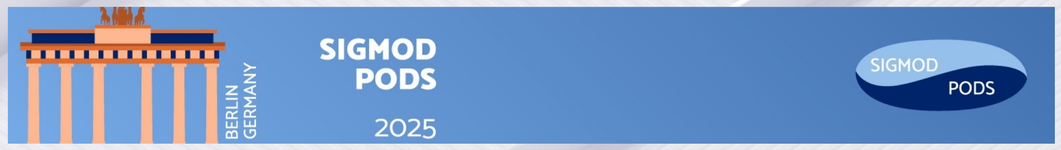


A Modular Graph-Native Query Optimization Framework

Bingqing Lyu, Xiaoli Zhou, Longbin Lai, Yufan Yang, Yunkai Lou, Wenyuan Yu, *Ying Zhang, Jingren Zhou

Tongyi Lab, Alibaba Group, China
*Zhejiang Gongshang University, China

Presented by **Longbin Lai**
2025/06/25

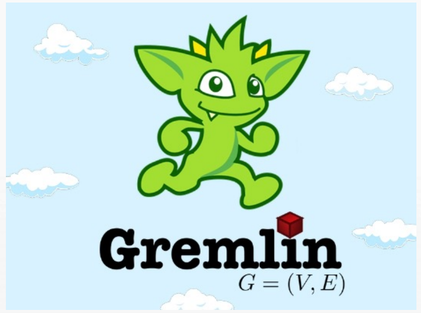
阿里云

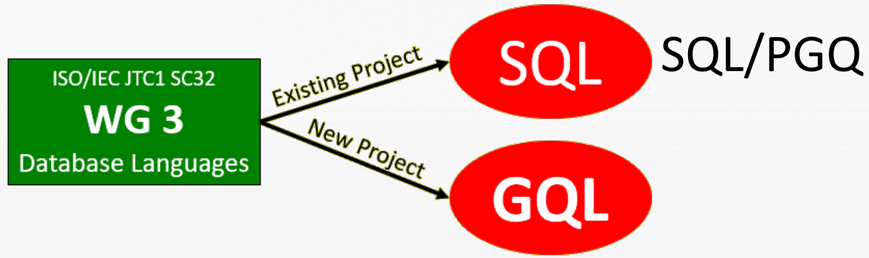
CONTENTS

阿里云

CONTENTS

Background





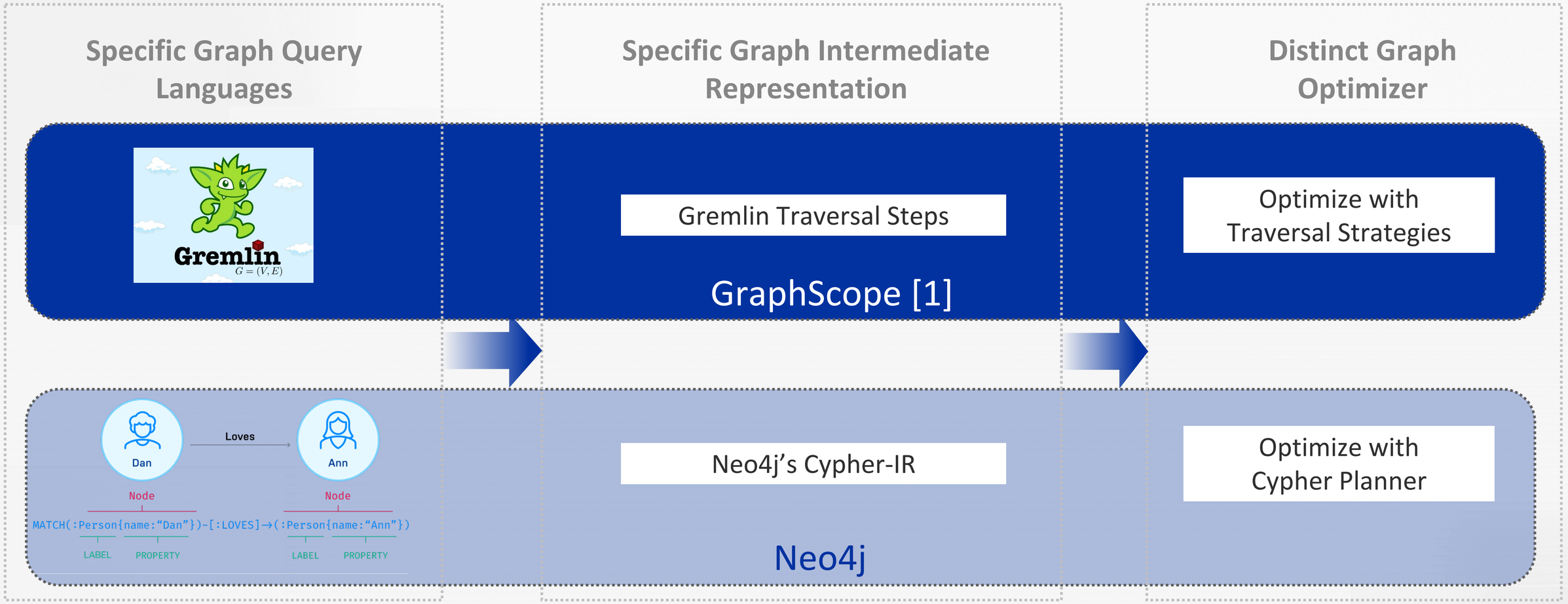
Declarative Query Languages

Query Optimizer is **a KEY component in (G)DBMS!**

Graph Database

Relational Database

A **Monolithic** Pipeline for query optimization



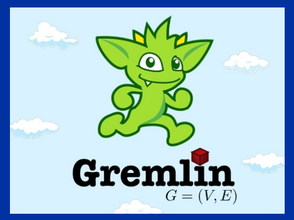
[1]GraphScope: a unified engine for big graph processing, Fan et al, VLDB 2021

A **Monolithic** Pipeline for query optimization (Cont.)

Specific Graph Query Languages

Specific Graph Intermediate Representation

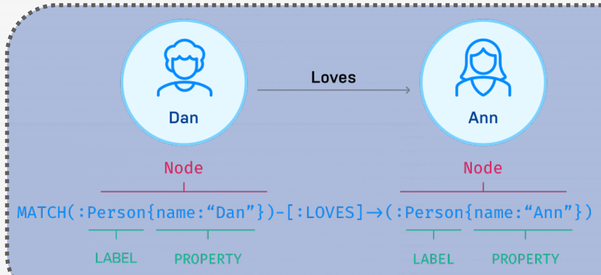
Distinct Graph Optimizer



Gremlin Traversal Steps

GraphScope

Optimize with Traversal Strategies



Neo4j's Cypher-IR

Neo4j

Optimize with Cypher Planner

Hard to extend the support of various graph query languages

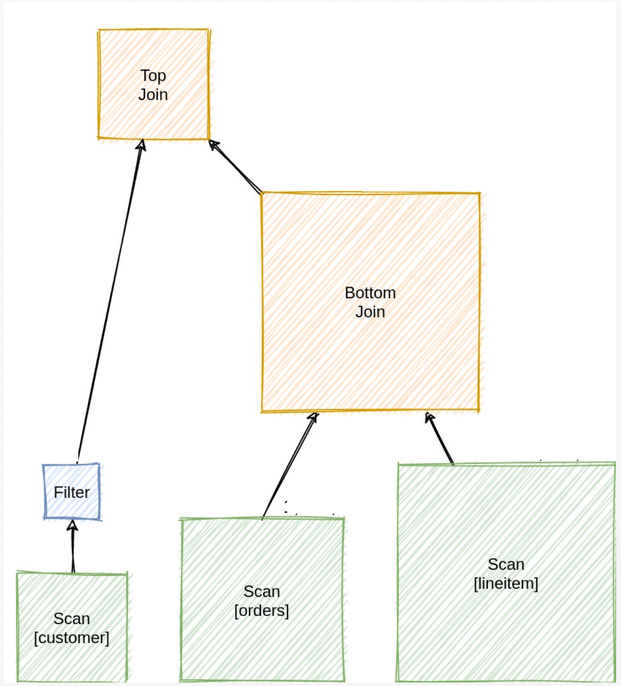
Lack advanced optimizations:

- No CBO in GraphScope
- No high-order statistics in Neo4j …

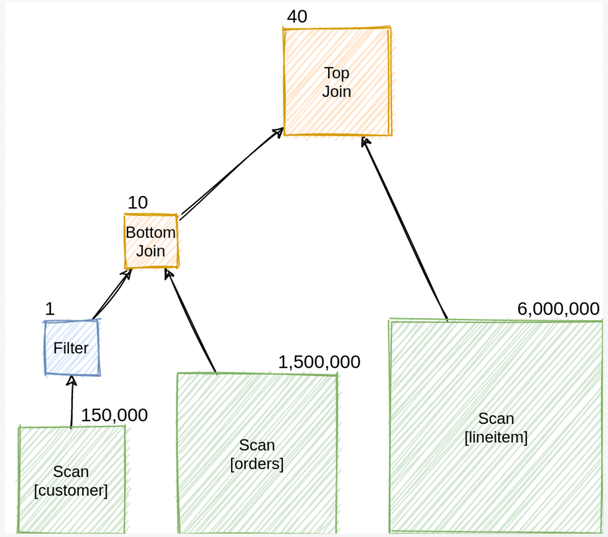
A Relational Optimization Pipeline

Standard Relational Query Language

Relational Algebra

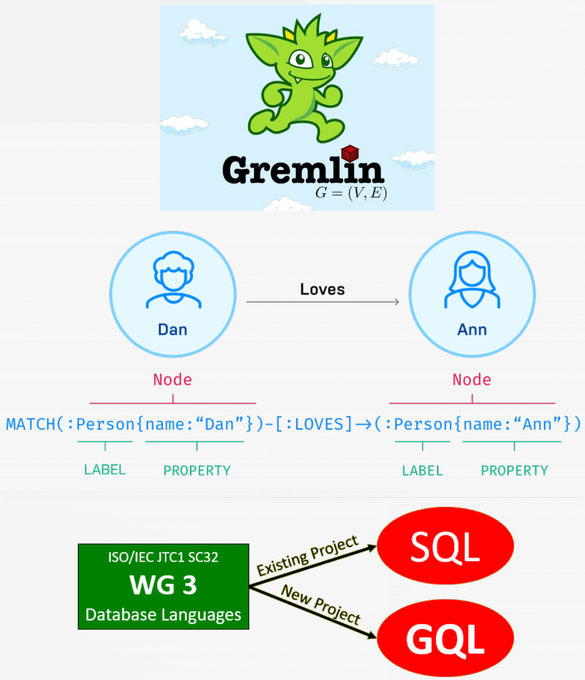


Relational Optimizer



A Graph Optimization Pipeline?

Various Graph Query Languages

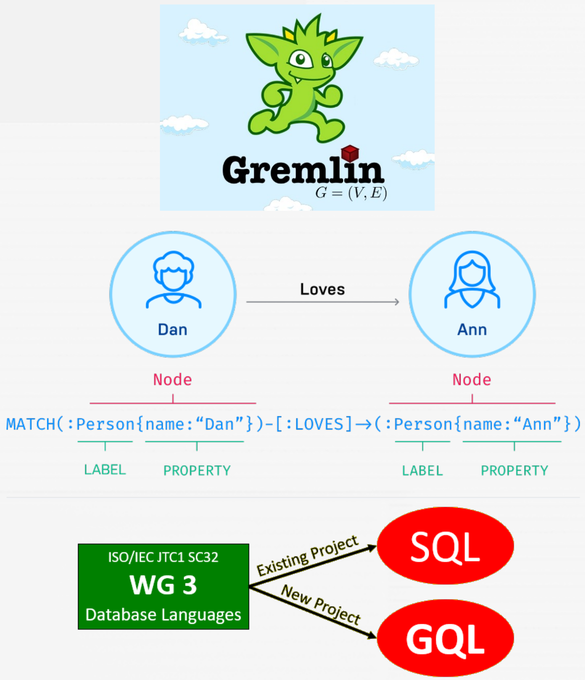


Unified Intermediate Representation?

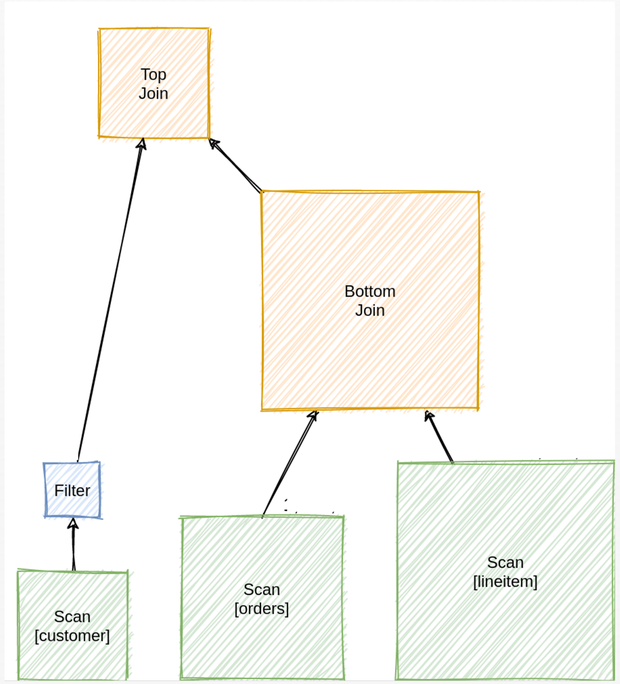
Optimizer?

