的位向量总共可达 2^{64} 位, 如此巨大的空间开销是难以忍受的。为了解决这个问题, 本文借鉴 *vEB* 树的概念, 设计了一种基于块的具有较低内存占用和访问延迟树形顶点索引。
2. 重新设计由缓冲区块 (*BufferBlock*) 和有序块 (*SortedBlock*) 组成的边链表, 在空间使用和更新速度之间取得平衡。邻接表为图更新提供了良好的支持, 而 CSR 则提供了出色的内存访问性能。考虑到现实世界中的大多数演化图都是稀疏的, 大多数顶点的度数相对较低^[97-98], 本文对于每个顶点都设计了一个有序块 (*Sortleblock*) 来像 CSR 一样紧凑地存储其邻居, 并使用一个由指示符索引的缓冲区块 (*BufferBlock*) 来替代链表结构来实现边的快速插入。由于高度数节点通常只占顶点的很小一部分, 它们的整体维护成本相对较低, 因此本文并不专门为它们设计复杂的结构。
3. 设计支持并发操作的轻量级并发控制协议。ROWEX 最显著的优势在于读者永远不会被阻塞。为了实现这一目标, ROWEX 的传统设计通常会采用

排他锁、Compare-And-Swap (CAS) 和 Read-Copy Update (RCU) 技术。然而,在复制块时,RCU 的实现会带来额外的空间和时间成本。此前还没有关于在 vEB 树上支持并行性的工作^[99],而 vEB 树中不存在节点拆分和合并,这就为在不使用 RCU 的情况下通过编辑节点/块中的数据来完善 ROWEX 提供了可能性。对于获取顶点邻居时的一致性需求,乐观锁是一个理想的解决方案。因此,ROWEX 和乐观锁的结合非常适合本文的树状数据结构。

3.2 数据结构设计

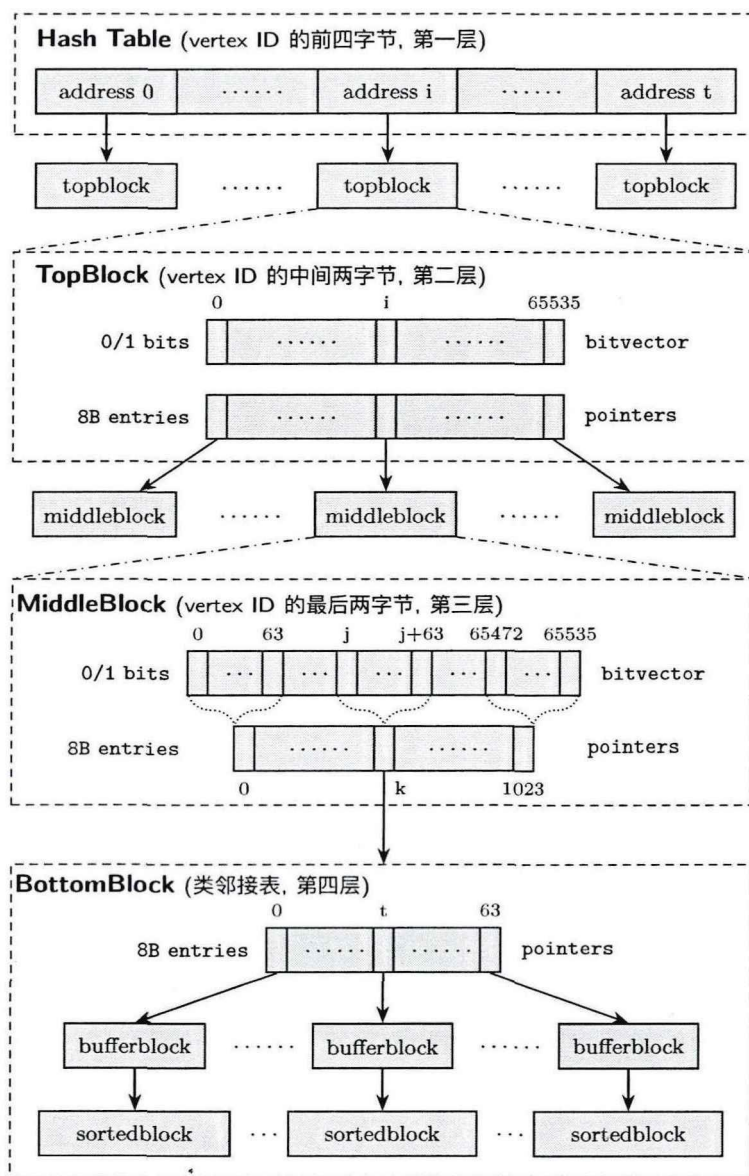


图 3.2 Spruce 的数据结构设计

本文所设计的动态图存储结构名为 **Spruce**。**Spruce** 的数据结构由类似 vEB 树的顶点索引和边存储块 (**BottomBlocks**) 组成,如图3.2所示。**Spruce** 的索引

由哈希表和多级的位向量组成。位向量被分割成固定大小的块 (TopBlock 和 MiddleBlock), 并按需进行分配。顶点的边及其属性存储在与 MiddleBlock 相连的 BottomBlock 中。对于顶点 ID, Spruce 支持最高 8 字节整数, 这足以容纳多达 2^{64} 个顶点, 也足以支持实际生活中常用的大规模图。在边属性的存储上, Spruce 支持固定长度属性 (或指向可变长度属性的固定长度指针)。在下面的小节中, 本文将详细介绍 Spruce 的结构及其操作。

3.2.1 顶点索引结构

Spruce 在索引结构中存储了明确的顶点 ID (即未重新编号的原始顶点 ID), 这些标识符将顶点映射到其连接的边。本文将 8 字节的顶点 ID 分为 $4 + 2 + 2$ 字节, 分别存储在哈希表、TopBlock 和 MiddleBlock 中。通过这样的划分, Spruce 中每一级的节点覆盖顶点的最大规模都以平方根的规模递减: 一个 MiddleBlock 最多可覆盖 2^{16} 个顶点, 一个 TopBlock 最多可覆盖 2^{32} 个顶点, 而处于索引顶端的哈希表最多可覆盖 2^{64} 个顶点。因此, 索引中有 $O(\lg \lg(u))$ 层 (u 为数域的大小), 动态操作 (如插入、删除) 只需 $O(\lg \lg(u))$ 时间即可完成。这些层级中的所有块都是在需要时动态分配和回收的, 这使得 Spruce 能够动态地适应动态图大小的变化。

Spruce 第一层是哈希表。图数据中顶点 ID 的前 4 个字节被哈希到哈希表中的各个单元, 每个单元中存储中间 2 个字节的块的地址 (即 TopBlock)。即所有具有相同 4 字节前缀的顶点都会映射到相同的 TopBlock 中。本文选择哈希表的原因有二: (1) 具有相同 4 字节前缀的顶点 ID 共享相同的哈希地址, 这可以大大节省空间; (2) 在哈希表中进行查找可以在 $O(1)$ 时间内完成。在 Spruce 中使用了 Junction Hash^[100], 一种使用基于线性探测的开放定址法解决哈希冲突问题的并发哈希表。

顶点 ID 的中间 2 个字节在 TopBlock 中编码。每个 TopBlock 包含一个表示中间 2 个字节的 2^{16} 位的位向量和一个存储指针的数组。位向量中的每一位都表示 2 字节对应的值的存在与否, 以及指向存储最后 2 个字节的块 (即 MiddleBlock) 的相应指针 (和 vEB 树的全局位向量类似)。位向量的优势在于只需要 1 位就能表示一个值。在这里, 本文用 1 表示值存在, 用 0 表示值不存在。例如, 如果 TopBlock 中的位向量是 000100..... 序列, 那么第四位表示中间 2 个字节的值 0x0003 存在, 并且指针数组中的第四个指针指向相应的 MiddleBlock。位向量的主要作用在于加速顶点的范围扫描, 这在删除 (详见第 3.2.3 节) 和许多图算法 (如 PageRank、WCC 和 LCC) 中都会用到。由于访问内存的次数更少, 扫描位向量比扫描指针数组要快得多, 可以提供更好的范围查询的性能。

顶点其余字节在 MiddleBlock 中编码存储。MiddleBlock 与 TopBlock 类似。

包含一个 2^{16} 位的位向量和一个由 2^{10} 组指针组成的指针数组。第 j 个组指针指向第 j 个 **BottomBlock**，其中聚合存储着从最后 2 个字节的 $(2^6 \cdot j)$ 到 $(2^6 \cdot (j+1) - 1)$ 的顶点 ID 的邻接边。在 **Spruce** 中，如果上层块的位向量的第 i 位是 1，那么第 i 个 **MiddleBlock** 就存在，并且 **MiddleBlock** 的位向量中至少有一个 1。也就是说，上层是下层的 **Summary**。需要注意的是，**TopBlock** 的数量比 **MiddleBlock** 的数量减少了 2^{16} ，所以本文不使用聚合技术来减少它们的空间。

3.2.2 边存储结构

Spruce 使用 **BottomBlock** 存储顶点的邻接边。**BottomBlock** 是一个类似于邻接链表的结构，用于存储一组顶点的邻接边（对应顶点 ID 的最后 6 位）。一个 **BottomBlock** 由三部分组成：指针数组、**BufferBlock** 和 **SortedBlock**。如图 3.2 所示，由第 k 个组指针指向的底层块中的指针数组有 64 个子指针，每个子指针指向一个 **BufferBlock**，该 **BufferBlock** 存储顶点最后 2 个字节在 $[j, j + 63]$ 范围内的邻接边。邻接信息存储在 **BufferBlock** 和 **SortedBlock** 中，**BufferBlock** 被设计为 **SortedBlock** 的插入缓冲区来提高写入效率。

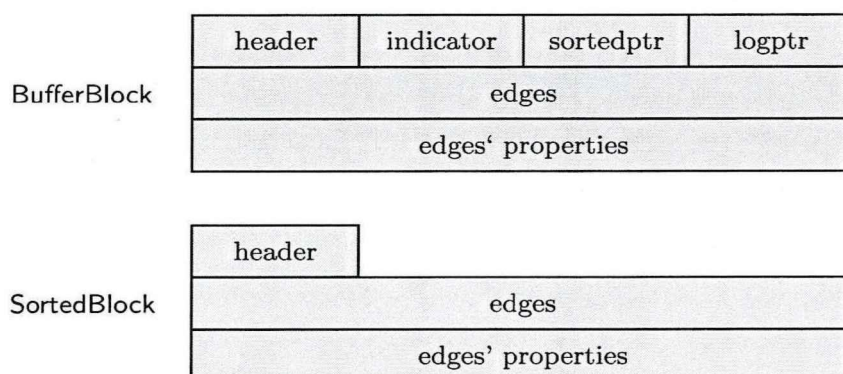


图 3.3 **BufferBlock** 和 **SortedBlock** 的结构设计

BufferBlock 用于临时存储新插入的边。**BufferBlock** 的数据布局如图所示。**BufferBlock** 包括一个头部、一个 8 字节的指示符、两个指针（**sortedptr** 和 **logptr**，一个指向 **sortedblock**，一个指向日志存储的相关结构）和两个数组（一个用于存储边，即该顶点邻居的 ID，另一个用于存储边的属性）。头部表示 **BufferBlock** 的类型，并包含与锁相关的信息（将在下一节讨论）。如果顶点的边的数量小于或等于 **BufferBlock** 的最大容量 C （本文将 C 设置为 64，以便与 8 字节的指示器保持一致），所有的边都会存储在 **BufferBlock** 中。否则 **Spruce** 会为该顶点分配一个 **SortedBlock** 来支持存储更多的边。在 **BufferBlock** 中，顶点邻居的 ID 和边的属性（可选）会按照相同顺序存储在两个数组中，指示符会显示数组的占用状态。**Spruce** 为数组设置了几种基本类型，使其能够随一个顶点的邻接边的规模而

动态变化,其类型存储在 **BufferBlock** 的头部中。如果指示器的第 i 位为 0,那么邻居 ID 的数组中的第 i 个位置为空。否则,该单元格存放着该顶点的邻居信息。这种设计具有以下优点:

- **加速插入:** 为了在数据块中找到一个空位置进行插入, **Spruce** 只需对指示符进行位操作即可得到空位的索引,而无需顺序扫描整个数组。
- **提高读取效率:** 在访问顶点的邻接边时, **Spruce** 可以根据指示器指示的 **BufferBlock** 的空闲状态从非空单元格中提取数据,避免了对整个数组的扫描。

SortedBlock 的设计目的是在 **BufferBlock** 之外存储更多的边。**SortedBlock** 包括一个头部、一个顶点邻居 ID 数组和一个边属性数组。头部用于指示该块的容量和块中已删除边的数量。**SortedBlock** 中的邻接边(删除的边除外)是无间隙连续存储的,并依照顶点 ID 从大到小排序。当删除一条边时, **Spruce** 会直接将该标记该边无效。当一个完整 **BufferBlock** 中的边被移动到 **SortedBlock** 中时,将触发合并排序以合并新添加的边。**SortedBlock** 的优点如下:

- **加速查询:** 在 **SortedBlock** 中可以使用二分搜索可以在对数时间内找到一条边。同时有序的数据组织形式有利于基本的图模式匹配算法,比如 LCC 算法(局部聚类系数)。
- **最小化空间使用:** **Sortedblock** 将边紧凑地存储在数组中,从而有效地利用了空间。与链表结构相比,它避免了指针的额外开销,增强了数据的局部性,同时通过批量插入的形式保持了良好的插入性能。

3.2.3 基本操作

和大部分图系统一样,在 **Spruce** 中,无向图藉由有向图实现,即每条无向边被视为两条有向边进行存储。一般来说,动态图操作包括顶点和边的插入、删除、更新和查询,下面本文将对这些操作在 **Spruce** 上的实现进行介绍:

顶点操作: 顶点操作包括顶点的定位 (**Locate**)、插入 (**Insert**)、删除 (**Delete**) 和获取顶点邻居 (**GetNeighbours**)。顶点定位是 **Spruce** 的基本操作。它检查 **Spruce** 中是否存在顶点 v ,并返回指向存储其邻接边的 **BufferBlock** 的指针。顶点定位的伪代码如算法3.1所示。要定位一个顶点 v , **Spruce** 将首先使用顶点 ID 的 4 个字节在哈希表中得到 **TopBlock** 的地址。然后它继续使用中间的 2 个字节,通过检查顶块内部的位向量来定位 **MiddleBlock**。**MiddleBlock** 和 **BottomBlock** 也将执行类似的过程,最终找到 **BufferBlock**。顶点插入操作将顶点添加到索引中。顶点插入的执行过程与顶点定位类似。不同的是,顶点在被插入时会根据顶点 ID 设置不同层级块的位向量中的位。如果块不存在, **Spruce** 会分配一个新的块,并在上层存储相应的指针。顶点删除操作删除索引中的顶点。首先,它会定位顶

点位置。然后，释放顶点的 *BufferBlock* 和 *SortedBlock*（如果存在），并将底层块中的子指针设置为 *NULL*。然后，进程将通过位向量检查 *BottomBlock* 是否为空。如果为空，则释放 *BottomBlock*，并清空 *MiddleBlock* 中的相应位和指针。最后，还会对 *TopBlock* 和哈希表进行类似的维护。获取顶点邻居操作会访问顶点的邻接边，并获取顶点所有邻接顶点的 ID。它首先会定位顶点位置，然后遍历 *BufferBlock* 和 *SortedBlock*，提取所有有效邻接顶点的 ID。

算法 3.1 Locate

Input: the root r of Spruce, a vertex identifier u
Output: the pointer to adjacency information of vertex u

```

1  $preidx = u \gg 32$ ;
2  $mididx = (u \& 0x0000ffff) \gg 16$ ;
3  $sufidx = u \& 0x000000ff$ ;
4 if  $!(topblock = r \rightarrow hashtable.get(u \gg 32))$  then
5   | return NULL;
6 end
7 if  $!(topblock \rightarrow bitvector.get(mididx))$  then
8   | return NULL;
9 else
10  |  $midblock = topblock \rightarrow pointers[mididx]$ ;
11 end
12 if  $!(midblock \rightarrow bitvector.get(sufidx))$  then
13   | return NULL;
14 else
15   |  $vertex\_pos = (midblock \rightarrow pointers[sufidx/64]) \rightarrow pointers[sufidx\%64]$ ;
16 end
17 return  $vertex\_pos$ ;
```

边操作：边操作包括边的插入（Insert）、更新（Update）和删除（Delete）。
Insert (v_1, v_2, w) 操作在 *BufferBlock* 中添加一条新的边。它会首先定位顶点 v_1 的 *BufferBlock*，然后通过位于 *BufferBlock* 中的指示符获取空位置。如果 *BufferBlock* 已满，则执行以下操作生成空位：如果 *BufferBlock* 的大小小于 C ，*BufferBlock* 将被扩展为原始大小的两倍。否则，*Spruce* 会将 *BufferBlock* 的边合并到 *SortedBlock* 中，然后清空 *BufferBlock*。最后，邻居标识符 v_2 和边属性 w 将被插入 *BufferBlock* 的数组中。对 *BufferBlock* 进行修改的伪代码如算法 3.2 所示。
Update (v_1, v_2, w) 操作更新一条边的属性。在找到顶点 v_1 的位置后，它会检查 *BufferBlock* 中的非空位置，以找到边 (v_1, v_2) 。如果边不在 *BufferBlock* 中，则会在 *SortedBlock* 中进行二分搜索以找到它。然后，边的相应属性 w 将被更新。
Delete (v_1, v_2) 从 *BufferBlock* 或 *SortedBlock* 中删除一条边。它使用与更新相同的方法来查找边。如果要删除的边在 *BufferBlock* 中，则将其位置对应的指示符位设置为 0；如果要删除的边在 *SortedBlock* 中，则将其存储位置的值标记为无效。

一个插入的运行示例如图 3.4 所示。要插入边 $(0, 3)$ ，*Spruce* 首先需要找到它的 *BufferBlock*。由于 *BufferBlock* 的父 *MiddleBlock* 不存在，*Spruce* 会分配一

算法 3.2 Insert Edge into Bufferblock**Input:** the pointer b to the bufferblock, the directed edge $e = (v_1, v_2, w)$ **Output:** the pointer b to the bufferblock

```

1 if !isfull( $b \rightarrow indicator, C$ ) then
2    $idx = rank_0(b \rightarrow indicator, 0)$ ;
3   if  $idx > b \rightarrow size$  then
4      $b = expand\_buffer(b)$ ; // double its size
5   end
6 else
7    $idx = 0$ ;
8   if !( $b \rightarrow sortedblock$ ) then
9      $b \rightarrow sortedblock = new\ sortedblock$ ;
10  end
11  sort( $b \rightarrow edges$ );
12  merge_move( $b \rightarrow edges, b \rightarrow sortedblock$ );
13  clear_bitvector( $b \rightarrow indicator$ );
14  set_value( $b \rightarrow edges[idx], e$ );
15  set_bit( $b \rightarrow indicator, idx$ );
16 return  $b$ ;

```

一个新的 MiddleBlock, 并将 TopBlock 的位向量中相应的位设置为 1 (图3.4(b))。然后, 它会分配一个新的 BufferBlock, 并将 MiddleBlock 位向量中的相应位设置为 1。最后, 直接向 BufferBlock 中插入 3 (图3.4(c))。要插入 $(2^{17}, 7)$, Spruce 会找到 2^{17} 的 BufferBlock (图3.4(d))。由于缓冲区已满, BufferBlock 中的边会合并到 SortedBlock 中以腾出空间。然后, Spruce 将数值 7 插入 BufferBlock (图3.4(e))。

