Implementation Strategies for Views over Property Graphs

作者：SOONBO HAN, ZACHARY G. IVES

原文链接：https://dl.acm.org/doi/pdf/10.1145/3654949

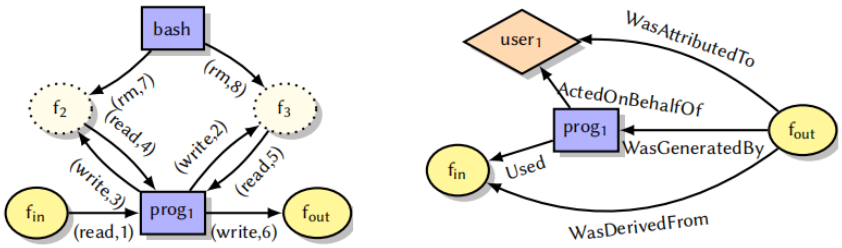
代码链接：https://github.com/PennGraphDB/pg-view

背景

随着对复杂查询的需求增大，人们对属性图的关注度逐渐上升

对于一些像数据溯源的图，会被编码为实体、操作、代理等节点，有些时候会需要在不同的细节级别上显示节点，可能会把某个子图直接替换为一个节点(即缩小操作)，也会把一个节点替换为一整个子图(即放大操作)。另外从查询速率的角度来说，提取查询的公共部分，提前存储视图结果，可以大大减少查询时间。

比如对于下面的图



(a) Raw event-level provenance graph (b) PROV graph

Fig. 1. An (a) event log-based provenance graph encodes I/O ($prog_1$ reads and writes files $f_{in}, f_2, f_3, f_{out}$); when (b) converting to PROV, we omit the temporary files, add the user, and add a variety of transitive edges.

f2和f3在程序运行完会被bash删除，所以希望有一个视图可以去掉这些被删除的临时文件，就像图(b)中那样

贡献

本文的贡献如下：

- 将图视图转换为关系形式，这样可以使用现有的方法计算视图并且用来表示查询
- 本文使用了标准图数据库查询语言GQL来支持视图语句的定义，并且定义了一个保持干净语义的良好行为视图的概念
- 本文在图数据库中实现了物化和更新视图的策略，在关系数据库中同时实现了物化视图和虚拟视图，并且在关系型数据库中使用索引加速了在图视图上的查询速率
- 实验结果表明在多个工作负载和不同的数据库管理系统上本文的方法都能够有不错的性能提升

视图相关定义

TGD

本文使用TGD来表示视图，这样比较清晰，并且易于分析。

对于以下的视图

```
CREATE VIEW view AS (
CONSTRUCT (f1)<-[u:Used]-(p) SET u.ts = 3
MATCH (f1:Entity)-[:read]->(p:Activity) )
```

可以表示为以下的TGD

$$N\ (f1,\text{“}Entity\text{”})\ \wedge\ N\ (p,\text{“}Activity\text{”})\ \wedge\ E(r,\ f1,\ p,\text{“}read\text{”})$$
$$\rightarrow\ \exists u(N'(f1,\text{“}Entity\text{”})\ \wedge\ N'(p,\text{“}Activity\text{”})\ \wedge\ E'(u,\ p,\ f1,\text{“}U\ sed\text{”})\ \wedge\ EP'(u,\text{“}prop\text{”},\ 3)).$$

就是将Match子句作为TGD的左半部分，Construct子句作为TGD的右半部分

映射默认规则

现在的视图本质上是对点和边做了映射，由于视图中除了创建新的关系，可能会出现一些不需要的关系需要被显式删除，所以需要有DELETE子句，如下

```
GRAPH VIEW NoTemps AS (
    DELETE f1
    MATCH [(f1:Entity)-[:rm]-(a2:Activity)
    WHERE a2.name='bash'] ON EventGraph
    UNION DEFAULT_MAPPING )
```

可以用Map来表示映射关系，如下：

N^M (n,n') Node to node ID mapping (n' = $\perp$ if n is deleted)
E^M (e, e') Edge to edge ID mapping (e' = $\perp$ if e is deleted)

这也可以用TGD来表示：

$$N(f_1, \text{“}Entity\text{”}) \wedge N(a_2, \text{“}Activity\text{”}) \wedge E(e_1, f_1, a_2, \text{“}rm\text{”}) \wedge NP(a_2, \text{“}name\text{”}, \text{“}bash\text{”})$$
$$\rightarrow \mathbf{N^M}(\mathbf{f}_1, \perp) \wedge \mathbf{E^M}(\mathbf{e}_1, \perp)$$