A Non-graph-native Pipeline for graph optimization

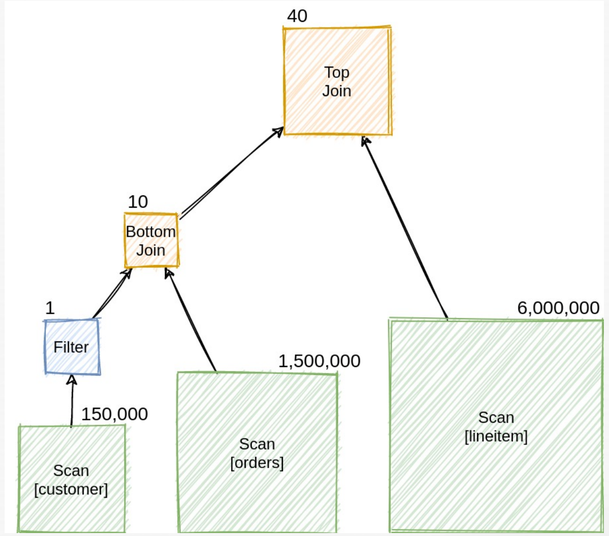
Various Graph Query Languages



Relational Algebra



Relational Optimizer

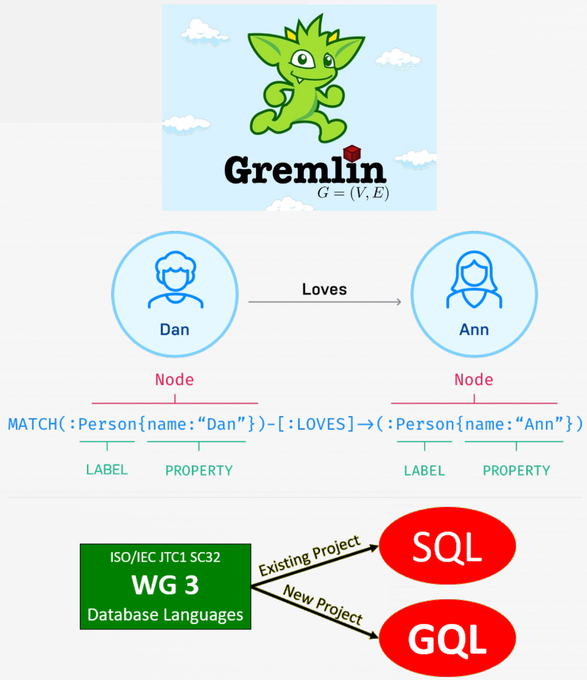


[1] Modern Techniques for Querying Graph-Structured Relations: Foundations, System Implementations, and Open Challenges, Amine Mhedhbi, Semih Salihoğlu, VLDB 2022

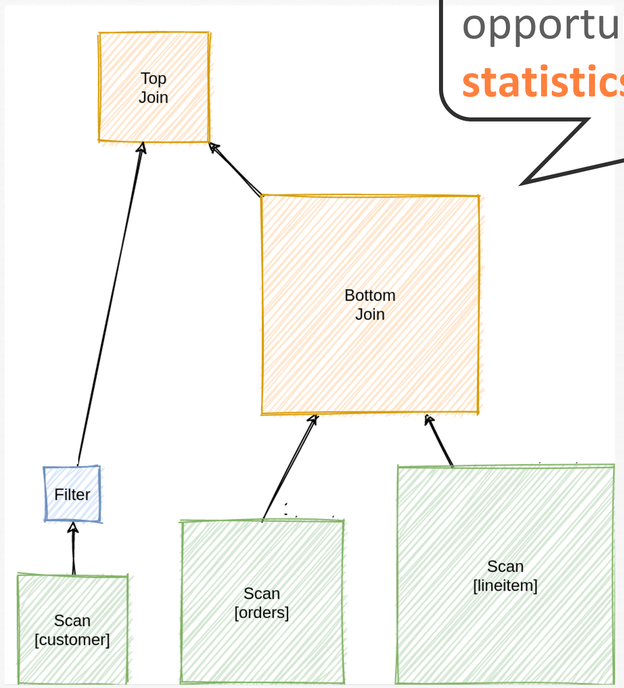
[2] Implementing Graph queries in a Relational Database, https://blog.whiteprompt.com/implementing-graph-queries-in-a-relational-database-7842b8075ca8?gi=fb6853dc262e

A Non-graph-native Pipeline for graph optimization (Cont.)

Various Graph Query Languages

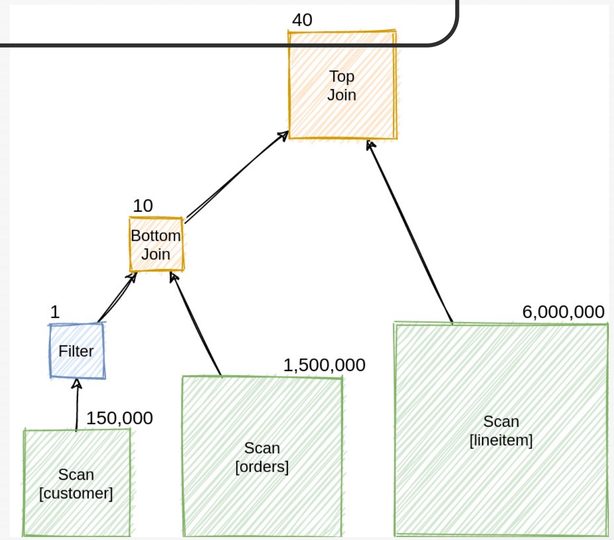


Relational Algebra



May miss graph query optimization opportunities: **hybrid join [1], high-order statistics [2], etc.**

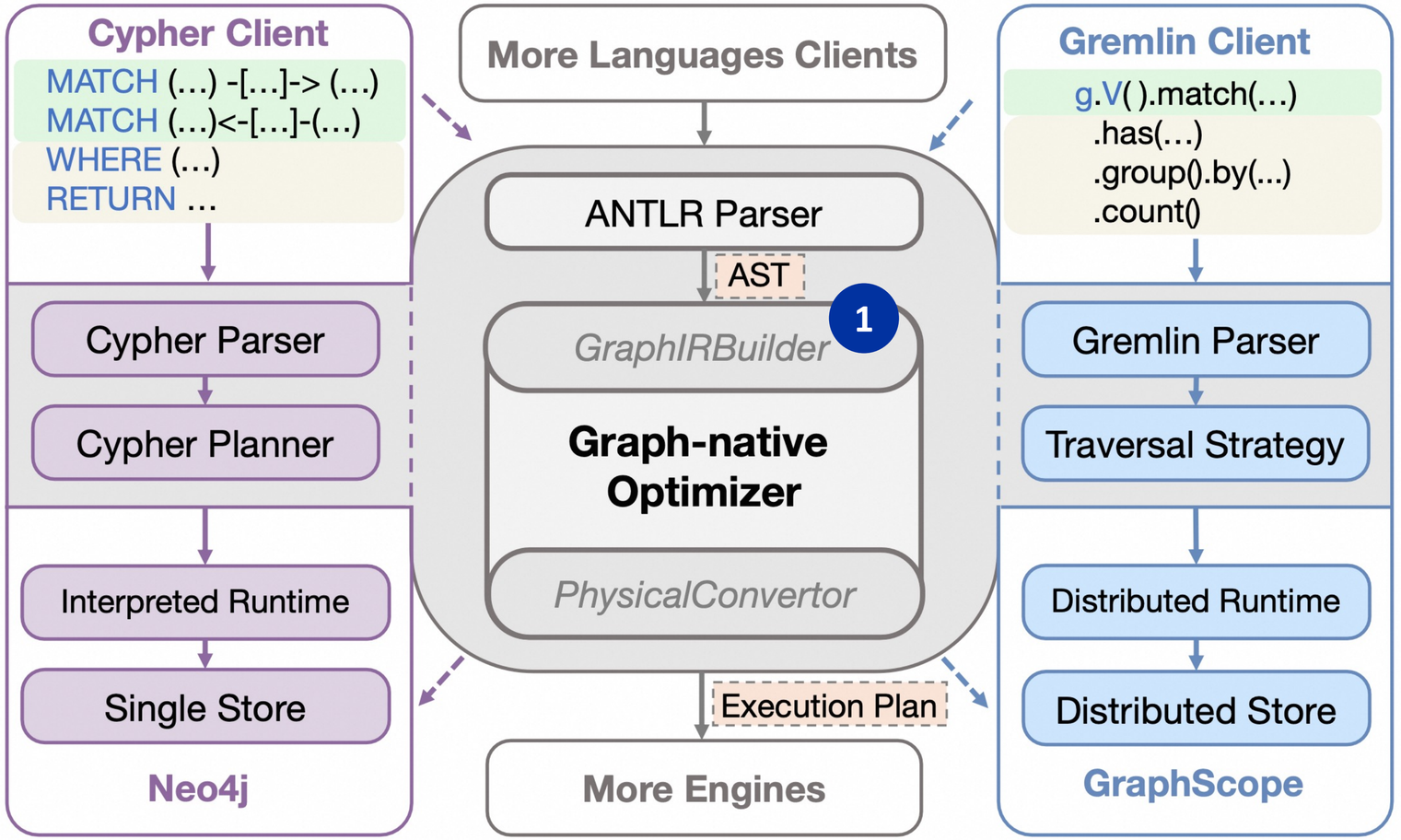
Relational Optimizer



[1] Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins, Amine Mhedhbi, Semih Salihoğlu, VLDB 2019

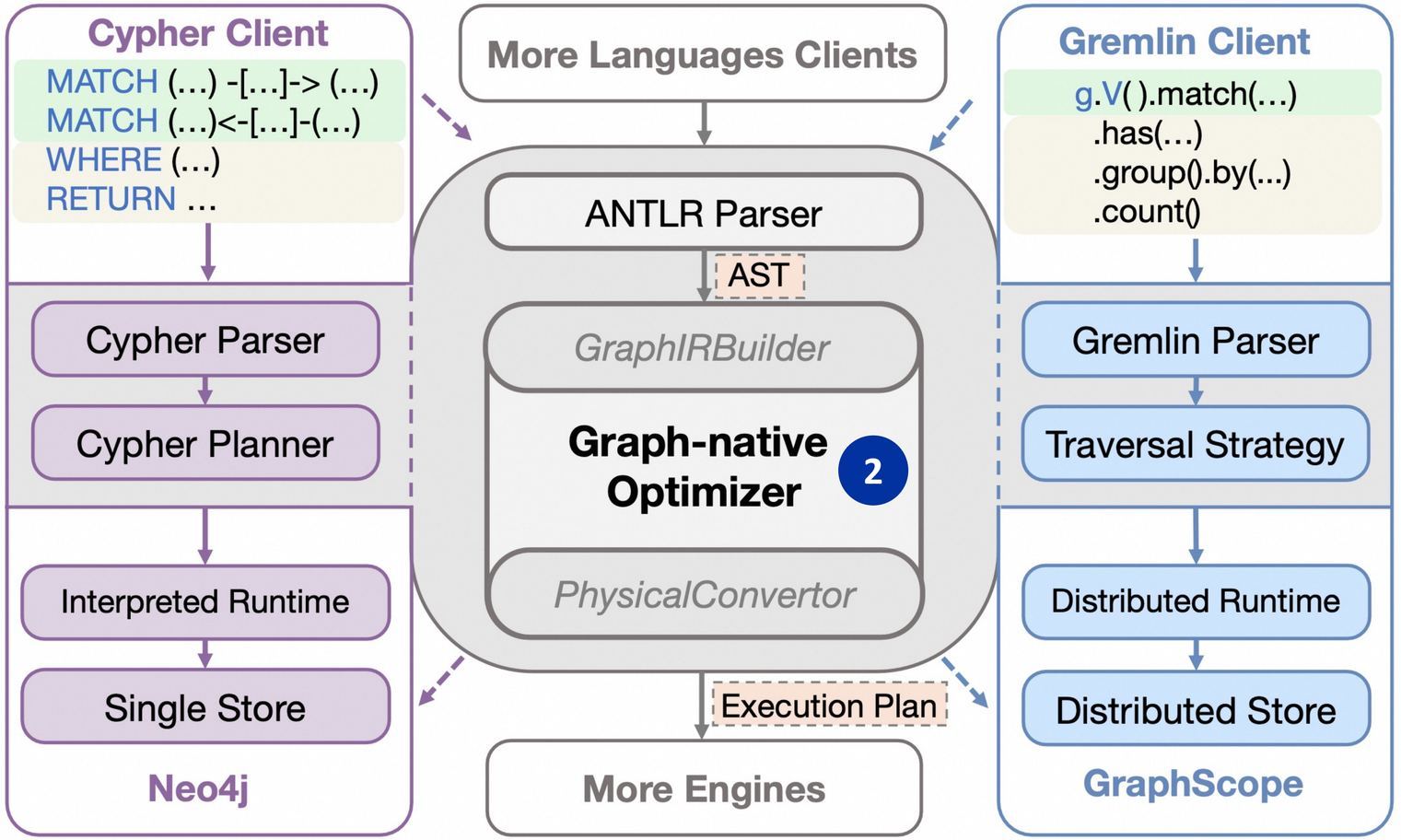
[2] Glogs: interactive graph pattern matching query at large scale, Longbin Lai et al, ATC 2023