3.2.4 复杂度分析及讨论

本小节对 Spruce 的时间和空间复杂度进行分析, 并讨论 Spruce 相较于其他动态图数据结构的优势。

基本操作的时间复杂度。表3.1中列出了 Spruce 和其它具有代表性的图存储结构上基本操作的时间复杂度。在顶点操作方面, Spruce 的时间复杂度仅次于使用单个哈希表来索引所有顶点的 Stinger。对于边操作 (首先需要找到边的对应顶点), Spruce 在边的更新和删除方面优于其他方法。在插入方面, 将边插入 BufferBlock 和将 BufferBlock 合并到 SortedBlock 的成本分别为 $O(1)$ 和 $O(d)$ (d 是顶点的度数)。由于现实世界中的图是稀疏的^[97-98], 因此对于大多数顶点来说, 合并操作不会频繁触发, Spruce 整体仍具有较好的性能。

空间性能: Spruce 使用类似 vEB 树的索引, 并通过共享相同前缀的块来表示索引, 与原始的 vEB 树相比减少了层次并节省了空间。与其他图索引相比, 该索引的最大区别在于, 在 Spruce 中, 顶点 ID 被分离并存储在不同的层级中并共享前缀以减少内存消耗, 而其他索引则需要存储整个顶点 ID。此外, 紧凑的数据结构 (Bitvector 和 SortedBlock) 有助于进一步降低内存消耗。由于减少了内存访问次数, Spruce 在节省空间的同时也使大多数操作的性能得到了提高。

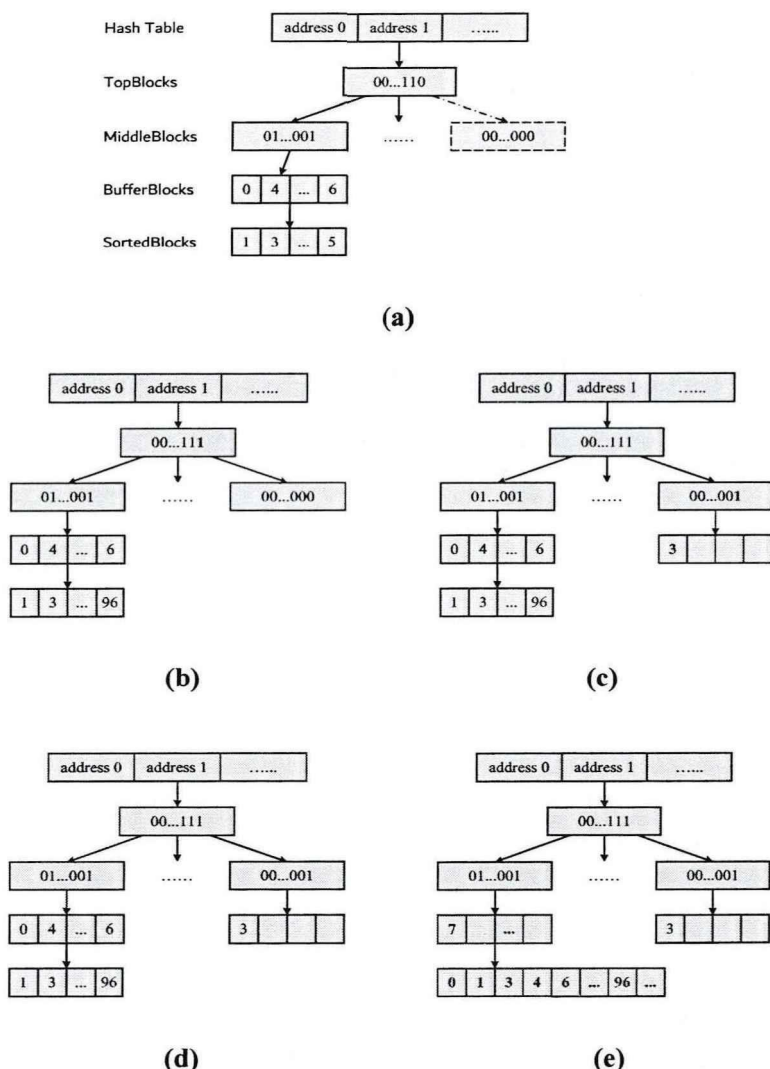


图 3.4 Spruce 的运行示例: (a) 初始状态, (b) 插入边 (0,3), (d) 插入边 (2^{17} ,7)。一个顶点的 ID 被分成三部分来定位其 BufferBlock。这里 0,0,0 被用于表示顶点 0; 0,2,0 表示顶点 2^{17} 。

为了方便进一步对 Spruce 的空间开销进行分析,我们定义间隙 (gap) 为有序集合中两个连续整数之差。设图 $G = (V, E)$, $n_v = |V|$, $n_e = |E|$, d_i 是连接到顶点 i 的边的数量。假设 $U = \{0, 1, 2, \dots, u-1\}$ 是数域中的整数集合, G, B 分别是 V 中连续顶点 ID 之间的平均间隙和块中位向量的大小。对于顶点 ID 集合 $V = \{v_1, v_2, \dots\}$, 其中 $v_i < v_{i+1}$, $G = (\sum_{i=1}^{n_v-1} (v_{i+1} - v_i)) / n_v = (v_{n_v} - v_1) / n_v$ 。因此, Gn_v 表示在 $[v_1, v_{n_v}]$ 范围内存储所有顶点 ID 所需的比特数。在 MiddleBlock 层, 我们以位向量块为单位存储比特, 因此块的数量不应超过 n_v 。因此, 考虑到块的大小, MiddleBlock 层占用的总空间为 $O(\min(B \lceil Gn_v / B \rceil, Bn_v)) = O(\min(Gn_v, Bn_v))$ 。由于上层的大小按平方根缩减, 总空间复杂度的上限为 $O(\min(\sum_{i=0}^{\lg \lg(u)-1} (Gn_v)^{2^{-i}}, Bn_v \lg \lg(u)))$ 。忽略低阶项, 即为 $O(\min(Gn_v, Bn_v \lg \lg(u)))$ 。对于图中大量 8 字节顶点 ID 按顺序编号 ($G=1 \sim 2$) 的常见情况, Gn_v 远远小于

表 3.1 不同动态图存储结构上基本操作的时间复杂度。 n 是顶点的数量, u 是数域的大小, d 是顶点的平均度数。

操作	Stinger	Teseo	Sortledton	Spruce
locate_vertex	$O(1)$	$O(\lg(u))$	$O(\lg(n))$	$O(\lg \lg(u))$
insert_vertex	$O(1)$	$O(\lg(u))$	$O(\lg(n))$	$O(\lg \lg(u))$
delete_vertex	$O(d)$	$O(\lg(u) + d)$	$O(\lg(n) + d)$	$O(\lg \lg(u) + d)$
insert_edge	$O(d)$	$O(\lg(u) + \lg(d))$	$O(\lg(n) + \lg(d))$	$O(\lg \lg(u) + d)$
delete_edge	$O(d)$	$O(\lg(u) + \lg(d))$	$O(\lg(n) + \lg(d))$	$O(\lg \lg(u) + \lg(d))$
update_edge	$O(d)$	$O(\lg(u) + \lg(d))$	$O(\lg(n) + \lg(d))$	$O(\lg \lg(u) + \lg(d))$
get_neighbours	$O(d)$	$O(\lg(u) + d)$	$O(\lg(n) + d)$	$O(\lg \lg(u) + d)$

$Bn_v \lg \lg(u)$, 因此 Spruce 的空间复杂度约为 $O(Gn_v)$, 在实际应用中的空间占用较低。根据 Spruce 索引, 当 $G=1\sim 2$ 时, MiddleBlock 级的位向量中只有 1~2 位用于表示顶点 ID, 并且 TopBlock 的位向量中的一位可被 $2^{15}\sim 2^{16}$ 个顶点 ID 共享, 哈希表的一个槽最多被 $2^{31}\sim 2^{32}$ 个顶点 ID 共享。最坏情况是每两个顶点 ID 之间的差距大于 $u^{1/2}$, 此时每个块 (不包括哈希表) 只有一个元素。不过, 这种情况在实践中并不多见, 可以通过重新标记来解决^[50]。对于边的存储, BufferBlock 和 SortedBlock 会根据存储的边的数量动态调整大小, 因此顶点 i 使用 $O(d_i)$ 空间来存储边。由于 $n_e = \sum_{i=1}^{n_v} d_i$, 易知在 Spruce 中存储图 G 中的边的总空间为 $O(n_e)$ 。

访存效率: 与其他方法相比, Spruce 在获取邻接边时访问的块更少。例如, 要找到一条有向边并将其删除, Spruce 只需访问源顶点的 BufferBlock 和 SortedBlock。Stinger 和 GraphOne 需要遍历链表中的 $O(d/B)$ (d 和 B 分别指顶点的度和块的大小) 个块, 而 Teseo 和 Sortledton 需要访问 $O(\lg(d/B))$ 个块。因此, BufferBlock 和 SortedBlock 的紧凑存储方案在访问内存时提供了更好的数据局部性, 从而进一步提高了性能。

3.3 并发策略设计

并发控制方法是影响图存储系统性能的另一个重要因素。一个理想的图系统的并发协议应该具有良好的可缩放性和空间效率, 同时又能很好地满足更新与计算并发运行的要求。为此, 本文利用 vEB 树不拆分和合并节点的特点, 提出了一种结合 ROWEX 和乐观锁的轻量级并发控制协议, 并进一步引入 MVCC 来支持事务并发操作。

3.3.1 基本思想

Spruce 的核心结构由两个主要部分组成: 顶点索引和边存储结构。本文为这两个组件分别设计了并发控制机制。对于顶点索引, 正如在 3.2 一节中所讨论

的, Spruce 索引中无需合并、拆分或调整块的大小。与索引相关的操作只有块的申请和释放。在此基础上, 本文引入了 *Improved ROWEX*, 它是 ROWEX 的改进版, 在写入过程中只使用互斥锁和原子操作, 从而消除了对 RCU 技术的需求。改进的 ROWEX 的基本原理很简单: 当一个块需要修改时, 写入程序只获取该块的锁 (而不是获取该程序块及其父程序块的锁), 并通过原子操作执行修改操作。在此过程中, 还会进行额外的验证以检查目标数据在获取锁后是否被其他进程修改过, 确保并发的正确性。如果验证失败, 写者就会释放锁, 并重新开始对该数据块的操作。在本文设计的数据结构中, 这种验证不需要时间戳。

对于边存储结构, 需要确保获取邻居操作的正确性, 这一操作与范围扫描类似。为此, 本文采用了乐观锁策略。然而, 传统的乐观锁具有一定的局限性: 当读者访问频繁修改的数据时, 可能会需要多次重启操作。为了解决这个问题, 本文为锁设计了升级策略。具体来说, 如果读者在获取数据时重启次数超过了预定的阈值, 读者就会将其乐观锁升级为互斥锁。进一步地, 本文还整合了版本日志, 使在 Spruce 中的执行图事务成为可能。

3.3.2 并发控制协议

在 TopBlock 和 MiddleBlock 中, 位向量中的每 64 位及其相应的指针共享一个互斥锁。在底块中, 子指针数组与底块指针使用相同的锁, 每个顶点的 BufferBlock 和 SortedBlock 都有一个乐观锁, 其中包括一个独占锁和一个时间戳。

顶点索引的并发协议: 本部分首先讨论写入协议。在 Spruce 中, 只有插入和删除可以修改索引结构。鉴于修改 TopBlock 和 MiddleBlock 的协议类似, 不妨以 MiddleBlock 为例。如算法3.3中的伪代码所示, 插入过程中对 MiddleBlock 的修改过程分为 4 步:

1. 根据顶点 ID 获取 MiddleBlock 中对应的锁。
2. 检查 MiddleBlock 是否过时。如果 MiddleBlock 无效 (在被锁定之前已经被移除), 则从上层重新开始插入; 否则, 继续这个过程。
3. 重新读取位向量, 检查要插入下层的 BufferBlock 是否已经存在。如果 BufferBlock 不存在, 则分配并插入一个新的 BufferBlock, 并修改相关的指针和位向量。如果不存在, 则分配并插入新块, 并修改相关的指针和位向量。在此过程中, BottomBlock 中的指针数组可能需要分配或修改。如果 BufferBlock 已经存在, 则直接获取指向它的指针。
4. 释放锁, 继续在 BufferBlock 中的插入过程。

删除过程中对 MiddleBlock 的修改是在删除顶点时触发的, 其执行方式与插入有所不同。它的基本流程如下:

算法 3.3 Concurrently Insert Value into MiddleBlock

Input: the pointer *midblock* to the middleblock, the value *sufix* (last 2 bytes of vertex id) to be inserted.

Output: the pointer *bufblock* to the value's corresponding bufferblock.

```

1 acquire(midblock → locks[sufix/64]);
2 if midblock → obsolete_flag then
3   | release(midblock → locks[sufix/64]);
4   | return NULL; // restart from upper level
5 end
6 if !(midblock → bitvector.get(sufix)) then
7   | if !(midblock → pointers(sufix/64)) then
8     |   botblock = new bottomblock;
9     |   midblock → pointers[sufix/64] = botblock;
10  | else
11  |   botblock = midblock → pointers[sufix/64];
12  |   bufblock = new bufblock;
13  |   botblock → pointers[sufix%64] = bufblock;
14  |   midblock → bitvector.set(sufix);
15 else
16   | bufblock = (midblock → pointers[sufix/64]) → pointers[sufix%64];
17 release(midblock → locks[sufix/64]);
18 return bufblock;

```

1. 根据顶点 ID 获取 MiddleBlock 中对应的锁。
2. 修改位向量和指针，指示顶点已经被删除。如果底部图块中的指针数组为空，那么底部图块就会被标记为过时 (obsolete)。
3. 释放锁。
4. 检查块中的整个位向量。如果所有位都是 0，则获取块中的所有锁并转到步骤 5。否则，删除将结束。整个块中的锁需要按顺序获取以避免死锁。
5. 重新检查块中的过时标记和整个位向量。如果数据块已经被标记为过时，那么删除操作将直接完成。如果位向量中的所有位都为 0，则清除该块并标记为过时，然后继续在树中上一层级的维护过程。否则，删除完成，因为位向量显示在步骤 3 和 5 之间插入了新数据。

对读者来说，它们只需忽略锁并原子读取数据。在读取的过程中它们还需要对过时标记进行检查以确保正确性。

边存储块的并发控制：要修改 BufferBlock 或 SortedBlock，写者首先要在修改数据前获得互斥锁，然后在不检查时间戳的情况下修改数据。在此过程中，时间戳会被修改两次：一次在数据修改前，另一次在数据修改后。对于读者，它们每位都持有一个重新启动计数器。访问顶点的 BufferBlock 和 SortedBlock 时，读者会检查读取前后的时间戳。如果时间戳没有变化，则读取成功。否则，读者会重新开始读取这些块，并将重启计数器加一。如果重启计数超过了预定义的阈值（即读者多次重启），读者将独占互斥锁进行读取。

3.3.3 改进的 ROWEX 的正确性证明

本小节将证明为 Spruce 设计的改进的 ROWEX 的正确性。这里以 MiddleBlock 为例, BottomBlock 和 TopBlock 的 ROWEX 证明类似。假设两个线程 T_1 和 T_2 同时访问同一个 MiddleBlock。如果它们都执行定位操作 (Locate), 不会发生冲突, 结果显然也是正确的。如果它们同时执行插入 (Insert) 操作, 并且需要设置 MiddleBlock 的位向量和相应指针的值, 则其正确性可以由锁保证, 并且 Insert 步骤 3 中的重新检查过程可以确保前者插入的值不会被后者覆盖。

如果 T_1 和 T_2 同时对 MiddleBlock 执行删除操作 (Delete), 情况与 Insert 类似。唯一不同的是, 删除位向量和指针数组中的值后, 线程需要在 Delete 的步骤 3 中判断块是否为空, 是否应该删除。如果数据块为空, 线程将获取该数据块中的所有锁, 并删除整个数据块。如果 T_1 和 T_2 都检测到块是空的, 那么后来获得锁的线程会发现过时标志为真, 并终止删除的过程 (因为这表明块已经被删除了)。

然后考虑 T_1 执行 Delete, T_2 执行 Insert。如果目标删除值不是 MiddleBlock 中的唯一值, 情况与上述类似; 否则, 可能出现以下情况:

- **情况 1: Delete 在 Insert 之前完成。**在这种情况下, MiddleBlock 在 Insert 之前被标记为过时。 T_2 会发现过时标记为真, 并从上层重新启动 Insert 操作以获取新的 MiddleBlock 进行插入。
- **情况 2: Insert 在 Delete 之前完成。**在 T_2 插入值后, T_1 会在删除的步骤 4 或步骤 5 发现位向量不为空, 并在不清除整个 MiddleBlock 的情况下完成 Delete。

当 T_1 执行 Locate, T_2 执行 Insert/Delete 时, 情况会更加复杂。Insert 和 Delete 的情况类似。由于篇幅限制, 本文只讨论在 MiddleBlock 上同时执行 Locate 和 Delete 的情况。Delete 首先将位向量中的相应位设置为 0 (然后修改指针数组中的指针), 而 Locate 则依次读取位向量和指针。如果 T_2 在 T_1 读取之前删除了整个 MiddleBlock, 那么 T_1 将发现该块已被删除, 并从上层重新开始读取以获得最新结果。对于其它情况, 假设 T_1 尝试读取 i , 而 T_2 尝试删除 j 。如果 $i \neq j$, 则正确性显而易见 (因为操作是以原子方式完成的)。否则, T_1 和 T_2 可能的操作顺序如下:

- **情况 1: T_2 在 T_1 读取之前修改了位向量。**在 T_1 访问该值之前, T_2 已经删除了该值, 因此 T_1 将返回 false, 表示该值不存在于索引中。
- **情况 2: T_2 在 T_1 读取之后修改了位向量。**在这种情况下, T_1 可能会得到一个空指针, 或者一个指向过时的 BottomBlock 的指针, 这将使 T_1 重新开始读取 MiddleBlock, 以得到最新的正确结果。

综上所述,改进的 ROWEX 的正确性得以证明。

3.3.4 版本控制

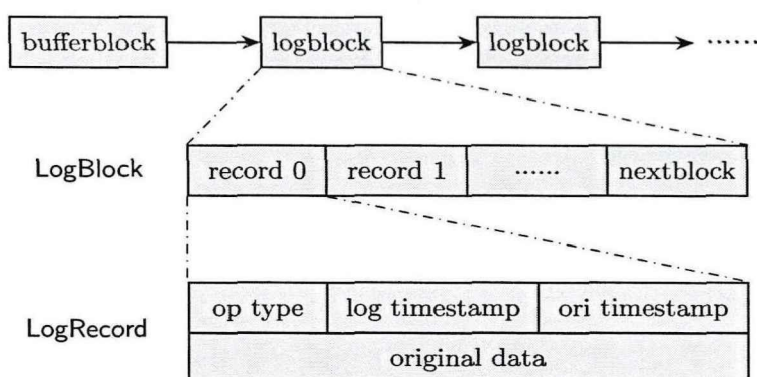


图 3.5 Spruce 的版本控制设计

许多实时图形分析都需要使用事务来访问图的一致快照。Spruce 通过日志块 (LogBlock) 支持多版本并发控制,以进一步支持图中事务级别的并发操作。Spruce 的目标之一是节省空间。因此,本文设计将一个顶点的所有历史版本的边收集到一个日志链表结构中,而不是为每条边维护一个版本链表。如图3.5所示,每个顶点都在其 BufferBlock 中关联了一个指针,该指针指向存储版本的日志块链接链表。日志块中的每条版本记录都由 4 个字段组成: (1) `op_type`, 用于指示对数据的操作类型 (如边的删除/更新); (2) `log_timestamp`, 表示提交此操作的时间; (3) `ori_timestamp`, 表示此记录之前的最后一次修改时间; (4) `original_data`, 存储与 `ori_timestamp` 相对应的版本边/顶点数据。在 Spruce 中,边或顶点的初始插入不会记录在日志块中。为了节省空间,本文选择将它们连同时间戳一起直接存储在 BufferBlock 和 SortedBlock 中。

