Kaskade: Graph Views for Efficient Graph Analytics

Joana M. F. da Trindade*, Konstantinos Karanasos†, Carlo Curino†, Samuel Madden*, Julian Shun*
*MIT CSAIL {jmf, jshun, madden}@csail.mit.edu, †Microsoft {ccurino, kokarana}@microsoft.com

***Abstract*—Graphs are a natural way to model real-world entities and relationships between them, ranging from social networks to data lineage graphs and biological datasets. Queries over these large graphs often involve expensive sub-graph traversals and complex analytical computations. These real-world graphs are often substantially more structured than a generic vertex-and-edge model would suggest, but this insight has remained mostly unexplored by existing graph engines for graph query optimization purposes. In this work, we leverage structural properties of graphs and queries to automatically derive materialized *graph views* that can dramatically speed up query evaluation. We present KASKADE, the first graph query optimization framework to exploit materialized graph views for query optimization purposes. KASKADE employs a novel *constraint-based view enumeration* technique that mines constraints from query workloads and graph schemas, and injects them during view enumeration to significantly reduce the search space of views to be considered. Moreover, it introduces a graph view size estimator to pick the most beneficial views to materialize given a query set and to select the best query evaluation plan given a set of materialized views. We evaluate its performance over real-world graphs, including the provenance graph that we maintain at Microsoft to enable auditing, service analytics, and advanced system optimizations. Our results show that KASKADE substantially reduces the effective graph size and yields significant performance speedups (up to 50X), in some cases making otherwise intractable queries possible.**

I. INTRODUCTION

Many real-world applications can be naturally modeled as graphs, including social networks [1], workflow, and dependency graphs as the ones in job scheduling and task execution systems [2], [3], knowledge graphs [4], [5], biological datasets, and road networks [6]. An increasingly relevant type of workload over these graphs involves analytics computations that mix traversals and computation, e.g., finding subgraphs with specific connectivity properties or computing various metrics over sub-graphs. This has resulted in several systems being designed to handle complex queries over such graphs [7].

In these scenarios, graph analytics queries require response times on the order of a few seconds to minutes, because they are either exploratory queries run by users (e.g., recommendation or similarity search queries) or they power systems making online operational decisions (e.g., data valuation queries to control replication, or job similarity queries to drive caching decisions). However, many of these queries involve the enumeration of large subgraphs of the input graph, which can easily take minutes to hours to compute over large graphs on modern graph systems. To achieve our target response times over large graphs, new techniques are needed.

We observe that the graphs in many of these applications have an inherent structure: their vertices and edges have specific types, following well-defined schemas and connectivity properties. For instance, social network data might include users, pages, and events, which can be connected only in specific ways (e.g., a page cannot "like" a user), or workload management systems might involve files and jobs, with all files being created or consumed by some job. As we discuss in § I-A, the provenance graph that we maintain at Microsoft has similar structural constraints. However, most existing graph query engines do not take advantage of this structure to improve query evaluation time.

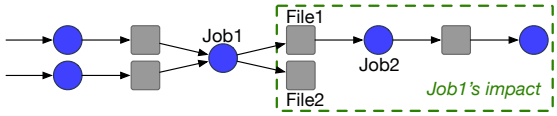

Fig. 1: Running example of query over a heterogeneous network: the "blast radius" impact for a given job in a data lineage graph (blue circles correspond to jobs; gray squares to files).

At the same time, we notice that *similar queries are often run repeatedly* over the same graph. Such queries can be identified and materialized as views to avoid significant computation cost during their evaluation. The aforementioned *structural regularity* of these graphs can be exploited to efficiently and automatically derive these materialized views. Like their relational counterparts, such *graph views* allow us to answer queries by operating on much smaller amounts of data, hiding/amortizing computational costs and ultimately delivering substantial query performance improvements of up to 50X in our experiments on real-world graphs. As we show, the benefits of using graph views are more pronounced in *heterogeneous* graphs[1] that include a large number of vertex and edge types with connectivity constraints between them.

A. Motivating example

At Microsoft, we operate one of the largest data lakes worldwide, storing several exabytes of data and processing them with hundreds of thousands of jobs, spawning billions of tasks daily [8]. Operating such a massive infrastructure requires us to handle data governance and legal compliance (e.g., GDPR), optimize our systems based on our query workloads, and support metadata management and enterprise search for the entire company, similar to the scenarios in [2], [3]. A natural way to represent this data and track datasets and computations at various levels of granularity is to build a *provenance graph* that captures data dependencies among jobs, tasks, files, file blocks, and users in the lake. As discussed above, only specific relationships among vertices are allowed, e.g., a user can submit a job, and a job can read or write files.

