AeonG：图数据库中的高效内置时态支持（扩展版）

Jiamin Hou Zhanhao Zhao Zhouyu Wang Wei Lu 中国人民大学 中国人民大学 中国人民大学 中国人民大学 jiaminhou@ruc.edu.cn zhanhaozhao@ruc.edu.cn zyu_wang@ruc.edu.cn lu-wei@ruc.edu.cn

Guodong Jin 滑铁卢大学
g35jin@uwaterloo.ca

Dong Wen 新南威尔士大学，澳大利亚
dong.wen@unsw.edu.au

Xiaoyong Du 中国人民大学
duyong@ruc.edu.cn

arXiv:2304.12212v2 [cs.DB] 1 Apr 2024

摘要

现实世界图通常具有动态性，并随时间演变。在图数据库中存储图的演变并对其进行查询至关重要。然而，现有工作要么存储开销高，要么缺乏高效的时态查询支持，或两者兼有。本文提出 AeonG，这是一种内置时态支持的新型图数据库。AeonG 基于一种新颖的时态图模型。为适配该模型，我们设计了存储引擎和查询引擎。我们的存储引擎采用混合架构：一个当前存储用于管理图对象的最新版本，另一个历史存储用于管理图对象的先前版本。这种分离使查询最新图对象版本时的性能下降尽可能小。为降低历史存储开销，我们提出了一种新颖的锚点+增量 策略：定期为图对象创建完整版本（即锚点），并维护同一对象相邻两个锚点之间的每项变更（即增量）。为加速时态查询处理，我们在查询引擎中提出了一种基于锚点的版本检索技术，以跳过不必要的历史版本遍历。我们在真实数据集和合成数据集上进行了大量实验。结果表明，与最先进的方法相比，AeonG 的存储消耗最多降低 5.73×，时态查询延迟最多降低 2.57×，同时支持时态特性仅带来 9.74% 的性能下降。

PVLDB 引用格式：
Jiamin Hou、Zhanhao Zhao、Zhouyu Wang、Wei Lu、Guodong Jin、Dong Wen 和 Xiaoyong Du。《AeonG：图数据库中的高效内置时态支持（扩展版）》。PVLDB，17(6)：1515–1527，2024年。
doi:10.14778/3648160.3648187

PVLDB 工件可用性：
源代码、数据和/或其他工件已发布于 https://github.com/hououou/AeonG.git。

Wei Lu 是通讯作者。

本作品采用知识共享署名-非商业性使用-禁止演绎 4.0 国际许可协议授权。访问 https://creativecommons.org/licenses/by-nc-nd/4.0/ 查看该许可协议副本。如需进行超出该许可协议范围的使用，请发送电子邮件至 info@vldb.org 获取许可。版权所有者为所有者/作者。出版权授权给 VLDB基金会。《VLDB基金会会刊》，第 17 卷，第 6 期，ISSN 2150-8097。doi:10.14778/3648160.3648187

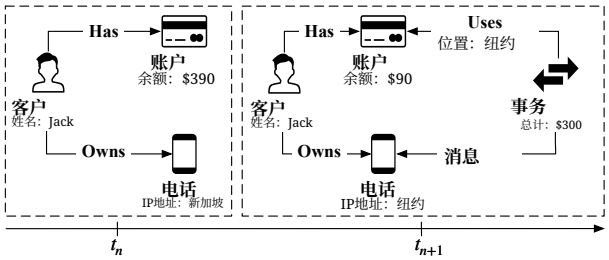


图1：客户购买图的演变

1 引言

图广泛用于建模现实世界实体之间的关系。许多图数据库，例如 Neo4j [1],ArangoDB [2],Dgraph [3], 和 Memgraph [4], , 都是为了高效管理图数据而开发的。尽管现实世界图通常具有动态性并会随时间演化，但这些数据库通常旨在管理最新的图数据：当图发生变化时，数据库只存储该图当前（最新）的状态，即顶点和边的最新值，同时丢弃任何先前的（历史）状态。然而，随时间演化的（时态）图数据包含图的最新状态和历史状态，在许多应用中都十分重要，例如金融欺诈检测 [5], 道路网络中的交通预测 [6], 等。

示例 1。图1展示了客户购买图的演变过程，其中客户、银行账户、手机和事务被建模为实体，实体之间的关系被建模为边。手机会在进行各种活动（例如接收消息或网页浏览）时记录其位置。每次客户购买都会调用一个事务来更新账户余额，并记录交易发生的位置。假设在时间 t_n，一位名为 Jack 的客户账户余额为390美元，其手机位置位于新加坡。我们存储了反映这一状态的图，如图1左侧所示。随后，在时间 t_{n+1}（即 t_n之后一分钟），Jack 发起了一笔总计300美元的购买事务，从而形成新的图状态，如图1右侧所示。该事务发生在纽约，与 Jack 手机所在的位置相同，因此看起来是合法的。然而，当比较 t_{n+1} 和 t_n的状态时，我们发现 Jack 的手机位置在一分钟内从新加坡变为纽约。考虑到 Jack 不可能如此迅速地跨越这么远的距离，这笔事务很可能是欺诈交易。我们想强调的是，仅手机位置的变化本身并不一定可疑，但

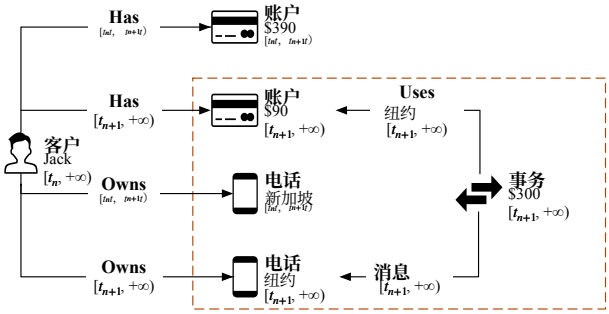


图 2：具有时间维度的客户购买图

当这种位置变化与一笔事务相关联时，就需要提高警惕。如上所述，只有通过追踪图结构随时间的演变，才能识别这种潜在的欺诈活动。因此，只维护t_{n+1}最新状态的传统图数据库将无法检测到此类欺诈交易。□

截至目前，已经提出了多种用于管理时态图数据的方法。若干研究 [7–13] 为图中的每个顶点或边分配时间戳属性，以反映其生命周期，如图 2 所示。这些方法并不在图发生变化时丢弃先前的状态，而是在同一图中同时维护当前状态和历史状态。例如，在时间 t_{n+1}，Jack 的两部电话同时存在，其中一部表示时间区间为 $[t_n, t_{n+1})$ 的先前状态，另一部表示具有 $[t_{n+1}, +\infty)$ 的当前状态。因此，如示例 1 所示，通过识别表明某笔事务的位置不一致的异常子图结构（红框高亮部分），这些方法能够检测欺诈性事务。然而，在这些将时间戳视为普通属性的方法中，执行时态查询（根据给定时间戳选择数据）通常需要遍历整个图。由于历史状态的增加，图规模不可避免地不断扩大，查询效率可能会随时间显著下降。另一类研究 [14–23] 通过定期物化整个图的快照，并记录两个连续快照之间的增量，来管理随时间变化的图数据。在这些方法中，查询历史状态首先需要根据时间戳确定最近的快照，然后使用该快照及相关增量重构完整的图状态。因此，这些方法由于需要维护完整快照而产生了大量存储开销，并且由于历史状态重构，可能导致查询性能欠佳。

我们的目标是设计一种图数据库，以高效管理时态图数据。然而，要实现这一目标，必须应对三个关键挑战，才能最大限度地降低存储开销并加快时态查询处理。首先，随着历史图状态的数量不断增加，要将存储开销降至最低并非易事❶。其次，面对大量历史状态，如何在保持数据一致性的同时高效处理时态查询并不简单❷。最后，数据库需要提供原生时态支持，使用户能够便捷地访问时态图数据❸。

本文提出了 AeonG，这是一种能够高效提供内置时态支持的新型图数据库。我们将广泛采用的静态属性图模型 [24–27] 与时间维度相结合，首先定义了时态属性图模型，以形式化表示和操作时态图数据。在该模型的指导下，我们进一步扩展常见的图数据库架构，设计出 AeonG。具体而言，我们增强了存储引擎、查询语言和查询引擎，使其具备高效的时态支持。我们构建了混合存储引擎❶，由当前存储和历史存储组成，以尽可能低的存储开销存储时态图数据。该引擎为每个顶点和边维护多个版本，最新版本保留在当前存储中，先前版本则存放在历史存储中。我们将时间维度融入数据布局，并基于多种现有图数据库 [2–4, 28–30]所采用的多版本存储引擎构建当前存储。我们提出了一种新颖的“锚点+增量”策略，用于在历史存储中紧凑地组织历史数据。具体而言，我们定期为图对象创建一个完整版本（即锚点），并维护同一对象相邻两个锚点之间的每一项变更（即增量），以减少历史存储开销。此外，我们引入异步迁移机制，将过时版本从当前存储转移至历史存储。我们不在每次顶点/边更新或删除时同步迁移先前版本，而是将迁移推迟到数据库定期执行垃圾回收时进行 [31]。该机制可确保迁移不会造成干扰，从而减少维护时态数据导致的性能下降。

随后，我们提出了一种时间增强查询语言❸，将 Cypher [25], 这一常用的图查询语言进行扩展，以便便捷地访问时态图数据。在混合存储引擎的基础上，我们引入了内置的时态查询引擎❷。我们继承了现有图数据库中的两种基本操作，即扫描和扩展，并对其进行扩展，以实现一致且高效的时态查询处理。我们提出了一种独特的基于锚点的版本检索技术，以最大限度地减少扫描和扩展算子中的不必要历史版本遍历。具体而言，我们直接定位与给定查询条件相匹配的最近锚点，并在所获得的锚点上应用后续增量，以重构所需版本，从而最大限度地降低历史版本遍历开销。

总而言之，我们作出以下贡献：

- 我们提出了 AeonG，这是一种提供高效内置时态支持的新型图数据库。AeonG 构建于时间增强查询语言、查询引擎和存储引擎之上，将时态特性视为一等公民，使操作时态图数据变得简单直观。
- 我们提出了一种混合存储引擎，采用具有“锚点+增量”策略的独立存储引擎，以减少历史数据的存储开销。我们进一步引入了一种异步迁移策略，以最大限度地降低维护时态图数据所造成的性能下降。
- 我们设计了一种时态查询引擎，采用基于锚点的版本检索技术，以提供一致且高效的

时态查询处理，同时将历史版本遍历开销降至最低。

• 我们基于 Memgraph [4], 这一真实世界中的原生图数据库实现了 AeonG。我们在真实数据集和合成数据集上进行了广泛实验，并将 AeonG 与两个最先进的时态图数据库 [7, 19]进行了比较。结果表明，在支持时态特性的同时，AeonG 可使存储消耗降低至多 5.73×，并使时态查询的延迟降低至多 2.57×，而性能下降仅为 9.74%。

本文其余部分的结构如下。下一节对时态图模型进行形式化，并介绍时态查询语言。第 3 节概述 AeonG 的架构。第 4 节详细介绍混合存储引擎，第 5 节阐述时态查询引擎。第 6 节介绍系统实现。第 7 节展示实验结果。第 8 节讨论相关工作，第 9 节总结全文。

2 建模与查询语言

本节对时态图模型进行形式化，并介绍 AeonG 中使用的时态查询语言。

2.1 时态属性图模型

我们通过在静态属性图模型 [24–27] 中加入时间维度来定义时态图模型。在属性图模型中，现实世界实体表示为顶点，这些实体之间的关系建模为边。每个顶点或边都有一个唯一标识符（简称 id），可以有多个标签（例如客户、电话）以及属性（例如姓名：Jack）。