GOpt: A Modular Graph-native Optimization Framework



1 A unified **Graph Intermediate Representation (GIR)**: uniform the logical expression of various graph query languages (Cypher, Gremlin)

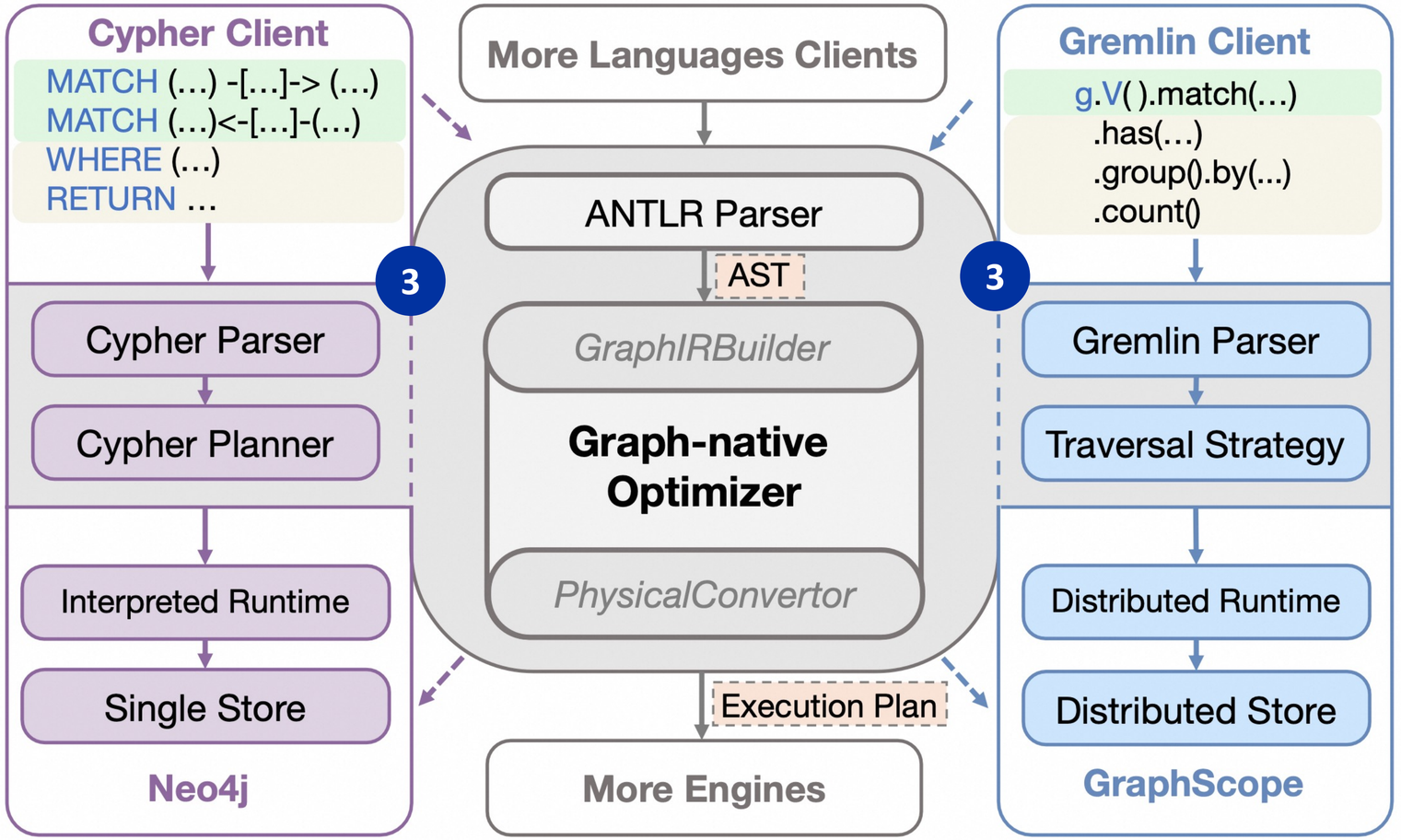
GOpt: A Modular Graph-native Optimization Framework



1 A unified **Graph Intermediate Representation (GIR)**: uniform the logical expression of various graph query languages

2 Advanced **graph query optimization techniques**: hybrid join, high-order statistics, and type inference

GOpt: A Modular Graph-native Optimization Framework



1. A unified **Graph Intermediate Representation (GIR)**: uniform the logical expression of various graph query languages

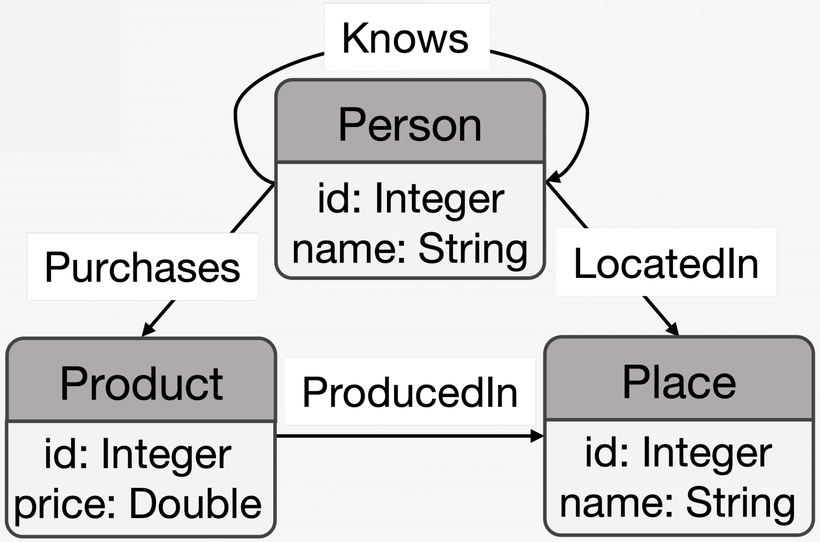
2. Advanced **graph query optimization techniques**: hybrid join, high-order statistics, and type inference

3. **Modular Design**: Enable easy integration into existing systems (Neo4j and GraphScope) through modular API design

阿里云

CONTENTS

A Simple Query Example



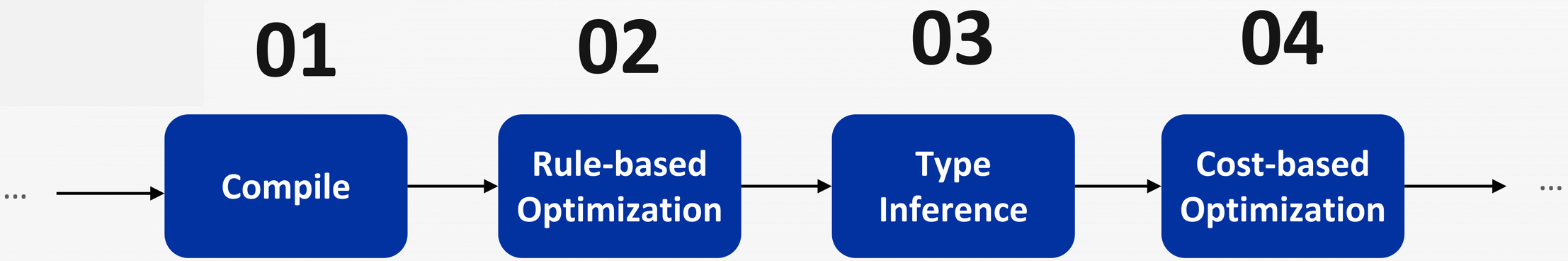
Schema

MATCH a triangle pattern containing a Place vertex, and then **FILTER** with a specified place name, **GROUP** the matchings by the name and the **TOP** 10 results are returned?

```
MATCH (v1) -[e1]-> (v2)
      (v2) -[e2]-> (v3)
MATCH (v1) -[e3]-> (v3:Place)
WHERE v3.name = “China”
WITH v2, COUNT(v2) as cnt
RETURN v2, cnt
ORDER BY cnt LIMIT 10
```

User Query

Typical Optimization Pipeline in GOpt



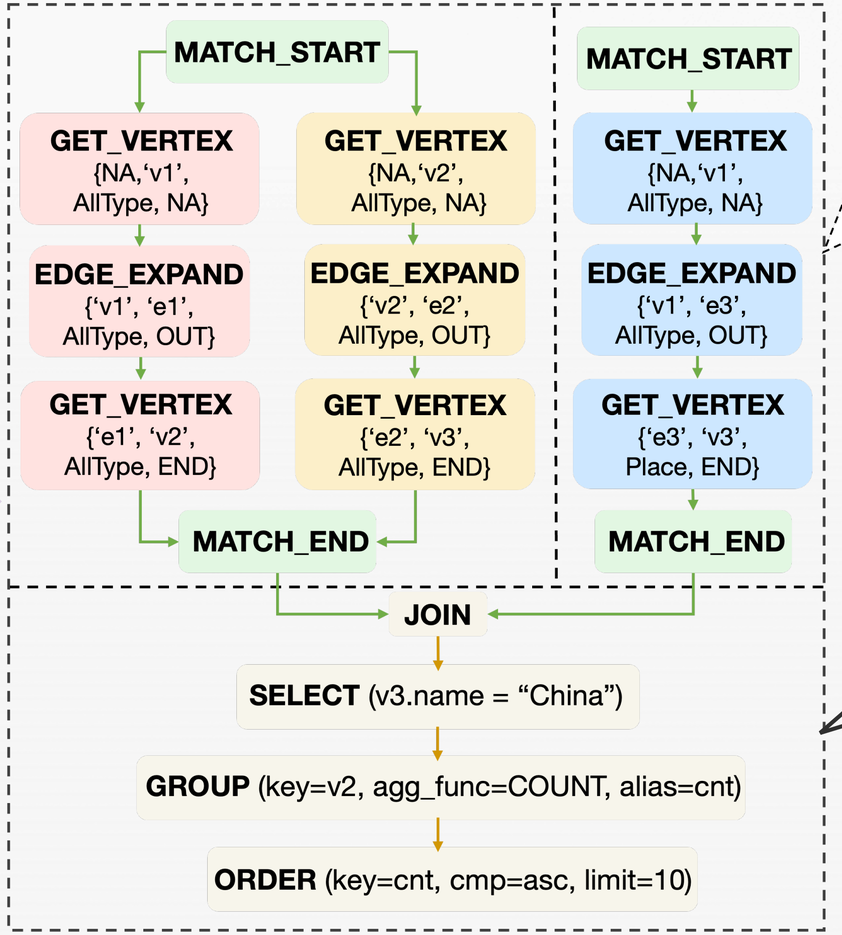
NOTE: Case can be more complex in practice

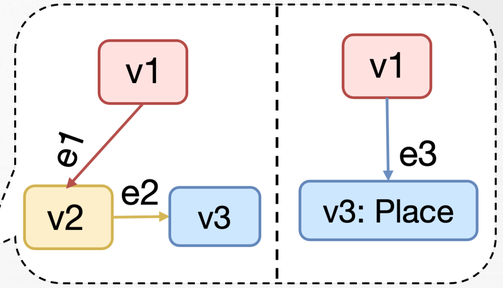
Step 1: Compile (Graph Intermediate Representation, GIR)

Context for pattern matching, including graph operators

```
MATCH  (v1) -[e1]-> (v2)
       (v2) -[e2]-> (v3)
MATCH  (v1) -[e3]-> (v3:Place)
WHERE v3.name = “China”
WITH v2, COUNT(v2) as cnt
RETURN v2, cnt
ORDER BY cnt LIMIT 10
```