[1]By heterogeneous, we refer to graphs that have more than one vertex types, as opposed to homogeneous with a single vertex type.

To enable these applications over the provenance graph, we need support for a wide range of *structural* queries. Finding files that contain data from a particular user or created by a particular job is an anchored graph traversal that computes the reachability graph from a set of source vertices, whereas detecting overlapping query sub-plans across jobs to avoid unnecessary computations can be achieved by searching for jobs with the same set of input data. Other queries include label propagation (i.e., marking privileged derivative data products), data valuation (i.e., quantifying the value of a dataset in terms of its "centrality" to jobs or users accessing them), copy detection (i.e., finding files that are stored multiple times by following copy jobs that have the same input dataset), and data recommendation (i.e., finding files accessed by other users who have accessed the same set of files that a user has).

We highlight the optimization opportunities in these types of queries through a running example: the *job blast radius*. Consider the following query operating on the provenance graph: *"For every job j, quantify the cost of failing it, in terms of the sum of CPU-hours of (affected) downstream consumers, i.e., jobs that directly or indirectly depend on j's execution."* This query, visualized in Fig. 1, traverses the graph by following read/write relationships among jobs and files, and computes an aggregate along the traversals. Answering this query is necessary for cluster operators and analysts to quantify the impact of job failures—this may affect scheduling and operational decisions.

By analyzing the query patterns and the graph structure, we can optimize the job blast radius query in the following ways. First, observe that the graph has structural connectivity constraints: jobs produce and consume files, but there are no file-file or job-job edges. Second, not all vertices and edges in the graph are relevant to the query, e.g., it does not use vertices representing tasks. Hence, we can prune large amounts of data by storing as a view only vertices and edges of types that are required by the query. Third, while the query traverses job-file-job dependencies, it only uses metadata from jobs. Thus, a view storing only jobs and their (2-hop) relationships to other jobs further reduces the data we need to operate on and the number of path traversals to perform. A key contribution of our work is that these views of the graph can be used to answer queries, and if materialized, query answers can be computed much more quickly than if computed over the entire graph.

B. Contributions

Motivated by the above scenarios, we have built KASKADE, a graph query optimization framework that employs graph views and materialization techniques to efficiently evaluate queries over graphs. Our contributions are as follows:

Query optimization using graph views. We identify a class of graph views that can capture the graph use cases we discussed above. We then provide algorithms to perform view selection (i.e., choose which views to materialize given a query set) and view-based query rewriting (i.e., evaluate a query given a set of already materialized views). As far as we know, this is the first work that employs graph views in graph query optimization.

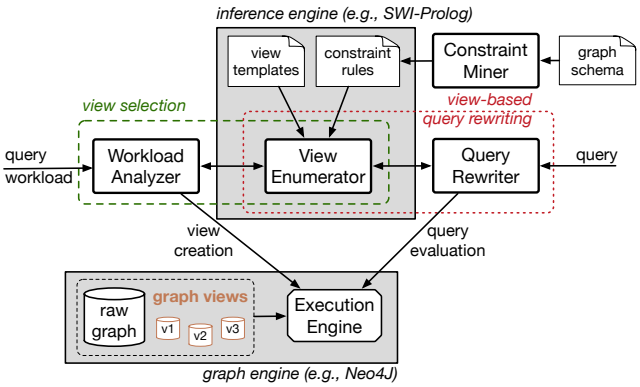

Fig. 2: Architecture of KASKADE.

Constraint-based view enumeration. Efficiently enumerating candidate views is crucial for the performance of view-based query optimization algorithms. To this end, we introduce a novel technique that mines constraints from the graph schema and queries. It then leverages view templates expressed as inference rules to generate candidate views, injecting the mined constraints at runtime to reduce the search space of views to consider. The number of views our technique enumerates is further lowered when using the query constraints.

Cost model for graph views. Given the importance of path traversals, we introduce techniques to estimate the size of views involving such operations and to compute their creation cost. Our cost model is crucial for determining which views are the most beneficial to materialize. Our experiments show that by leveraging graph schema constraints and associated degree distributions, we can estimate the size of various path views in a number of real-world graphs reasonably well.

Results over real-world graphs. We have incorporated all of the above techniques in our system, KASKADE, and have evaluated its efficiency using a variety of graph queries over both heterogeneous and homogeneous graphs. KASKADE is capable of choosing views that, when materialized, speed up query evaluation by up to 50X on heterogeneous graphs.