对于没有被映射的节点和边，应用默认规则：

$N^{DM}(n, n')$ Node:node ID mapping by default rules

$E^{DM}(e, e')$ Edge:edge ID mapping by default rules

$$N^{M}(n, n') \wedge n' \neq \perp \rightarrow N^{DM}(n, n')$$

$$N(n, l) \wedge \neg N^{M}(n, n') \rightarrow N^{DM}(n, n) \wedge N'(n, l)$$

$$E^{M}(e, e') \wedge e' \neq \perp \rightarrow E^{DM}(e, e')$$

$$E(e, s, d, l) \wedge \neg E^{M}(e, \perp) \wedge N^{DM}(s, s') \wedge N^{DM}(d, d') \rightarrow E'(e, s', d', l) \wedge E^{DM}(e, e)$$

如果是已经显式指定映射的点和边，即n'和e'不为空，那么则应用默认规则映射

如果是没有显式指定映射或删除的点，应用默认规则映射到自己，保留ID和标签

如果是没有显式指定映射或删除的边，当它的源点和终点都被映射的话，那么该边也保留ID和标签映射到自己，源点和终点为映射后的源点和终点

视图的良好行为

在将输入图映射到输出图时，需要满足两个限制条件，才可以被认为是Well-Behaved

- 对于相同的点或边不能有冲突的映射
- 映射的结果不能违反Schema的限制，比如标签或者属性的唯一性等等

对于第一点的冲突映射，主要是点或边不能既被删除又被映射，采用静态检查的方式，本文将其建模成SMT问题判断，由于属性图查询中的模式图大小非常小，所以时间可以忽略。

Schema的限制可以用DBMS的内置功能在运行时检查视图和Schema的主键、外键约束等等

实现视图

图数据库

本文提供了两种图数据库中的物化视图：

- Update In-place：直接在原图数据上创建视图的点或边，这样在定义其他视图的时候需要重新加载以前的视图
- Overlay: 给点和边添加额外的属性，类似于版本控制，这样可以定义视图的序列(在视图上定义视图)，基图的创建时间是0，创建时间为k的视图包括创建时间<=k，删除时间>k的点和边，这样可以区分开原图和视图

关系数据库

每个视图转换规则对应于一个TGD，而TGD可以分解为一系列的Datalog rules(Datalog语言由一组规则和事实组成。规则用于推导新的信息，事实表示已知的信息，支持递归查询)，转换成Datalog之后可以用基于Datalog的LogicBlox数据库创建视图；同时Datalog rule也可以用已有的标准化技术转换为SQL语句。

在查询方面，可以把图数据库主要的Match语句转换为连接的Datalog规则，Where语句作为限制条件，再将Datalog语句转换成SQL语句

使用视图加速查询

物化视图可以加速查询，但是带来的维护代价和空间成本很高，在虚拟视图(完全不物化)和物化视图之间，本文提供了折中的方式，即subgraph substitution relation (SSR)，它可以对输入图上的转换规则带来的局部变换进行编码，比如之前的TGD可以分解为如下的两个TGD

EXAMPLE 5.1 (SSR). *The TGD (t_1) representing a local transformation from input pattern to output pattern can be decomposed into two TGDs:*

$$N(f_1, \text{“Entity”}) \wedge N(p, \text{“Activity”}) \wedge E(r, f_1, p, \text{“read”}) \rightarrow \exists u(\mathbf{ssr}_2(f_1, u, p)), \qquad (t_8)$$

$$\mathbf{ssr}_2(f_1, u, p) \rightarrow N'(f_1, \text{“Entity”}) \wedge N'(p, \text{“Activity”}) \wedge E'(u, p, f_1, \text{“Used”}) \wedge EP'(u, \text{“prop”}, 3). \qquad (t_9)$$

实验部分

实验同时在图数据库和两个关系数据库上做实验，有一个虚拟数据集LSQB和四个真实世界数据集，每个数据集有相应的查询和视图定义

Dataset	View	Type	Operation	Queries
LSQB	V_1: Subgraphs of posts and comments linking authors	Standard	Match	Q_1
	V_2: Subgraphs of friends: Three connected individuals from the same country	Standard	Match	Q_2
PROV	V_3: Substitute subgraphs in event log-based graph into PROV (zoom-in)	Transformation	Substitute subgraphs	$Q_3 \sim Q_6$
OAG	V_4: Add co-authors and the author-venue relationships	Transformation	Add new edges	$Q_7 \sim Q_{10}$
SOC	V_5: Add friend recommendation edges	Transformation	Add new edges	$Q_{11} \sim Q_{14}$
WORD	V_6: Abstract interconnected entities (zoom-out)	Transformation	Collapse nodes	$Q_{15} \sim Q_{18}$

Table 3. Summary of views and queries used for evaluation (https://github.com/PennGraphDB/pg-view).