Compile





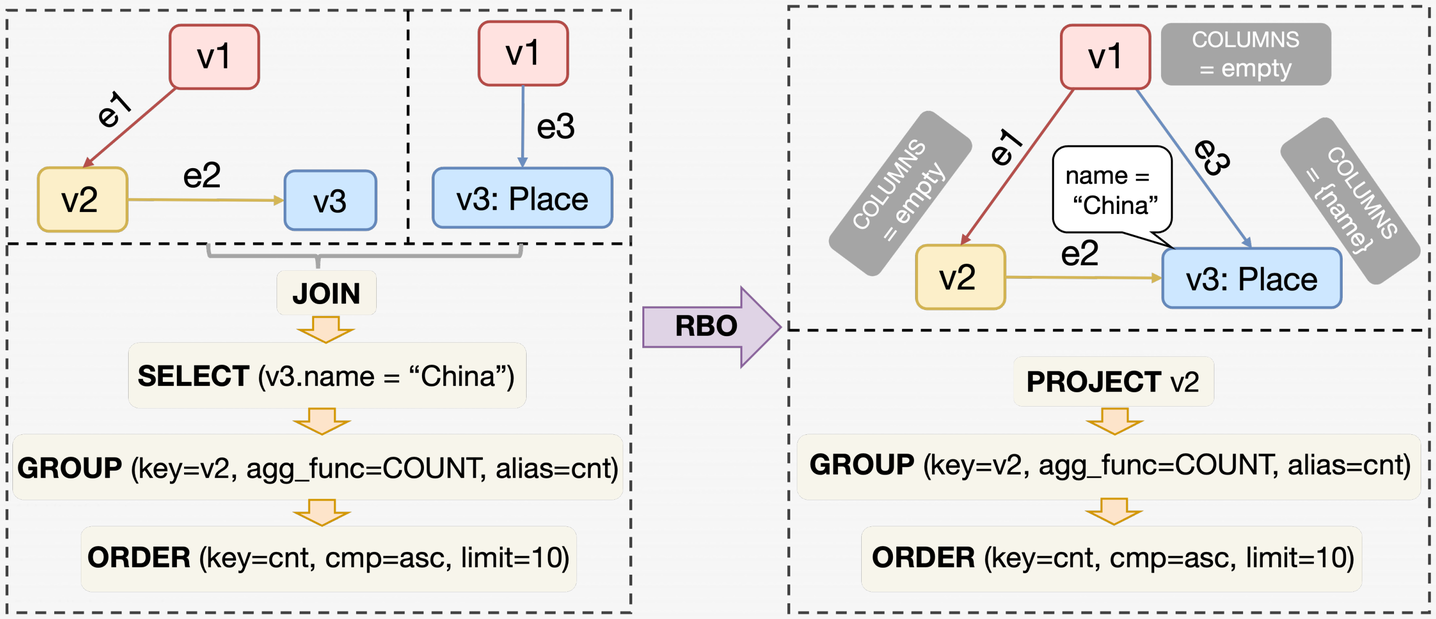
Relational Operators

(a) Cypher Query

(b) Logical Plan in GIR

Step 2: Rule-based Optimization (RBO)

- **Built-in Predefined Rules**
- **Customized User-Defined Rules**

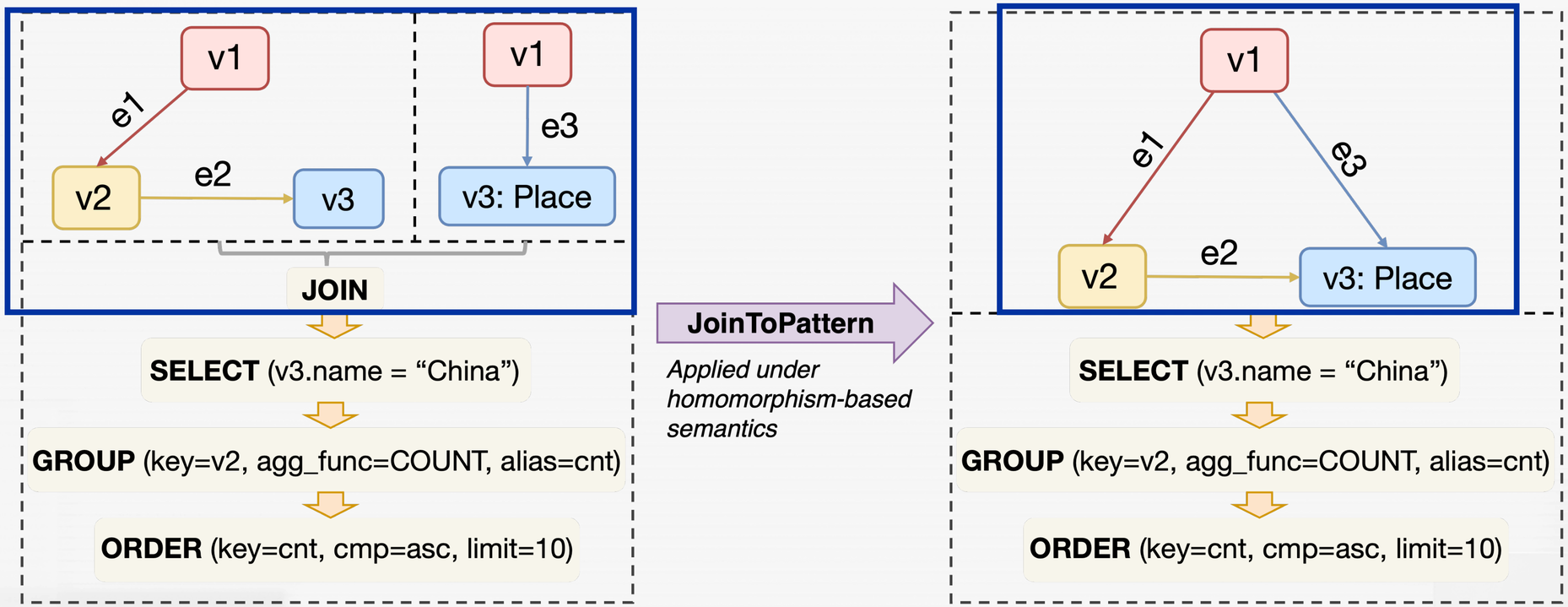


(a) Logical Plan

(b) Optimized Plan After RBO

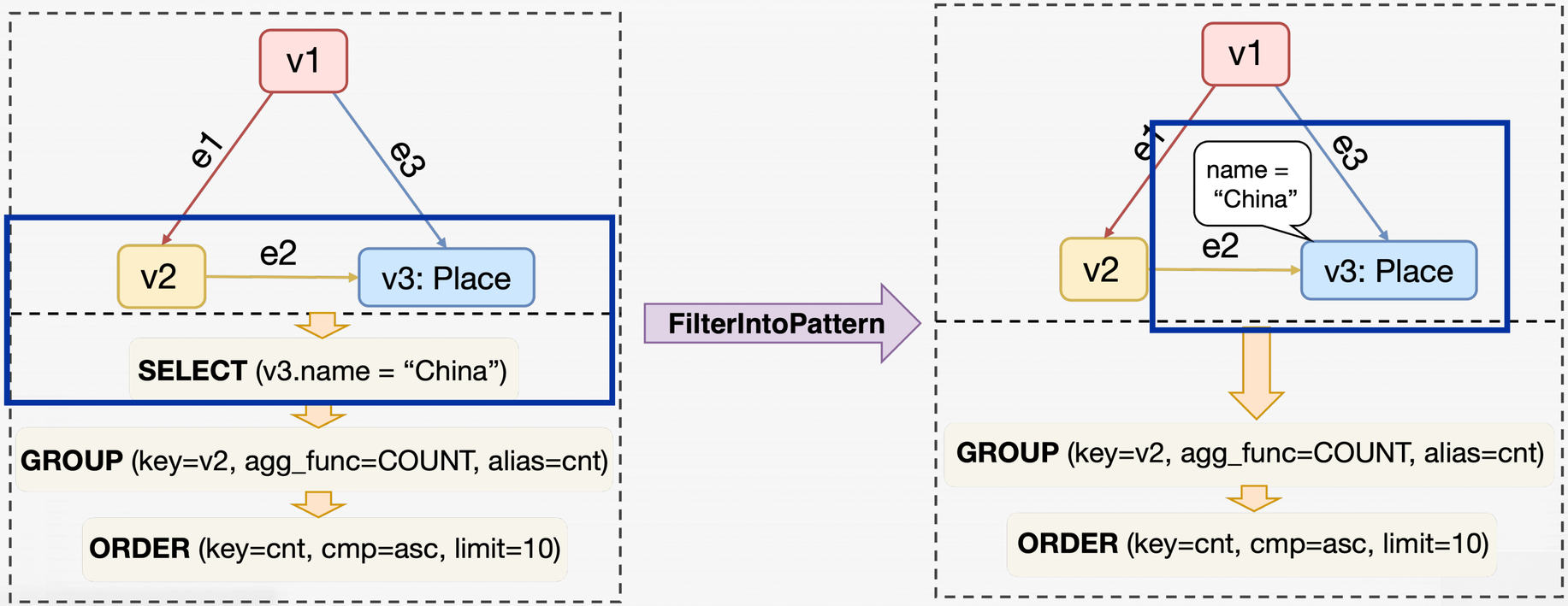
Step2: RBO (Cont.)

Built-in Rule: JoinToPattern

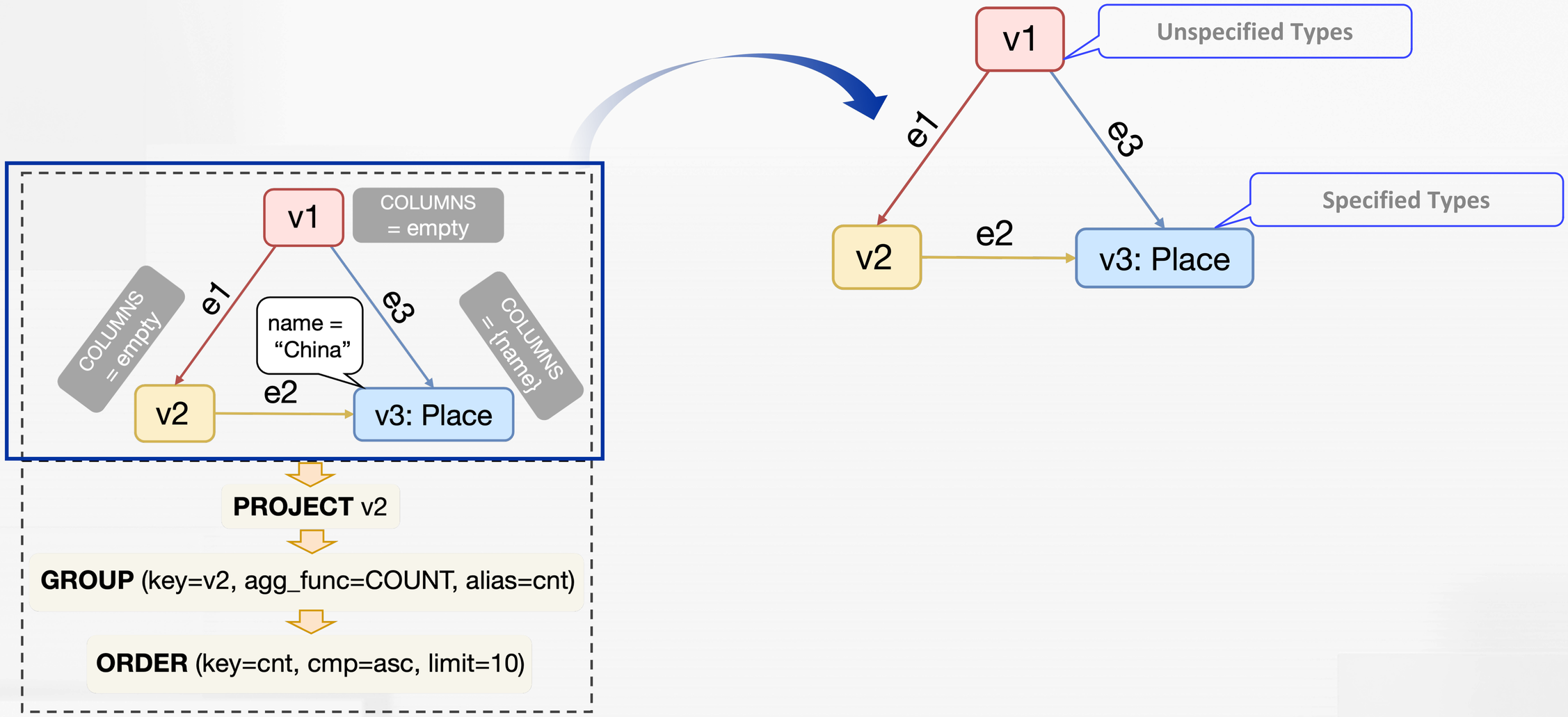


Step 2: RBO (Cont.)