3.3.5 基本图算法的并行化支持

在实现并发控制的基础上,本文进一步在 Spruce 上实现了基本图算法的并行化支持。Spruce 上支持的图算法有广度优先搜索 (Breadth-First Search, BFS), PageRank, 单源最短路径 (Single-Source Shortest Paths, SSSP), 聚类系数 (Clustering Coefficient, CC), 弱连通分量 (Weakly Connected Components) 等。

广度优先搜索 (Breadth-First Search, BFS): 本节基于双重队列的方法实现了 BFS 的并行执行,如算法3.4和算法3.5所示。算法通过交替使用两个队列, `c_frontier` 代表当前层的顶点,而 `n_frontier` 存储下一层将要访问的顶点。每次迭代中,算法并行扩展 `c_frontier` 中的所有顶点,对它们的未访问邻居进行访问,并将这些邻居加入 `n_frontier`。处理完成后, `c_frontier` 被更新为 `n_frontier`,而 `n_frontier` 被清空,准备下一轮的迭代。当在迭代的起始阶段 `c_frontier` 中没有顶

算法 3.4 Parallel Top-Down Breadth-First Search

Input: undirected graph $G = (V, E)$, initialized visited array vs , set of nodes to be visited in the current level $c_frontier$, set of nodes to be visited in the next level $n_frontier$, source vertex src , parent map $parent$

Output: parent map $parent$

```

1  $vs[src] \leftarrow 1$ ;
2  $parent[src] \leftarrow src$ ;
3  $c\_frontier \leftarrow \{src\}$ ;
4 while  $c\_frontier$  is not empty do
5   |  $TDStep(G, vs, c\_frontier, n\_frontier, parent)$ ;
6 end
7 return  $parent$ ;

```

算法 3.5 TDStep

Input: $G, vs, c_frontier, n_frontier, parent$

```

1 parallel for  $v \in c\_frontier$  do
2   | for  $u \in v.neighbours$  do
3     | if  $!vs[u]$  then
4       |    $parent[u] \leftarrow v$ ;
5       |    $vs[u] \leftarrow 1$ ;
6       |    $n\_frontier \leftarrow n\_frontier \cup \{u\}$ ;
7     | end
8   | end
9 endfor
10  $c\_frontier \leftarrow n\_frontier$ ;
11  $n\_frontier \leftarrow \emptyset$ ;

```

点时，表示自源顶点起的所有可达顶点均已被访问，算法结束。需要指出的是，与第四章不同，在本章实现的是一般的自上而下的 BFS 算法，未针对动态图进行诸如负载均衡和内存访问等特殊的优化。

PageRank: 本节实现的 PageRank 在传统的 PageRank 算法上基于顶点进行顶点并行化，并加入了最大迭代轮数限制。首先算法为图中的每个顶点分配初始 PageRank 值，并计算每个顶点对其所有邻居的贡献。本文通过并行 for 循环，在每次迭代中计算每个顶点的新 PageRank 值（包括计算来自每个顶点的所有入边的 PageRank 贡献，更新该顶点的 PageRank 值，并计算总误差以检查是否达到收敛条件）。如果在任何迭代步骤中，总误差小于给定的阈值 ϵ ，算法将停止迭代并返回最终的 PageRank 结果。否则，它将继续执行，直到达到最大迭代次数 max_iters 。

单源最短路径 (Single-Source Shortest Paths, SSSP): 本节实现了基于 Delta-Stepping 的 SSSP 算法，其大致伪代码如算法 3.7 所示。在一开始，所有顶点到源顶点的距离初始化为无穷大，源顶点到自身的距离为 0。顶点被存储在基于距离划分的桶中，每个桶存储一定距离范围 (Delta) 内的顶点。每个线程处理并处理当前全局桶中的顶点，尝试通过该顶点放松 (Relax) 相邻顶点的距离，同时维护自己的本地桶来收集即将被放松的顶点。处理完成后，不同线程间进行同步，

算法 3.6 Parallel PageRank Algorithm

Input: Graph G , maximum number of iterations max_iters , convergence threshold $epsilon$

Output: PageRank scores for all nodes in G

```

1  $init\_score \leftarrow 1.0/g.num\_nodes()$ 
2  $base\_score \leftarrow (1.0 - kDamp)/g.num\_nodes()$ 
3  $scores \leftarrow$  vector of  $G.num\_nodes()$  size, initialized to  $init\_score$ 
4  $outgoing\_contrib \leftarrow$  vector of  $G.num\_nodes()$  size
5 parallel for each node  $n$  in  $G$  do
6    $outgoing\_contrib[n] \leftarrow init\_score/g.out\_degree(n)$ 
7 endfor
8 for  $iter \leftarrow 0$  to  $max\_iters$  do
9    $error \leftarrow 0$ 
10  parallel for each node  $u$  in  $G$  do
11     $incoming\_total \leftarrow 0$ 
12    for each node  $v$  in  $G.in\_neigh(u)$  do
13       $incoming\_total \leftarrow incoming\_total + outgoing\_contrib[v]$ 
14    end
15     $old\_score \leftarrow scores[u]$ 
16     $scores[u] \leftarrow base\_score + kDamp \times incoming\_total$ 
17     $error \leftarrow error + |scores[u] - old\_score|$ 
18     $outgoing\_contrib[u] \leftarrow scores[u]/g.out\_degree(u)$ 
19  endfor
20  if  $error < epsilon$  then
21    break
22  end
23 end
24 return  $scores$ 

```

合并本地桶到全局桶中。算法重复上述过程直到所有桶都被处理且没有新的顶点可以被放松为止。

聚类系数 (Clustering Coefficient, CC): 本文实现借助三角计数的 CC 计算, 其核心算法代码如算法 3.8 所示。算法开始时, 首先初始化一个用于统计三角形总数的变量 $total$ 为 0。然后该算法并行遍历图中的每个节点 u , 对于 u 的每个邻居 v , 再遍历 u 的邻居 w , 通过检查每个可能的三元组 (u, v, w) 来统计三角形数量。为了确保每个三角形只被计数一次, 算法使用有序性进行约束: 只有当 $u > v > w$ 时才计数。如果发现 $v > u$, 则提前终止内层循环。完成所有节点的遍历后, 算法返回在图中找到的三角形的总数。在 Spruce 上, 由于 BufferBlock 的无序性, 在访问邻居时需要额外对顶点所有的邻居进行一次排序来确保算法的正确性。

弱连通分量 (Weakly Connected Components): 本文实现了使用了 Shiloach-Vishkin 算法进行优化的 WCC 算法, 其伪代码如算法 3.9 所示。首先每个节点被赋予一个与其索引相同的连通分量 ID。在每次迭代中, 算法对图中的每个顶点并行地都会检查其邻居。如果当前顶点和邻居的连通分量 ID 不同, 则会进行“钩连”操作, 即将较大的连通分量 ID 更新为两者中较小的那个。在每次迭代的最

算法 3.7 Parallel Δ -Stepping SSSP Algorithm

Input: A weighted graph $G = (V, E)$, a source vertex *source*, and a step size δ
Output: Shortest distances *dist* from the source to all other vertices

```

1 dist[v]  $\leftarrow \infty$  for all  $v \in V$  except dist[source]  $\leftarrow 0$ 
2 frontier  $\leftarrow$  initialize a sufficiently large array
3 frontier[0]  $\leftarrow$  source
4 shared_indexes  $\leftarrow$  [0, MAX_BIN]
5 frontier_tails  $\leftarrow$  [1, 0]
6 while there are vertices to be processed do
7   parallel for  $i = 0$  to curr_frontier_tail do
8      $u \leftarrow$  frontier[ $i$ ]
9     if dist[ $u$ ]  $\geq \delta \times$  curr_bin_index then
10      RelaxEdges( $g, u, \delta, dist, local\_bins$ )
11    end
12  endfor
13  for each local bin do
14    if bin is not empty and bin size is below threshold then
15      Move vertices from local bin to shared bin
16    end
17  end
18  Update shared indexes and bounds for the next iteration
19 end
20 return comp return dist

```

算法 3.8 Triangle Counting Algorithm

Input: An undirected graph $G = (V, E)$ with sorted neighborhoods
Output: The total number of triangles in G

```

1 total  $\leftarrow 0$ 
2 parallel for  $u = 0$  to  $|V| - 1$  do
3   for  $v \in out\_neigh(u)$  do
4     if  $v > u$  then
5       break
6     end
7      $it \leftarrow begin(out\_neigh(v))$ 
8     for  $w \in out\_neigh(u)$  do
9       if  $w > v$  then
10        break
11      end
12      while  $*it < w$  do
13         $it \leftarrow it + 1$ 
14      end
15      if  $w == *it$  then
16         $total \leftarrow total + 1$ 
17      end
18    end
19  end
20 endfor
21 return total

```

算法 3.9 Shiloach-Vishkin Algorithm for Weakly Connected Components

Input: An undirected graph $G = (V, E)$
Output: Array *comp* indicating the connected component ID for each vertex

```

1 parallel for each vertex  $v \in V$  do
2   |  $comp[v] \leftarrow v$ 
3 endfor
4  $change \leftarrow true$ 
5  $num\_iter \leftarrow 0$ 
6 while  $change$  do
7   |  $change \leftarrow false$ 
8   |  $num\_iter \leftarrow num\_iter + 1$ 
9   | parallel for each vertex  $u \in V$  do
10    |   foreach neighbor  $v$  of  $u$  do
11    |       |  $comp_u \leftarrow comp[u]$ 
12    |       |  $comp_v \leftarrow comp[v]$ 
13    |       | if  $comp_u \neq comp_v$  then
14    |       |   |  $high\_comp \leftarrow \max(comp_u, comp_v)$ 
15    |       |   |  $low\_comp \leftarrow \min(comp_u, comp_v)$ 
16    |       |   | if  $high\_comp == comp[high\_comp]$  then
17    |       |   |   |  $change \leftarrow true$ 
18    |       |   |   |  $comp[high\_comp] \leftarrow low\_comp$ 
19    |       |   end
20    |       end
21    |   end
22   | endfor
23   | parallel for each vertex  $n \in V$  do
24   |   | while  $comp[n] \neq comp[comp[n]]$  do
25   |   |   |  $comp[n] \leftarrow comp[comp[n]]$ 
26   |   | end
27   | endfor
28 end

```

后,通过一个额外的并行循环来“压缩”连通分量路径,即如果一个节点的连通分量 ID 不直接等于其父节点的 ID,那么就更新它来加速后续的查找和更新操作。当一次迭代中没有任何连通分量 ID 的更新,则所有可能的连通节点都已经归入正确的连通分量中,算法结束。

3.4 实验结果与分析

本节在一台配备了英特尔 Xeon Gold 5120 @ 2.20GHz 处理器和 500GB DDR4 内存的双插槽机器上进行实验。每个处理器有 19.25 MB 三级缓存,14 个内核,最多支持 28 个线程。所有代码均使用 GCC 11.3.0 和 O3 参数编译。

本节使用了多种图来评估时间和空间性能,如表 3.2 所示。*Com-Livejournal* 是 LiveJournal 社交网络的无向图^[30]。*DOTA-League* 是一个加权图,代表了许多游戏实体之间的关系^[101]。*Orkut* 是来自 Orkut 社交网络的社区关系图。*Yahoo-Songs* 是一个基于用户和歌曲二分评级网络,其中包含 100 多万用户对超过 2.5

表 3.2 实验中使用的图数据集。**K**”、**”M**”、**”B**” 分别代表**”千”**、**”百万”**、**”十亿”**。前**0.1%** 度数”表示这个度数超过了其他所有顶点度数的**99.9%**。

图数据集	V	E	平均度数	最高度数	前 0.1% 度数	平均 Gap	最大 Gap	大小 (GB)
com-livejournal	4M	34M	17.35	14815	473	1.010	86	0.47
dota-league	61K	51M	1663.24	17004	12988	5.194	720	0.76
orkut	3M	117M	76.28	33313	1174	1.000	12	1.6
yahoo-songs	1.6M	256M	513.04	468425	23751	1.000	1	3.0
graph500-24	9M	260M	58.70	406416	7019	1.891	18	4.1
uniform-24	9M	260M	58.70	103	84	1.500	2	4.1
graph500-26	33M	1B	64.13	1003338	5340	2.046	21	17.5
uniform-26	33M	1B	64.13	111	90	1.500	2	17.5
com-friendster	65M	2B	55.06	5214	1612	1.903	1874	30.1

亿首歌曲的评级^[102]。 *Graph500-X* 和 *Uniform-X* 数据集分别是顶点度数符合幂律和均匀分布的图。 *Com-friendster* 是 Friendster 社交网络的无向图^[19]。

本文使用的 Benchmark 是 GFE (Graph Framework Evaluation) Driver, 一个用 C++ 进行编写的用于评估动态结构图的更新和分析性能的工具。对于图算法, 本文使用了来自 LDBC Graphalytics^[103] 的执行框架, 这是一个工业级基准, 其中包括多种涵盖现实世界工作负载的框架, 并支持广度优先搜索 (BFS)、单源最短路径 (SSSP)、PageRank (PR)、聚类系数 (CC) 等图算法。

在竞争对手方面, 本节选择最近具有代表性的动态图数据存储结构进行比较: *Stinger*^[43]、*GraphOne*^[46]、*LiveGraph*^[48]、*Teseo*^[50] 和 *Sortledton*^[34]。 *LiveGraph*、*Teseo* 和 *Sortledton* 都是支持事务的图系统, 而 *Stinger* 和 *GraphOne* 只提供并行化操作支持。

3.4.1 基本操作时间性能

这一小节中评估多线程动态操作 (包括插入和删除) 在各种图上的吞吐量 (通量), 主要的测量单位为每秒百万条边 (millions of edges per second, meps)。

1. 插入性能

本实验从图数据集中随机插入带有权重 (如果原始数据不包含权重, 则随机生成) 的所有边 (以及顶点) 到初始为空的结构中, 并统计平均吞吐量。图 3.6(a) 显示了具有特定分布的图的插入吞吐量。对于幂律图, *Spruce* 在所有方法中表现最佳。对于均匀分布图, 虽然 *Stinger* 在 *uniform-24* 上实现了最高吞吐量, 但它无法在限定时间内完成 *uniform-26* 和 *graph500-26* 中所有边的插入。这是因为 *Stinger* 会执行线性搜索来检查边是否存在, 从而导致超大图的运行时间无法承受。 *Spruce* 在 *uniform-26* 和 *uniform-24* 上分别获得了第一名和第二名。需要指出的是, 对事务的支持确实会带来性能开销, 这也是 *Spruce* 和 *Sortledton* 等支持事务的图存储结构在某些数据集 (如 *uniform-24*) 上的吞吐量低于非事务图存储结构 (*Stinger*) 的重要原因。

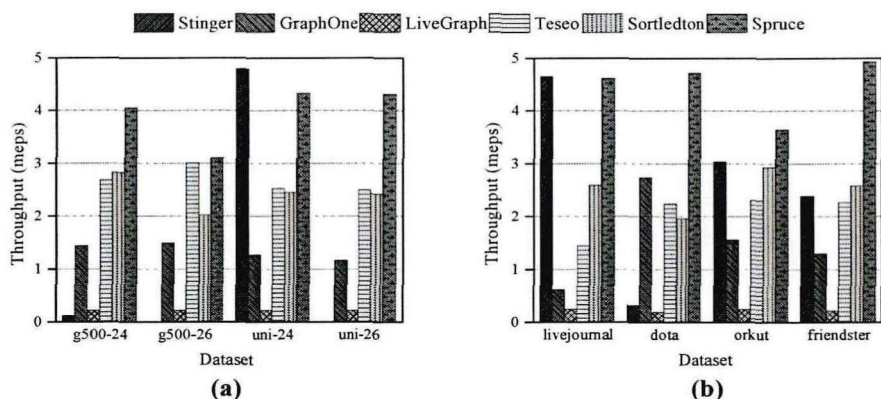


图 3.6 在 (a) 特定分布图和 (b) 真实世界图上随机插入的性能。

与具有特定分布的图数据不同，真实世界的图呈现的数据分布形式相对复杂一些。因此，本文也测试了不同工作在真实世界图上的操作性能。图3.6(b)显示了它们在四个不同真实世界图上进行插入操作的吞吐量。结果表明，**Spruce**对现实世界图的适应性更好，其插入速度比 **GraphOne** 快 7.5 倍，比 **Teseo** 快 3.2 倍，比 **Sortedton** 快 2.4 倍。**LiveGraph** 每秒仅插入不超过 100 万条边，这主要是因为其复杂的事务设计。

与 **Teseo** 的 ART 和 **Sortedton** 的多级向量等先进的工作相比，**Spruce** 的索引不涉及节点的合并、拆分或大小调整，具有更好的时间复杂性，从而提高了顶点索引的动态性能。**BufferBlock** 的设计以及 **SortedBlock** 的紧凑存储方案还为边的修改提供了可编辑性和良好的数据局部性，使之适用于现实世界中大多数顶点都具有较低度数的图（如表3.2所示）。

2. 删除性能

删除实验从已经构建好的图数据结构开始，以随机顺序删除图中的所有边。图3.7(a)显示了所有方法在特定分布图上的删除吞吐量。对于幂律图，**Spruce** 比 **Teseo** 快 1.7 倍，比 **Sortedton** 快 1.9 倍。对于均匀图，**Teseo** 和 **Sortedton** 在 *uniform-24* 上比 **Spruce** 稍快，而在 *uniform-26* 上比 **Spruce** 慢。**Stinger** 不支持边的版本控制。由于在小型均匀图上，线性搜索找到需要删除的边所需的时间和维护哈希索引所需的时间都相对较短。因此，它在 *uniform-24* 上的表现优于所有其他竞争对手。

图3.7(b)显示了不同图存储方法在现实世界图上的删除吞吐量。**Spruce** 的吞吐量超过了大多数竞争对手，达到 4.0 ~ 4.6 meps，是 **Stinger** 的 0.9 ~ 15.8 倍和 **Sortedton** 的 1.4 ~ 1.5 倍。其他竞争对手的吞吐量相对较低，不到 3 meps。

需要指出，在删除实验中，大部分修改操作都集中在顶点的邻接边上，高效的顶点索引有助于快速访问这些顶点，提高删除的效率。由于缺乏有效的索引，**GraphOne** 和 **LiveGraph** 在该实验中表现不佳。**Stinger** 在不同数据集上的表现差