视图物化及查询时间

结果如下：

View	Impl.	Materialization		Query execution		
		MV	SSR	$Q_1(G)$	Q_1(MV)	Q_1(SSR)
LSQB$_{V_1}$	LB	43.144	43.048	2.341	0.141	**0.106**
	PG	14.639	**6.975**	3.196	0.485	0.437
	N4-UP	1.614	-	0.295	0.075	-
	N4-OV	**1.120**	-	**0.223**	**0.032**	-
		MV	SSR	$Q_2(G)$	Q_2(MV)	Q_2(SSR)
LSQB$_{V_2}$	LB	54.514	53.675	**0.560**	**0.301**	**0.160**
	PG	47.740	**9.414**	7.918	4.067	0.431
	N4-UP	6.723	-	0.771	0.516	-
	N4-OV	**2.168**	-	0.614	11.524	-

View	Materialization							
	$PROV_{V_3}$		OAG_{V_4}		SOC_{V_5}		$WORD_{V_6}$	
Impl.	MV	SSR	MV	SSR	MV	SSR	MV	SSR
LB	32.221	18.709	350.355	124.915	24.869	**18.412**	**10.709**	**4.202**
PG	77.552	**11.267**	413.047	**69.787**	60.985	49.881	170.508	159.627
N4-UP	**3.462**	-	**5.929**	-	**14.935**	-	15.941	-
N4-OV	3.741	-	6.520	-	19.421	-	154.716	-

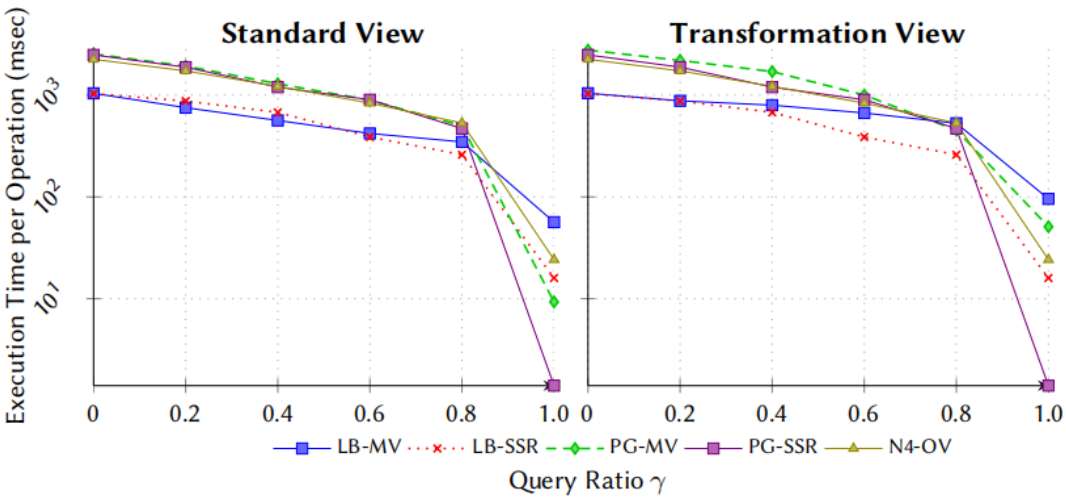
Table 6. Times for creating views with default rules (in sec).