Build-in Rule: FilterIntoPattern

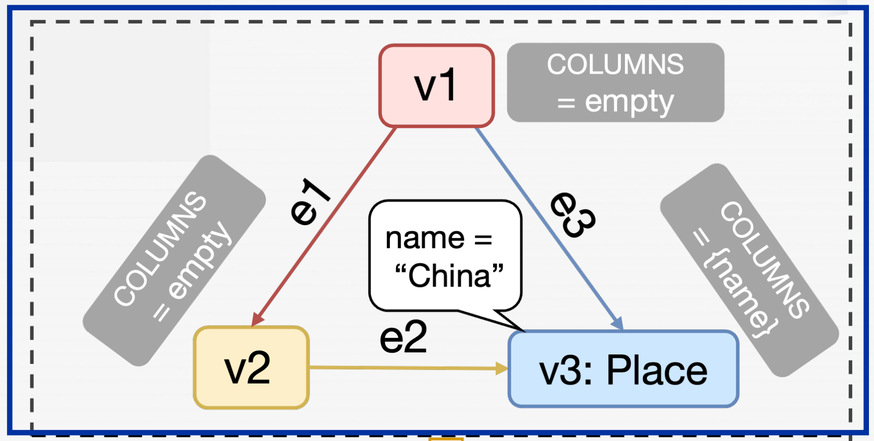


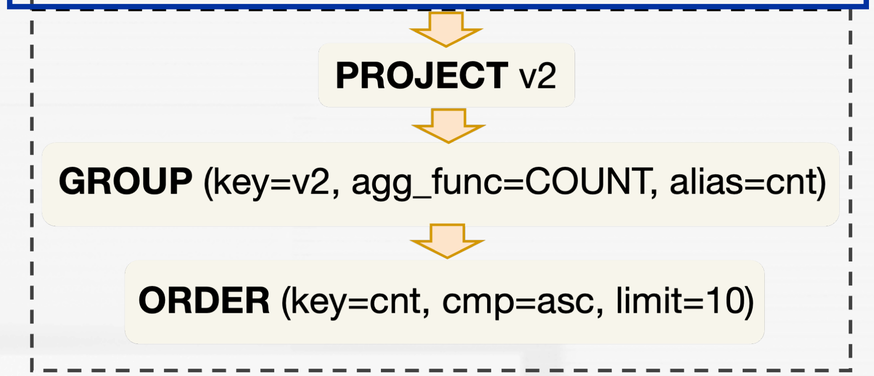
Step 3: Type Inference



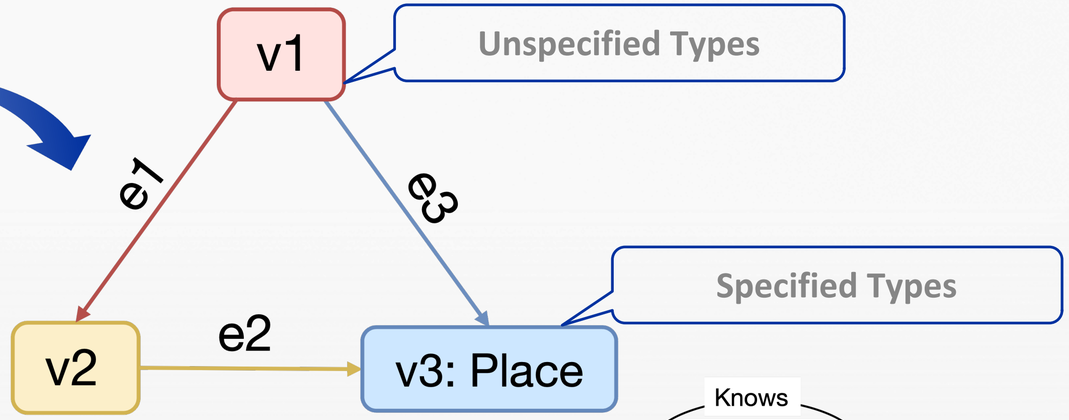
Optimized Plan After RBO

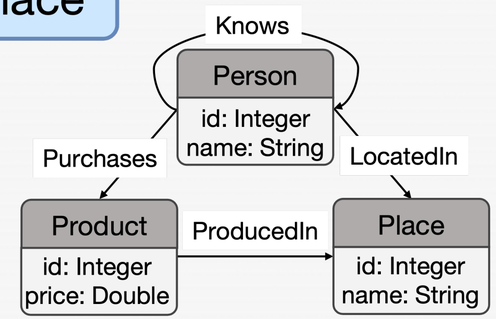
Step 3: Type Inference (Cont.)





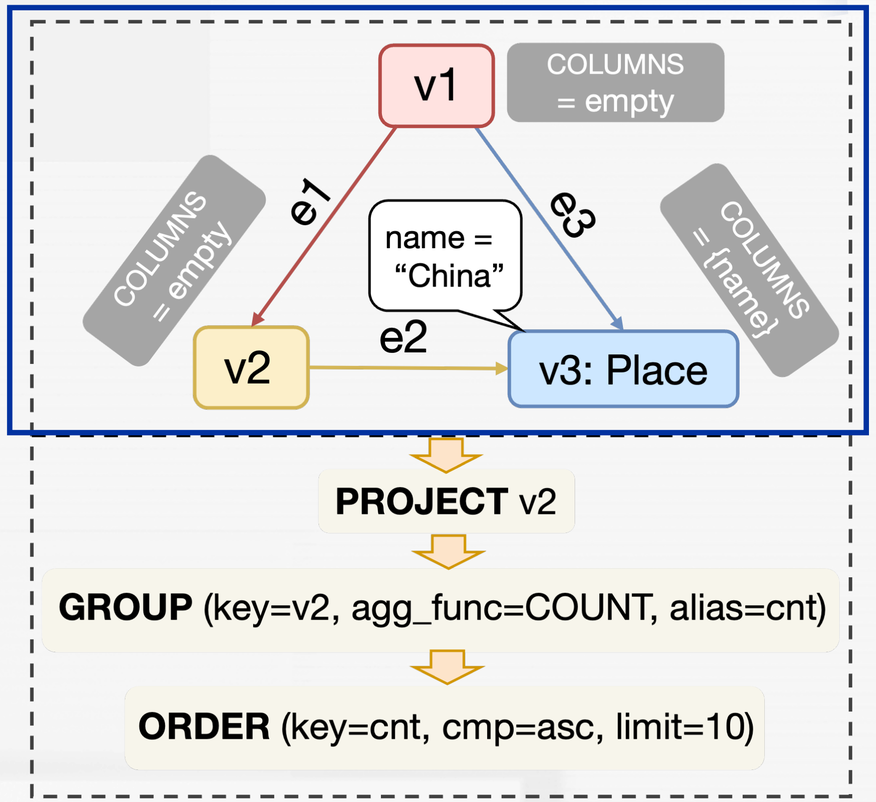
Optimized Plan After RBO



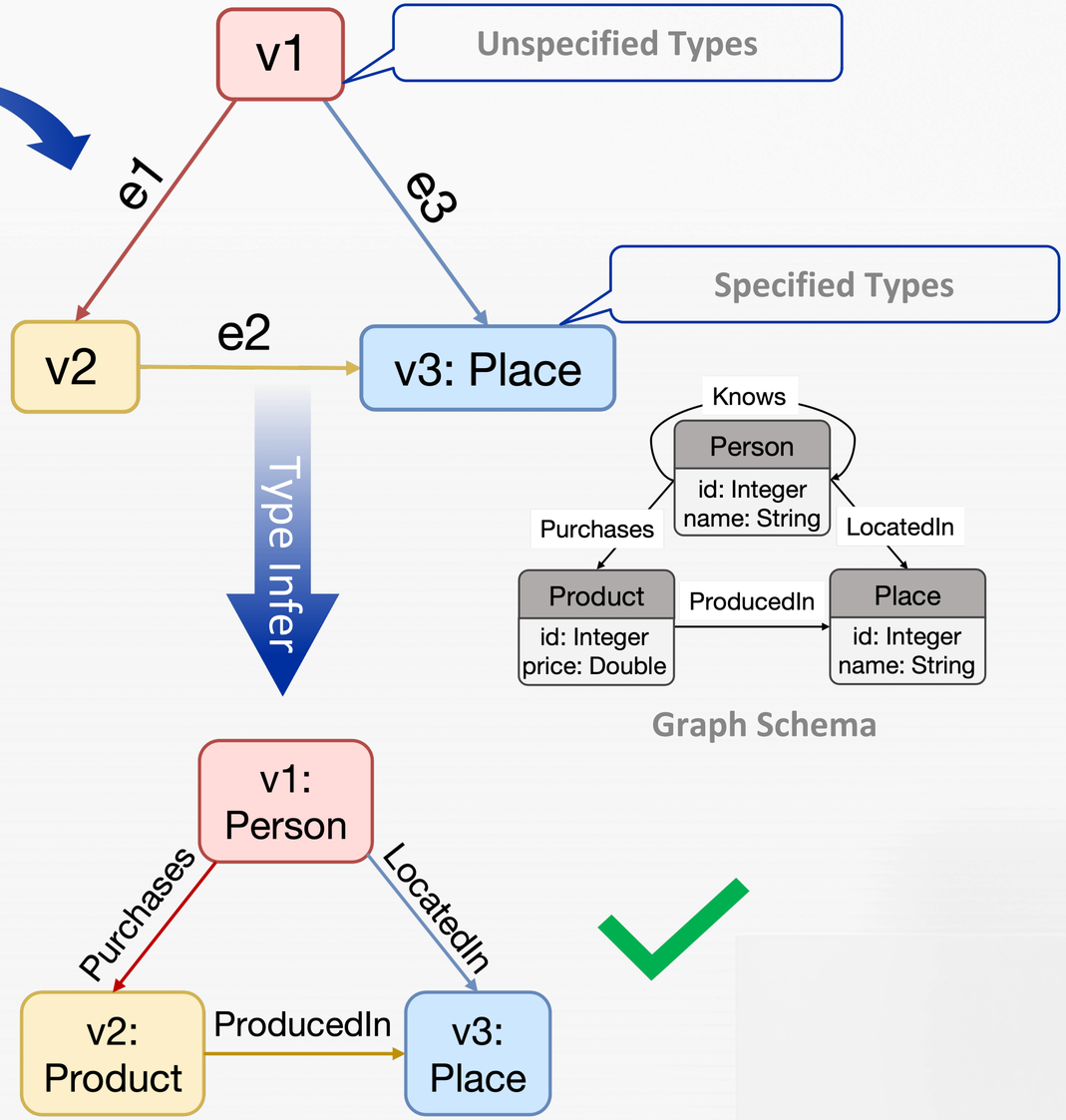


Graph Schema

Step 3: Type Inference (Cont.)