定义 1（属性图）。令 N 和 E 分别表示顶点标识符集合和边标识符集合。设 L、K 和 V 是可数的标签、属性名称和属性值集合。属性图是一个元组 $G = N,\ E,\ \rho,\ \lambda,\ \pi$，其中：

• N 是 N 的有限子集，其元素称为 G 的顶点；

• E 是 E 的有限子集，其元素称为 G 和 $N \cap E = \emptyset$ 的边；

• ρ: $E \to (N \times N)$ 是一个将每条边映射到其源顶点和目标顶点的全函数；

• λ：$(N \cup E) \to 2^{\mathcal{L}}$ 是一个将顶点和边映射到有限标签集（包括空集）的全函数；

• π：$(N \cup E) \times \mathrm{K} \to \mathrm{V}$ 是一个偏有限函数，将顶点/边和属性键映射到一个值。

属性图模型最初是为静态图设计的，缺乏捕捉图随时间演化的固有能力。在关系数据库的语境中，“Trans-”这一概念提出了操作时间（Action Time）[32]，为关系模型引入时间维度。该事务时间由数据库系统本身创建和维护，用于跟踪系统中每个数据项的生命周期。受事务时间启发，我们将时间维度整合到属性图模型中，以正式定义时态属性图模型。

定义 2 (时态属性图)。时态属性图是一个元组 $G = \Omega,\ N,\ E,\ \rho,\ \lambda,\ \pi,\ \sigma,\ \tau$，其中：

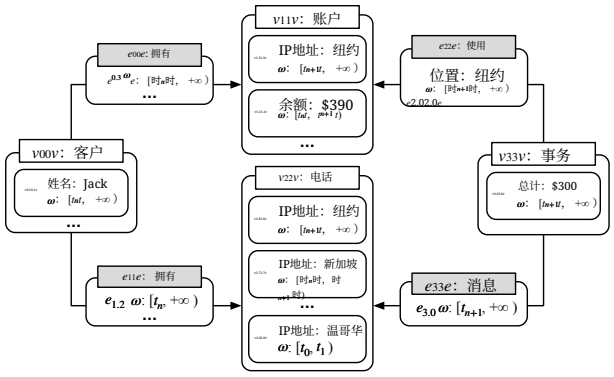


图 3：时态属性图示例

• Ω 是一个时态域，由一组连续的时间戳构成，即，对于某些 $a,\ b \in T$，有 $\Omega = \{i \in T | a \le i \le b\}$，使得 $a \le b$。T 表示时间点全集；

• N, E, ρ, λ, π 继承定义 1 中的定义；

• $\sigma: (N \cup E) \times \Omega \to \{true, false\}$ 是一个全函数，它将顶点或边以及一个时间段 ω 映射到一个布尔变量，以指示该顶点或边在时间段 ω 内是否存在；

• τ：$(N \cup E) \times \mathrm{K} \times \Omega \to \mathrm{V}$ 是一个偏函数，它将顶点或边、属性键和时间段 ω 映射到一个值。

约束。在我们的时态属性图模型中，我们施加了两个约束，以确保图在任意时间点都对应于一个有效的属性图。在任意时间点 t：（1）仅当源顶点和目标顶点在 t 存在时，一条边才存在。形式化地，若 $e \in E, \sigma(e, t) = true$，且 $\rho(e) = (v_1, v_2)$，则 $\sigma(v_1, t) = true$ 且 $\sigma(v_2, t) = true$；（2）属性只能在相应顶点或边存在的时间段内取值。形式化地，若 $\tau(o, k, t) = val$，其中 $o \in (N \cup E)$、$k \in \mathrm{K}$、$val \in \mathrm{V}$，则 $\sigma(o, t) = true$。

在我们的模型中，每个图对象由多个对应的版本组成，包括一个当前/最新版本，以及可能存在的若干历史版本。不同于 T-GQL 等现有研究为每个图对象分配时间段，我们的模型为图对象（顶点或边）的每个版本分配时间段。例如，如图 2 所示，考虑更新实体“电话”的情况。在现有模型中，此更新会导致在同一图中保留两个完整的“电话”顶点，从而产生两条冗余且未发生变化的“拥有”边。相比之下，我们会创建“电话”顶点的新版本，并将“拥有”边重新连接到该版本，同时将发生变化的属性存储在历史版本中。因此，我们的模型通过避免创建冗余的顶点和边，降低了复杂性并提高了效率。在任意给定时间 t，具有 $\omega = [st, ed)$ 的图对象版本被称为**合法的** 当且仅当 $st \le t < ed$。如果图对象版本在当前时间合法，我们将其归类为当前版本；否则，将其归类为历史版本。

图操作。我们的时态模型支持以下多种图操作。假设这些图操作由一个在时间 t_1 提交的事务发出。

• 创建顶点或边：这涉及添加一个具有当前版本且时间段为 $\omega = [t_1, +\infty$ 的顶点或边）。

列表 1：融入时态特性的 Cypher 表达式、查询和子句的语法

```
// 表达式
表达式 ::=值 |变量 | f(表达式_列表) v ∈V a ∈A f ∈F                                                  值/变量
        |表达式。k| {} | {属性_列表}                                                                  maps
        |[] | [表达式_列表] | expr IN 表达式 |表达式[表达式] | 表达式[表达式..] | 表达式[..表达式] | 表达式[表达式..表达式]   列表
        |表达式START WITH 表达式 |表达式ENDS WITH 表达式 |表达式CONTAINS expr                                  字符串
        |表达式 OR 表达式| 表达式 AND 表达式| 表达式 XOR 表达式| 非 表达式 |表达式为空 |表达式不为空                  逻辑
        |表达式 < 表达式| 表达式 <= 表达式| 表达式 >= 表达式| 表达式 > 表达式 |表达式 = 表达式 | 表达式 <> expr                 不等式
表达式_列表 ::=表达式 | 表达式 , 表达式_列表                                                          表达式列表

// 查询
查询 ::=查询° | 查询 并集 查询 | 查询 全部并集 查询                                                  联合查询
查询°::=返回 返回项 | 子句 查询°                                                                  子句序列
返回项 ::= * | expr [AS a] | 返回项 , expr [AS a]                                                  返回列表

// 子句
子句 ::= [可选] 匹配 模式_元组 [WHERE 表达式] [FORTTASOF表达式 |FORTTFROM表达式TO表达式]                 匹配子句
        |WITH 返回项 [WHERE 表达式] | 展开 表达式 作为 a a ∈ 𝒜                                        关系子句
模式_元组 ::= 模式 | 模式 , 模式_元组                                                                 模式元组
```

- 删除当前版本为 $[st, +\infty$ ω $[st, t_1$ $\omega =$ 的顶点或边：这涉及更新为）。
- 更新一个当前版本为 with$\omega = [st, +\infty)$ 的顶点或边：这会将当前版本标记为历史版本，方法是将 ω 更新为 $[st, t_1)$，并生成一个具有 $\omega = [t_1, +\infty)$ 的新当前版本，以表示最新语义。

示例 2。根据定义 2，我们在图 3 中展示示例 1 对应的时态属性图。这里，$\Omega = [t_0, +\infty]$, $N = [v_0, v_1, v_2, v_3]$, $E = [e_0, e_1, e_2, e_3]$。每个顶点和边都拥有一个当前版本，以及从 t_0 到 t_n的若干历史版本。为简洁起见，我们省略了 t_n之前的图状态。在 t_n，存在三个顶点（v_0、v_1 和 v_2）以及两条边（e_0：（v_0，v_1））和 e_1：（v_0，v_2））。例如，v_2 拥有一个当前版本 $v_{2.7}$，其唯一 ID 为 2，标签为“电话”，包含键值对（IP地址，新加坡）的属性，以及生命周期 $\omega = [t_n, \infty)$。随后，在 t_{n+1}，假设发生了一笔客户购买事务。该事务更新了 v_1 和 v_2的属性，从而为它们分别生成新版本。以 v_2 为例：它将 $v_{2.7}$ 标记为历史版本，并生成一个新的当前版本 $v_{2.8}$。具体而言，τ 将 $v_{2.8} \times IP \times [t_{n+1}, +\infty)$ 映射为纽约，并将 $v_{2.7} \times IP \times [t_n, t_{n+1})$ 映射为新加坡。我们认为 $v_{2.7}$ 在 t_n时合法，但在 t_{n+1}时不合法。此外，该事务还创建了 v_3、e_2 和 e_3，它们仅拥有具有 $\omega = [t_{n+1}, +\infty)$ 的当前版本。上述所有图操作均遵循已定义的约束。例如，只有在验证关联的顶点 v_3 和 v_1 于 t_{n+1} 时存在后，才能在 t_{n+1} 成功创建 e_2（约束 1）。□

2.2 时间图查询语言

AeonG 集成了增强时态功能的 Cypher [25],，扩展了 OpenCypher [25] 中定义的标准语法，以支持时态查询。如列表 1 所示，AeonG 在 MATCH 子句中引入了两种时态语法扩展（列表 1 中带下划线标出的部分）：(1) FOR TT AS OF t，用于检索在时间 t 合法的所有图对象；(2) FOR TT FROM t_1 TO t_2，用于定位在从 t_1 到 t_2 的时间范围内始终合法的所有图对象。前者称为“时间点”查询，后者称为“时间切片”查询。用户可以为时态查询应用任意时间条件，时间范围可从最早的历史记录一直延伸到最近的更新。此外，除了使用时态查询检索用户感兴趣的时态图数据外，AeonG 还允许用户使用标准 Cypher 语法提交常规（非时态）查询和数据操作（创建、更新和删除）。

示例 3. 考虑查询“Jack 在 t_n 时的手机 IP 地址是什么”。该查询可以通过执行以下语句来回答，其中时态语法带有下划线：“MATCH (:Customer name: ‘Jack’)-[r]-(p:Phone) FOR TT AS OF t_n 返回 p.IP。”

---

3 系统架构

本节介绍 AeonG 的系统架构，如图 4 所示。AeonG 包含一个事务管理器，用于支持对一系列图操作进行具有 ACID 属性的处理。我们通过采用多版本并发控制（MVCC） [33]来处理事务。MVCC 确保事务只能看到对其**可见** 的数据一致快照，从而使多个事务能够并发工作而不会相互干扰 [33–38]。鉴于我们主要关注时态数据管理，下面将介绍如何利用 MVCC机制有效管理时态数据。AeonG 通过两个主要组件提供内置时态特性：存储引擎和查询引擎。

3.1 存储引擎

AeonG 的存储引擎拥有两个物理隔离的存储：当前存储和历史存储。当前存储通常维护图对象的当前版本。相比之下，历史存储管理图对象的历史版本，这些历史版本会从当前存储异步迁移而来。

当前存储。如第2.1节所述，图会因各种图操作而发生变化。为高效记录这些变化，AeonG 将当前存储构建为多版本存储，为每个图对象维护多个版本。每个图对象都包含一个保存最新状态的当前版本，并关联一个历史版本列表，用于保存先前的状态。更新图对象时，我们不会直接覆盖数据，而是创建一个新的当前版本，并将先前的版本移至历史版本列表中。我们进一步将时间维度