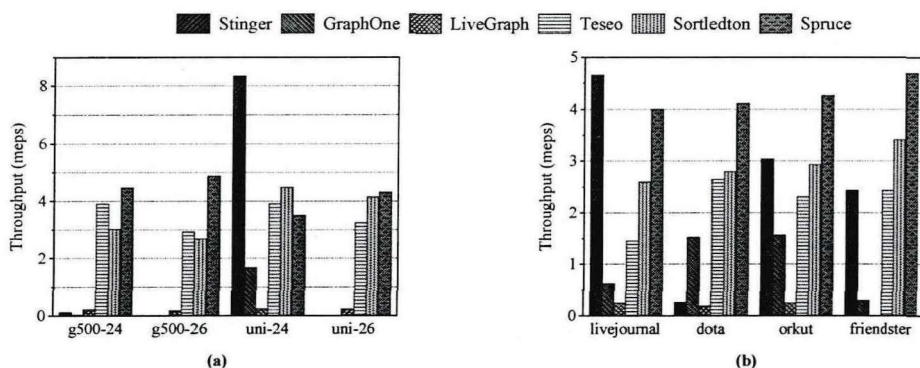


图 3.7 在 (a) 特定分布图和 (b) 真实世界图上随机删除的性能。

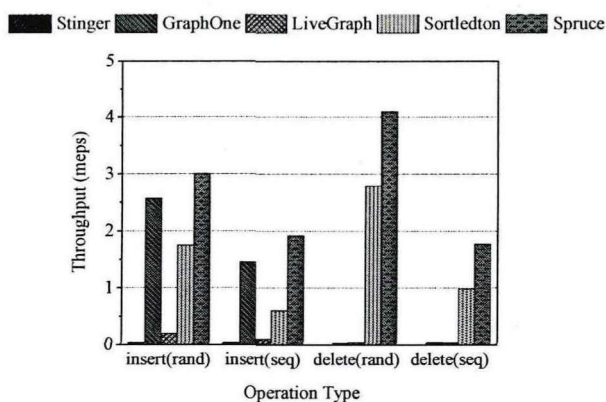


图 3.8 顺序操作与随机操作性能的比较

异很大，因为它没有对邻接边进行排序，而且在链表中进行线性搜索的时间成本主要取决于顶点的度数。由于高性能的索引和紧凑的排序边存储方案，Spruce 在删除操作上的性能优于其他方法。

3. 顺序操作性能

现实世界中的许多动态图都呈现出一种顺序访问模式。即在一定时间内对边的修改与同一组顶点有关，并表现出很强的时间局部性。例如，某个网站（可视为顶点）报道了一则轰动性新闻，并在短时间内被大量引用（可视为边）。在本实验中，本文对 *Yahoo-Songs* 的顺序操作性能进行了评估，它呈现出很强的顺序访问模式（顶点和边按照顶点 ID 的顺序插入/删除）。在实验中边是被连续不断地插入/删除的，本文统计的是整个过程的平均吞吐量。

图 9 以柱状图的形式显示了评估结果。GraphOne、LiveGraph、Sortledton 和 Spruce 均能在规定时间内完成所有操作。结果表明，在执行顺序操作时，Spruce 在所有方法中排名第一。在顺序插入方面，Spruce 达到每秒 3.0 meps，比 GraphOne 快 1.9 倍。在顺序删除方面，Spruce 达到每秒 1.8 meps，比 Sortledton 快 1.8 倍。与随机操作相比，所有的方法在顺序执行时都会出现不同程度的性能下降。这主要是由于资源访问竞争造成的，尤其是对于使用以顶点为中心的锁策略（为一个顶点分配一个锁），如 Livegraph 和 Sortledton。GraphOne 使用边表和批处理的

方法进行插入,在顺序插入时减少了冲突,因此顺序插入性能较好。尽管 Spruce 也采用的是以顶点为中心的锁策略,但缓冲区设计一定程度上弥补了这种设计的缺陷。再加之以上高效的顶点索引, Spruce 最终在众多方法中胜出。

3.4.2 图存储的空间性能

本小节测试不同方法下用于存储动态图的实际内存开销。在实验中本文使用 Ubuntu 系统下 `/proc/[pid]/statm` 中的 Resident Set Size (RSS)^[104] 统计图系统的内存占用情况。下面,本节首先分析不同方法建立完整图的内存消耗,然后测试在图的建立过程中的内存变化情况。

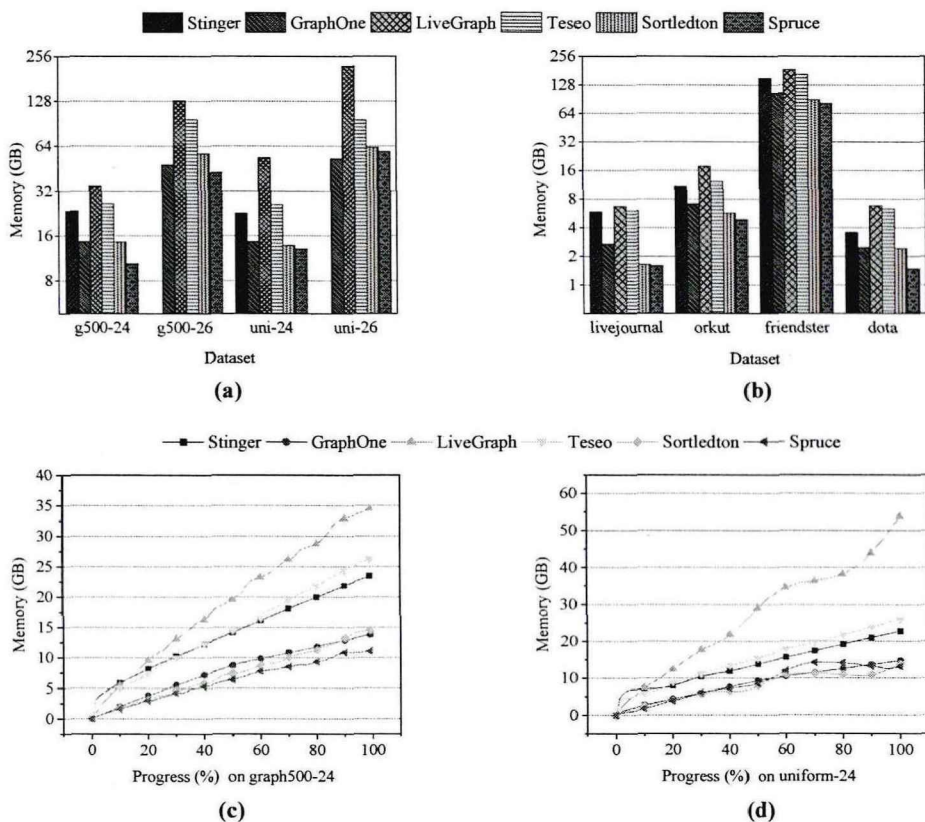


图 3.9 不同图存储方法的内存开销。(a) 显示了特定分布图的内存开销。(b) 显示了现实世界图的内存开销。(c) 和 (d) 分别显示在 *graph500-24* 和 *uniform-24* 上进行实验时系统的内存随实验进程的变化。

图3.9(a)和图3.9(b)展示了加载整个图数据集的内存开销。Y轴是以对数标度显示的内存开销,X轴代表不同的图数据集。所有系统使用的物理内存都高于原始数据的大小,这是因为:(1)一条无向边会存储为两条有向边;(2)插入过程中会为每条边随机生成8字节的权重;(3)存在时间戳和指针等附加结构。如图3.9(a)所示,对于具有特定分布的图, Spruce 在动态加载图时使用的内存最少。与最好的竞争对手 (Sortledton) 相比, Spruce 节省了约 5.5% ~ 28.7% 的内存。对于真实世界的图 (图 3.9(b)), 这一差距进一步拉大: Spruce 的内存消耗

比 Sortledton 低 38.5%，比 GraphOne 低 41.0%。LiveGraph 的空间开销很大，因为它为每条边存储了两个时间戳。尽管 Teseo 实现了压缩稀疏行（CSR）来存储图，但由于 Packed Memory Array 中的间隙需要额外的开销，因此内存消耗仍然很高。Spruce 得益于其压缩动态索引和紧凑存储的边，在大多数数据集上都具有最小的空间开销。

图3.9(c)和图3.9(d)记录了在两个具有相同规模的图数据集上图存储结构的内存占用随插入进度的变化（*graph500-24* 和 *uniform-24*）。所有方法的内存消耗与系统中存储的边之间都呈近似线性关系，其中 Spruce 的平均斜率最低。与 *graph500-24* 相比，LiveGraph 在 *uniform-24* 上的曲线上升得更快，这是因为当一个块已满时，LiveGraph 会将其数据复制到一个两倍于当前大小的新块中^[48]，并不适合顶点度数均匀分布的图。

3.4.3 图算法的时间性能

这一部分使用本节开头提到的 LDBC BenchMark 中的基准方法来测试所有图存储结构运行图算法的时间。本节测试了 BFS、SSSP、PR、WCC 和 CC 等具有不同图数据访问模式的算法的性能。表3.3展示了这些图算法的运行时间。本文以 Compressed Sparse Row (CSR) 为 Baseline，其他方法的运行时间均对标到 CSR。对于每种算法，本文都取了 5 次实验的平均运行时间，表中除 Baseline 外的最佳运行时间用粗体标出。

Breadth-First Search (BFS) 和 Single-Source Shortest Paths (SSSP): 对于所有存储方法，本文测试了不带方向优化的基本并行 BFS 算法^[24]和基于 Delta-Stepping 的 SSSP 算法^[105]。这两种算法都表现出很强的顶点随机访问模式和邻接边顺序访问模式。如表3.3所示。在所有方法中，除了 SSSP 在 *dota-league* 上的运行时间次与 Sortledton 外，Spruce 的运行时间是最短的，这归功于其索引中顶点的快速定位能力和图顶点邻居数据的良好局部性。

PageRank 和 Weakly Connected Components (WCC): 本文实现的 PageRank 和 WCC 的数据访问模式类似，都需要顺序访问所有顶点。PageRank 在 Spruce 上的运行时间约 Baseline 的 1.01 倍~2.97 倍，优于所有其他方法。对于 WCC，Spruce 除去在 *dota-league* 上的结果仅次于 Sortledton (2.39×Baseline)，在所有数据集上都给出了最佳结果。这说明对于小型图上的索引， $O(\lg \lg u)$ 时间优于 $O(\lg u)$ 时间的优势并没有明显体现出来。

Clustering Coefficient (CC): 对于 Spruce，本文采用了与 Teseo 和 Sortledton 相同的 CC 算法，该算法要求对顶点的邻接边进行排序。表3.3显示，在所有数据集上，Sortledton 和 Spruce 都明显优于其他方法。具体来说，Spruce 的运行时间在 *dota-league* 上位居第一，而 Sortledton 则在其他图数据集上排名第一。Sortledton

表 3.3 不同图算法的性能。对于 **Baseline**，表中显示的是确切的执行时间。对于其它所有方法，表中显示的是相对于 **Baseline** 所花费的时间比例。“-”表示图算法未在规定时间内完成。

图数据集	方法	BFS	PR	SSSP	WCC	CC
graph500-24	baseline	0.54s	2.11s	5.56s	8.24s	81.31s
	Stinger	1.75×	3.30×	1.75×	4.09×	-
	GraphOne	5.27×	3.67×	1.68×	2.17×	-
	LiveGraph	5.75×	3.67×	1.69×	2.35×	-
	Teseo	4.19×	3.52×	2.47×	3.35×	5.34×
	Sortledton	2.80×	2.26×	1.34×	3.55×	1.72×
	Spruce	1.45×	1.80×	1.07×	1.91×	2.90×
uniform-24	baseline	0.51s	3.14s	6.39s	1.12s	3.29s
	Stinger	2.27×	2.87×	2.12×	2.31×	18.45×
	GraphOne	11.45×	3.37×	2.46×	55.21×	19.70×
	LiveGraph	4.27×	3.21×	3.02×	2.25×	12.55×
	Teseo	2.65×	3.18×	3.33×	2.32×	22.46×
	Sortledton	1.91×	2.42×	2.50×	1.48×	2.05×
	Spruce	1.36×	1.55×	1.43×	1.39×	5.50×
orkut	baseline	0.13s	0.54s	1.72s	0.16s	2.80s
	Stinger	3.06×	5.58×	2.16×	6.99×	63.89×
	GraphOne	40.46×	12.23×	4.70×	161.54×	33.06×
	LiveGraph	5.02×	4.91×	2.21×	4.35×	31.70×
	Teseo	3.81×	3.32×	2.38×	5.33×	196.1×
	Sortledton	3.77×	3.78×	2.13×	5.14×	2.63×
	Spruce	1.69×	2.97×	1.23×	3.04×	5.22×
dota-league	baseline	0.12s	2.04s	0.48s	0.33s	61.41s
	Stinger	8.90×	4.80×	2.61×	6.28×	19.13×
	GraphOne	383.40×	24.70×	10.52×	133.84×	6.14×
	LiveGraph	9.94×	2.09×	1.37×	2.51×	7.92×
	Teseo	5.19×	1.52×	1.25×	2.45×	8.57×
	Sortledton	3.63×	1.23×	0.92×	1.95×	1.04×
	Spruce	3.04×	1.01×	1.11×	2.39×	1.01×

展示了对 CC 的良好支持，在 *graph500-24*、*uniform-24* 和 *orkut* 上取得了最佳运行时间。其他方法的执行时间明显落后于 Sortledton 和 Spruce。在 4 个数据集中，Spruce 有 3 个数据集的执行时间慢于 Sortledton，这主要是由于 BufferBlock 中的边无序造成的。

3.4.4 动态图的并发性能

本小节中主要是测试不同动态图存储结构的并发读写性能。

1. 多线程可扩展性

为了评估多线程的可扩展性，本文在 *graph500-24* 上使用 1~56 个写线程测试了插入性能。图 3.10 描述了所有方法的多线程性能。其中 x 轴为线程数，y 轴为插入操作的吞吐量。Spruce、Stinger 和 Teseo 最多可扩展至 56 个线程。Spruce 的扩展能力优于 Teseo 和 Stinger，其峰值吞吐量为 4.2 meps，这是因为它的索引使用了更轻量级的并发协议，在图不断演化的情况下无需使用 Read-Copy-Update

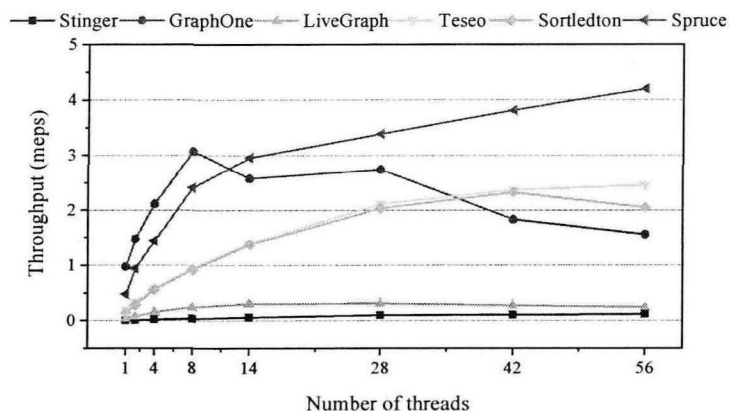


图 3.10 不同图存储方法在 *graph500-24* 上的多线程可扩放性。

技术进行更新。由于以顶点为中心的锁竞争激烈，Sortlepton 和 LiveGraph 只能分别扩放到 42 和 28 个线程。GraphOne 仅能扩放到 8 个线程，超过 28 个线程后，其性能会明显降低，这是由于不同线程之间的工作量不平衡造成的。

2. 并发读写性能

本部分测试并发读写工作负载的性能。实验设置与 Teseo^[50] 相同，如下所示。对于写者（写线程），Benchmark 会执行大约 $10 \cdot |E|$ 的更新操作，其中 $|E|$ 是原始图中的边的数量。前 10% 操作用于构建初始的图，其余 90% 操作是插入和删除操作，其中插入和删除操作的数量基本相当以维持图的规模和数据分布稳定。对于读者（读线程）而言，当初始图构建完成时它们就会执行图分析算法（BFS/PageRank）。本文选择 LiveGraph 作为比较对象的，它支持基于 MVCC 并发事务操作。本小节测试了各种图算法在不同读写工作量下的运行时间和吞吐量。

图 3.11(a)~(c) 分别展示了 0、16 和 28 个写线程时的 BFS 的执行时间。Spruce 和 LiveGraph 都能很好地支持在正在更新的图上进行图分析操作，由图可见随着阅读器（读线程）数量的增加，它们的运行时间会显著缩短。与没有写线程的情况相比，在最重写工作负载（28 个写线程）下，Spruce 和 LiveGraph 上 BFS 耗费的时间分别为 $1.04 \times$ 和 $\times$ 。对于 PageRank，也有类似的结果（图 3.11(d)~(f)）。在并发工作负载上，Spruce 的 PageRank 算法平均比 LiveGraph 快 1.9 倍。与无写入线程时的运行时间相比，Spruce 在动态图上的运行时间约为 $1.32 \times \sim 1.46 \times$ 。对于 LiveGraph，该值为 $1.32 \times \sim 1.49 \times$ 。Spruce 的 PR 执行速度最多降低 32.4%，LiveGraph 的 PR 执行速度最多降低了 42.3%。由于 LiveGraph 使用多版本日志来支持高级的快照隔离使读者和写者在互不干扰的情况下工作，因此随着分析线程的增加，LiveGraph 的曲线下落幅度更大^[48]。

图 3.11(g) 和图 3.11(h) 显示了 16 个和 28 个写线程下的更新吞吐量。当写线程的数量保持不变时，两种方法的更新吞吐量都不会受到读线程数量的明显影

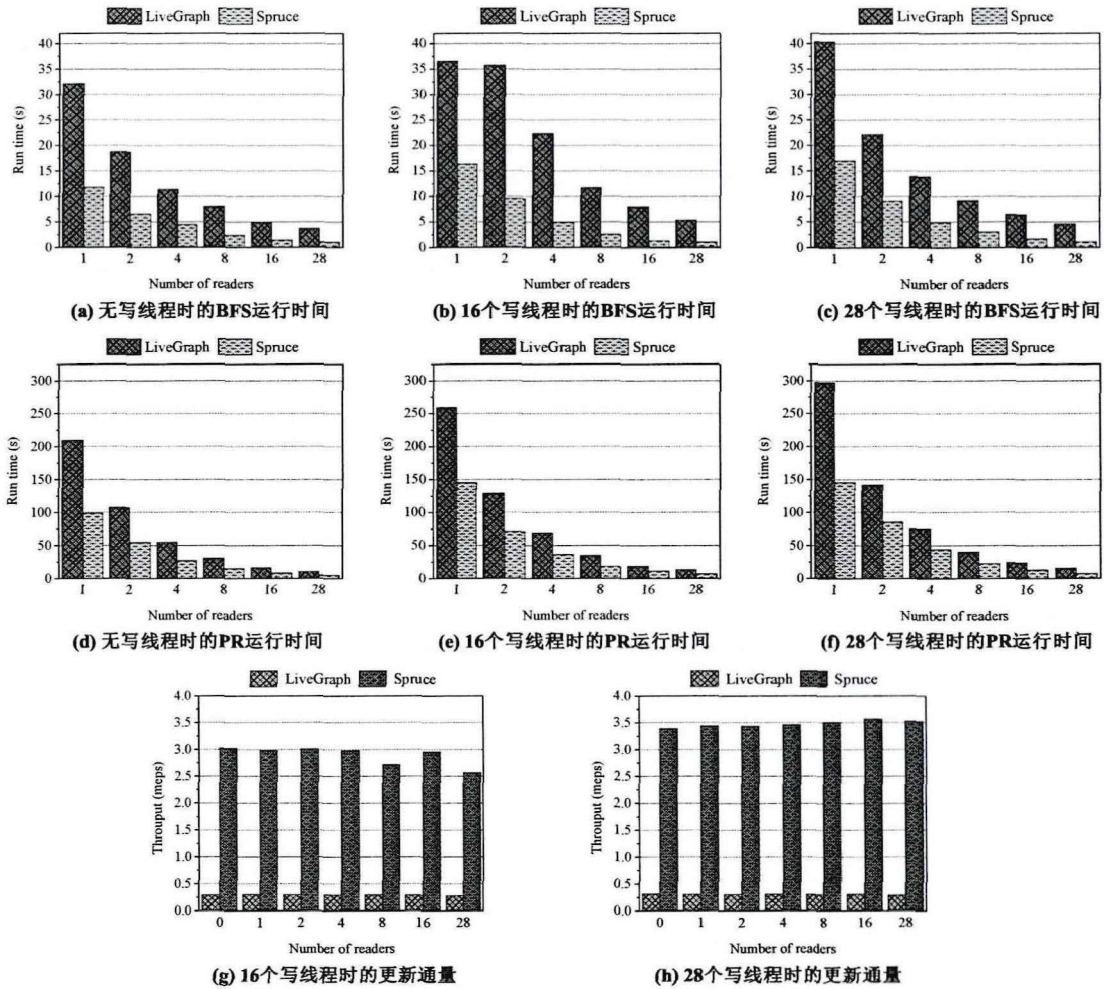


图 3.11 LiveGraph 和 Spruce 并发执行图更新和图分析的性能表现

响。当写线程数量从 16 个增加到 28 个时, LiveGraph 的吞吐量出现了小幅度的下降 (平均 3.3%), 而 Spruce 的吞吐量则出现了显著增加 (平均 21.7%)。本文将这种差异归因于两种结构所使用的并发协议不同。LiveGraph 中用于事务的细粒度 MVCC 协议给写线程带来了沉重的负担, 限制了它们的吞吐量, 而 Spruce 中的轻量级锁策略则让写线程更容易修改数据。