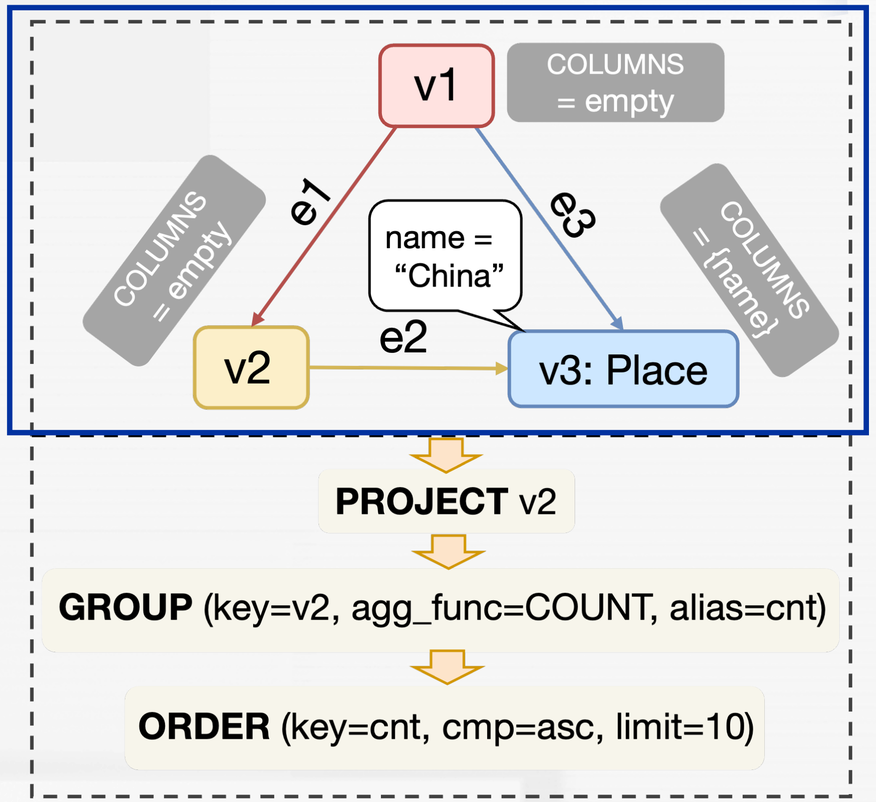


Optimized Plan After RBO

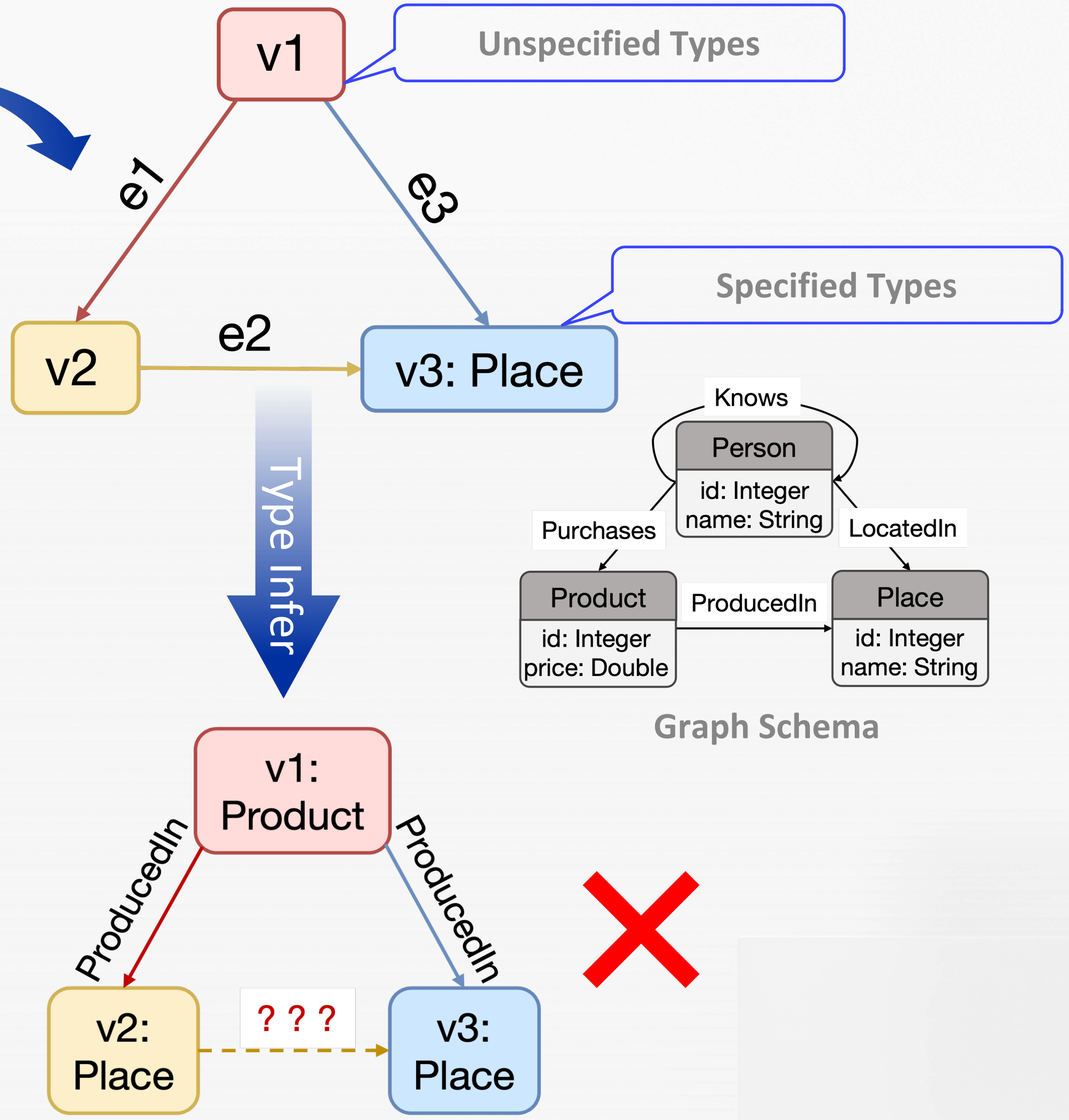


Graph Schema

Step 3: Type Inference (Cont.)

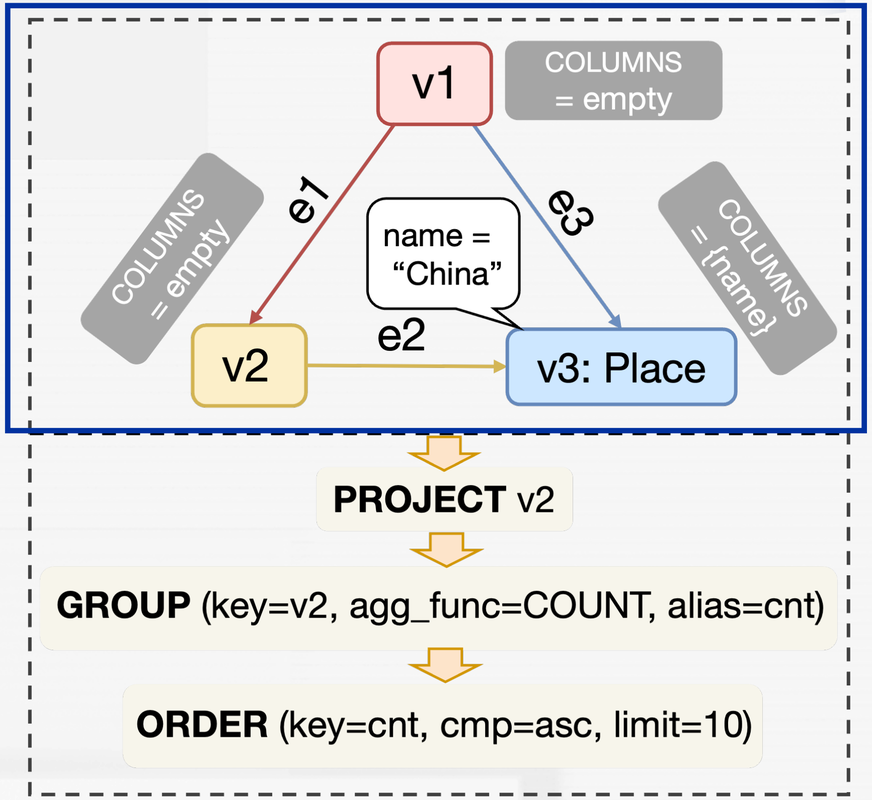


Optimized Plan After RBO

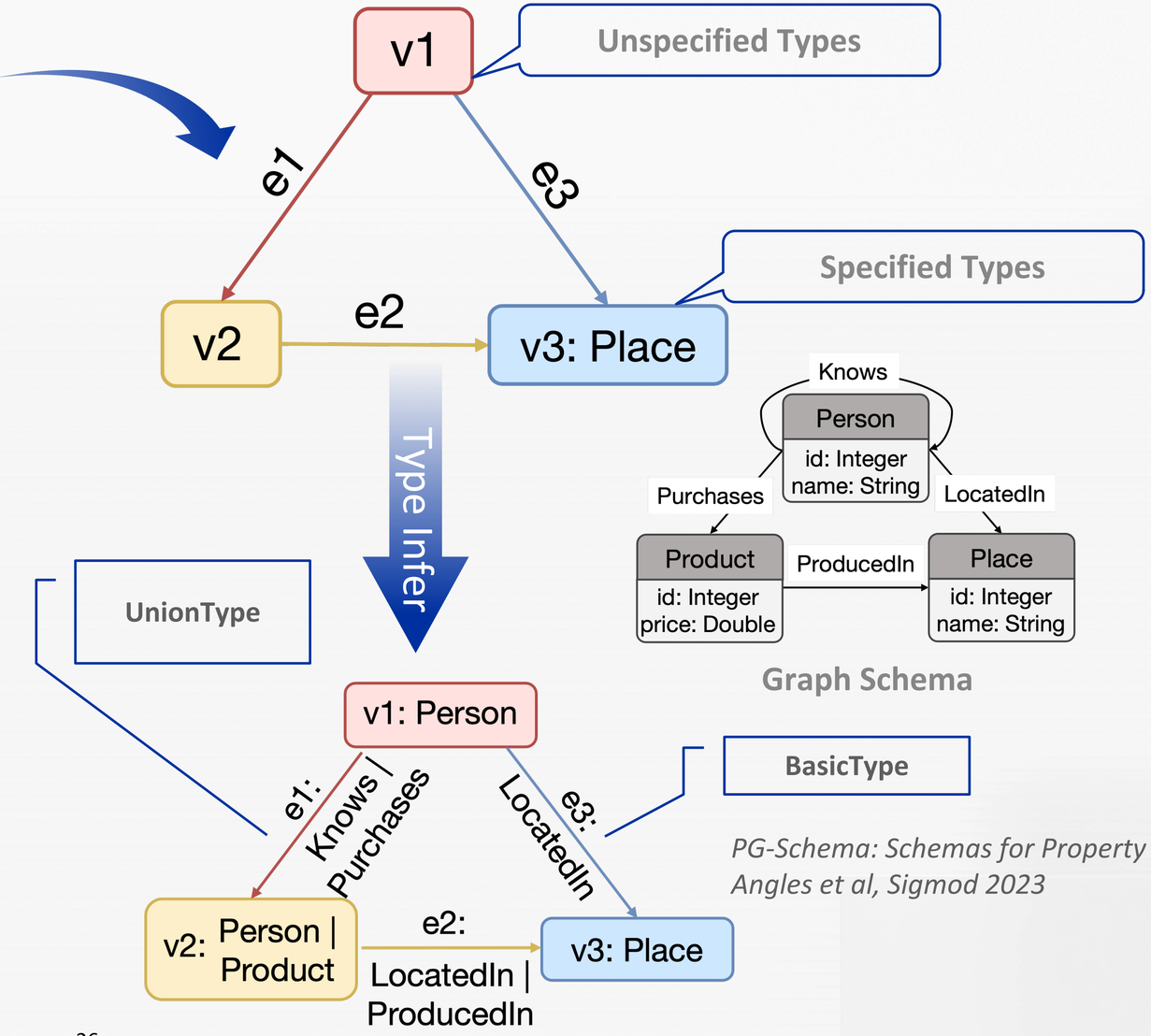


Graph Schema

Step 3: Type Inference (Cont.)



Optimized Plan After RBO

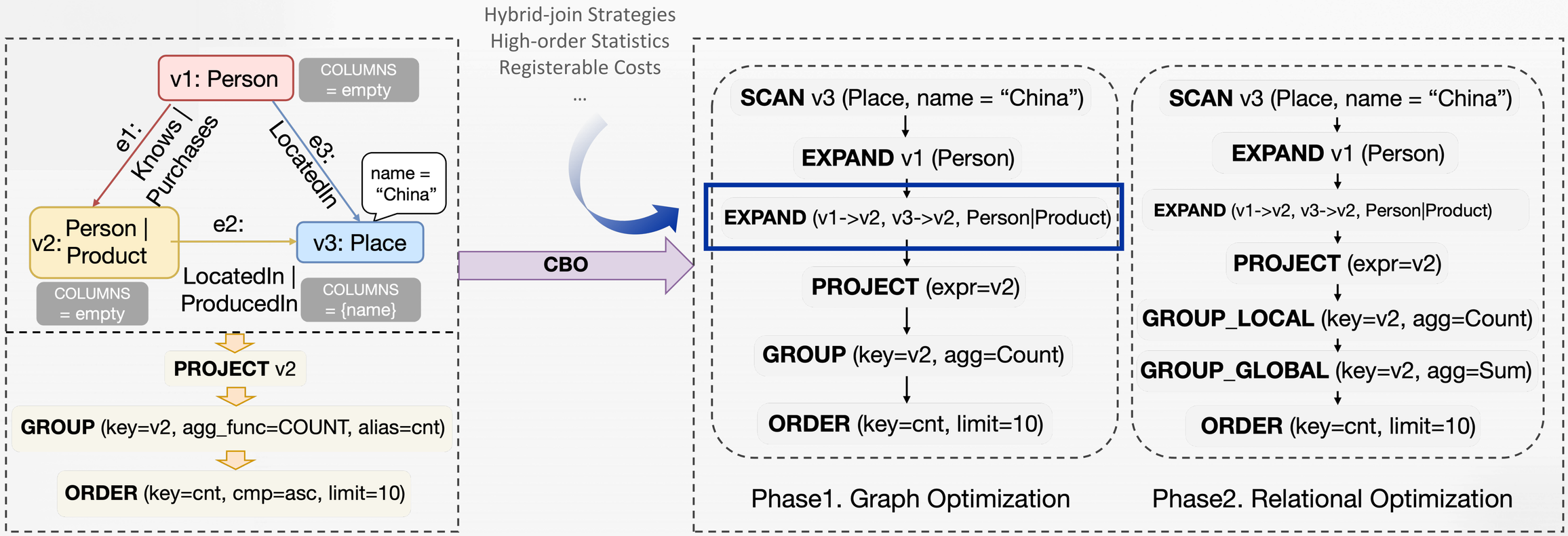


Graph Schema

PG-Schema: Schemas for Property Graphs, Angles et al, Sigmod 2023

Step 4: Cost-based Optimization (CBO)

GLogS: Interactive Graph Pattern Matching Query At Large Scale, Lai et al, ATC 2023