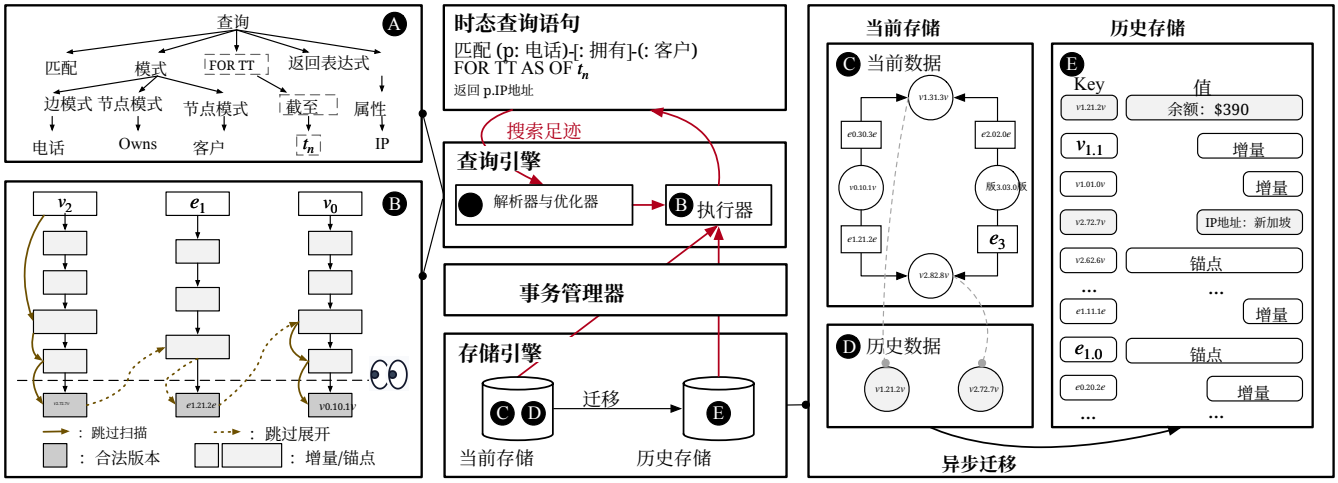


图 4：AeonG 概览 AeonG - AeonG 由时态查询引擎、混合存储引擎和基于 MVCC 的事务管理器组成。我们采用“锚点+增量”策略来降低历史存储开销，同时使用基于锚点的版本检索技术，以确保高效的时态查询处理。

融入数据布局和修改范式，以准确追踪图演化。详情将在第 4.1 节中介绍。

历史存储。 AeonG 不会将历史版本永久存储在当前存储中，而是将其迁移到历史存储中进行长期维护。为处理可能大量的历史数据，我们对历史存储进行了适当压缩。我们以键值格式组织迁移后的历史版本。键包含历史版本的元数据，包括顶点/边 ID 和版本的生命周期 ω，而值则保存该版本的详细属性。我们不为每个版本保留所有属性，而是以“锚点+增量”的方式组织版本。我们使用增量记录后续版本之间的相对差异，从而最大限度地降低不断增长的历史数据的存储成本。此外，在一系列增量之后，我们维护一个锚点，以保留图对象的完整状态，从而在执行时态查询时便于重构。具体细节将在第 4.2 节中介绍。

异步迁移。 AeonG 采用异步迁移方式，将历史数据从当前存储转移到历史存储。该迁移不会在更新后立即触发，而是延迟到多版本并发控制的垃圾回收期间执行。这种设计确保不断增长的历史数据转移过程保持轻量，从而最大限度地降低其对当前存储的开销。我们将在第 4.2 节介绍异步迁移。

示例 4. 在图 4 的右侧，我们展示了 AeonG 如何存储示例 2 中所示的客户购买图。在当前存储中，组件 记录了 t_{n+1} 处的当前版本。此外， t_n 处的历史版本（本例中为$v_{1.2}$ 和 $v_{2.7}$）存储在组件 中。以 v_2 为例。为了记录 v_2的 IP地址在t_{n+1}从新加坡变为纽约这一变化，AeonG 执行两个步骤。首先，它原地更新 v_2 ，以创建一个新的当前版本 $v_{2.8}$。其次，为了保留先前的状态，AeonG 会生成一个历史版本 $v_{2.7}$，并将其链接到 $v_{2.8}$ 形成一条链，由多版本并发控制进行管理。我们将历史数据从组件 异步迁移到历史存储（组件 ）。在历史存储中，历史版本$v_{1.2}$ 和 $v_{2.7}$被组织为一个锚点（以长矩形表示）和一个增量（以短矩形表示）。□

3.2 查询引擎

查询引擎负责处理用户发出的查询，并从混合存储引擎中检索相关图数据。遵循“教科书式”的组件分离原则，AeonG 由解析器、优化器和执行器组成。在继承现有图数据库中的这些组件的同时，AeonG 进一步扩展了它们，以支持时态查询。

解析器和优化器。解析器翻译查询，并生成供查询优化器使用的相应语法树。为了适应时态查询的语法，AeonG 扩展了词法、语法和语义分析，使其能够识别第 2.2 节定义的时间限定符。优化器利用生成的语法树，为执行器生成执行计划。

执行器。 AeonG 在传统图数据库的两项核心基本操作——扫描和扩展——之上构建并加以扩展。扫描算子为每个查询检索所需的顶点版本，而扩展算子则获取相关的边和相邻顶点版本。我们增强了这些算子，以便一致且高效地处理时态查询。为确保查询结果一致，我们分别从两个存储引擎获取当前数据和历史数据，再将结果合并。对于当前数据，我们遵循传统的查询机制：仅在当前存储上执行计划，并通过多版本并发控制的快照可见性检查捕获可见的图对象版本 [33]。然而，仅从历史存储访问历史数据可能会导致结果不完整。由于异步迁移，部分数据仍保留在当前存储中。为解决这一问题，我们引入了一种合法性检查机制，从两个存储中检索相关数据。该机制会验证

某个版本在给定时间条件下是否合法，以提取适当的版本。请注意，从当前存储中检索历史版本时，需要进行快照可见性检查。这些步骤确保所请求的版本来自一致快照，从而保证一致性。

为了从大量历史数据中高效遍历历史版本，我们提出了基于锚点的版本检索技术，以最大限度地减少不必要的遍历。对于扫描算子，我们获取满足所提供时态条件的相关顶点版本。为了重构所需版本，我们直接定位生命周期为 ω 且与查询时间约束一致的最近锚点。随后，我们从获取的锚点开始遍历后续增量，并应用所有符合条件的增量。对于扩展算子，我们利用已获取的顶点版本，直接定位相应的边和相邻顶点锚点，从而进一步消除不必要的遍历。更多详细说明请参见第 5 节。

示例 5. 图 4 展示了给定时态查询语句的简化语法树。基于此，AeonG 随后利用执行器从混合存储引擎中获取查询结果。图 4 展示了给定查询语句的搜索足迹，以回答“截至 t_n所有客户的手机 IP 地址是什么”。我们只重构四个相关的图顶点/边。我们首先扫描感兴趣的顶点 v_2。我们跳转以寻找其最近锚点，并收集所有相关增量，从而重构所需的合法版本 $v_{2.7}$。随后，我们扩展 $v_{2.7}$，获取其关联边 $e_{1.2}$ 及相邻顶点 $v_{0.1}$，而无需遍历整个版本链中的 e_1 和 v_0。□

4 混合存储引擎

本节将详细介绍 AeonG 的混合存储引擎设计。

4.1 当前存储

继承现有原生图数据库的设计，[39],AeonG 将图数据组织为三个存储组件：(i) 顶点属性（VP）、(ii) 边属性（EP）以及 (iii) 图拓扑，即顶点的入边和出边（VE）。与大多数原生图数据库一样，[1, 4, 40], 我们将拓扑保存在顶点内，从而能够对每个顶点进行快速的邻域遍历。然而，在这种设计下记录图演化并非易事。图不仅可能在语义上发生变化，即图对象的属性发生变化，也可能在结构上发生变化。我们必须识别不同类型的应用于图的操作。例如，当顶点相关的图拓扑发生变化但其属性保持不变时，我们应避免创建新的顶点版本。为了解决这一问题，我们将时间维度与每个独立的存储组件关联起来，以分别记录语义变化和结构变化。

数据布局。 如图5所示，数据存储由两个组件构成：顶点存储，用于维护顶点对象列表；以及边存储，用于存储边对象列表。每个顶点对象 v 都拥有唯一的图标识符 Gid、一个 VP 部分、一个 VE 部分，以及一个指向历史版本链表（版本链）的指针。每个边对象 e 则拥有唯一标识符 Gid、一个 EP 部分，以及一个指向历史版本链表的指针。具体而言，

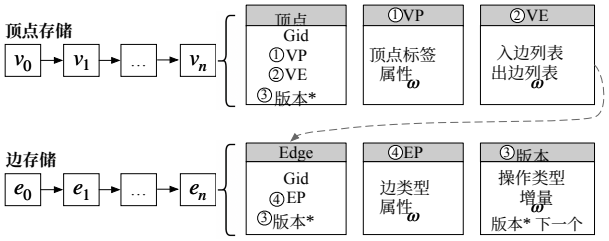


图5：当前存储的数据布局

• VP 部分存储与 v当前版本相关联的一组顶点标签和属性值对，以及一个时间段 ω，用于指示 v当前的语义生命周期。

• VE 部分跟踪 v的入边和出边的当前版本，其中每个条目都是由（边 Gid、邻接顶点 Gid）组成的列表。VE 部分还包括 ω，用于记录 v当前的结构生命周期。

• EP 部分存储 e当前版本的边类型和属性值对，以及一个时间段 ω，用于表示 e当前的语义生命周期。

• 每个历史版本包含：用于指示对 VP 部分、VE 部分或 EP 部分所做更改的操作类型；用于记录撤销更改并恢复先前版本步骤的增量；用于记录历史版本生命周期的时间段 ω；以及指向下一个历史版本的指针。根据多版本并发控制机制，同一事务生成的所有历史版本都会聚集在一个撤销缓冲区中。

修改范式。 现在我们讨论 AeonG 如何演化图，以处理各种图操作。假设图操作由事务 T_i 调用，其提交时间为 t_i。图数据布局上的修改范式如下。(1) 创建顶点时，我们创建一个顶点对象，将其 VP 部分的时间设置为 $\omega = [t_i, +\infty)$，将其 VE 部分的时间设置为 $\omega = [-\infty, +\infty)$，并将其链接到顶点对象列表中。(2) 创建边时，我们创建一个边对象，将其 EP 部分的时间设置为 $\omega = [t_i, +\infty)$，并将其链接到边对象列表中。我们还通过将相关顶点对象的 VE 部分时间设置为 $\omega = [t_i, +\infty)$，在它们先前的 VE 部分时间为 $[-\infty, +\infty)$ 时，与相关顶点对象建立连接。(3) 使用 $\omega = [t_j, +\infty)$ 更新、创建或删除顶点对象的属性值时，我们首先更新 VP 部分中的相关属性值，并将其时间设置为 $\omega = [t_i, +\infty)$。接下来，我们创建一个历史 VP 版本，用于记录修改前顶点的状态，并将其 ω设置为 $[t_j, t_i)$。随后，该 VP 版本会被链接到顶点的版本链中。更新边的属性值遵循相同的逻辑。(4) 删除顶点时，我们首先按照 (3) 范式删除相关顶点对象的所有属性值，然后按照 (5) 范式删除所有相连的边。(5) 删除边时，我们将其分解为删除所有属性值，以及删除与相关顶点的连接。前者作用于边对象，并遵循 (3) 范式。后者作用于源顶点对象和目标顶点对象。以使用 $\omega = [t_j, +\infty)$ 的源顶点对象为例。我们首先更新其 VE 部分，从

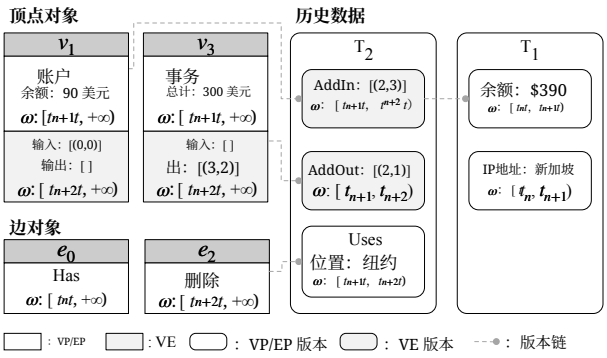


图 6：当前存储布局示例

出边列表中删除该边，并将其 VE 部分的时间设置为 $\omega = [t_i, +\infty)$。然后，我们创建一个 VE 版本，用于记录被删除的边及其 $\omega = [t_j, t_i)$，并将其链接到该边的版本链。