3.5 本章小结

本章详细地介绍了 Spruce, 一种高性能且空间节约的动态图存储方法。本章先是分析了现有的基于邻接表的图存储结构并指出它们难以同时兼顾空间和时间性能、并发操作繁琐等问题, 然后从三个方面介绍了 Spruce 的设计动机。此后本章介绍了 Spruce 的数据结构设计, 分析了其时间、空间复杂度并和其它工作进行了对比。进一步地, 本章还为 Spruce 的结构设计了新的并发控制协议、并就其正确性进行了证明, 还实现了多种并行的图分析算法。最后, 本文通过多项实验测试了 Spruce 的空间性能、时间性能、图分析性能和并发性能。

第4章 并行广度优先搜索算法的优化

本章概要：本章节主要介绍 ABi-BFS, 一种共享体系上优化的双向并行广度优先搜索 (Breadth-First Search, BFS) 算法。本章先分析共享体系上并行广度优先搜索的搜索模式, 结合现有的一些优化搜索算法指出不足和可改进之处, 顺势引出设计动机和思路。然后, 本章详细介绍 ABi-BFS 的算法设计, 从自上而下搜索和自下而上搜索两个方面介绍其优化方法。最后本章通过在不同图结构、不同图数据集上的实验验证了设计的优化算法的有效性。

4.1 并行广度优先搜索的分析

4.1.1 并行广度优先搜索的执行流程及分析

并行化是提高 BFS 性能的最有效方法之一。一般来说, 并行 BFS (Parallel Breadth-First Search, PBFS) 的执行是一个逐层访问整个图的迭代过程, 每次迭代可分为两个阶段: 节点扩展 (node expanding) 和任务分配 (task assigning)。如算法3.4和算法3.5所示, 经典的 PBFS 算法采用自顶向下的搜索模式, 在节点扩展阶段必须访问待扩展节点 (称为前沿, frontier) 的所有邻接边, 这给内存访问带来了沉重的负担。另一方面, 在扩展节点和分配任务的迭代过程中存在大量同步。此外, 由于 BFS 的随机数据访问模式, 分配给线程的工作量通常是不平衡的, 经常会导致不可忽略的同步开销^[106]。

由于经典的自顶向下 BFS 需要在每次迭代中检查与当前 frontier 节点相连的所有边, 当较大时, 由于内存访问频繁且不规则, 因此性能相对较差。Beamer S. 等人提出了双向 BFS (Direction-Optimizing BFS)^[24], 其主要贡献是设计了一种自下而上 (Bottom-Up) 的搜索模式, 以减少 BFS 迭代中需要检查的边。

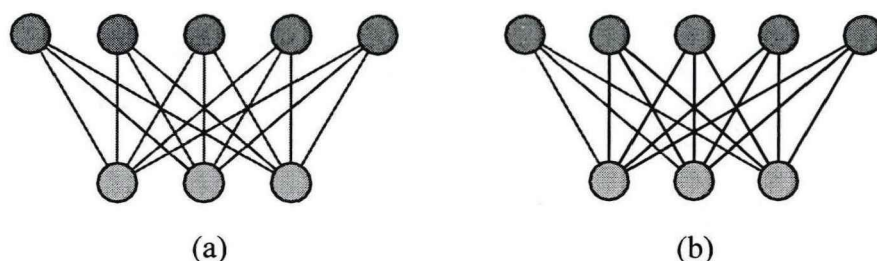


图 4.1 广度优先搜索的 (a) 自上而下搜索模式; (b) 自下而上的搜索模式。其中蓝色节点为当前的前沿节点, 绿色节点为未被访问的节点。

如算法 4.1 所示, 自下而上搜索模式 (BUStep) 会并行检查所有尚未进入 BFS

树的节点（即未访问节点），而不是访问当前前沿节点的邻居。对于每个这样的顶点，它都会使用 for 循环来检查它在当前 BFS 的前沿中是否有邻居。如果有，则将该顶点添加到 BFS 树的下一级（即下一前沿， $n_ext_frontier$ ），并终止循环。当前沿大小较大时该方法十分有效，因为它可以避免检查前沿中节点的所有边。例如，在图4.1中，如果使用自上而下的搜索模式，需要访问 15 条边，而如果采用自下而上的访问模式，则只需要访问 3 条边。随着 BFS 的推进，越来越多的节点被访问，在自下而上的搜索模式中许多未被访问的节点可能只需一次或几次检查就能在当前前沿中找到被访问的邻居，从而减少了整体的边检查。双向 BFS 结合了自顶向下搜索和自底向上搜索。通常双向的 BFS 算法从自上而下的搜索模式开始，一旦前沿的大小超过某个阈值，它就会切换到自下而上的搜索模式。如果前沿大小再次变小，它也可以切换回自上而下的搜索模式。

算法 4.1 BUSStep

Input: undirected graph $G = (V, E)$, visited array vs , set of nodes to be visited in the current level $c_frontier$, set of nodes to be visited in the next level $n_frontier$, parent map $parent$

```

1 parallel for  $v \in G.V$  do
2   if  $!vs[v]$  then
3     for  $u \in v.neighbours$  do
4       if  $u \in c\_frontier$  then
5          $parent[v] \leftarrow u$ ;
6          $vs[v] \leftarrow 1$ ;
7          $n\_frontier \leftarrow n\_frontier \cup \{v\}$ ;
8         break;
9       end
10    end
11  end
12 endfor
13  $c\_frontier \leftarrow n\_frontier$ ;
14  $n\_frontier \leftarrow \emptyset$ ;

```

4.1.2 现有并行图搜索算法的优化方法的不足

在算法及其执行的层面，已有的方法主要有两类，分别是面向负载均衡的优化和面向内存访问的优化。

负载均衡方法的目的是尽可能在不同计算节点间均匀分配 BFS 的工作负载。图分割被广泛用于分布式计算系统中的工作量分配。在大型图系统中最常被使用的是图分割方法，这类方法通过对图中的边或者顶点进行切割，将图尽可能均匀的分配到各个计算节点本地进行计算。^[39-40,42]。还有些方法侧重于解决高度数节点造成的工作量不平衡问题。这类解决方案包括使用额外的线程来处理高程度节点^[85]；根据节点的度数对节点进行分类，然后在不同阶段访问它们^[107-108]。此外，还有一些方法使用动态调度来重新平衡线程间的工作量^[36-37]。

由于 BFS 是一种受内存访问限制的算法，因此另一类方法着重于优化其内

存访问。很多工作都旨在改进图的数据存储结构,以提高数据的局部性,从而获得更好的内存访问效率。Agarwal V. 等人提出可以使用位向量来表示 BFS 的前沿节点以降低内存访问成本^[79]。Ligra+ 使用差分编码和 nibble 编码来压缩节点的边,并使用并行解码技术来加速图算法的读取过程^[31]。Yong-Yeon J. 等人设计了一种新的图存储数据布局,其中节点和边的顺序遵循 BFS 的遍历顺序^[109]。Chen Z. 等人提出了一种基于规则的图压缩技术^[110],使用特定的规则来表示邻接边的重复序列,从而可以通过缓存和重复使用规则的处理结果来减少冗余计算。此外,还有一些方法侧重于优化内存访问过程。Chhugani J. 等人提出了一种邻接预取技术,以隐藏内存访问的延迟^[80]。Paredes M. 等人使用邻接表的矢量化和 SIMD 指令分批访问/修改节点^[81]。Yasui Y. 等人提出了一种度感知的自下而上搜索方法^[111]。他们在自下而上搜索时根据节点的度数对邻接列表中的节点进行降序排序,从而进一步减少了在节点扩展阶段检查的边。他们还提出了一种根据节点度数排序和分配节点的技术,以提高不同线程间的调度效率^[112]。Zhang C. 等人利用轮循洗牌技术对其进行了改进,并提出了一种在位图上使用 SIMD 指令的跳转技术,以加速自下而上搜索中节点的访问速度^[23]。

需要指出的是,尽管侧重于平衡工作量的方法减少线程之间的等待时间,但任务分配阶段和节点扩展阶段之间的同步仍然存在于不同线程之间。在自下而上的 BFS 搜索上,随着 BFS 的推进,访问的节点越来越多,其搜索的性能仍然会收到较差局部性的制约。近年来,越来越多基于图分析的应用建立在动态图之上^[113-114]。依赖于图数据存储布局的 BFS 的许多优化方法(如图分割和顶点排序)主要是针对静态图设计的,因此并不适用于动态图。例如,当图发生变化时,基于图划分的图分析方法需要重新计算整个图的拓扑结构,并在计算节点之间重新分配节点和边才能有效实现负载平衡。

4.1.3 设计动机与思路

在对以往工作的限制性的洞察的基础上,本文现在给出 ABi-BFS 的设计动机与思路:

1. **使用异步的任务分配方法减小同步时延。**目前为止,现有的共享体系上的 BFS 优化工作中任务分配/调度(例如,算法 3.5,第 1 行)和节点扩展(例如,算法 3.5,第 2~8 行)的过程是分开执行的。也就是说,任务分配阶段的执行总是在等待节点扩展阶段结束。本文注意到,任务分配阶段可以通过与节点扩展过程混合的方式异步执行,从而减少同步延迟。
2. **使用动态的结构筛选自下而上搜索过程中需要检查的节点。**现有的自下而上步骤优化工作主要集中在减少要检查的边或使用紧凑的数据结构来提高其性能^[23,111-112]。然而,在每个自下而上的步骤中,所有这些方法仍然需

要检查整个 *visited* 数组来过滤未访问的节点（算法 4.1, line 1~2）。当前沿着 BFS 的进行而变小时，由于内存访问成本较高，这种检查会浪费较多的时间。因此，本文提出了一种在自下而上的步骤中动态选择前沿节点的策略，以避免这种不必要的内存访问，同时提高数据的局部性。

3. **使用动态的任务划分方法实现负载均衡：**许多方法都依赖于图预先计算的特征（如图中顶点之间的排序顺序和簇的划分）来平衡节点之间的工作量，难以应用到实时的动态的图分析中。其他方法（如工作偷取）可能会导致线程间不规则、频繁的通信，使得动态图系统的负载雪上加霜。在 ABi-BFS 中，本文在滑动队列的基础上进行了新的尝试，以在任务分配阶段实现负载均衡，从而降低通信开销并消除计算图特征的成本。

4.2 ABi-BFS: 高性能的双向图搜索算法

ABi-BFS 是共享内存系统上的一种通用双向 PBFS 算法。在 ABi-BFS 算法中，本文提出了两种优化方法来减少同步开销和提高数据局部性。下面将对此进行详细介绍。

4.2.1 算法概览

ABi-BFS 优化了 BFS 算法中的自顶向下搜索和自底向上搜索。它遵循方向优化 BFS 的基本流程，如图 4.2 所示。如果总度小于或等于转换阈值，算法就会使用自顶向下搜索来遍历图，否则就会使用自底向上搜索。对于自顶向下搜索，本文提出了一种异步模式，将扩展节点和分配任务两个阶段混合在一起。对于自下而上的搜索，本文采用了一种动态选择前沿候选者的策略，以提高数据局部性。

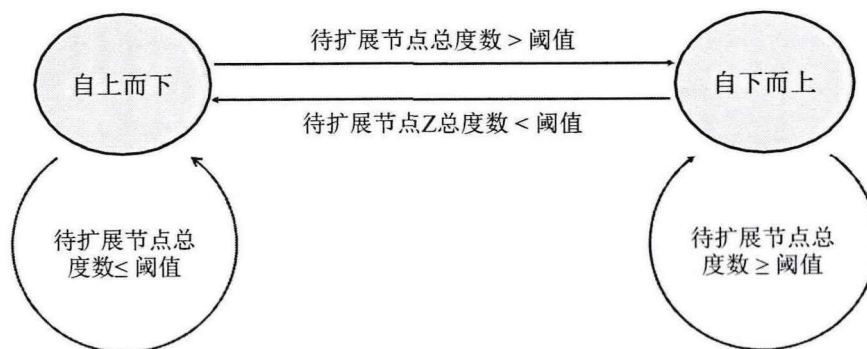


图 4.2 ABi-BFS 执行模式示意图

在 ABi-BFS 中，存储任务的滑动队列（Sliding Queue）取代节点前沿作为任务分配和节点扩展的动态数据结构。滑动队列是一种无锁队列，它使用双缓冲区技术来支持并发操作。滑动队列的基本结构和机制如图 4.3 所示。对于读者，滑

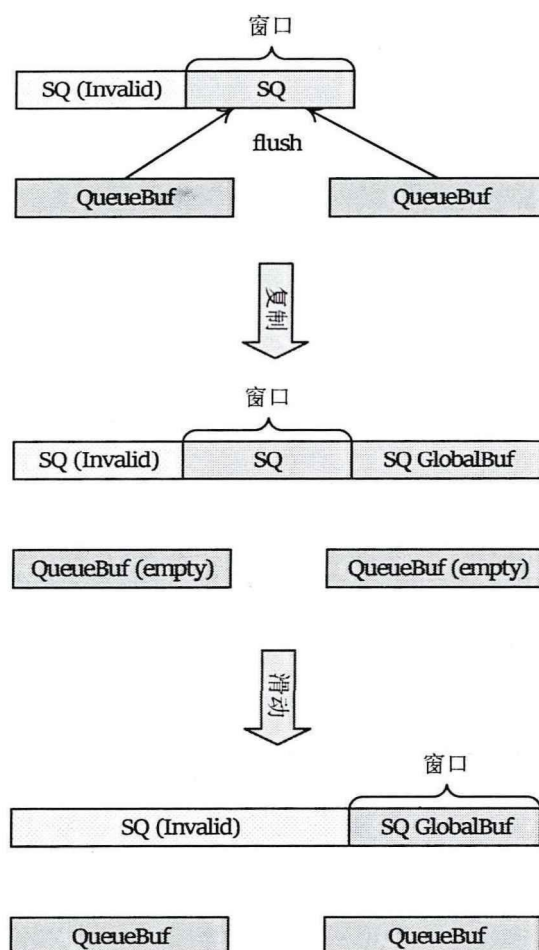


图 4.3 滑动队列的基本结构及操作

滑动队列使用滑动窗口机制来控制队列中元素的可访问性。也就是说，读者只能访问窗口内的数据。对于写者，每个写者都有一个本地队列缓冲区。要将元素推入滑动队列，写者首先要将项目插入其本地缓冲区。当缓冲区满时（或由程序员决定），缓冲区将被刷新，其内容将被复制 (flush) 到滑动队列中。flush 方法使用原子 `fetch_and_add` 来获取滑动队列中用于复制的位置，从而确保多个线程可以无冲突地获得唯一、不重叠的区域用于复制。最后，队列上的窗口被滑动，以进行入队和出队操作。算法中有两种队列可以存储任务。`g_queue` 是一个共享的全局滑动队列，用于存储扩展低度节点的任务；`l_queue[i]` 用于存储线程 `i` 扩展高度节点的分区任务。每个任务包含三个字段：(1) `node_id`，表示任务属于哪个节点；(2) `size`，表示要检查的邻居数量；(3) `pos`，记录要访问的第一个邻居的位置。

算法4.2展示了 ABi-BFS 的执行过程。算法从源节点开始，初始将源节点标记为已访问节点，并将其父节点设置为自身（第 1~2 行）。然后它根据源节点的

算法 4.2 ABi-BFS

Input: undirected graph $G = (V, E)$, source vertex src , visited array vs , global task queue g_queue , thread-local task queues $l_queue[NUM_THREADS]$, dynamic set of nodes bu_queue , parent map $parent$

Output: the BFS tree $parent$

```

1  $vs[src] \leftarrow 1$ ;
2  $parent[src] \leftarrow src$ ;
3  $e\_cnt \leftarrow src.degree$ ;
4  $g\_queue.push(\{src, src.degree, 0\})$ ;
5  $g\_queue.slide()$ ;
6 while  $!g\_queue.empty() \text{ or } !l\_queue[0].empty()$  do
7   if  $e\_cnt \leq D\_THRESHOLD$  then
8      $e\_cnt \leftarrow 0$ ;
9     Asynchronous_TDStep(...);
10  else
11    ConvQueue( $c\_frontier, g\_queue, l\_queue$ );
12     $n\_frontier \leftarrow \emptyset$ ;
13    while  $e\_cnt > D\_THRESHOLD$  do
14       $e\_cnt \leftarrow 0$ ;
15      Select-Optimized_BUStep(...);
16    end
17    ConvBitmap( $c\_frontier, g\_queue, l\_queue$ );
18  end
19 end
20 return  $parent$ ;

```

信息初始化边计数（用来统计前沿节点总度数）和任务队列（第3~4行）。当任务队列不为空时，它会选择自上而下搜索或自顶向下搜索来扩展节点。如果队列中节点的总度数小于或等于自下而上阈值，算法就会使用自上而下搜索来遍历图。否则，算法将使用自下而上搜索（第6~13行）。除非另外说明，本文使用 $|E|/20$ 作为自下而上的阈值，其中 $|E|$ 代表输入图 $G = (V, E)$ 中的边的数量。在自下而上的步骤中，本文使用位图来提高数据局部性，即在自下而上的搜索算法中使用一个包含了图中顶点数个比特的位图来代替 $visit$ 数组，其中的每一位用于标记顶点是否被访问过。位图和队列之间的转换在进入和退出自下而上搜索模式时执行（第11、17行）。最后，完成 BFS 进程并返回 BFS 树（第20行）。在下面的小节中将详细介绍自上而下搜索和自下而上搜索的优化方法。

4.2.2 使用异步任务分配的自上而下搜索

ABi-BFS 采用异步模式在自顶向下搜索中扩展节点和分配任务。其基本思想是，下一轮迭代将要执行的任务分配无需等待节点扩展阶段结束。也就是说，某个线程的任务由许多不同的线程异步分配。考虑到负载平衡，在分配过程中，扩展高阶节点的任务被分割并分配给多个线程，而扩展低阶节点的任务只分配给一个线程。这样做的原因是高度数节点在容易导致 PBFS 进程中的负载不平衡^[108]，进一步增加线程间因为等待而导致的同步开销，将它们的扩展工作量分