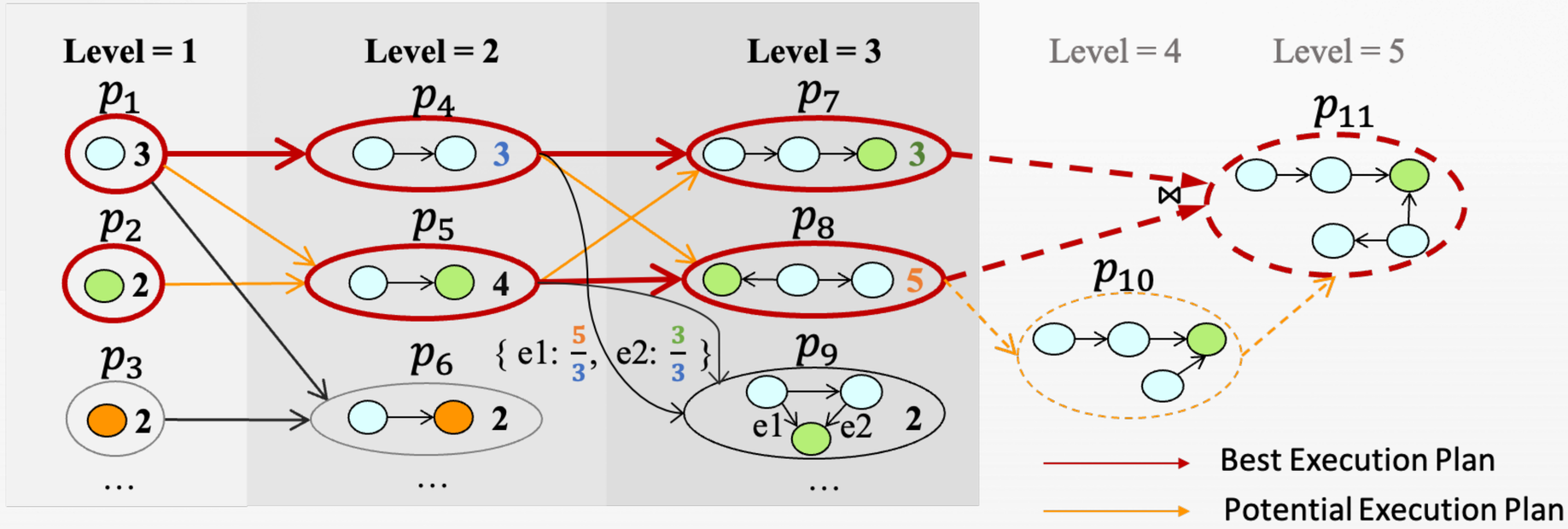
(a) Optimized Plan After Type Infer

(b) Optimized Plan After CBO

Step 4: CBO (Cont.) : Cost Model

- $\mathrm{Plan}(p) = (\Phi = \{p_1, p_2, \ldots, p_n = p\}, \Gamma = [\tau_1, \tau_2, \ldots, \tau_m])$
  - p_i: intermediate pattern
  - τ_i: operation on intermediate patterns, Join or Vertex Expansion
  - $\mathcal{F}(p_i)$: Frequency(count number) of the p_i in the graph.
  - $\mathrm{Cost}(\mathrm{Plan}(p)) = \sum_{p' \in \Phi} \mathcal{F}(p') + \sum_{\tau \in \Gamma} \mathrm{Cost}(\tau)$
- GLogueQuery provides $\mathcal{F}(p_i)$ based on **high-order statistics**
  - GLogue provides precomputed statistics of small patterns
  - If pattern is not in GLogue , compute by $\mathcal{F}(\mathrm{p}) = \mathrm{Avg}_{\mathrm{p}_1,\mathrm{p}_2} \frac{\mathcal{F}(\mathrm{p}_1) \times \mathcal{F}(\mathrm{p}_2)}{\mathcal{F}(\mathrm{p}_1 \cap \mathrm{p}_2)}, \mathrm{p} = \mathrm{p}_1 \cup \mathrm{p}_2$
- **$\mathrm{Cost}(\tau_i)$ is registrable** according to backend implementations, e.g.,
  - Neo4j (ExpandInto): $\mathrm{Cost}(\mathrm{Expand}(p_s \to p_t)) = \alpha_{ve} \mathcal{F}(p_s) \sum_{i=1}^{k} \overline{\sigma_{e_i}}$
  - GraphScope (WCOJ): $\mathrm{Cost}(\mathrm{Expand}(p_s \overset{28}{\Rightarrow} p_t)) = n\mathcal{F}(p_1)$

Step 4: CBO (Cont.): Plan Optimizer



A top-down search framework utilizing registered operator costs and cardinality estimation from GLogueQuery to find optimal plans with minimum estimated costs

阿里云

CONTENTS

01

Background

02

GOpt Overview

03

GOpt Integration

- GOpt Architecture
- Integration with Neo4j
- Integration with GraphScope

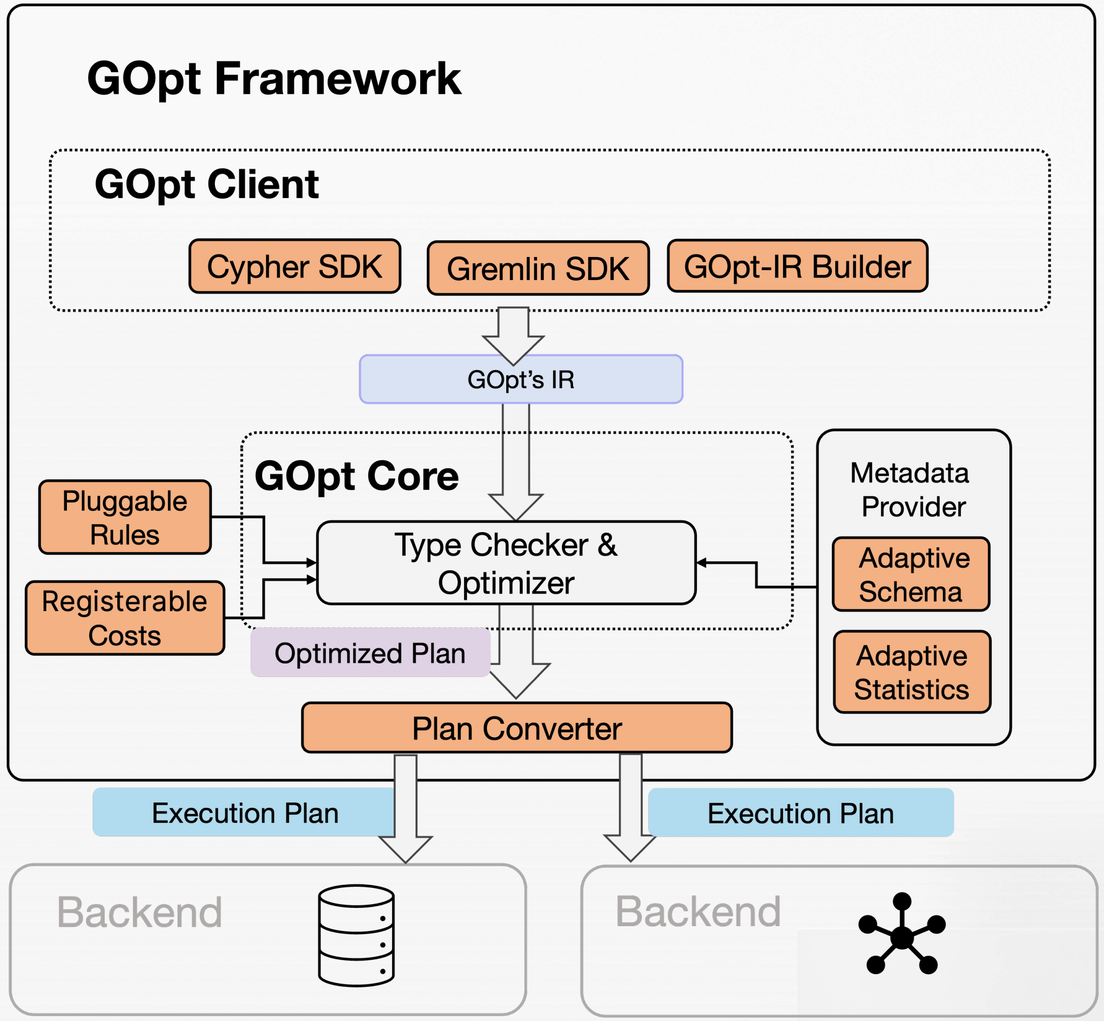
04

Experimental Results

System Architecture of GOpt

User-friendly interfaces in GOpt

- **Input:** various clients to the unified GIR
- **Output:** convert to alternative execution plans
- **Metadata:** how to retrieve schema and statistics
- **Rules:** heuristic rules for using in RBO
- **Costs:** cost of physical operations for using in CBO



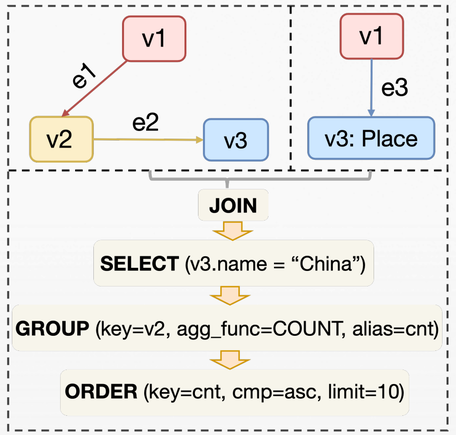
GOpt Integration – Neo4j

Input

```
MATCH (v1) -[e1]-> (v2)
      (v2) -[e2]-> (v3)
MATCH (v1) -[e3]-> (v3:Place)
WHERE v3.name = “China”
WITH v2, COUNT(v2) as cnt
RETURN v2, cnt
ORDER BY cnt LIMIT 10
```

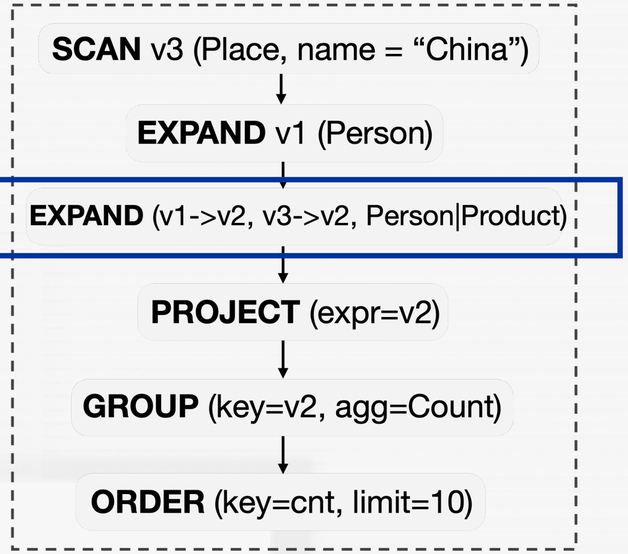
Cypher Query

Antlr Tool & GIR-Builder



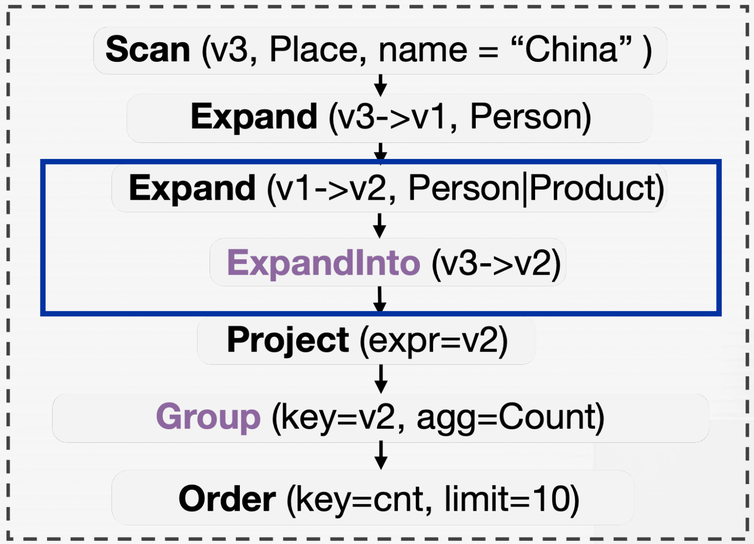
Logical Plan

Output



Optimized Plan

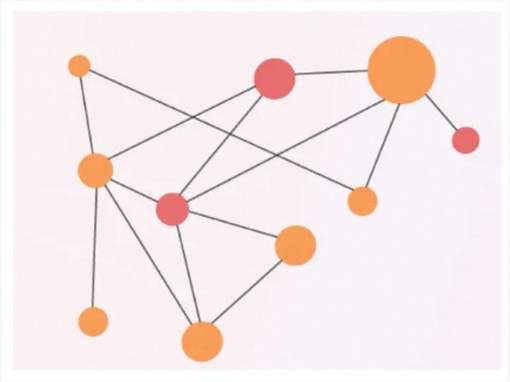
Physical Converter



Neo4j’s Execution Plan

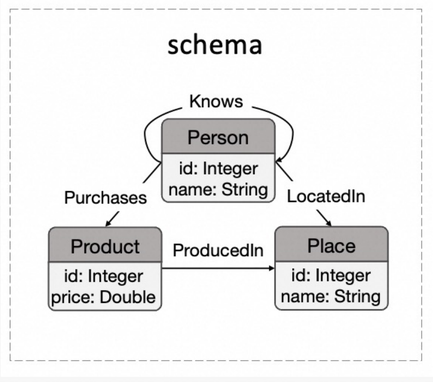
GOpt Integration – Neo4j (Cont.)

Metadata



Data Graph

Neo4j APOC



Graph Schema

Rules

FieldTrim | FilterIntoPattern | ...

Build-in Rules

TrimTypeFilter

User-define Rules

To remove unnecessary type filtering in Neo4j

Costs

$$\mathrm{Cost}\left(\mathrm{ExpandInto}(p_s \rightarrow p_t)\right) = \alpha_{ve}\mathcal{F}(p_s)\sum_{i=1}^{k}\overline{\sigma_{e_i}}$$

...