示例 6。我们说明当前存储的数据布局。如图 6 所示，我们重新考察示例 2。为了进一步表示结构变化，我们假设在 t_{n+2} 处发生了删除 e_2 的事件。我们关注两个顶点 v_1 和 v_3，以及两条边 e_0 和 e_2，以展示图演化。在 t_{n+1}，表示客户购买的事务 T_1 被提交。该事务更新了 v_1 和v_2 的 VP 部分，生成了与它们关联的两个 VP 版本，并创建了图对象 e_2、 e_3 和 v_3。由于篇幅限制，图 6 展示了除 e_3 和 v_2 之外的这些元素。在 t_{n+2}，提交了 T_2 以删除 e_2，这会影响 v_1、 v_3 和 e_2 对象。它首先作用于 e_2 的 EP 部分，清除所有语义信息，并生成一个 EP 版本以记录之前的边状态。然后，它作用于 v_1 和 v_3 的 VE 部分，并生成两个 VE 版本。□

4.2 历史存储

在多版本并发控制（MVCC）中，历史版本不会永久保留在当前存储中。相反，一旦任何活动事务都不再需要这些版本，就会通过垃圾回收（GC）安全地将其移除，以优化当前存储的性能。AeonG 利用这一机制，将无法访问的版本转移到历史存储中进行长期维护。为便于交流，当前存储中的历史版本称为“未回收”版本，而历史存储中的历史版本称为“已回收”版本。本小节首先介绍用于存储历史版本的优化键值格式，然后概述将未回收版本迁移到历史存储中的过程。

键值格式。已回收版本以键值格式组织，其中键表示版本的元数据，值存储相应的详细信息。AeonG 将三种类型的历史版本（顶点分区（VP）、边属性（EP）和顶点边（VE）版本）分别归入各自的数据段。在每个数据段中，键由三个元素组合而成：*Type* 前缀、图标识符 *Gid* 和版本的 ω。*Type* 前缀表示数据段中所含版本的类型：“V”表示 VP 版本，“E”表示 EP 版本，“VE”表示 VE 版本。*Gid* 是与该版本关联的图对象的唯一标识符，而

ω 表示该版本的生命周期。至于值字段，其中包含剩余的语义信息，即仅记录该版本相较于前一版本的数据变更的增量

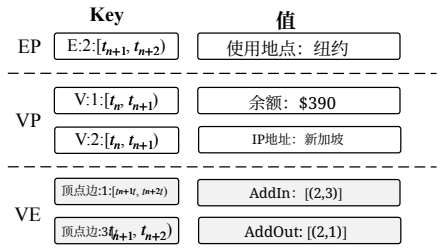


图7：历史存储中的键值格式

。通过这种方式组织数据，键中具有相同前缀的数据会在物理上聚集在一起，存储于跳表 [41]，中，这确保同一实体的不同版本会根据其生命周期 ω 自动排序。因此，按时间顺序检索变得高效。图 7 展示了图 6 中未回收历史版本的键值格式。

锚点+增量。我们利用增量来降低存储开销。然而，检索已回收的图对象需要将最新版本与所有先前的增量进行组装，对于较长的检索历史，这会产生显著的重构开销。为缓解这一问题，我们在增量数据中按一定间隔引入锚点，其中锚点表示图对象的完整状态。为了在键值存储中区分锚点和增量，我们在键的 *Type* 后附加一个一位字符后缀，其中“A”表示锚点，“D”表示增量。具体而言，要重构某个已回收版本o_1，我们查找其最新的锚点 o_2，收集从 o_2 到o_1的所有增量，并将它们组合起来以重构 o_1。

我们提出了一种自适应锚定方法，为不同的图对象分配不同的锚点间隔 u。较大的u 会使连续锚点之间产生更多增量，这可能增加查询延迟，但能够降低存储开销。因此，我们为更新频繁的对象分配较大的 u，以在查询延迟和存储效率之间取得平衡。给定一个图对象o，我们使用公式 1，根据其更新频率 f（即对 o执行的更新次数）o来确定其 u。

$$u_o = \begin{cases} \tau_1 * c & f(o) \le \tau_1 \quad \text{low frequency} \\ \tau_2 * c & \tau_1 < f(o) \le \tau_2 \quad \text{medium frequency} \\ \tau_2^2/\tau_1 * c & \tau_2 \le f(o) \quad \text{high frequency} \end{cases} \tag{1}$$

该公式使用两个阈值（τ_1 和 τ_2）将更新频率分为三个级别（低、中和高）。每个频率级别都分配了特定的锚点间隔，该间隔通过将相应的阈值与预定义参数 c 相乘，以启发式方式计算得出。目前，AeonG 支持用户在数据库初始化和运行时设置公式 1 中的参数，例如 τ_1。

数据迁移。在多版本并发控制中，当相关的提交事务不再处于活动状态时，未回收的历史版本将在异步垃圾回收阶段从当前存储中被物理移除。AeonG 会收集这些版本，并将其迁移到历史存储中进行长期维护，具体如算法 1 所示。我们使用 *undo* 维护待迁移的未回收版本（第 2 行），并使用 *kv* 以键值格式存储已回收数据（第 3 行）。*CT*中的每个未回收版本 *undo*最初都会被编码为键值对 *kv*（第 5 行）。随后，我们将其存储在历史键值存储中（第 6 行）。最后，我们锁定版本链中的 *undo* 并将其物理删除（第 7 行）。

算法 1： 数据迁移

```
1 函数 迁移(CT)：
      输入： CT，已提交事务不再处于活动状态；
2     undo ← ∅；//未回收版本；
3     kv ← ∅；//已回收版本；
4     对每个 undo ∈ CT 执行
5         kv=encode2KV(undo);
6         KV_store::put(kv);
7         物理删除撤销记录；
8 结束函数
```

5 时态查询引擎

AeonG 继承并扩展了扫描和扩展算子，以实现一致且高效的时态查询处理。

5.1 扫描算子

AeonG 使用扫描算子高效获取顶点版本，同时确保当前数据和历史数据的数据一致性。我们从每个存储组件获取数据的角度对其进行详细说明。从当前存储获取数据时，必须确保在存在并发事务的情况下能够一致地捕获数据。为此，我们首先定位感兴趣的相关顶点对象。对于每个顶点对象，我们首先使用快照可见性检查 [33]，以查找给定事务的可见版本。版本链中位于该可见版本之前的所有版本，都是我们可能需要的候选合法版本。然后，我们使用合法性检查机制，依据以下公式验证每个候选版本 v_i 对于给定查询时间条件是否合法。

$$\omega.st \le C.t_2 \wedge \omega.ed > C.t_1 \tag{2}$$

其中，$\omega.st$ 和 $\omega.ed$ 分别表示 v_i'的生命周期的开始时间和结束时间；C 表示给定查询的时间条件，其开始时间为 t_1，结束时间为 t_2。对于时间点查询，$t_1 = t_2$。

从历史存储中获取数据时，无需处理事务冲突，因为历史存储仅用于用户无法更改其中数据的只读查询。因此，我们直接采用合法性检查机制来获取所需版本。为进一步挖掘历史版本，我们采用基于锚点的跳过检索策略来重构所需版本。要还原特定合法版本v_i，我们通过探测前缀“AV:*id*:*C*”直接在键值存储中查找最近锚点 v，其中“AV”表示 VP 段中的锚点，id 是目标顶点的唯一 ID，C 是给定的查询时间约束。然后，我们将 v_i 与从 v 到 v_i的所有先前版本组装起来。得益于历史存储中特殊设计的键值格式，我们可以利用探测前缀快速找到最近锚点。

算法 2 展示了从混合存储引擎获取顶点的伪代码。我们从顶点对象 v 开始扫描，它要么是整个图的第一个顶点，要么是索引指向的顶点（第 2 行）。我们首先从当前存储中检索数据（第 5–8 行）。我们检查 v 及其历史未回收版本对当前事务是否可见（第 6 行）。我们还根据公式 1，使用 TemporalCheck() 函数检查它们是否合法（第 7 行）。接下来，

算法 2： 检索顶点

```
1  函数VertexRead(C)：
       输入： C，时间条件；
       输出： Σ，结果集；
2      v ← 要扫描的顶点；
3      当 v 时执行
4          // 从当前存储中获取；
5          对每个 v′ ∈ (v ∪ v.versions) 执行
6              如果 !SnapshotCheck(v′)则 继续；
7              如果 时间检查(v′.ω, C)则
8                  Σ ← Σ ∪重构(v′);
9          // 从历史存储获取;
10         从键值存储获取(v.id, C, Σ);
11         v ← v.next();
12     返回 Σ;
13 函数 从键值存储获取(id, C, Σ)：
14     kv_a ←KV_存储::seeknext(id, C);
15     kv_d ← KV_store::seeknext(id, kv_a.key);
16     while kv_d ∧ kv_d.ω.st ≤ C.t_2 do
17         kv_a ← 合并 (kv_a, kv_d);
18         if 时间检查(kv_d.ω, C) then
19             Σ ← Σ ∪ kv_a;
20         kv_d ← kv_d.next();
```

使用 FetchFromKV() 函数从历史存储中获取数据（第 10 行）。我们首先根据探测前缀找到最新的锚点 kv_a（第 15 行）。然后，我们查找所有满足时间检查条件的先前增量 kv_d（第 19 行），并将它们与 kv_a 组装起来，以获得所需版本（第 18 行）。

复杂度分析。扫描算子查询数据集中某个顶点 v 的版本，该数据集总计包含 n 个顶点。此过程包含两部分：(1) 定位 v 的当前版本；(2) 查询 v的历史版本。定位当前版本的复杂度取决于当前存储中选用的具体检索机制，例如，B$^+$-树索引查找的复杂度为 $log(n)$，非索引查找的复杂度为 n，记作 $O(\iota(n))$。查询历史版本的复杂度取决于引入时态特性的方式。在 AeonG 中，我们首先定位最近的锚点。由于我们使用跳表在键值存储中组织历史版本，此过程的复杂度为 $O(log(A_v))$，其中 A_v 表示顶点的平均锚点数。随后，我们从锚点开始依次扫描增量，直到满足查询时间条件；其时间复杂度为 $O(u)$，其中 u 表示公式 1 中定义的 u_o 的平均长度。总之，扫描算子的复杂度为 $O(\iota(n) + log(A_v) + u)$。

5.2 扩展算子

AeonG 利用扩展算子获取关联边和邻接顶点版本。扩展算子的总体设计思路与扫描算子类似，后者在两个独立的存储引擎中采用不同的检索策略。此外，扩展算子还考虑了图结构的检索。下面我们详细说明扩展算子如何获取给定顶点的边版本和邻居顶点版本。

算法 3： 扩展顶点

```
1 函数扩展顶点 (v, C)：
      输入： v，需要扩展的图顶点； C，时间
             条件；
      输出： Σ，结果集；
2     Σ_ve ← VERead(v, C) //获取邻接表；
3     对于每个 (e_id, nv_id) ∈ Σ_ve 执行
4         Σ_e ← EdgeRead(e_id, f(C, v.ω));
5         对于每个 e ∈ Σ_e 执行
6             Σ_nv ← VertexRead(nv_id, f(C, e.ω));
7             对于每个 nv ∈ Σ_nv do
8                 Σ ← Σ ∪ (e, nv) ;
9     返回 Σ;
```