算法 4.3 Asynchronous TStep

Input: undirected graph $G = (V, E)$, visited array vs , global task queue g_queue , thread-local task queues $l_queue[NUM_THREADS]$, parent map $parent$, edge count e_cnt

```

1 parallel for each thread do
2    $t_{id} = \text{get\_thread\_num}()$ ;
3    $c\_size = g\_queue.size() / NUM\_THREADS$ ;
4    $g\_start = t_{id} * c\_size$ ;
5   if  $t_{id} \neq NUM\_THREADS$  then
6      $g\_stop = g\_start + c\_size$ ;
7   else
8      $g\_stop = g\_queue.size()$ ;
9   end
10  for  $i \in [g\_start, g\_stop)$  do
11     $\text{ProcessTask}(g\_queue[i], \dots)$ ;
12  end
13  for  $t \in l\_queue[t_{id}]$  do
14     $\text{ProcessTask}(t, \dots)$ ;
15  end
16 endfor
17 parallel for  $q \in l\_queue$  do
18    $q.slide()$ ;
19 endfor
20  $g\_queue.slide()$ ;

```

算法 4.4 ProcessTask

Input: $task$, $G = (V, E)$, vs , g_queue , $parent$, $l_queue[NUM_THREADS]$, e_cnt

```

1  $v = task.node\_id$ ;
2 for  $j \in [task.pos, task.pos + task.size)$  do
3    $u = v.neighbours[j]$ ;
4   if  $!vs[v.neighbours[j]]$  then
5      $parent[u] \leftarrow v$ ;
6      $vs[u] \leftarrow 1$ ;
7      $e\_cnt \leftarrow e\_cnt + u.degree$ ;
8     if  $u.degree \leq THRESHOLD$  then
9        $g\_queue.push(\{u, u.degree, 0\})$ ;
10    else
11       $\text{PartitionTask}(u, l\_queue)$ ;
12    end
13  end
14 end

```

配给不同的线程可以显著降低同步成本。相比之下，低度数节点不存在这个问题，因此与低度节点相关的每个任务只分配给一个线程。

为了实现这一目标，每个线程都拥有一个本地滑动队列来存储任务。在自顶向下搜索步骤中，每个线程处理其本地任务队列，并将每个未访问的邻居（下一次迭代中要访问的节点）分配给与其度数相对应的一个或多个线程的任务队列。在这一阶段，本文设计了一种“预扩展”技术，即在任务分配阶段就定位节点邻居的位置，并将其存储在任务队列中。在下次迭代之前滑动窗口并不会滑动，即新添加的任务是不可见的，从而保证了 BFS 算法的正确性。

异步任务分配和节点扩展算法的伪代码如算法 4.3 和 4.4 所示。在自顶向下搜索的每次迭代中，每个线程首先使用其线程 ID 获取其分配任务在全局滑动队列中的位置（算法 4.3，第 2~9 行）。然后，线程依次处理全局滑动队列和滑动队列中的任务（算法 4.3，第 10 行）。在此过程中，线程会访问任务列表中节点的邻居（算法 4.4，第 2~6 行）。对于之前未访问过的每个节点，线程都会获取其度来估计工作量。如果节点的度数较低，该节点的扩展任务将被推入全局滑动队列，否则任务将被平均分割并推入所有线程的本地任务队列（算法 4.4，第 7~12 行，由于预扩展阶段取决于图存储结构，因此算法中已隐去）。最后，当所有任务都完成后，队列的窗口就会滑动，为下一次迭代做准备（算法 4.3，第 17 行）。

本文采用异步模式扩展节点和分配任务的主要优势有两个方面。首先，它减少了任务分配/调度的时间，因为任务是直接异步分配给每个线程的。其次，它能以较低的计算成本在不同线程之间平衡工作量，从而进一步缩短同步时间。与动态调度和工作偷取等其他负载平衡方法相比，本文方法中的本地滑动队列提供了更高的数据局部性，并降低了线程间执行任务时的通信成本。

4.2.3 使用节点筛选策略的自下而上搜索

自下而上的 BFS 方法需要扫描整个 *visited* 数组以获取未访问节点进行扩展。随着 BFS 深度的增加，未访问节点会越来越少，并且稀疏地分布在所有节点中。此外，图中通常存在一些没有邻居的孤立节点，如果它们不是源节点，BFS 算法将永远不会遍历它们。在这种情况下，自下而上的 BFS 会产生大量冗余工作（即检查节点是否已被访问过），并表现出很差的数据局部性。

本文注意到，与自顶向下搜索不同，自底向上 BFS 有一个重要特性，即下一次迭代中要扩展的节点总是当前迭代中扩展的节点的子集。其正确性是显而易见的，因为未访问节点集的大小总是随着 BFS 的进行而不断缩小。受此启发，本文提出了一种动态集合，用于在自下而上的搜索中选择候选节点（可能被标记为下一个前沿的节点）。该集合初始化为全部的未访问节点，在后续的执行流程中它将依据筛选策略进行动态的更新。将节点筛选进集合的选择策略很简单：对于当前迭代中扩展的每个节点，如果该节点被标记为已访问节点或没有邻居，就将其从当前集合中删除，否则予以保留。

在实现过程中，本文使用滑动队列来构造这种动态集合。优化的自下而上搜索的伪代码如算法 4.5 和 4.6 所述。如果是第一次执行自下而上搜索，它将并行遍历图中的所有节点，并处理未访问的非孤立节点的邻接节点，为下一次自下而上搜索选择节点（算法 4.5，第 1~7 行）。否则，它只会并行处理滑动队列中的节点。（算法 4.5，第 8 行~12）。Process 函数会检查它在当前 BFS 边界中是否有邻居。如果有，则将该节点添加到 BFS 树的下一个前沿，并终止循环（算法 4.6，

算法 4.5 Select-Optimized BUStep

Input: undirected graph $G = (V, E)$, visited array vs , set of nodes to be visited in the current BUStep bu_queue , set of nodes to be expanded in the current level $c_frontier$, set of nodes to be expanded in the next level $n_frontier$, parent map $parent$, the flag indicates whether it is the first time for bottom-up search $first$, edge count e_cnt

```

1 if  $first$  then
2    $first = 0$ ;
3   parallel for  $v \in G.V$  do
4     if  $!vs[v]$  and  $v.degree \neq 0$  then
5       ProcessBUNode( $v, \dots$ );
6     end
7   endfor
8 else
9   parallel for  $v \in bu\_queue$  do
10    ProcessBUNode( $v, \dots$ );
11  endfor
12 end
13  $c\_frontier \leftarrow n\_frontier$ ;
14  $n\_frontier \leftarrow \emptyset$ ;
15  $bu\_queue.slide()$ ;
```

算法 4.6 ProcessBUNode

Input: $v, vs, bu_queue, c_frontier, n_frontier, parent, e_cnt$

```

1  $next\_flag \leftarrow true$ ;
2 for  $u \in v.neighbours$  do
3   if  $u \in c\_frontier$  then
4      $parent[v] \leftarrow u$ ;
5      $vs[v] \leftarrow 1$ ;
6      $n\_frontier \leftarrow n\_frontier \cup \{v\}$ ;
7      $e\_cnt \leftarrow e\_cnt + v.degree$ ;
8      $next\_flag \leftarrow false$ ;
9     break;
10  end
11 end
12 if  $next\_flag$  then
13    $bu\_queue.push(v)$ ;
14 end
```

第2~11行)。否则，该节点将被选中推入滑动队列（算法4.6，第11行）。最后，算法更新BFS和队列的前沿节点（算法4.5，第13行）。

需要指出的是，与传统的自下而上搜索方法相比，在本文的方法中，前沿的候选节点被紧凑地存储在滑动队列中，而不是稀疏地分布在所有节点的集合（如数组或链表）中，从而带来了更好的数据局部性。在自下而上的搜索中，每个节点的任务量是不确定的（因为循环可能提前终止，算法4.6，第9行）。此时筛选候选节点的方法还有助于平衡自下而上搜索的工作量，因为孤立节点（在节点扩展阶段的工作量要小得多）会在扩展阶段之前被过滤掉，避免了某些线程过早地闲置下来。

4.3 实验结果与分析

本节使用的实验设备和前文一样，是一台英特尔装载两片 Xeon Gold 5120 @ 2.20GHz 处理器和 500GB DDR4 内存的双插槽服务器。所有代码均使用 GCC 11.3.0 和 O3 参数编译。

本文使用由 Graph500 基准的 Kronecker 生成器生成的无向图以及现实生活中的网络图。这些图自然符合常见的网络属性，例如低直径和幂律程度分布^[98]。生成图的大小与输入给生成器的两个参数有关：规模（scale）和边系数（edge-factor）。一个规模为 s 、边系数为 f 的图有 2^s 个节点和 $2^s \cdot f$ 条边，在后文中命名为 *g500-scale-edgefactor*。Orkut 社交网络有 300 万个节点和 1.17 亿条边，*twitter-2010* 网络有 4100 万个节点和 14.68 亿条边。

在图数据结构的选择方面，本文首先使用了 Compressed Sparse Row (CSR)，目前最常见的静态图存储结构来验证算法的有效性。更进一步，本文使用了 Spruce，一种源于 van Emde Boas 树 (vEB 树)，结合了邻接表和 CSR 的优点，具有高并发性能的动态图存储结构来验证算法的普适性。

4.3.1 消融分析

首先，本文通过实验验证 ABi-BFS 中使用的优化方法的效果。对于每个结果，本文均取 50 次试验的中值。本文选择位图优化的双向 PBFS 作为 Baseline 方法，衡量 ABi-BFS 中提出的优化方法的独立优化效果和累积优化效果。

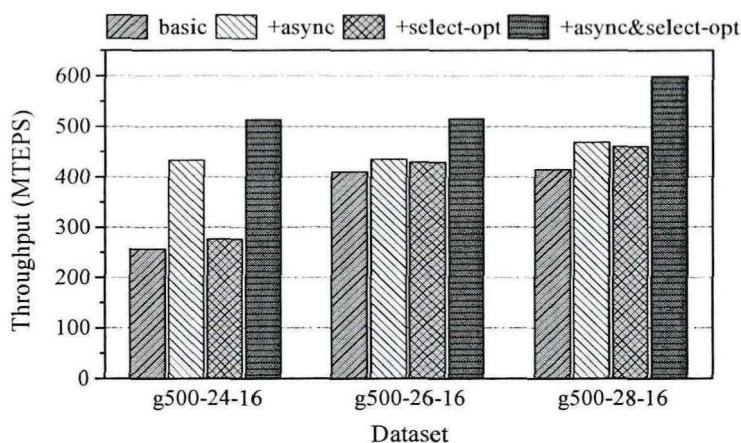


图 4.4 使用不同优化方法时 PBFS 的运行性能

本文首先对 PBFS 运行的数据通量进行统计，其结果以每秒遍历数百万条边 (MTEPS) 为单位进行评估，计算方法为 PBFS 的遍历边数与运行时间之比。其实验结果如图 4.4 所示。与 Baseline 相比，只使用异步任务分配（异步方法）的平均改进率为 29.3%，而节点筛选策略（选择优化方法）的平均改进率约为 7.8%。使用这两种方法的累积效果更为显著，性能提高了 56.9%。随着图规模的增大，选

择优化方法的效果也在增加 ($g500-24-16$ 为 7.7%, $g500-28-16$ 为 11.1%)。这是因为更大的图需要更多的运行时内存来执行,而数据局部性的改善在这方面更为有效。

更进一步地,本文对 PBFS 运行过程中各个线程的运行时间进行了测试和分析。图4.5显示了 Baseline 和 ABi-BFS 在 $g500-24-16$ 数据集上第3层和第5层的线程执行时间的累积分布情况(为了方便展示,此处选取了具有前沿节点较多的层级进行比较),其中各层的运行时间均进行了归一化处理。从图中可以看出,无论是在自下而上搜索(图4.5 (a))还是自上而下搜索(图4.5 (b)),Baseline 方法存在着显著的工作负载不均衡的情况,而 ABi-BFS 则可以平衡工作负载并减少 BFS 搜索的执行时间。

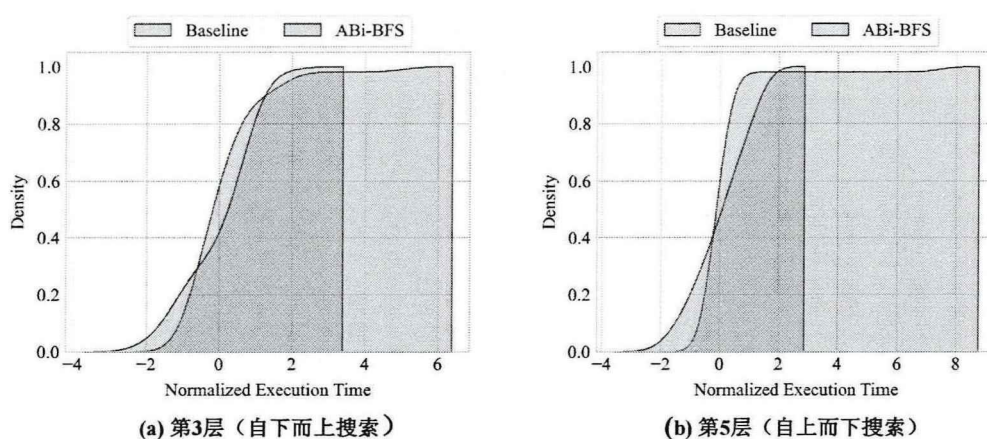


图 4.5 graph500-24-16 上优化前后线程运行时间的累积函数分布图

4.3.2 可扩展性实验

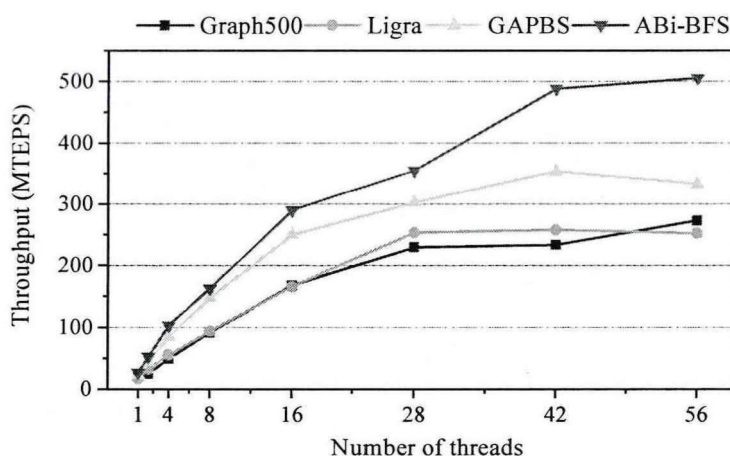


图 4.6 不同 PBFS 方法的可扩展性

在这一部分,本文在 $g500-24-16$ 数据集上使用不同线程(从 1 到 56)运

行PBFS算法,测试其可扩展性,并统计其吞吐量。如图4.6所示,Graph500的参考PBFS和ABi-BFS均可扩展至56个线程,其中ABi-BFS的吞吐量是Graph500的两倍,而Ligra和GAPBS只能扩展至42个线程。在所有方法中,ABi-BFS的吞吐量最高,这一结果归功于ABi-BFS中减少的同步步骤和更平衡的工作负载。

4.3.3 图规模对算法时间性能的影响

本小节中分析不同PBFS算法在不同规模图上的性能。本文选择中Graph500中默认的 $edgefactor = 16$ 和 $scale \in [21, 28]$ 作为生成图的参数进行评估,并测试了与Graph500的参考的PBFS相比的吞吐量和速度提升。这里的加速比使用 $t_{Graph500}/t_{method}$ 计算,其中 $t_{Graph500}$ 是Graph500的PBFS算法的耗时, t_{method} 是其他方法的耗时。

图4.7(a)显示了所有BFS方法的吞吐量。ABi-BFS的峰值吞吐量为609 MTEPS,而余下方法中效果最好的GAPBS的峰值吞吐量为424 MTEPS。与ABi-BFS和GAPBS相比,Graph500和Ligra的吞吐量相对较低,仅为300 MTEPS左右。图4.7(b)显示了Ligra、GAPBS和ABi-BFS相较于Graph500的PBFS的运行时间的加速比。ABi-BFS的平均运行速度比Graph500快27.9倍,在所有方法中排名第一。该图显示Ligra、GAPBS和ABi-BFS的运行时间比Graph500短得多,这主要是因为双向BFS大幅减少了BFS中需要检查的边。

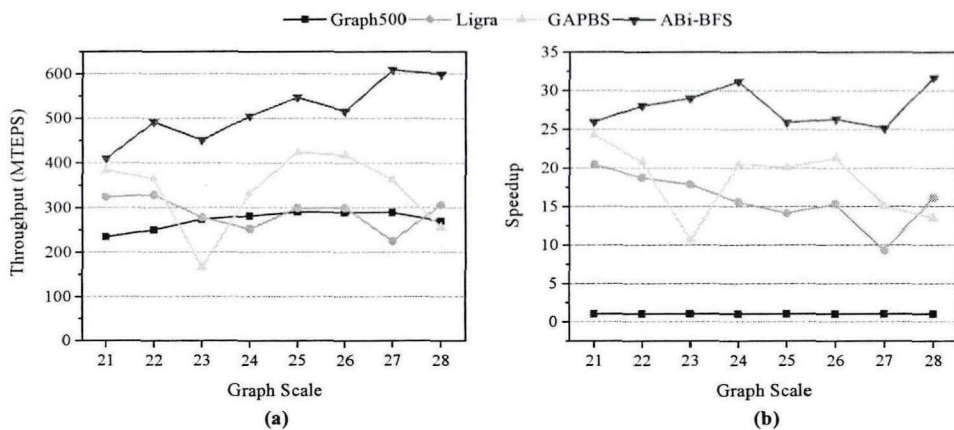


图 4.7 不同 PBFS 方法在不同规模的图上的性能

4.3.4 图密度对算法时间性能的影响

这一小节分析 PBFS 算法在不同密度的图上的性能。本文使用以 $scale = 24$ 和 $edgefactor \in [16, 44]$ 为参数的图,并统计与Graph500的参考PBFS相比的吞吐量和加速度提升。

图4.8(a)描述了所有方法的吞吐量,图4.8(b)描述了Ligra、GAPBS和ABi-BFS的相较于Graph500的执行时间加速比。ABi-BFS的吞吐量平均约为

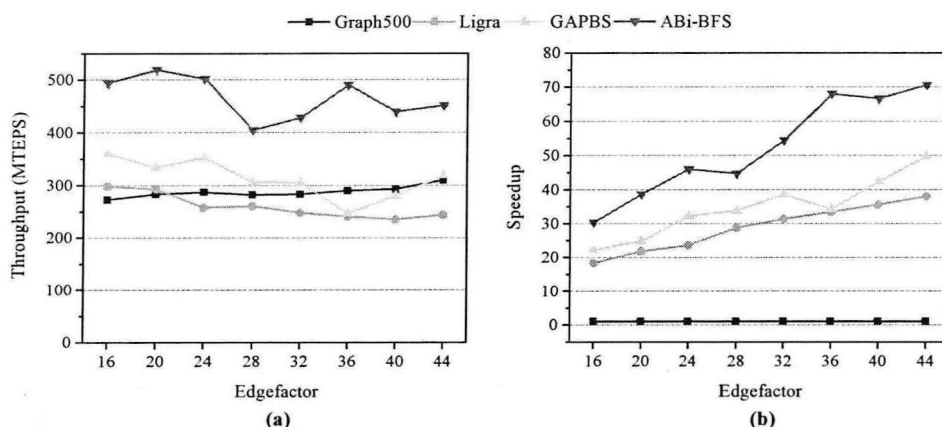


图 4.8 不同 PBFS 方法在不同密度的图上的性能

466 MTEPS, 而 Graph500、Ligra 和 GAPBS 分别为 288 MTEPS、260 MTEPS 和 313 MTEPS。在所有方法中, ABi-BFS 的加速比也是第一名, 平均为 Graph500 的 52.7 倍, Ligra 的 1.81 倍, GAPBS 的 1.51 倍。随着图密度的增加, Ligra、GAPBS 和 ABi-BFS 的速度也在增加, 不过所有方法的吞吐量都没有显著增加。原因是 Graph500 只使用自上而下搜索, 必须检查前沿中节点的所有邻接边, 导致 BFS 执行速度减慢。而其他方法使用的是自下而上的搜索, 其耗时不会受到图密度的太大影响。与竞争对手相比, ABi-BFS 的加速曲线呈现出最陡峭的上升趋势, 这是因为在执行过程中工作量更加均衡, 数据局部性更好。