	Query execution											
	View = $PROV_{V_3}$											
Impl.	Q_3(VV)	Q_3(MV)	Q_3(SSR)	Q_4(VV)	Q_4(MV)	Q_4(SSR)	Q_5(VV)	Q_5(MV)	Q_5(SSR)	Q_6(VV)	Q_6(MV)	Q_6(SSR)
LB	t/o	0.169	**0.117**	t/o	**0.116**	**0.122**	t/o	**0.114**	**0.093**	t/o	0.153	**0.088**
PG	**83.773**	0.431	0.419	**29.818**	0.418	0.422	**55.047**	0.435	0.420	t/o	0.465	0.420
N4-UP	-	0.086	-	-	1.227	-	-	1.241	-	-	**0.026**	-
N4-OV	-	**0.057**	-	-	2.539	-	-	2.493	-	-	-	0.037
	View = OAG_{V_4}											
Impl.	Q_7(VV)	Q_7(MV)	Q_7(SSR)	Q_8(VV)	Q_8(MV)	Q_8(SSR)	Q_9(VV)	Q_9(MV)	Q_9(SSR)	Q_{10}(VV)	Q_{10}(MV)	Q_{10}(SSR)
LB	t/o	**0.116**	**0.111**	t/o	**0.409**	**0.110**	t/o	0.102	**0.094**	t/o	0.335	**0.109**
PG	t/o	0.990	0.613	t/o	0.913	0.621	t/o	0.585	0.593	t/o	0.609	0.590
N4-UP	-	16.996	-	-	19.948	-	-	**0.062**	-	-	**0.069**	-
N4-OV	-	31.121	-	-	7.243	-	-	0.100	-	-	0.109	-
	View = SOC_{V_5}											
Impl.	Q_{11}(VV)	Q_{11}(MV)	Q_{11}(SSR)	Q_{12}(VV)	Q_{12}(MV)	Q_{12}(SSR)	Q_{13}(VV)	Q_{13}(MV)	Q_{13}(SSR)	Q_{14}(VV)	Q_{14}(MV)	Q_{14}(SSR)
LB	t/o	**0.148**	**0.127**	t/o	**0.169**	**0.091**	t/o	0.668	**0.104**	t/o	**0.839**	**0.167**
PG	**47.583**	0.301	0.129	**103.547**	0.665	0.125	**94.494**	**0.657**	0.137	**182.078**	1.616	0.777
N4-UP	-	0.505	-	-	0.541	-	-	0.849	-	-	5.380	-
N4-OV	-	0.994	-	-	1.041	-	-	1.545	-	-	50.578	-
	View = $WORD_{V_6}$											
Impl.	Q_{15}(VV)	Q_{15}(MV)	Q_{15}(SSR)	Q_{16}(VV)	Q_{16}(MV)	Q_{16}(SSR)	Q_{17}(VV)	Q_{17}(MV)	Q_{17}(SSR)	Q_{18}(VV)	Q_{18}(MV)	Q_{18}(SSR)
LB	t/o	0.097	**0.099**	t/o	**0.203**	**0.117**	t/o	**0.205**	**0.129**	t/o	0.461	**0.128**
PG	**3.476**	0.171	0.178	**37.302**	0.349	0.228	**25.661**	0.315	0.262	**56.060**	0.432	0.243
N4-UP	-	**0.011**	-	-	0.480	-	-	0.463	-	-	**0.181**	-
N4-OV	-	0.102	-	-	4.498	-	-	1.117	-	-	50.997	-

Table 7. Times for query execution times over views with default rules (in sec).

可以看到SSR的物化时间基本比MV短，在查询上，SSR和MV都有比较好的效果，并且LogicBlox和PostgreSQL在SSR的帮助下效果甚至比图数据库还好。

视图维护效果分析



除了查询优化，本文还探讨了视图维护的效果，由于实际的更新大多为插入，所以本文只实现了插入的维护，同样在LSQB图上测试，横轴为查询所占比例，1则都是查询，0则都是插入语句；纵轴为每个查询或更新的平均时间

当查询比例为百分之90左右，MV和SSR都非常有效，但是查询比例继续降低时，维护代价开始严重影响性能

在查询Ratio为0时，SSR和MV的方式差不多，维护代价区别不大，但是随着查询比例增加，尤其是在查询Ratio为1时，SSR的加速效果明显优于MV，所以总体来说SSR是更优的选择，它在任何情况下都不会比基线差，在查询比例比较高的时候，有非常好的优化效果

总结

本文从GQL的视图定义出发，在属性图上实现物化视图，定义了TGD以及SSR的概念将其推广到关系型数据库，在关系型数据库上实现了图视图，并且实验得到的效果超越了属性图上的视图。

优点：

- 在多个数据库上实现完整的视图并进行比较，方法的通用性很强
- 使用TGD这样的一阶逻辑概念描述视图，对视图的定义更加明确

缺点：

- 可能由于篇幅限制，文中没有具体的算法，例子比较少，很难读懂，对于增量维护的介绍并不多
- 实验部分的测试图比较简单，视图的应用范围比较窄，只能应用于特定的查询