如算法 3 所示，给定一个顶点 v，我们首先使用函数 VERead() 从 VE 部分/分段中访问 v 的邻接表版本（第 2 行），其中包含由（边ID e_{id}，邻居顶点ID nv_{id}）组成的列表。函数 VERead() 与算法 2 中的函数 VertexRead() 采用类似的逻辑，将当前数据与历史数据相结合，以获取所需版本。然后，我们根据边和邻接顶点的唯一 ID，获取特定语义信息的合法版本（第 3-8 行）。首先，我们使用 EdgeRead() 函数获取关联边版本，该函数遵循与 VertexRead() 类似的逻辑（第 4 行）。为加快关联边版本的搜索速度，我们优化了用于锚点定位的跳过查找策略，以跳过更多不必要的版本。根据第 2.1 节定义的约束 1，边对于其连接的顶点而言必须是合法的。这意味着，边版本的生命周期必须与其连接的顶点版本的生命周期相交。由于我们已经持有一个扫描得到的顶点版本 v，因此可以利用 v 的生命周期 ω，根据以下方程细化探测时间范围 C。

$$f(C, \omega) = [max(C.t_1, \omega.st), min(C.t_2, \omega.ed)] \tag{3}$$

通过实施这一方法，我们能够高效地跳过更多不必要的版本，从而提升查询性能。随后，我们基于 nv_{id}（第 5-6 行），以类似的逻辑过程获取每个所持有边版本的邻居顶点版本。最后，我们获得最终结果（第 7-8 行）。扫描和扩展算子的示例详见我们的扩展稿件 [42]。

复杂度分析。 扩展算子通过以下三个步骤检索特定顶点版本的关联边和相邻顶点。首先，我们获取邻接表版本。与扫描算子的复杂度类似，相关复杂度为 $O(log(A_{ve}) + u)$，其中 A_{ve} 是邻接表的锚点总数， u 是 u_o 的平均长度。接下来，对于邻接表中的每一对 (e_{id}, nv_{id})，我们获取对应的边版本。最后，我们定位邻居顶点版本。与第一步类似，这两个步骤的复杂度分别为 $O(log(A_e) + u)$ 和 $O(log(A_v) + u)$，其中 A_{ve} 和 A_v 分别是边和顶点的锚点总数。总之，扩展算子的总体复杂度为$O(log(A_{ve}) + u + D \times (log(A_e) + u + log(A_v) + u))$，其中 D 是顶点度数的平均值。我们希望强调，算法 2 和算法 3 的复杂度均处于 $O(log(n))$级别，这在图查询处理 [28, 43, 44] 中是合理且通常可接受的。

5.3 正确性的证明

现在，我们证明事务能够从两个相互隔离的存储引擎中获取一致的图数据。

定理 1. 给定一个只读事务 T，在条件时间 C下获取时态图数据，可以验证其已从具有一致状态的混合存储中检索到数据。

证明。首先，时态图数据在全局范围内是一致的。每个版本的生命周期 ω 是事务的提交时间，由系统根据全局时钟分配。因此，两个相互隔离的存储的时间线遵循相同的串行化顺序。其次，我们采用获取策略，根据生命周期 ω返回一致的请求数据。对于当前存储中的版本 o，我们验证其对 T的可见性，然后将 o的 ω 与 C 对齐（第 6-8 行，算法 2）。对于历史存储中的 o，我们仅将 o的 ω 与 C 对齐（第 10 行，算法 2）。通过这样做，我们可以保证每个独立存储中的数据一致性。某个版本 o 可能同时存在于当前存储和历史存储中。当前存储中的未回收版本可能会在异步迁移过程中从版本链中截断之前被转移到历史存储（算法 1）。然而，这不会导致错误，因为两个版本具有相同的生命周期 ω 并包含完全相同的信息。在这种情况下，我们执行去重步骤，仅保留当前存储中的版本。综合考虑所有数据获取场景，我们确保了整体数据一致性。□

6 实现

AeonG 构建于 Memgraph[4] 和 RocksDB[41]之上。Memgraph 是一种商业原生图数据库，支持属性图模型、Cypher 和多版本并发控制。我们利用并扩展 Memgraph，将其作为 AeonG 的主数据库引擎，为 AeonG 提供基本的查询引擎和当前存储引擎。随后，我们将流行的键值存储 RocksDB 集成到 Memgraph 中，作为历史存储，以管理历史数据。我们提出的方法通常适用于支持多版本并发控制的原生图数据库。

查询引擎。 AeonG 扩展了 Memgraph 查询引擎的 解析器 和 执行器 组件，以支持时态查询。AeonG 扩展 解析器，以识别第 2.2 节中定义的时态查询，将时态限定符纳入 Cypher.g4，并增强 CypherMainVisitor() 以识别时态限定符。此外，AeonG 通过修改两个基本算子来增强 执行器： 扫描算子和扩展算子。在 ScanAllCursor.Pull() 函数中，除了检索当前顶点外，我们还引入了 AddHistoricalVertices() 函数，以捕获未回收和已回收的历史版本（算法 2）。在 ExpandCursor.Pull() 函数中，我们进行了类似的调整，加入 AddHistoricalEdges() 函数，以获取历史边和邻接顶点（算法 3）。

存储引擎。 AeonG 的存储引擎采用混合架构，由当前 存储和历史 存储组成，具体内容详见第 4 节。当前 存储源自 Memgraph 的存储，其中 Vertex 结构映射到顶点对象，EdgeRef

表1：工作负载特征

	T-mgBench	T-LDBC	T-gMark			
			Bib	WD	LSN	SP
顶点数量	10K	3,181K	100K	103K	100K	100K
边数量	122K	17,256K	121K	93K	200K	385K
密度	12.17	5.42	1.2	0.90	2	3.85
顶点标签数	1	8	5	24	15	7
边标签数	1	25	4	82	27	7
图操作数 用于数据生成	320K	1M	320K	320K	320K	320K

结构映射到边对象，而 Delta 结构表示历史未回收版本。随后，我们将时间维度与这些结构关联起来，以引入时态支持，具体讨论见第 4.1 节。相关事务提交时，全局时钟会为其分配时间戳，时间粒度为毫秒。AeonG 将 RocksDB 集成到 Memgraph 中作为历史 存储：Memgraph 实例启动时，同时启动一个 RocksDB 进程。我们在 Memgraph 的 CollectGarbage() 函数中引入 Migrate() 函数，将未回收数据转换为键值对，随后迁移到 RocksDB（算法 1）。我们还使用 TiKV [45], 这一高效的分布式键值存储，通过将 RocksDB 的接口替换为 TiKV，实现了 AeonG 的分布式版本 AeonG-D，并将其用于历史存储。我们在 AeonG 中引入系统参数 retention_period，用于设置历史数据保留期限。例如，将 retention_period 设置为一个月后，系统便可定期从历史存储中移除一个月前生成的历史数据。

7 评估

本节首先介绍实验设置。随后，我们将 AeonG 与两个最先进的时间系统 Clock-G 和 T-GQL 进行比较，并使用两个指标对 AeonG 进行深入的性能分析：1）时态查询/图操作的延迟；2）时态图数据的存储开销。

7.1 实验设置

AeonG 构建于 Memgraph $v2.2.0$、RocksDB $v6.14.6$ 和 TiKV $v7.1.2$ 之上，用于评估。我们在一个最多包含 5 个节点的集群中运行实验。每个节点配备 32 个 Intel(R) Xeon(R) Gold 5220 CPU @ 2.20GHz、128 GB 内存，并运行 CentOS 7.9。

7.1.1 基准系统。 我们将 AeonG 与两个支持时态特性的基准系统进行比较：

T-GQL [7]: 一种最先进的图数据库，为每个图对象（顶点或边）分配一个时间段，以提供时态支持。由于我们无法获得 T-GQL 的源代码，为了进行公平比较，我们基于 Memgraph 实现了 T-GQL。需要注意的是，T-GQL 将所有顶点和边存储在内存中。

Clock-G [19]: 一种先进的图存储引擎，通过定期创建整个数据库的快照来管理时态数据。由于 Clock-G 未开源，我们将其时态数据管理方法实现于我们的代码库中，以便进行公平比较。我们在 RocksDB 中记录快照以及连续快照之间的日志。我们还引入了一个类似于 AeonG 所用的查询引擎，以支持时态查询。默认情况下，Clock-G 每执行 80k 次图操作后创建一个快照。

表 2：T-mgBench 的时态查询

查询	语句
Q1	匹配 (n: User {id: $id}) FOR TT AS OF t 返回 n
Q2	匹配 (n: User {id: $id}) FOR TT From t_1 to t_2 返回 n
Q3	匹配 (n: User {id: $id})-[e]->(m) FOR TT AS OF t 返回 n,e,m
Q4	匹配 (n: User {id: $id})-[e]->(m) FOR TT 从 t_1 到 t_2 返回 n,e,m

7.1.2 工作负载。我们使用三种增强了时态支持的工作负载进行实验，其特征详见表1。接下来我们将分别介绍这三种工作负载。

T-mgBench 基于现实世界的 Pokec 数据集 [46], , 该数据集用于 Memgraph 的 mgBench [47] 工作负载。如表 2 所述，T-mgBench 包含四个时态查询，它们是在 mgBench 的非时态查询中加入时间维度扩展而来的。具体而言，我们在 Q1 和 Q3 中加入“FOR TT AS OF t”，形成“时间点”查询；在 Q2 和 Q4 中加入“FOR TT FROM t_1 到 t_2”，形成“时间切片”查询。

T-LDBC源自 LDBC [48], 这一知名的合成图工作负载，方法是在 LDBC IS 查询（IS1-IS7）中加入“FOR TT AS OF t”。有关这些查询的更多细节，请参见附录中的表 I。

T-gMark 基于 gMark[49], 这一知名的合成图工作负载。如表 1 所示，它包含四个数据集。我们使用 gMark 的查询生成工具创建非时态查询，然后通过加入时间条件“FOR TT AS OF t”将其转换为时态查询。查询生成遵循 gMark 的默认配置，其中包括元数约束（0-4）、查询形状（25% 链、25% 星链、25% 环和 25% 星形）、选择率（ 33% 常量、 33% 线性和 33% 二次）、递归概率（50%）以及查询大小（[1,1], [3,4], [1,3], [2,4]）。

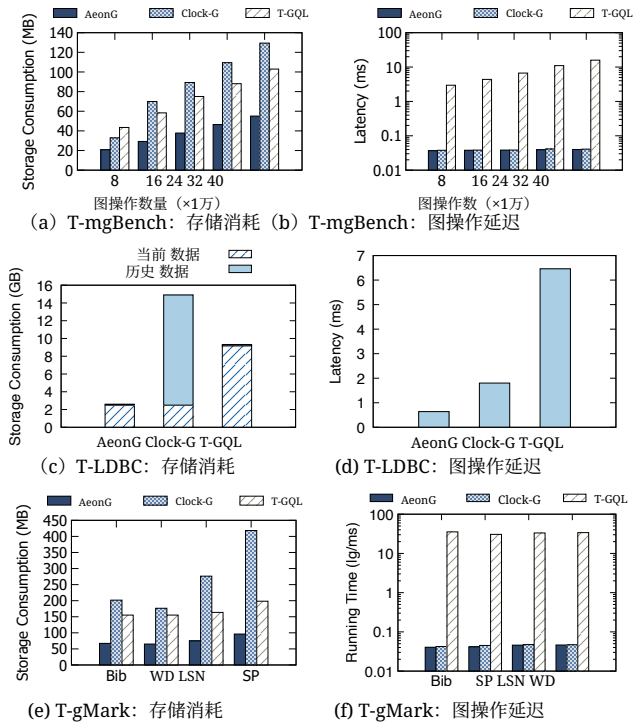
为了有效评估 AeonG 和基准系统中时态特性的效率，对于每种工作负载，我们都会在评估前生成额外的历史数据。除非另有说明，我们首先使用原始工作负载中的数据生成工具创建初始数据集，然后执行由 80% 更新、10% 创建和 10% 删除组成的图操作，以生成历史数据。更新操作和查询的访问分布遵循 Zipf [50] 分布，以模拟真实世界中的图操作场景。