4.3.5 动态图存储结构上的算法性能

本文在 Packed CSR 和 Spruce 上均实现了 ABi-BFS 算法。Packed CSR 通过使用 Packed Memory Array 替代 CSR 中的静态数组实现 CSR 对简单图更新操作的支持。Spruce 是基于 vEB 树、结合邻接表和 CSR 特性设计的动态图数据结构, 支持高效的图更新操作和并发图更新和图分析。本节的测试中使用生成的图和现实世界中的图, 比较了 ABi-BFS 和 GAPBS 中的 PBFS 两种双向 PBFS 算法在 PCSR 上 Spruce 上的性能表现。

图4.9 (a) 和图4.9 (b) 分别显示了 PBFS 在 PCSR 和 Spruce 上的遍历边的吞吐量。在 PCSR 上, 与 GAPBS 相比, ABi-BFS 在 g500-24-16、g500-26-16、orkut 和 twitter-2010 上的吞吐量分别提高了 59.8%、21.1%、8.3% 和 16.8%。在 Spruce 上, 改进幅度分别为 33.6%、87.3%、14.4% 和 86.0%。这两种算法在 PCSR 上的性能差距尚可, 而在 Spruce 图上的差距就较为明显了。这是因为在 PCSR 中, BFS 过程中的大部分时间用于遍历数组, 呈现出一种顺序内存访问模式。而在 Spruce 中, 更多时间用于访问顶点索引中的节点, 呈现出一种随机存储器访问模式。在自下而上搜索中的优化中, 动态节点选择策略帮助 Spruce 显著减少了顶点索引中的随机内存访问, 因此与 PCSR 相比, Spruce 上的 ABi-BFS 有了更显

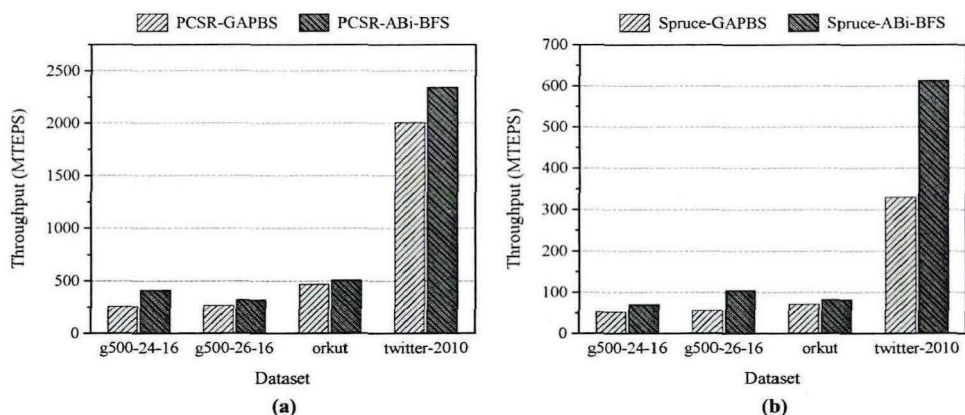


图 4.9 GAPBS 中的 PBFS 算法和 ABi-BFS 在 Spruce 上的性能表现对比

著的改进。此外，随着图密度的增加，不平衡的工作量也成为影响 GAPBS 算法性能的重要因素之一。在动态图结构中，节点是通常是通过顶点索引访问的，而不是直接使用标识符作为数组偏移来获取其存储位置。因此，在 twitter-2010 等密度较高的图上，BFS 的总体吞吐量要高于 orkut 等密度较低的图。在 PCSR 上，PBFS 的遍历边的吞吐量高于 Spruce。这主要是因为 PCSR 本质上仍然是由数组构成的图存储结构，牺牲了更新性能来换取读取效率：当图规模较大时（如百万个顶点），PCSR 由于数组中进行 Rebalance 的开销较大，基本已经失去了即时更新的能力。除此之外，相较于 Spruce，PCSR 也不支持版本控制、并发图更新等操作。

需要指出，不同的图存储结构对于 ABi-BFS 的性能也有着一定的影响。结合本小节与 4.3.1 小节的内容可以看出，对于静态图存储结构（CSR）或者动态能力支持较差的图存储结构（PCSR），ABi-BFS 的相较于其它算法的优化效果尚可。对于动态性能更好的图（Spruce），ABi-BFS 对于广度优先搜索速度的提升效果则更为显著，这得益于 ABi-BFS 的动态任务分配方式和节点选择策略对于支持快速图更新的图存储结构的良好适应性。

4.4 本章小结

本章主要介绍了本文的第二项工作，ABi-BFS，一种通用的双向并行广度优先搜索的算法。针对并行 BFS 中同步开销高、数据局部性差、难以适用于动态图的挑战，本节为任务分配和节点扩展阶段提出了一种异步执行模式以降低同步成本，并为前沿节点提出了一种选择策略，以加速双向广度优先搜索中的自下而上搜索。实验表明，与共享内存系统上最先进的 PBFS 算法实现相比，ABi-BFS 在可扩展性、吞吐量和耗时方面都取得了最佳效果。

第5章 总结与展望

5.1 本文工作

近年来社交平台、网络媒体、生物信息学、金融和物流等诸多领域日益增长的复杂数据使得图数据库正发挥着越来越重要的作用。图作为能够原生表示和查询数据实体之间错综复杂的关系数据格式，在复杂网络数据的表示上有着与生俱来的优势。一张图通常由节点、边及其属性构成。节点代表实体（如人、企业或任何感兴趣的对象），边定义这些实体之间的关系，属性提供键值对，提供有关节点或边的附加信息。这种结构允许用户对数据内的关系进行高效查询和遍历，使应用程序能够快速访问连接的数据点。图存储技术作为图数据库的基石在这其中发挥着不可或缺的作用。

大数据时代下数据的日新月异也要求数据库更加有效地支持动态数据的存储。在图数据的存储方面，现有的应用需求不仅要求数据库支持图数据的动态更新，还要求它们具有良好的空间效率和图分析能力。高效的动态图存储和分析离不开图存储的结构和图算法的优化。从动态图存储结构上来看，目前的主流工作往往由图顶点索引、边存储结构和与之相匹配的并发控制协议组成。常见的图顶点索引主要从键值数据库直接迁移而来，边存储结构通常是在传统图存储结构如邻接表、CSR 上进行改进而来，而并发控制则往往照搬关系型数据库的设计，没有很好地结合现实生活中大图的特性为动态图数据设计专门的存储方法。从图算法上而言，大部分工作主要针对静态图设计，在动态图上的适配性较差。因此，本文设计了一种既高性能又空间节约的动态图存储结构，并针对动态图存储结构的特点对图算法进行了特殊的优化。本文创新性的工作如下：

1. 高性能、空间节约的动态图存储方法设计

本文充分考虑了动态图的稀疏性、小世界特性和分布特征，设计了 Spruce，一种高效的内存动态图数据存储方法。Spruce 的索引采用了类似 vEB 树的位向量表示方法，通过共享前缀和层级规模开根递减实现了对图中顶点的低空间兼高性能表示，能够在双对数时间内有效地支持动态操作。Spruce 的边存储结构对邻接表和 CSR 取其精华，去其糟粕，通过缓冲区加紧凑数组的设计兼顾的空间效率和更新速度。在此基础上，本文进一步根据 Spruce 索引中没有节点分裂合并的特性和图分析事务的需求，基于 ROWEX、乐观锁和多版本并发控制为其量身打造了轻量级的并发控制协议。此后，本文在 Spruce 上实现了主流的并行图算法，以满足常见的图分析需求。和 Sortledton, LiveGraph, Teseo 等先进的图存储技术的对比实验结果表明，无论是空间效率还是操作性能，Spruce 都位居前列。

2. 并行广度优先搜索算法的优化

本文从图算法中最为基础的广度优先搜索入手对图算法进行优化,提出了 ABi-BFS, 一种减少了同步开销、提高图分析执行时数据局部性的并行双向 BFS 算法。相较于以往的并行 BFS 算法, ABi-BFS 创新性地使用了异步任务分配策略, 能有效地减少任务分配与节点扩展步骤之间的同步操作。在分配的过程中, ABi-BFS 使用了局部队列和全局队列相结合的方法, 对大小度数的顶点分别处理以达到负载均衡。ABi-BFS 还提出了一种动态节点选择策略来在自下而上的搜索中动态筛选顶点, 提高数据局部性, 避免无效的访存操作。在共享内存服务器上的实验结果表明, ABi-BFS 可以显著提升 BFS 的遍历速度和执行时间, 在静态和动态图数据结构以及不同规模、不同密度的图上均有着良好的适应性。

5.2 工作展望

本文对动态图存储结构和图算法的优化进行了深入研究, 提出了新的动态图存储方法和高效的图分析优化方法, 并通过实验验证了其有效性。尽管如此, 本文的研究工作仍然存在着一些不足之处和可以进一步探索的方向, 具体如下:

1. 分布式设计与数据持久性保证

目前设计的图存储结构仍然局限在单节点的共享内存系统中。随着数据规模的增大, 存储结构会难以避免地遇到内存不足的情况。此时, 将该结构推广到分布式系统、云平台 and 磁盘存储阵列就尤为必要。可以考虑结合 The Log-Structured Merge-Tree (LSM-Tree)、远程直接内存访问 (Remote Direct Memory Access, RDMA)、网络拓扑乃至更先进的 Compute Express Link (CXL) 架构进行专门的设计和优化来满足更大规模的图数据存储与分析需求。

2. 大数据压缩技术

目前的图存储系统里尚不涉及到数据的压缩技术。未来随着数据量的增加, 使用数据压缩技术节约空间也不失为一个好的解决方案。目前在这方面主要面临的挑战是现有的数据压缩技术无法兼顾数据的压缩率与动态性能, 对于数据的修改等操作往往需要经历解压缩-修改-再压缩的繁琐步骤。在未来可以尝试考虑设计合适的免解压的压缩技术用于动态图数据的存储。

3. 适用于动态图的图分析执行框架

本文目前只将 BFS 结合动态图数据进行了优化设计。实际上, 大多数图算法都有着对顶点或是邻居节点与 BFS 或多或少相似的访问模式。因此可以考虑将本文的优化方法进行一般性推广得到动态图的图分析执行框架。更进一步地, 可以考虑针对动态图增量更新的特点, 设计专用的增量动态图算法, 通过对图数据的更新部分的特殊分析处理, 在原有的图分析结果上进行补充得到更新后的图分析结果。

参考文献

- [1] ZAFAROVICH T S. The concept and features of big data[J/OL]. International journal of advanced research in education, technology and management, 2023, 2(2). <https://doi.org/10.5281/zenodo.7662875>.
- [2] SAEED N, HASSAN ABED A. Big data with column oriented nosql database to overcome the drawbacks of relational databases[J/OL]. International Journal of Advanced Networking and Applications, 2020, 11: 4423-4428. DOI: 10.35444/IJANA.2020.11057.
- [3] ARSHAD M, BROHI M N, SOOMRO T R, et al. Nosql: Future of bigdata analytics characteristics and comparison with rdbms[M/OL]. Cham: Springer International Publishing, 2023: 1927-1951. https://doi.org/10.1007/978-3-031-12382-5_106.
- [4] HAN J, E H, LE G, et al. Survey on nosql database[C/OL]//2011 6th International Conference on Pervasive Computing and Applications. 2011: 363-366. DOI: 10.1109/ICPCA.2011.6106531.
- [5] MATSUNOBU Y, DONG S, LEE H. Myrocks: Lsm-tree database storage engine serving facebook's social graph[J/OL]. Proc. VLDB Endow., 2020, 13(12): 3217 – 3230. <https://doi.org/10.14778/3415478.3415546>.
- [6] ZHANG Z, WANG X, ZHU W. Automated machine learning on graphs: A survey[C/OL]//ZHOU Z. Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021. ijcai.org, 2021: 4704-4712. <https://doi.org/10.24963/ijcai.2021/637>. DOI: 10.24963/IJCAI.2021/637.
- [7] LIU W, ZHANG Y, WANG J, et al. Item relationship graph neural networks for e-commerce [J/OL]. IEEE Transactions on Neural Networks and Learning Systems, 2022, 33(9): 4785-4799. DOI: 10.1109/TNNLS.2021.3060872.
- [8] 徐曜. 基于图数据的电商用户特征分析方法[J]. 中国新通信, 2023, 25(12): 19-20+26.
- [9] DB-Engines - Knowledge Base of Relational and NoSQL Database Management Systems — db-engines.com[EB/OL]. <https://db-engines.com/en/>.
- [10] LU C, LAM W, ZHANG Y. Twitter user modeling and tweets recommendation based on wikipedia concept graph[J]. AAAI Workshop - Technical Report, 2012: 33-38.
- [11] SHARMA N, DE P K. Application of machine learning to predict climate change consequences arising due to investments by banks in fossil fuel sectors[M/OL]. Singapore: Springer Nature Singapore, 2023: 49-90. https://doi.org/10.1007/978-981-19-5244-9_3.
- [12] 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/OL]//SIGMOD '21: Pro-

- ceedings of the 2021 International Conference on Management of Data. New York, NY, USA: Association for Computing Machinery, 2021: 513–527. <https://doi.org/10.1145/3448016.3457263>.
- [13] NAYAK A, PORIYA A, POOJARY D. Article: Type of nosql databases and its comparison with relational databases[J]. International Journal of Applied Information Systems, 2013, 5: 16-19.
- [14] DE LEO D, BONCZ P. Teseo and the analysis of structural dynamic graphs[J/OL]. Proc. VLDB Endow., 2021, 14(6): 1053–1066. <https://doi.org/10.14778/3447689.3447708>.
- [15] MAURER W D, LEWIS T G. Hash table methods[J/OL]. ACM Comput. Surv., 1975, 7(1): 5–19. <https://doi.org/10.1145/356643.356645>.
- [16] ATHANASSOULIS M, AILAMAKI A. Bf-tree: approximate tree indexing[J/OL]. Proc. VLDB Endow., 2014, 7(14): 1881–1892. <https://doi.org/10.14778/2733085.2733094>.
- [17] DE LEO D, BONCZ P. Packed memory arrays - rewired[C/OL]//2019 IEEE 35th International Conference on Data Engineering (ICDE). 2019: 830-841. DOI: 10.1109/ICDE.2019.00079.
- [18] KUNG H T, ROBINSON J T. On optimistic methods for concurrency control[J/OL]. ACM Trans. Database Syst., 1981, 6(2): 213–226. <https://doi.org/10.1145/319566.319567>.
- [19] LESKOVEC J, KREVL A. SNAP Datasets: Stanford large network dataset collection [EB/OL]. 2014. <http://snap.stanford.edu/data>.
- [20] WIJS A. Bfs-based model checking of linear-time properties with an application on gpus[C]//CHAUDHURI S, FARZAN A. Computer Aided Verification. Cham: Springer International Publishing, 2016: 472-493.
- [21] ARTILES O, SAEED F. Turbobfs: Gpu based breadth-first search (bfs) algorithms in the language of linear algebra[C/OL]//2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). 2021: 520-528. DOI: 10.1109/IPDPSW52791.2021.00084.
- [22] Lü Y, GUO H, HUANG L, et al. Graphpeg: Accelerating graph processing on gpus[J/OL]. ACM Trans. Archit. Code Optim., 2021, 18(3). <https://doi.org/10.1145/3450440>.
- [23] ZHANG C, CAO H, YE X, et al. Highly efficient breadth-first search on cpu-based single-node system[C/OL]//2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS). 2019: 2066-2071. DOI: 10.1109/HPCC/SmartCity/DSS.2019.00286.
- [24] BEAMER S, ASANOVIC K, PATTERSON D. Direction-optimizing breadth-first search [C/OL]//SC '12: Proceedings of the International Conference on High Performance Comput-

- ing, Networking, Storage and Analysis. 2012: 1-10. DOI: 10.1109/SC.2012.50.
- [25] CHEN R, SHI J, CHEN Y, et al. Powerlyra: Differentiated graph computation and partitioning on skewed graphs[J/OL]. ACM Trans. Parallel Comput., 2019, 5(3). <https://doi.org/10.1145/3298989>.
- [26] HARARY F. The determinant of the adjacency matrix of a graph[J/OL]. SIAM Review, 1962, 4(3): 202-210. <https://doi.org/10.1137/1004057>.
- [27] JEREMY SIEK A L, Lie-Quan Lee. The boost graph library (bgl)[EB/OL]. 2022. https://www.boost.org/doc/libs/1_80_0/libs/graph/doc/index.html.
- [28] ROY A, MIHAILOVIC I, ZWAENEPOEL W. X-stream: edge-centric graph processing using streaming partitions[C/OL]//SOSP '13: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2013: 472-488. <https://doi.org/10.1145/2517349.2522740>.
- [29] LIU W, VINTER B. Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication[C/OL]//ICS '15: Proceedings of the 29th ACM on International Conference on Supercomputing. New York, NY, USA: Association for Computing Machinery, 2015: 339-350. <https://doi.org/10.1145/2751205.2751209>.
- [30] BOLDI P, VIGNA S. The webgraph framework i: compression techniques[C/OL]//WWW '04: Proceedings of the 13th International Conference on World Wide Web. New York, NY, USA: Association for Computing Machinery, 2004: 595-602. <https://doi.org/10.1145/988672.988752>.
- [31] SHUN J, DHULIPALA L, BLELLOCH G E. Smaller and faster: Parallel processing of compressed graphs with ligra+[C/OL]//2015 Data Compression Conference. 2015: 403-412. DOI: 10.1109/DCC.2015.8.
- [32] KARYPIS G, KUMAR V. A fast and high quality multilevel scheme for partitioning irregular graphs[J/OL]. SIAM Journal on Scientific Computing, 1998, 20(1): 359-392. <https://doi.org/10.1137/S1064827595287997>.
- [33] DHULIPALA L, KABILJO I, KARRER B, et al. Compressing graphs and indexes with recursive graph bisection[C/OL]//KDD '16: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. New York, NY, USA: Association for Computing Machinery, 2016: 1535-1544. <https://doi.org/10.1145/2939672.2939862>.
- [34] FUCHS P, MARGAN D, GICEVA J. Sortledton: a universal, transactional graph data structure[J/OL]. Proceedings of the VLDB Endowment, 2022, 15(6): 1173 - 1186. DOI: 10.14778/3514061.3514065.
- [35] KRUSKAL C P, RUDOLPH L, SNIR M. Efficient parallel algorithms for graph problems [J/OL]. Algorithmica, 1990, 5(1-4): 43-64. DOI: 10.1007/bf01840376.