Operator Costs according to Neo4j's Implementations

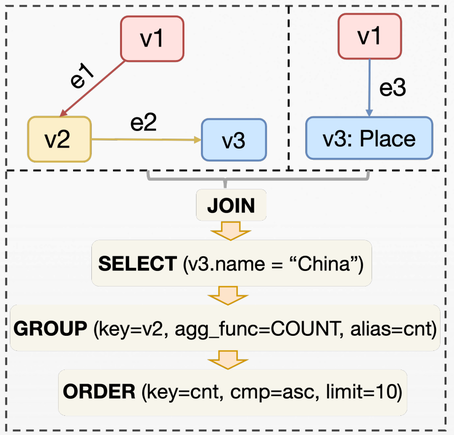
GOpt Integration – GraphScope

Input

```
g.V( ).match(
    __.as(‘v1’).out( ).as(‘v2’),
    __.as(‘v2’).out( ).as(‘v3’))
.match(
    __.as(‘v1’).out( ).as(‘v3’))
.select(‘v3’).has(‘name’, ‘China’)
.hasLabel(‘Place’)
.groupCount( ).by(‘v2’)
.order( ).by(values).limit(10)
```

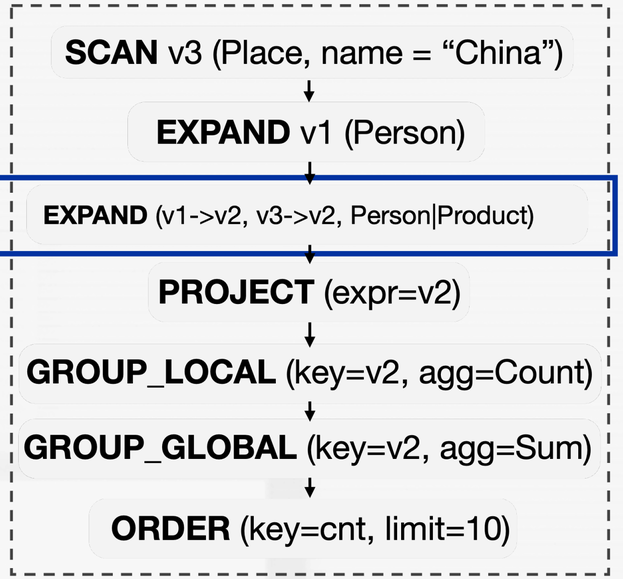
Gremlin Query

Antlr Tool & GIR-Builder



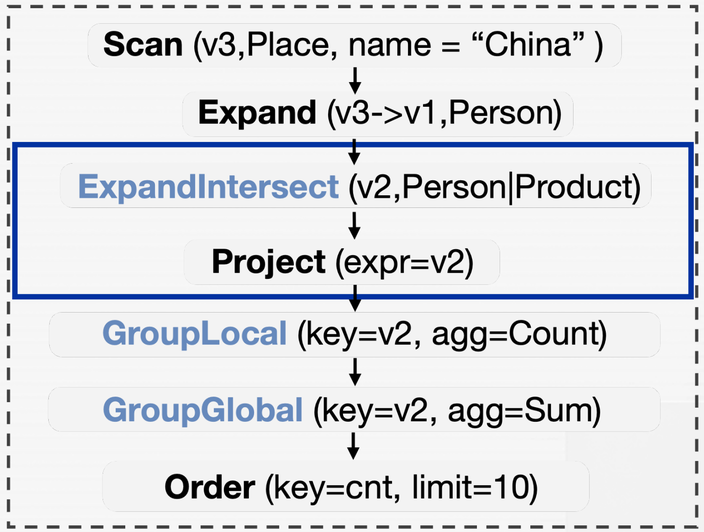
Logical Plan

Output



Optimized Plan

Physical Converter



GraphScope’s Execution Plan

GOpt Integration – GraphScope (Cont.)

Rules

FilterIntoPattern | FieldTrim | JoinToPattern | ...

Build-in Graph Rules

LocalGlobalAggregation | ...

Build-in Relational Rules

Costs

$\mathrm{Cost}\big(\mathrm{ExpandIntersect}(p_s \to p_t)\big) = n\mathcal{F}(p_1)$ | ...

Operator Costs according to GraphScope's Implementations

阿里云

CONTENTS

Experimental Settings

Dataset

LDBC Graph scaled at sf=30, 100, 300, 1000, up to **2.7 billion** vertices and **17.8 billion** edges.

Queries

LDBC Interactive (IC) and **Business Intelligence (BI)** workloads, challenging the systems with real-world scenarios.

Compared Systems

- **Neo4j** with Cypher on a single machine,
- **GraphScope** with Gremlin on distributed machines

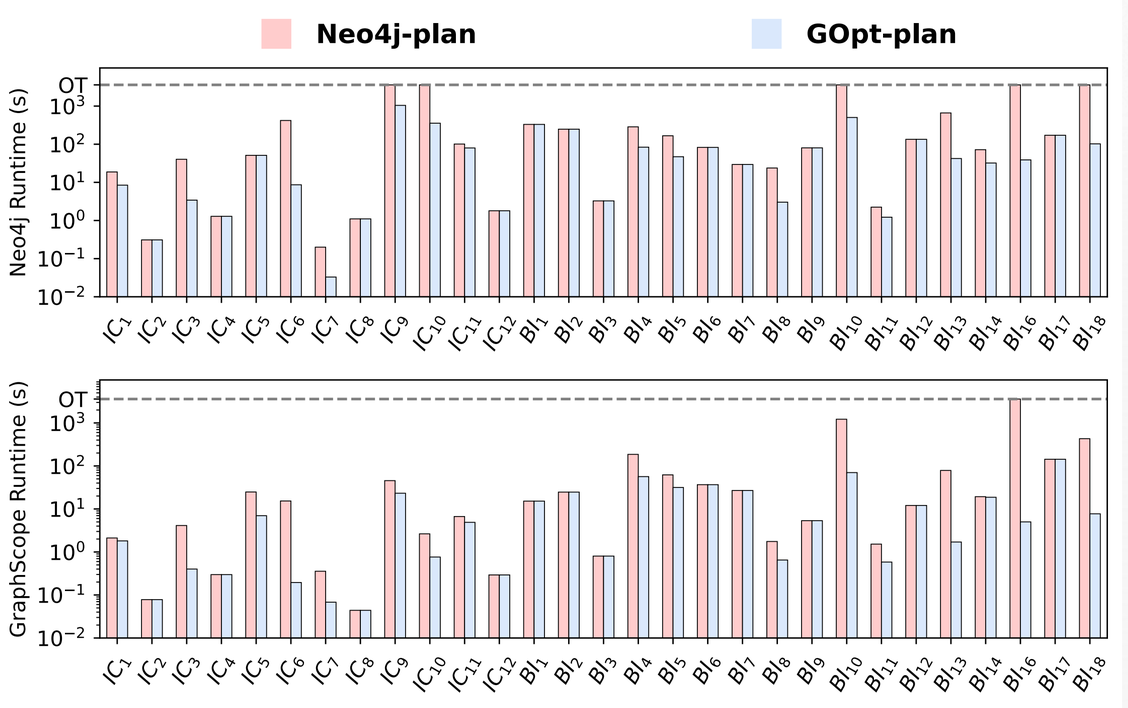
Optimization Comparison

- **Neo4j-Plan**: Plans by Neo4j Optimizer
- **Gopt-plan**: Plans by GOpt

Experimental Results: Comprehensive Comparison

- Neo4j-plan v.s. GOpt-plan on Neo4j
  - GOpt-plan outperformed Neo4j-plan in **16/29** queries
  - Avg. speedup 9.2x
  - Max. speedup **48.6x** (IC6)

- Neo4j-plan v.s. GOpt-plan on GraphScope
  - GOpt-plan outperformed Neo4j-plan in **17/29** queries
  - Avg. speedup 33.4x
  - Max. speedup **78.7x** (IC6) since hybrid-join strategy and optimized wcoj implementation



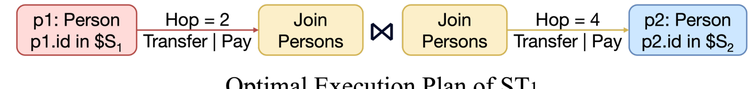
Comparison of Neo4j-plan and GOpt-plan on Neo4j and GraphScope resp.

Case Study: Money Mule

Money Mule: Fraudsters transfer funds through multiple intermediaries before another fraudster withdraws the money

```
MATCH (p1: PERSON)-[p:*6]->(p2: PERSON)
WHERE p1.id IN $S1 AND p2.id IN $S2
RETURN p
```

(a) Cypher Query Money Mule Detection

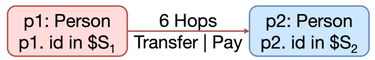


Optimal Execution Plan of ST_1

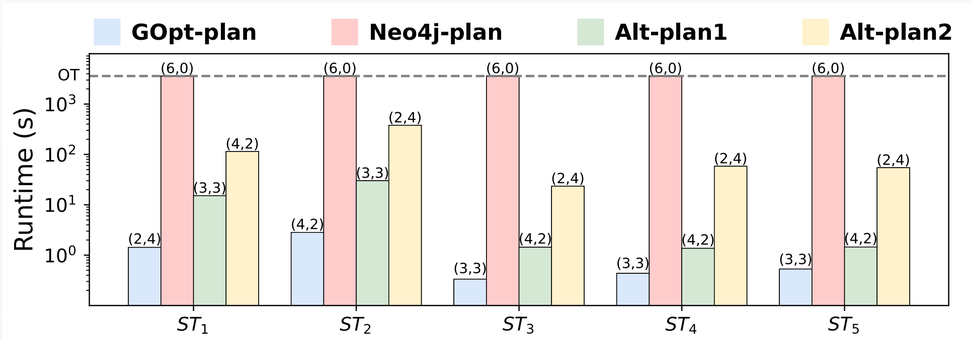
Optimal Execution Plan of ST_2

Optimal Execution Plan of ST_3, ST_4, ST_5

(b) GOpt-Plan for ST_1 to ST_5 with Different Src/Dst Vertices



(c) Neo4j-Plan for ST_1 to ST_5 with Different Src/Dst Vertices
(all plans are single direction expansion)

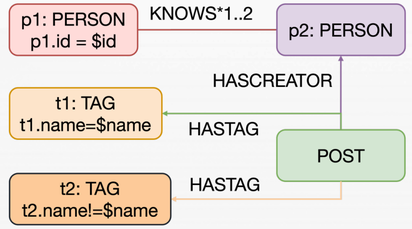


- GOpt **automatically** determines the best join vertex based on CBO
- GOpt-plans outperform alternatives by **3x - 20x**
- All Neo4j-plans run OT

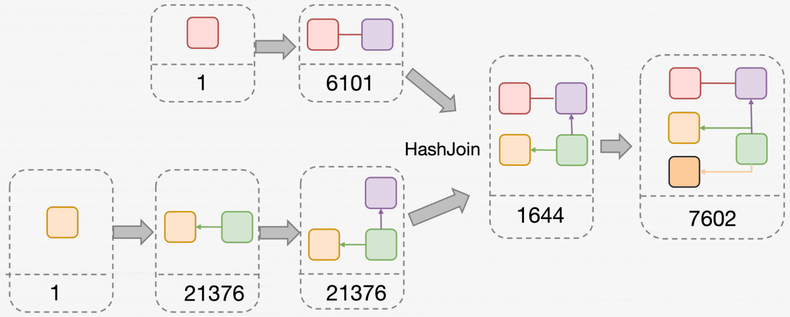
Case Study (cont.): LDBC SNB Bnechmark

The Case of IC6 in LDBC SNB

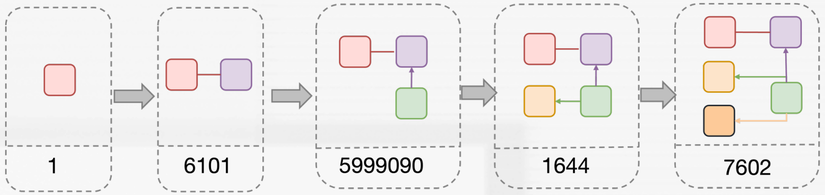
```
MATCH (p1:PERSON {id: $id})-[:KNOWS*1..2]-(p2:PERSON),
      (p2)<-[:HASCREATOR]-(p:POST),
      (p)-[:HASTAG]->(t1:TAG {name: $tagName}),
      (p:POST)-[:HASTAG]->(t2:TAG)
WHERE p2.id <> $id AND t2 <> t1
WITH DISTINCT t2, p
RETURN t2.name as name, count(p) as postCnt
ORDER BY postCnt desc, name asc
LIMIT 10;
```



(a) Cypher Query and Graph Pattern of IC6



(b) Execution Order of IC6 in GOpt-plan



(c) Execution Order of IC6 in Neo4j-plan

AUDITED RESULTS USING A DECLARATIVE QUERY LANGUAGE

Benchmark setup	SF	Hardware	Performance	Performance (price-adj.)	Documents
System: **GraphScope Flex 0.31.3** Test sponsor: **Alibaba Cloud** Date: **2025-04-21** Query language: **Cypher** System cost: **808,444.48 RMB** (110.745,82 USD)	30	Alibaba Cloud ecs.g8a.48xlarge 96×AMD EPYC 9T24 @ 3.7GHz vCPUs, 768GiB RAM	37,650.00 ops/s	339.97	Full disclosure report Executive summary Signatures Supplementary package
	100	Alibaba Cloud ecs.g8a.48xlarge 96×AMD EPYC 9T24 @ 3.7GHz vCPUs, 768GiB RAM	79,244.18 ops/s	715.55	
	300	Alibaba Cloud ecs.g8a.48xlarge 96×AMD EPYC 9T24 @ 3.7GHz vCPUs, 768GiB RAM	80,510.79 ops/s	726.99	

The full disclosure report from LDBC

More Details in GOpt Technique Report

阿里云

THANKS

Please star us on GraphScope. We will soon announce a **new open-source product** for better graph data management