7.1.3 默认配置。 默认情况下，Zipf 分布因子设置为 1.1。公式 1 中定义的自适应锚定方法的参数配置为 $\tau_1 = 1k$、$\tau_2 = 10k$ 和 $c = 1\%$。第 6 节讨论的保留_期限设置为 0，表示所有历史数据都将永久保留。

7.2 AeonG 与基准系统

现在，我们将 AeonG 与两个基准系统 Clock-G 和 T-GQL 进行比较。由于 T-GQL 是内存数据库，为确保比较公平，我们通过将 RocksDB 的 MemTable 大小配置为 640MB，确保 AeonG 和 Clock-G 中的所有数据都缓存在内存中。

7.2.1 存储消耗实验。我们首先使用 T-mgBench 进行实验，并在图8(a)中绘制了不同图操作数量下各系统的存储消耗。如图所示，随着图操作数量的增加，与 Clock-G 相比，AeonG 最多可减少 2.4× 的存储消耗；与 T-GQL 相比，最多可减少 2.09× ，其中



（a）T-mgBench：存储消耗（b）T-mgBench：图操作延迟

（c）T-LDBC：存储消耗 (d) T-LDBC：图操作延迟

(e) T-gMark：存储消耗 (f) T-gMark：图操作延迟

图8：存储消耗和图操作延迟对比

AeonG 的存储消耗最低。这得益于我们的“锚点+增量”策略：该策略将大部分历史图数据以增量形式紧凑存储，从而最大限度地降低维护时态数据所需的存储消耗。相比之下，T-GQL 的图模型使其无法紧凑地存储历史图数据，而 Clock-G 则会定期创建整个图的历史快照。这两个系统产生的存储开销都高于 AeonG。

我们还使用 T-LDBC 和 T-gMrak 开展实验，以评估这些系统的存储消耗。如图8(c)所示，在 T-LDBC 下，与 Clock-G 和 T-GQL 相比，AeonG 的存储消耗最多分别降低了 5.73× 和 3.59×。此外，如图8(e)所示，在 T-gMrak 下，AeonG 的存储消耗最多分别比 Clock-G 和 T-GQL 低 4.34× 和 2.39×。这一趋势与图8(a)中的观察结果一致，表明在处理大型且复杂的图工作负载时，AeonG 仍能实现更低的存储开销。

7.2.2 图操作延迟实验。随后，我们使用 T-mgBench，在图操作数量不断变化的情况下评估图操作延迟。如图8(b)所示，AeonG 的性能与 Clock-G 相近，但明显优于 T-GQL，最高可达 397.06×。随着图操作数量增加，从 80k 增长到 400k 时，AeonG 和 Clock-G 的性能下降均为 5.4×，而 T-GQL 的性能下降幅度则大得多，达到 34.95×。性能差异主要是因为 T-GQL 没有将当前数据和历史数据分开存储。因此，更新等图操作需要遍历更多的图对象（包括当前数据和历史数据）才能找到要更新的特定图对象。然而，AeonG 将当前数据和历史数据分开存储，因此延迟开销要小得多。在 T-mgBench 中，AeonG 与 Clock-G 的性能相近，这是因为在处理规模相对较小的图时，创建快照的开销并不显著。因此，为了进一步评估更大规模图上的存储操作延迟，我们使用 T-LDBC 工作负载开展了额外实验。如图8(d)所示，AeonG 的图操作延迟比 Clock-G 和 T-GQL 分别最高低 2.82× 和 10.11×。由于 T-LDBC 的规模更大，Clock-G 需要额外的 CPU 和 IO 资源来定期创建大型历史快照，这可能会因资源竞争而对图操作性能产生负面影响。如图8(f)所示，AeonG 的性能与 Clock-G 相近，并明显优于 T-GQL，最高可达 766×。这一趋势与图8(b)中观察到的趋势一致。

7.2.3 时间查询延迟实验。现在，我们分析不同配置下时态查询的性能。首先，我们针对 T-mgBench 中的各种时态查询开展性能评估，并在图9(a)中绘制不同时态查询的查询延迟。默认情况下，我们将 Q2 和 Q4 的时间片长度设置为 100s。可以观察到，与 Clock-G 相比，AeonG 将查询延迟降低了 2.57×；与 T-GQL 相比，降低了 37.57×。AeonG 的优异性能得益于其内置查询引擎，该引擎采用高效的基于锚点的版本检索技术，避免了不必要的版本遍历。相比之下，为访问所需的历史图元素，T-GQL 必须遍历整个图，而 Clock-G 则需要获取相应的历史快照并在其上追加日志，因此性能较慢。

随后，我们研究了随着图操作数量变化，时态查询 Q1 的延迟。图 9(b) 显示，AeonG 的性能最高可比 Clock-G 提升 1.43×，相比 T-GQL 则最高可提升 47.18×。随着图操作数量增加，这一性能差距变得愈发明显。如前所述，得益于所提出的优化时态查询引擎，AeonG 展现出更优的性能。我们进一步评估了“冷”“温”和“热”查询下 Q3 的查询延迟。我们根据这些查询所访问的顶点对其进行分类。例如，访问“热”顶点的查询被归类为“热”查询。我们根据访问可能性将数据库中的所有顶点划分为“冷”“温”和“热”三类，该访问可能性基于 Zipf 分布，范围为 0% 至 100%。如图 9(c) 所示，在所有查询类别中，AeonG 的性能均优于次优系统 Clock-G，最高可提升 2.21×。我们还观察到，需要访问更多历史数据的“热”查询通常比“温”和“冷”查询的性能更低。然而，在 AeonG 中，“热”查询仅比“温”查询慢 1.29×，而在 T-GQL 和 Clock-G 中，“热”查询的性能分别比“温”查询低最多 2.14× 和 3.77×。如前所述，AeonG 较小的性能差距可归因于我们的基于锚点的版本检索技术，该技术避免了不必要的版本遍历。我们还研究了时间片长度从 1s 变化到 200s 时，时态查询 Q4 的延迟。如图 9(d) 所示，AeonG 相比 Clock-G 和 T-GQL 的性能最高可提升

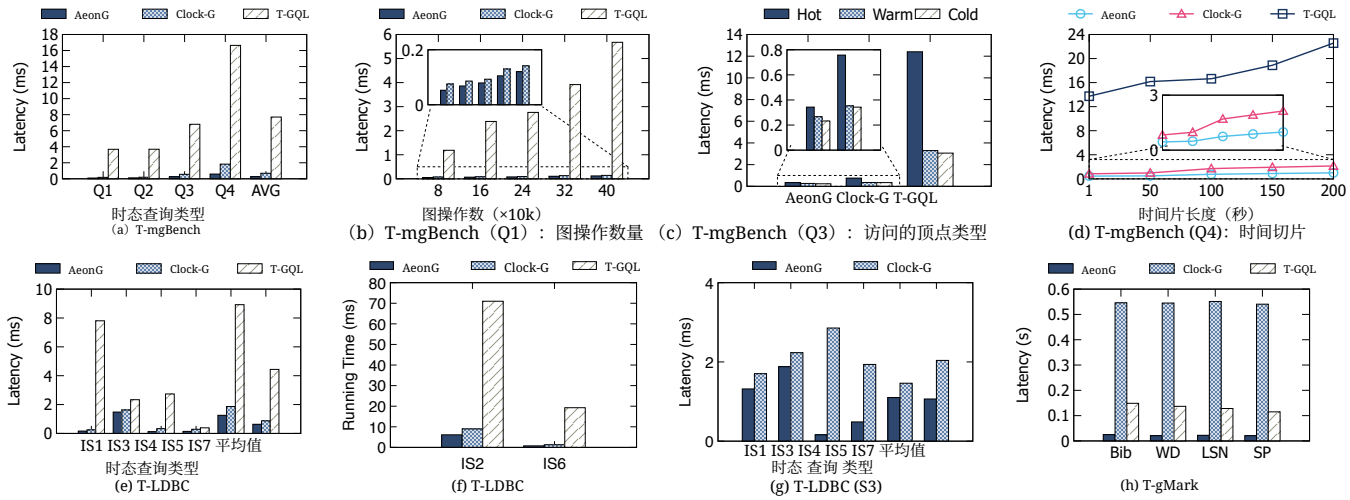


图 9：时态查询延迟比较

2.27× 和 33.23×，分别表明其在不同时间片长度下始终保持更优的性能。

我们还在 T-LDBC 上评估了时态查询性能。如图 9(e) 和 9(f) 所示，AeonG 在所有时态查询类型中均表现更优，其延迟比 Clock-G 和 T-GQL 最多低 1.46× 和 12.07×，这与图 9(a) 中观察到的趋势一致。我们还使用包含 9.28M 个顶点和 52.7M 条边的更大图，在 T-LDBC 上重新进行了实验。在这些实验中，由于图规模增大，T-GQL 遇到了内存溢出问题。然而，如图 9(g) 所示，AeonG 仍然比 Clock-G 的性能最高提升 2.06×，这与图 9(e) 所示的结果一致。此外，我们使用 T-gMrak 进行了实验。如图 9(h) 所示，AeonG 在所有数据集上始终表现更优，其时态查询性能比 Clock-G 最多快 26.16×，比 T-GQL 最多快 6.56×。

7.3 AeonG 性能分析

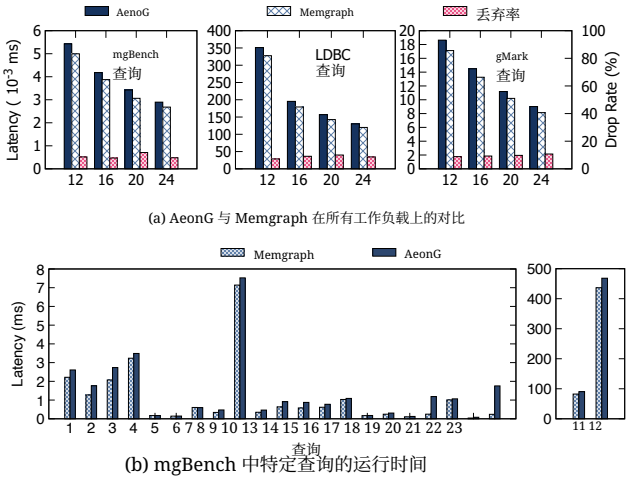
现在，我们对 AeonG 在不同配置下的性能进行深入分析。在接下来的实验中，我们将 RocksDB 的 MemTable 大小固定为默认值 64MB。

7.3.1 非时态查询的性能。 我们首先分析 AeonG 在非时态查询上的性能，以研究在其基础系统 Memgraph 中引入时态特性所带来的影响。我们使用三个原始的、未扩展的工作负载中定义的各种非时态查询：mgBench、LDBC 和 gMark。我们基于 T-mgBench、T-LDBC 和 T-gMark 生成的数据集运行相应的查询，并在图10中绘制每个工作负载的平均查询延迟。结果表明，与 Memgraph 相比，AeonG 的性能下降幅度最高可达 9.74%，处于可接受范围内，这是其支持时态查询所需付出的代价。此外，我们还在图10(b)中详细展示了 mgBench 中每个具体查询的性能分解。我们可以观察到，在大多数测试查询中，AeonG 的性能与 Memgraph 相当。AeonG 采用了一种将当前数据库，并异步传输历史数据，确保对主要的非时态查询影响最小。