- [36] ZHANG Y, HANSEN E. Parallel breadth-first heuristic search on a shared-memory architecture[J]. MA @BULLET July, 2006, 17.
- [37] CONG G, KODALI S, KRISHNAMOORTHY S, et al. Solving large, irregular graph problems using adaptive work-stealing[C/OL]//2008 37th International Conference on Parallel Processing. 2008: 536-545. DOI: 10.1109/ICPP.2008.88.
- [38] PEARCE R, GOKHALE M, AMATO N M. Faster parallel traversal of scale free graphs at extreme scale with vertex delegates[C/OL]//SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 2014: 549-559. DOI: 10.1109/SC.2014.50.
- [39] MALEWICZ G, AUSTERN M H, BIK A J, et al. Pregel: a system for large-scale graph processing[C/OL]//SIGMOD '10: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. New York, NY, USA: Association for Computing Machinery, 2010: 135-146. <https://doi.org/10.1145/1807167.1807184>.
- [40] LOW Y, BICKSON D, GONZALEZ J, et al. Distributed graphlab: a framework for machine learning and data mining in the cloud[J/OL]. Proc. VLDB Endow., 2012, 5(8): 716-727. <https://doi.org/10.14778/2212351.2212354>.
- [41] GONZALEZ J E, LOW Y, GU H, et al. Powergraph: Distributed graph-parallel computation on natural graphs[C/OL]//THEKKATH C, VAHDAT A. 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012. USENIX Association, 2012: 17-30. <https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez>.
- [42] XIN R S, GONZALEZ J E, FRANKLIN M J, et al. Graphx: a resilient distributed graph system on spark[C/OL]//GRADES '13: First International Workshop on Graph Data Management Experiences and Systems. New York, NY, USA: Association for Computing Machinery, 2013. <https://doi.org/10.1145/2484425.2484427>.
- [43] EDIGER D, MCCOLL R, RIEDY J, et al. Stinger: High performance data structure for streaming graphs[J/OL]. 2012 IEEE Conference on High Performance Extreme Computing, 2012, 1: 1-5. DOI: 10.1109/hpec.2012.6408680.
- [44] MACKOP, MARATHE V J, MARGO D W, et al. Llama: Efficient graph analytics using large multiversioned arrays[C/OL]//2015 IEEE 31st International Conference on Data Engineering. 2015: 363-374. DOI: 10.1109/ICDE.2015.7113298.
- [45] 毛志雄, 刘志楠, 高叙宁, 等. 面向幂律图的动态图存储结构 Power-PCSR[J]. 计算机科学, 2024(1-10): 1-10.
- [46] KUMAR P, HUANG H H. Graphone: A data store for real-time analytics on evolving graphs [J/OL]. ACM Trans. Storage, 2020, 15(4). <https://doi.org/10.1145/3364180>.

- [47] DHULIPALA L, BLELLOCH G E, SHUN J. Low-latency graph streaming using compressed purely-functional trees[C/OL]//PLDI 2019: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA: Association for Computing Machinery, 2019: 918–934. <https://doi.org/10.1145/3314221.3314598>.
- [48] ZHU X, FENG G, SERAFINI M, et al. Livegraph: a transactional graph storage system with purely sequential adjacency list scans[J/OL]. Proceedings of the VLDB Endowment, 2020, 13(7): 1020–1034. DOI: 10.14778/3384345.3384351.
- [49] FIRMLI S, TRIGONAKIS V, LOZI J P, et al. CSR++: A Fast, Scalable, Update-Friendly Graph Data Structure[C/OL]//BRAMAS Q, OSHMAN R, ROMANO P. Leibniz International Proceedings in Informatics (LIPIcs): volume 184 24th International Conference on Principles of Distributed Systems (OPODIS 2020). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021: 17:1–17:16. <https://drops.dagstuhl.de/opus/volltexte/2021/13502>. DOI: 10.4230/LIPIcs.OPODIS.2020.17.
- [50] LEOD D, BONCZ P. Teseo and the analysis of structural dynamic graphs[J/OL]. Proceedings of the VLDB Endowment, 2021, 14(6): 1053–1066. DOI: 10.14778/3447689.3447708.
- [51] LEIS V, KEMPER A, NEUMANN T. The adaptive radix tree: Artful indexing for main-memory databases[C/OL]//2013 IEEE 29th International Conference on Data Engineering (ICDE). 2013: 38–49. DOI: 10.1109/ICDE.2013.6544812.
- [52] DHULIPALA L, BLELLOCH G E, GU Y, et al. Pac-trees: Supporting parallel and compressed purely-functional collections[C/OL]//PLDI 2022: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. New York, NY, USA: Association for Computing Machinery, 2022: 108 – 121. <https://doi.org/10.1145/3519939.3523733>.
- [53] FUCHS P, MARGAN D, GICEVA J. Sortedledton: a universal graph data structure[J/OL]. SIGMOD Rec., 2023, 52(1): 17–25. <https://doi.org/10.1145/3604437.3604442>.
- [54] GUO W, LI Y, SHA M, et al. Parallel personalized pagerank on dynamic graphs[J/OL]. Proc. VLDB Endow., 2017, 11(1): 93–106. <https://doi.org/10.14778/3151113.3151121>.
- [55] SHUKLA K, REGUNTA S C, TONDOMKER S H, et al. Efficient parallel algorithms for betweenness- and closeness-centrality in dynamic graphs[C/OL]//ICS '20: Proceedings of the 34th ACM International Conference on Supercomputing. New York, NY, USA: Association for Computing Machinery, 2020. <https://doi.org/10.1145/3392717.3392743>.
- [56] LEIS V, SCHEIBNER F, KEMPER A, et al. The art of practical synchronization[C/OL]//DaMoN '16: Proceedings of the 12th International Workshop on Data Management on New Hardware. New York, NY, USA: Association for Computing Machinery, 2016. <https://doi.org/10.1145/2933349.2933352>.

- [57] SHALEV O, SHAVIT N. Split-ordered lists: Lock-free extensible hash tables[J/OL]. J. ACM, 2006, 53(3): 379–405. <https://doi.org/10.1145/1147954.1147958>.
- [58] PAGH R, RODLER F F. Cuckoo hashing[J/OL]. Journal of Algorithms, 2004, 51(2): 122–144. <https://www.sciencedirect.com/science/article/pii/S0196677403001925>. DOI: <https://doi.org/10.1016/j.jalgor.2003.12.002>.
- [59] HERLIHY M, SHAVIT N, TZAFRIR M. Hopscotch hashing[C]//TAUBENFELD G. Distributed Computing. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008: 350–364.
- [60] FREDKIN E. Trie memory[J/OL]. Commun. ACM, 1960, 3(9): 490–499. <https://doi.org/10.1145/367390.367400>.
- [61] BINNA R, ZANGERLE E, PICHL M, et al. Hot: A height optimized trie index for main-memory database systems[C/OL]//SIGMOD '18: Proceedings of the 2018 International Conference on Management of Data. New York, NY, USA: Association for Computing Machinery, 2018: 521–534. <https://doi.org/10.1145/3183713.3196896>.
- [62] COMER D. Ubiquitous b-tree[J/OL]. ACM Comput. Surv., 1979, 11(2): 121–137. <https://doi.org/10.1145/356770.356776>.
- [63] BENDER M A, HU H. An adaptive packed-memory array[J/OL]. ACM Trans. Database Syst., 2007, 32(4): 26–es. <https://doi.org/10.1145/1292609.1292616>.
- [64] BENDER M A, FARACH-COLTON M, MOSTEIRO M A. Insertion sort is $o(n \log n)$ [J/OL]. Theor. Comp. Sys., 2006, 39(3): 391–397. <https://doi.org/10.1007/s00224-005-1237-z>.
- [65] WHEATMAN B, XU H. Packed compressed sparse row: A dynamic graph representation [J/OL]. 2018 IEEE High Performance extreme Computing Conference (HPEC), 2018, 00: 1–7. DOI: 10.1109/hpec.2018.8547566.
- [66] PUGH W. Skip lists: a probabilistic alternative to balanced trees[J/OL]. Commun. ACM, 1990, 33(6): 668–676. <https://doi.org/10.1145/78973.78977>.
- [67] PLATZ K, MITTAL N, VENKATESAN S. Concurrent unrolled skiplist[C/OL]//2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS). 2019: 1579–1589. DOI: 10.1109/ICDCS.2019.00157.
- [68] OKASAKI C. Purely functional data structures[M]. Cambridge University Press, 1999.
- [69] LANGVILLE A N, MEYER C D. Google's pagerank and beyond: The science of search engine rankings[M]. Princeton university press, 2006.
- [70] ABUSALIM S W, IBRAHIM R, SARINGAT M Z, et al. Comparative analysis between dijkstra and bellman-ford algorithms in shortest path optimization[C]//IOP Conference Series: Materials Science and Engineering: volume 917. IOP Publishing, 2020: 012077.
- [71] ZHANG H, XIE J, ZHANG X. Communication-efficient Δ -stepping for distributed computing systems[C/OL]//2023 19th International Conference on Wireless and Mobile Computing,

- Networking and Communications (WiMob). 2023: 369-374. DOI: 10.1109/WiMob58348.2023.10187809.
- [72] TARJAN R E. A new algorithm for finding weak components[J/OL]. Information Processing Letters, 1974, 3(1): 13-15. <https://www.sciencedirect.com/science/article/pii/0020019074900404>. DOI: [https://doi.org/10.1016/0020-0190\(74\)90040-4](https://doi.org/10.1016/0020-0190(74)90040-4).
- [73] VISHKIN U. An optimal parallel connectivity algorithm[J/OL]. Discrete Applied Mathematics, 1984, 9(2): 197-207. <https://www.sciencedirect.com/science/article/pii/0166218X84900192>. DOI: [https://doi.org/10.1016/0166-218X\(84\)90019-2](https://doi.org/10.1016/0166-218X(84)90019-2).
- [74] WATTS D J, STROGATZ S H. Collective dynamics of ‘small-world’ networks[J/OL]. Nature, 1998, 393(6684): 440-442. <https://doi.org/10.1038/30918>.
- [75] EDMONDS N G, WILLCOCK J, LUMSDAINE A. Design of a large-scale hybrid-parallel graph library[C/OL]//2010. <https://api.semanticscholar.org/CorpusID:16985245>.
- [76] RAFIEV A, YAKOVLEV A, TARAWNEH G, et al. Synchronization in graph analysis algorithms on the partially ordered event-triggered systems many-core architecture[J/OL]. IET Computers & Digital Techniques, 2022, 16(2-3): 71-88. <https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/cdt2.12041>. DOI: <https://doi.org/10.1049/cdt2.12041>.
- [77] ACAR U A, CHARGUÉRAUD A, RAINEY M. Data structures and algorithms for robust and fast parallel graph search[C/OL]//2014. <https://api.semanticscholar.org/CorpusID:6668524>.
- [78] GURUNG A, DEKA A, BARTOCCI E, et al. Parallel reachability analysis for hybrid systems[C/OL]//2016 ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE). 2016: 12-22. DOI: 10.1109/MEMCOD.2016.7797741.
- [79] AGARWAL V, PETRINI F, PASETTO D, et al. Scalable graph exploration on multicore processors[C/OL]//SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. 2010: 1-11. DOI: 10.1109/SC.2010.46.
- [80] CHHUGANI J, SATISH N, KIM C, et al. Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency[C/OL]//2012 IEEE 26th International Parallel and Distributed Processing Symposium. 2012: 378-389. DOI: 10.1109/IPDPS.2012.43.
- [81] PAREDES M, RILEY G, LUJÁN M. Exploiting parallelism and vectorisation in breadth-first search for the intel xeon phi[J/OL]. IEEE Transactions on Parallel and Distributed Systems, 2020, 31(1): 111-128. DOI: 10.1109/TPDS.2019.2927451.
- [82] WEN H, ZHANG W. Improving parallelism of breadth first search (bfs) algorithm for accelerated performance on gpus[C/OL]//2019 IEEE High Performance Extreme Computing Conference (HPEC). 2019: 1-7. DOI: 10.1109/HPEC.2019.8916551.
- [83] HARISH P, NARAYANAN P J. Accelerating large graph algorithms on the gpu using cuda

- [C]//ALURU S, PARASHAR M, BADRINATH R, et al. High Performance Computing – HiPC 2007. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007: 197-208.
- [84] DONG R, CAO H, YE X, et al. Highly efficient and gpu-friendly implementation of bfs on single-node system[C/OL]//2020 IEEE Intl Conf on Parallel and Distributed Processing with Applications, Big Data and Cloud Computing, Sustainable Computing and Communications, Social Computing and Networking (ISPA/BDCLOUD/SocialCom/SustainCom). 2020: 544-553. DOI: 10.1109/ISPA-BDCLOUD-SocialCom-SustainCom51426.2020.00094.
- [85] HONG S, KIM S K, OGUNTEBI T, et al. Accelerating cuda graph algorithms at maximum warp[C/OL]//PPoPP '11: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming. New York, NY, USA: Association for Computing Machinery, 2011: 267-276. <https://doi.org/10.1145/1941553.1941590>.
- [86] CHUNG F R K. Graph theory in the information age[C/OL]//2010. <https://api.semanticscholar.org/CorpusID:1366784>.
- [87] GALLIAN J. A dynamic survey of graph labeling[J]. Electronic Journal of Combinatorics, 2018, 1(DynamicSurveys).
- [88] CAI Z, LOGOTHETIS D, SIGANOS G. Facilitating real-time graph mining[C/OL]//CloudDB '12: Proceedings of the Fourth International Workshop on Cloud Data Management. New York, NY, USA: Association for Computing Machinery, 2012: 1 – 8. <https://doi.org/10.1145/2390021.2390023>.
- [89] SHI X, CUI B, SHAO Y, et al. Tornado: A system for real-time iterative analysis over evolving data[C/OL]//SIGMOD '16: Proceedings of the 2016 International Conference on Management of Data. New York, NY, USA: Association for Computing Machinery, 2016: 417-430. <https://doi.org/10.1145/2882903.2882950>.
- [90] MURRAY D G, MCSHERRY F, ISARD M, et al. Incremental, iterative data processing with timely dataflow[J/OL]. Commun. ACM, 2016, 59(10): 75-83. <https://doi.org/10.1145/2983551>.
- [91] BAYER R, MCCREIGHT E. Organization and maintenance of large ordered indices[C/OL]//SIGFIDET '70: Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control. New York, NY, USA: Association for Computing Machinery, 1970: 107-141. <https://doi.org/10.1145/1734663.1734671>.
- [92] MÄSKER M, SÜß T, NAGEL L, et al. Hyperion: Building the largest in-memory search tree [C/OL]//SIGMOD '19: Proceedings of the 2019 International Conference on Management of Data. New York, NY, USA: Association for Computing Machinery, 2019: 1207-1222. <https://doi.org/10.1145/3299869.3319870>.
- [93] ON S T, HU H, LI Y, et al. Lazy-update b+-tree for flash devices[C/OL]//2009 Tenth Interna-

- tional Conference on Mobile Data Management: Systems, Services and Middleware. 2009: 323-328. DOI: 10.1109/MDM.2009.48.
- [94] AMIR A, EFRAT A, INDYK P, et al. Efficient regular data structures and algorithms for dilation, location, and proximity problems[J/OL]. *Algorithmica*, 2001, 30(2): 164-187. <https://doi.org/10.1007/s00453-001-0013-y>.
- [95] GONZALEZ J E, XIN R S, DAVE A, et al. Graphx: Graph processing in a distributed dataflow framework[C]//OSDI'14: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. USA: USENIX Association, 2014: 599-613.
- [96] Neo4j graph database platform[Z]. 2024.
- [97] CHAKRABARTI D, ZHAN Y, FALOUTSOS C. R-mat: A recursive model for graph mining [M/OL]. 442-446. <https://epubs.siam.org/doi/abs/10.1137/1.9781611972740.43>.
- [98] LESKOVEC J, CHAKRABARTI D, KLEINBERG J, et al. Kronecker graphs: An approach to modeling networks[J]. *J. Mach. Learn. Res.*, 2010, 11: 985-1042.
- [99] AGRAWAL K, SHUN J, GU Y, et al. Parallel longest increasing subsequence and van emde boas trees[J/OL]. *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*, 2023: 327-340. DOI: 10.1145/3558481.3591069.
- [100] PRESHING J. junction[J/OL]. GitHub repository, 2018. <https://github.com/preshing/junction>.
- [101] GUO Y, IOSUP A. The game trace archive[C/OL]//2012 11th Annual Workshop on Network and Systems Support for Games (NetGames). 2012: 1-6. DOI: 10.1109/NetGames.2012.6404027.
- [102] ROSSI R A, AHMED N K. The network data repository with interactive graph analytics and visualization[C/OL]//AAAI. 2015. <https://networkrepository.com>.
- [103] IOSUP A, HEGEMAN T, NGAI W L, et al. Ldbc graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms[J/OL]. *Proc. VLDB Endow.*, 2016, 9 (13): 1317-1328. <https://doi.org/10.14778/3007263.3007270>.
- [104] SIEBENMANN C. Understanding resident set size and the rss problem on modern unixes [Z]. 2012.
- [105] MEYER U, SANDERS P. Delta-stepping: A parallelizable shortest path algorithm[J/OL]. *J. Algorithms*, 2003, 49(1): 114-152. [https://doi.org/10.1016/S0196-6774\(03\)00076-2](https://doi.org/10.1016/S0196-6774(03)00076-2).
- [106] AMER A, LU H, BALAJI P, et al. Characterizing mpi and hybrid mpi+threads applications at scale: Case study with bfs[C/OL]//2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. 2015: 1075-1083. DOI: 10.1109/CCGrid.2015.93.
- [107] NASRE R, BURTSCHER M, PINGALI K. Data-driven versus topology-driven irregular computations on gpus[C/OL]//2013 IEEE 27th International Symposium on Parallel and Dis-

- tributed Processing. 2013: 463-474. DOI: 10.1109/IPDPS.2013.28.
- [108] WANG C, LU Y, ZHANG B, et al. An optimized bfs algorithm: A path to load balancing in mic[C/OL]//2015 IEEE International Conference on Computer and Communications (ICCC). 2015: 199-206. DOI: 10.1109/CompComm.2015.7387567.
- [109] JO Y Y, JANG M H, KIM S W, et al. A data layout with good data locality for single-machine based graph engines[J/OL]. IEEE Transactions on Computers, 2022, 71(8): 1784-1793. DOI: 10.1109/TC.2021.3107725.
- [110] CHEN Z, ZHANG F, GUAN J, et al. Compressgraph: Efficient parallel graph analytics with rule-based compression[J/OL]. Proc. ACM Manag. Data, 2023, 1(1). <https://doi.org/10.1145/3588684>.
- [111] YASUI Y, FUJISAWA K, SATO Y. Fast and energy-efficient breadth-first search on a single numa system[C]//KUNKEL J M, LUDWIG T, MEUER H W. Supercomputing. Cham: Springer International Publishing, 2014: 365-381.
- [112] YASUI Y, FUJISAWA K. Fast and scalable numa-based thread parallel breadth-first search [C/OL]//2015 International Conference on High Performance Computing & Simulation (HPCS). 2015: 377-385. DOI: 10.1109/HPCSim.2015.7237065.
- [113] SONG W, XIAO Z, WANG Y, et al. Session-based social recommendation via dynamic graph attention networks[C/OL]//WSDM '19: Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining. New York, NY, USA: Association for Computing Machinery, 2019: 555-563. <https://doi.org/10.1145/3289600.3290989>.
- [114] AGARWAL M K, RAMAMRITHAM K, BHIDE M. Real time discovery of dense clusters in highly dynamic graphs: identifying real world events in highly dynamic environments[J/OL]. Proc. VLDB Endow., 2012, 5(10): 980-991. <https://doi.org/10.14778/2336664.2336671>.