7.3.2 历史数据迁移的影响。 接下来，我们使用 T-mgBench 工作负载，通过调整 GC 间隔来控制历史数据迁移的频率，从而研究不同情况下的查询性能。我们在图 11(a) 中绘制了不同数据类型（当前数据、已回收数据和未回收数据）以及图操作的查询延迟。如图所示，当前数据和未回收数据的查询性能相对相近，且二者均优于已回收数据查询，优势最高可达 25.8%。造成这一差异的原因是，查询当前数据和未回收数据都需要遍历当前存储中的版本链，而查询已回收数据则需要额外使用历史存储中的锚点和增量重构历史版本，具体内容详见第4节。此外，我们注意到，将 GC 间隔从 1 秒增加到 1000 秒，会使图操作延迟降低 17.3%，并使查询延迟增加 9.5%。这是预期结果，因为降低迁移频率可以减少与图操作之间的竞争，从而提升图操作性能。相反，较低的迁移频率会导致当前存储中的版本链遍历更长，进而对查询性能产生负面影响。

7.3.3 锚点间隔分析。 我们使用 T-LDBC 工作负载评估自适应锚定方法的有效性。如图 11(b)所示，当我们为每个图对象分配固定锚点间隔 u，并将 u 从 1 调整到 1000 时，我们观察到历史存储的存储消耗减少了 2.69×，而时态查询延迟增加了 2.15×。相比之下，面对所有固定锚点间隔设置，我们的自适应锚定方法始终能实现近乎最优的查询性能和存储效率，因为它能够恰当地平衡查询延迟与存储开销效率。

7.3.4 历史保留期限的影响。我们使用 T-LDBC 工作负载评估历史存储开销和



(a) AeonG 与 Memgraph 在所有工作负载上的对比

(b) mgBench 中特定查询的运行时间

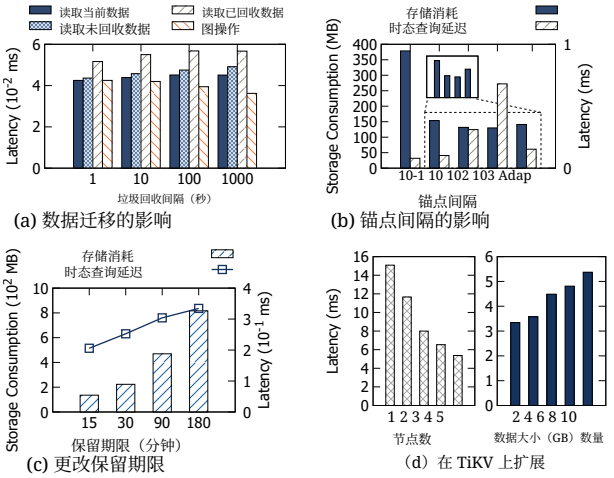
图 10：AeonG 与 Memgraph 在非时态查询上的对比

不同历史数据保留期限下的时态查询延迟。我们将一天的图演化量模拟为一分钟，方法是假设每天执行 100k 次操作，并在一分钟内完成这些操作。因此，我们将历史数据保留期限设为 15 、 30 、 90 和 180 分钟，分别模拟现实中的半个月、一个月、一个季度和半年。如图 11(c) 所示，数据保留期限从 15 分钟延长至 180 分钟时，存储消耗增加了 6.02×，查询性能下降了 1.62×。这一趋势符合预期，因为保留期限越长，保留的历史数据就越多，从而导致查询性能下降。基于这一观察，我们考虑允许用户设置合适的保留_期限，以在存储开销、历史数据时长和查询性能之间取得平衡。

7.3.5 AeonG-D 的可扩展性。 我们默认将 AeonG-D 和 TiKV 部署在 5 个节点上，并在这些节点之间对历史数据进行水平分区。首先，我们使用 10GB 数据量和 T-LDBC 工作负载，评估增加服务器数量对性能的影响。结果如图 11(d) 左侧所示，服务器数量从 1 个增加到 5 个时，AeonG-D 的时态查询延迟最多降低了 2.8×。AeonG-D 的可扩展性归因于增加服务器所带来的并行性提升：每台服务器都可以通过其 TiKV 实例独立处理历史数据检索请求。其次，我们使用 T-LDBC 评估时态查询延迟，同时将数据量从 2GB 增加到 10GB。如图 11(d) 右侧所示，AeonG-D 的延迟最多增加了 1.6×；由于从更大的数据库中获取图对象的成本更高，这一结果符合预期。在 [19, 23, 43] 中也报告了类似趋势。

8 相关工作

时态数据管理在关系数据库系统领域已经发展到较为成熟的阶段 [51–57]。然而，图环境中的时态数据管理仍相对有限。本节将从两种主要方法出发，对时态图数据



(a) 数据迁移的影响

(b) 锚点间隔的影响

(c) 更改保留期限

（d）在 TiKV 上扩展

图 11：AeonG 上的性能分解分析

管理系统进行综述：一种是在应用层面，另一种是在内部系统层面。

一种支持时间维度的方法是利用商业图数据库，并附加时间元数据。Cat- tuto 等人 [8] 提出了一种时态模型，其中时态数据表示为帧，帧被定义为时间聚合的最细粒度单位。Debrouvier 等人改进了属性图模型，以存储时态图 [7]。他们假设对象、属性和值以传统属性图顶点的形式存储，而时间区间则作为这些顶点的属性存储。时间边则以传统边的形式存储，时间区间是其属性之一。其他研究通过将时间维度表示为图对象的区间属性来利用图数据库 [9–13]。这些系统是在现有数据库系统之上于应用层实现的。由于底层存储和查询引擎是为静态图而非不断演变的图设计的，其性能可能难以预测。另一条研究路线聚焦于底层存储引擎，以组织时态图。其主要目标是在支持高效查询的同时处理不断增长的历史数据。为此，两种存储方案 *Copy* 和 *Log*被广泛用于管理时态图数据 [14–23]。*Copy* 方案 [14, 15] 会在每批更新发生时存储完整的图状态。虽然这简化了图查询，却导致存储的图信息冗余过多。相比之下，*Log* 方案 [16, 17] 会将每次图更新活动记录到日志中。这种方案更紧凑，但执行时间图查询时需要耗费大量资源进行重构。作为查询性能与空间开销之间的折衷，*Copy+Log* 方案 [18–23] 将有限数量的快照与其间的增量列表结合起来。然而，我们认为，*Copy+Log* 仍非最优方案，因为它依然需要大量存储开销来物化整个图。此外，以上所有方法主要关注离线历史数据，限制了它们在在线查询中的适用性。更糟的是，据我们所知，这些方法既不支持功能强大的时态图

数据模型，也不支持声明式时态查询语言，因而降低了用户使用的便利性。

9 结论

本文介绍了 AeonG，这是一种能够高效提供内置时态支持的新型图数据库。AeonG 基于一种新颖的时态属性图模型。为适配该模型，我们设计了时态增强型存储引擎和查询引擎。首先，我们提供了一种支持异步迁移的混合存储引擎，既能确保当前存储的性能下降幅度较小，又能优化历史存储的存储效率。随后，我们介绍了一种时态查询引擎，该引擎采用合法性检查技术来确保数据一致性，并采用基于锚点的版本检索技术来提升查询性能。我们使用三种工作负载进行了广泛的实验评估。结果表明，与次优系统相比，AeonG 的存储开销最高可降低 5.73×，图操作延迟可降低 2.82×，时态查询延迟可降低 2.57×；同时，为支持时态特性所引入的性能下降仅为 9.74%。

参考文献

[1] Neo4j，“https://neo4j.com”。访问日期：2024年2月。[2] ArangoDB，“https://www.arangodb.com”。访问日期：2024年2月。[3] Dgraph，“https://dgraph.io”。访问日期：2024年2月。[4] Memgraph，“https://memgraph.com”。访问日期：2024年2月。[5] A. Abdallah、M. A. Maarof 和 A. Zainal，“欺诈检测系统：综述”，*J. Netw. Comput. Appl.*，第 68 卷，第 90–113 页，2016 年。[6] W. Laddada 和 C. Ray，“基于图的海上活动模式分析”，载于*GAST Workshop*，第*20*届法语知识提取与管理研讨会（*EGC*）论文集，第 1–14 页，2020 年。[7] A. Debrouvier、E. Parodi、M. Perazzo、V. Soliani 和 A. A. Vaisman，“时态图数据库的模型与查询语言”，*VLDB J.*，第 30 卷，858，2021 5，825no. pp. – 。[8] C. Cattuto、M. Quaggiotto、A. Panisson 和 A. Averbuch，“图数据库中的时变社交网络：一个 Neo4j 用例”，载于*GRADES*，第 11 页，CWI/ACM，2013 年。[9] Z. Liu、C. Wang 和 Y. Chen，“时态图上的关键词搜索”，载于*ICDE*，第 1807–1808 页，IEEE计算机学会，2018 年。[10] C. Rost、K. Gómez、M. Täschner、P. Fritzsche、L. Schons、L. Christ、T. Adameit、M. Junghanns 和 E. Rahm，“使用 GRADOOP 进行分布式时态图分析”，*VLDB J.*，第 31 卷，第 2 期，第 375–401 页，2022 年。[11] M. Yu，“用于表示和量化四维自然现象的基于图的时空数据框架——以沙尘事件为例”，*ISPRS Int. J. Geo Inf.*，第 9 卷，第 2 期，第 127 页，2020 年。[12] L. Zheng、L. Zhou、Z. Xin、L. Li 和 W. Liu，“基于图数据库的时空数据建模与应用”，载于*2017*年第*4*届国际信息科学与控制工程会议（*ICISCE*），2017 年。[13] G. C. Durand、M. Pinnecke、D. Broneske 和 G. Saake，“积压记录与区间时间戳：支持图数据库时态查询的构件”，载于*EDBT/ICDTWorkshops*，第 1810 卷*CEURWorkshop Proceedings*，CEUR-WS.org，2017 年。[14] U. Khurana 和 A. Deshpande，“历史图数据的高效快照检索”，载于*ICDE*，第 997–1008 页，IEEE计算机学会，2013 年。[15] J. Byun、S. Woo 和 D. Kim，“Chronograph：支持时态图遍历，以实现随时间变化的信息扩散分析”，*IEEE Trans. Knowl. Data Eng.*，第 32 卷，第 3 期，第 424–437 页，2020 年。[16] B. A. Steer、F. Cuadrado 和 R. G. Clegg，“Raphtory：分布式时态图的流式分析”，*Future Gener. Comput. Syst.*，第 102 卷，464，2020 453pp. – 。[17] P. Macko、V. J. Marathe、D. W. Margo 和 M. I. Seltzer，“LLAMA：利用大型多版本数组实现高效图分析”，载于*ICDE*，第 363–374 页，IEEE计算机学会，2015 年。[18] Y. Miao、W. Han、K. Li、M. Wu、F. Yang、L. Zhou、V. Prabhakaran、E. Chen 和 W. Chen，“Immortalgraph：用于时态图存储与分析的系统”，*ACM Trans. Storage*，第 11 卷，第 3 期，第 14:1–14:34 页，2015 年。[19] M. Massri、Z. Miklós、P. R. Parvédy 和 P. Meye，“Clock-g：一种采用节省空间存储技术的时态图管理系统”，载于*ICDE*，第 2263–2276 页，IEEE，2022 年。

[20] P. Kumar 和 H. H. Huang，“Graphone：面向演化图实时分析的数据存储”，ACM Trans. Storage，第 15 卷第 4 期，第 29:1–29:40 页，2020 年。[21] W. Han、Y. Miao、K. Li、M. Wu、F. Yang、L. Zhou、V. Prabhakaran、W. Chen 和 E. Chen，“Chronos：用于时态图分析的图引擎”，载于 EuroSys，第 1:1–1:14 页，ACM，2014 年。[22] W. Han、K. Li、S. Chen 和 W. Chen，“Auxo：时态图管理系统”，Big Data Min. Anal.，第 2 卷第 1 期，第 58–71 页，2019 年。[23] U. Khurana 和 A. Deshpande，“大规模存储和分析历史图数据”，载于 EDBT，第 65–76 页，OpenProceedings.org，2016 年。[24] R. Angles、M. Arenas、P. Barceló、A. Hogan、J. L. Reutter 和 D. Vrgoc，“现代图数据库查询语言基础”，ACM Comput. Surv.，第 50 卷第 5 期，第 68:1–68:40 页，2017 年。[25] N. Francis、A. Green、P. Guagliardo、L. Libkin、T. Lindaaker、V. Marsault、S. Plantikow、M. Rydberg、P. Selmer 和 A. Taylor，“Cypher：一种不断演化的属性图查询语言”，载于 SIGMOD Conference，第 1433–1445 页，ACM，2018 年。[26] R. Angles、A. Bonifati、S. Dumbrava、G. Fletcher、A. Green、J. Hidders、B. Li、L. Libkin、V. Marsault、W. Martens、F. Murlak、S. Plantikow、O. Savkovic、M. Schmidt、J. Sequeda、S. Staworko、D. Tomaszuk、H. Voigt、D. Vrgoc、M. Wu 和 D. Zivkovic，“Pg-schema：属性图的模式”，Proc. ACM Manag. Data，第 1 卷第 2 期，第 198:1–198:25 页，2023 年。[27] R. Angles、A. Bonifati、S. Dumbrava、G. Fletcher、K. W. Hare、J. Hidders、V. E. Lee、B. Li、L. Libkin、W. Martens、F. Murlak、J. Perryman、O. Savkovic、M. Schmidt、J. F. Sequeda、S. Staworko 和 D. Tomaszuk，“Pg-keys：属性图的键”，载于 SIGMOD Conference，第 2423–2436 页，ACM，2021 年。[28] X. Zhu、M. Serafini、X. Ma、A. Aboulnaga、W. Chen 和 G. Feng，“LiveGraph：具有纯顺序邻接表扫描的事务型图存储系统”，Proc. VLDB Endow.，第 13 卷第 7 期，第 1020–1034 页，2020 年。[29] A. Dubey、G. D. Hill、R. Escriva 和 E. G. Sirer，“Weaver：基于可细化时间戳的高性能事务型图数据库”，Proc. VLDB Endow.，第 9 卷第 11 期，第 852–863 页，2016 年。[30] H. Chen、C. Li、C. Zheng、C. Huang、J. Fang、J. Cheng 和 J. Zhang，“G-Tran：让分布式图事务更快”，CoRR，第 abs/2105.04449 卷，2021 年。[31] Y. Wu、J. Arulraj、J. Lin、R. Xian 和 A. Pavlo，“内存中多版本并发控制的实证评估”，Proc. VLDB Endow.，第 10 卷第 7 期，第 781–792 页，2017 年。[32] K. G. Kulkarni 和 J. Michels，“SQL：2011 中的时态特性”，SIGMOD Rec.，第 41 卷第 3 期，第 34–43 页，2012 年。[33] T. Neumann、T. Mühlbauer 和 A. Kemper，“面向主存数据库系统的快速可串行化多版本并发控制”，载于 SIGMOD Conference，第 677–689 页，ACM，2015 年。[34] J. Kim、K. Kim、H. Cho、J. Yu、S. Kang 和 H. Jung，“重新思考 MVCC 数据库中的扫描”，载于 SIGMOD Conference，第 938–950 页，ACM，2021 年。[35] H. Berenson、P. A. Bernstein、J. Gray、J. Melton、E. J. O’Neil 和 P. E. O’Neil，“对 ANSI SQL 隔离级别的批评”，载于《1995 年 ACM SIGMOD 数据管理国际会议论文集》，美国加利福尼亚州圣何塞，1995 年 5 月 22 日至 25 日（M. J. Carey 和 D. A. Schneider 编），第 1–10 页，ACM Press，1995 年。[36] H. Li、Z. Zhao、Y. Cheng、W. Lu、X. Du 和 A. Pan，“在基于 MVCC 的关系数据库管理系统中高效提取时间区间数据”，万维网，2018 年。[37] D. Ports 和 K. Grittner，“PostgreSQL 中的可串行化快照隔离”，《VLDB基金会会刊》，第 5 卷第 12 期，第 1850–1861 页，2012 年。[38] Z. Zhao、W. Lu、H. Zhao、Z. He、H. Li、A. Pan 和 X. Du，“T-SQL：一种用于在基于 MVCC 的关系数据库管理系统中实现内置时态支持的轻量级实现”，IEEE Trans. Knowl. Data Eng.，第 35 卷第 1 期，第 1028–1042 页，2023 年。[39] P. Gupta、A. Mhedhbi 和 S. Salihoglu，“图数据库管理系统的列式存储和基于列表的处理”，Proc. VLDB Endow.，第 14 卷第 11 期，第 2491–2504 页，2021 年。[40] C. Kankanamge、S. Sahu、A. Mhedhbi、J. Chen 和 S. Salihoglu，“Graphflow：一种主动图数据库”，载于 SIGMOD Conference，第 1695–1698 页，ACM，2017 年。[41] O. Facebook，“RocksDB：面向快速存储环境的持久化键值存储”，2019 年。[42] J. Hou、Z. Zhao、Z. Wang、W. Lu、G. Jin、D. Wen 和 X. Du，“AeonG：图数据库中的高效内置时态支持（扩展版）。” https://hououou.github.io/AeonG/aeong-extended-version-vldb24.pdf。[43] A. G. Labouseur、J. Birnbaum、P. W. Olsen、S. R. Spillane、J. Vijayan、J.-H. Hwang 和 W.-S. Han，“g* 图数据库：高效管理大型分布式动态图”，Distributed and Parallel Databases，第 33 卷，第 479–514 页，2015 年。[44] D. D. Leo 和 P. A. Boncz，“Teseo 与结构动态图分析”，Proc. VLDB Endow.，第 14 卷第 6 期，第 1053–1066 页，2021 年。[45] D. Huang、Q. Liu、Q. Cui、Z. Fang、X. Ma、F. Xu、L. Shen、L. Tang、Y. Zhou、M. Huang 等，“TiDB：基于 Raft 的 HTAP 数据库”，《VLDB基金会会刊》，第 13 卷第 12 期，第 3072–3084 页，2020 年。[46] L. Takac 和 M. Zabovsky，“公共社交网络中的数据分析”，载于国际科学会议及“当今创新趋势”国际研讨会，第 1 卷，2012 年。[47] Mgbench，“https://memgraph.com/benchgraph”。访问时间：2024 年2月。[48] A. Iosup、T. Hegeman、W. L. Ngai、S. Heldens、A. Prat-Pérez、T. Manhardt、H. Chafi、M. Capota、N. Sundaram、M. J. Anderson、I. G. Tanase、Y. Xia、L. Nai 和 P. A. Boncz，“LDBC Graphalytics：面向大规模图分析的基准测试

并行与分布式平台”，《*VLDB*基金会会刊》，第 9 卷第 13 期，第 1317–1328 页，2016 年。[49] G. Bagan、A. Bonifati、R. Ciucanu、G. H. L. Fletcher、A. Lemay 和 N. Advokaat，“gMark：图和查询的模式驱动生成”，*IEEE Trans. Knowl. Data Eng.*，第 29 卷第 4 期，第 856–869 页，2017 年。[50] G. K. Zipf，《人类行为与最省力原则：人类生态学导论》。Addison-Wesley，1949 年。[51] A. Bolour、T. L. Anderson、L. J. Dekeyser 和 H. K. T. Wong，“时间在信息处理中的作用：综述”，*SIGART Newsl.*，第 80 卷，第 28–46 页，1982 年。[52] R. T. Snodgrass 和 I. Ahn，“数据库中的时间分类法”，载于 *SIGMOD Conference*，第 236–246 页，ACM Press，1985 年。[53] G. Özsoyoglu 和 R. T. Snodgrass，“时态数据库与实时数据库：综述”，*IEEE Trans. Knowl. Data Eng.*，第 7 卷第 4 期，第 513–532 页，1995 年。

[54] A. U. Tansel、J. Clifford、S. K. Gadia、S. Jajodia、A. Segev 和 R. T. Snodgrass 编，时态数据库：理论、设计与实现。Benjamin/Cummings，1993 年。[55] W. Lu、Z. Zhao、X. Wang、H. Li、Z. Zhang、Z. Shui、S. Ye、A. Pan 和 X. Du，“TDSQL 中轻量且高效的时态数据库管理系统”，*Proc. VLDB Endow.*，第 12 卷，第 12 期，第 2035–2046 页，2019 年。[56] Z. Zhao、W. Lu、H. Zhao、Z. He、H. Li、A. Pan 和 X. Du，“T-SQL：一种在基于多版本并发控制的关系数据库管理系统中实现内置时态支持的轻量级方案”，*IEEE Trans. Knowl. Data Eng.*，第 35 卷，第 1 期，第 1028–1042 页，2023 年。[57] B. Salzberg 和 V. J. Tsotras，“随时间演化数据的访问方法比较”，*ACM Comput. Surv.*，第 31 卷，第 2 期，第 158–221 页，1999 年。

表1：T-LDBC 工作负载的时态查询

IS1	匹配 (n:Person {id:xx})-[:位于]-(p:Place) **FOR TT AS OF** *t* 返回 n.firstName AS 名字, n.lastName AS 姓氏, n.birthday AS 生日, n.locationIP AS 位置IP, n.使用的浏览器 AS 使用的浏览器，n.性别 AS 性别, n.创建日期 AS 创建日期，p.id AS 城市ID；
IS2	匹配 (:人物 {id:xx})<-[:具有创建者]-(m)-[:回复*0..]->(p:帖子) 匹配 (p)-[:具有创建者]->(c) **FOR TT AS OF** *t* 返回 m.id AS 消息ID, CASE m.content 不为 null WHEN true THEN m.content, ELSE m.imageFile, END AS messageContent, m.creationDate AS messageCreationDate，p.id AS originalPostId, c.id AS 原始帖子作者ID, c.名字 as 原始帖子作者名字, c.lastName as originalPostAuthorLastName ORDER BY messageCreationDate DESC LIMIT 10;
IS3	匹配 (n:人物 {id:xx})-[r:认识]-(朋友) **FOR TT AS OF** *t* 返回 friend.id AS 人员编号, friend.firstName AS 名字, friend.lastName AS 姓氏, r.creationDate AS 友谊创建日期 排序依据 友谊创建日期 DESC, 转换为整数(人员编号) ASC;
IS4	匹配 (m:消息 {id:xx}) **FOR TT AS OF** *t* 返回 CASE exists(m.content) WHEN true THEN m.content ELSE m.imageFile END AS messageContent, m.creationDate as messageCreationDate;
IS5	匹配 (m:消息 {id:xx})-[:具有创建者]->(p:人物) **FOR TT AS OF** *t* 返回 p.id AS 人员编号, p.firstName AS 名字, p.lastName AS 姓氏;
IS6	匹配 (m:消息 {id:xx})-[:回复*0..]->(p:帖子)<-[:CONTAINER_OF]-(f:论坛)-[:HAS_MODERATOR]->(mod:人物) **FOR TT AS OF** *t* 返回 f.id AS 论坛ID, f.title AS 论坛标题, mod.id AS 版主ID, mod.firstName AS 版主名字, mod.lastName AS 版主姓氏;
IS7	匹配 (m:消息 {id:xx})<-[:回复]-(c:评论)-[:具有创建者]->(p:人物) 可选匹配 (m)-[:具有创建者]->(a:人物)-[r:认识]-(p) **FOR TT AS OF** *t* 返回 c.id AS 评论ID, c.content AS 评论内容, c.creationDate AS 评论创建日期, p.id AS 回复作者ID, p.firstName AS 名字, p.lastName AS 姓氏, CASE r WHEN null THEN false ELSE true END AS 回复作者认识原消息作者 排序依据 评论创建日期 DESC, 转换为整数(回复作者ID) ASC;

T-LDBC 的时态查询

我们在表1中列出了实验所用 T-LDBC 工作负载的时态查询。顶点ID的值取自用于评估这些查询的相应数据集。