II. OVERVIEW

KASKADE is a graph query optimization framework that materializes graph views to enable efficient query evaluation. As noted in § I, it is designed to support complex enumeration queries over large subgraphs, often involving reporting-oriented applications that repeatedly compute filters and aggregates, and apply various analytics over graphs.

KASKADE's architecture is depicted in Figure 2. Users submit queries in a language that includes graph pattern constructs expressed in Cypher [9] and relational constructs expressed in SQL. They use the former to express path traversals, and the latter for filtering and aggregation operations. This query language, described in § III, is capable of capturing many of the applications described above.

KASKADE supports two main view-based operations: (i) *view selection*, i.e., given a set of queries, identify the most useful views to materialize for speeding up query evaluation, accounting for a space budget and various cost components; and (ii) *view-based query rewriting*, i.e., given a submitted

query, determine how it can be rewritten given the currently materialized views in the system to improve the query's execution time by leveraging the views. These operations are detailed in § VI. The *workload analyzer* drives view selection, whereas the *query rewriter* is responsible for the view-based query rewriting.

An essential component in both of these operations is the *view enumerator*, which takes as input a query and a graph schema, and produces candidate views for that query. A subset of these candidates will be selected for materialization during view selection and for rewriting a query during view-based query rewriting. As we show in § V, KASKADE follows a novel constraint-based view enumeration approach. In particular, KASKADE's *constraint miner* extracts (explicit) constraints directly present in the schema and queries, and uses constraint mining rules to derive further (implicit) constraints. KASKADE employs an inference engine (we use Prolog in our implementation) to perform the actual view enumeration, using a set of view templates it expresses as inference rules. Further, it injects the mined constraints during enumeration, leading to a significant reduction in the space of candidate views. Moreover, this approach allows us to easily include new view templates, extending the capabilities of the system, and alleviates the need for writing complicated code to perform the enumeration.

KASKADE uses an execution engine component to create the views that are output by the workload analyzer, and to evaluate the rewritten query output by the query rewriter. In this work, we use Neo4j's execution engine [10] for the storage of materialized views, and to execute graph pattern matching queries. However, our query rewriting techniques can be applied to other graph query execution engines, so long as their query language supports graph pattern matching clauses.

III. PRELIMINARIES

A. Graph Data Model

We adopt property graphs as our data model [11], in which both vertices and edges are typed and may have properties in the form of key-value pairs. This *schema* captures constraints such as domain and range of edge types. In our provenance graph example of § I-A, an edge of type "read" only connects vertices of type "job" to vertices of type "file" (and thus never connects two vertices of type "file"). As we show later, such schema constraints play an essential role in view enumeration (§ V). Most mainstream graph enginesprovide support for this data model (including the use of schema constraints).

B. Query Language

To address our query language requirements discussed in § II, KASKADE combines regular path queries with relational constructs in its query language. In particular, it leverages the graph pattern specification from Neo4j's Cypher query language [9] and combines it with relational constructs for filters and aggregates. This hybrid query language resembles that of recent industry offerings for graph-structured data analytics, such as those from AgensGraph DB [12], SQL Server [13], and Oracle's PGQL [14].

Listing 1 Job blast radius query over raw graph.

```
SELECT A.pipelineName, AVG(T_CPU) FROM (
  SELECT A, SUM(B.CPU) AS T_CPU FROM (
    MATCH (q_j1:Job)-[:WRITES_TO]->(q_f1:File)
      (q_f1:File)-[r*0..8]->(q_f2:File)
      (q_f2:File)-[:IS_READ_BY]->(q_j2:Job)
      RETURN q_j1 as A, q_j2 as B
  ) GROUP BY A, B
) GROUP BY A.pipelineName
```

As an example, consider the job blast radius use case introduced in § I (Fig. 1). Lst. 1 illustrates a query that combines OLAP and anchored path constructs to rank jobs in its blast radius based on average CPU consumption.

Specifically, the query in Lst. 1 ranks jobs up to 10 hops away in the downstream of a job (`q_j1`). In the example query, this is accomplished in Cypher syntax by using a *variable length path* construct of up to 8 hops (`-[r*0..8]->`) between two file vertices (`q_f1` and `q_f2`), where the two files are endpoints of the first and last edge on the complete path, respectively.

Although we use the Cypher syntax for our graph queries, our techniques can be coupled with any query language, as long as it can express graph pattern matching and schema constructs.

C. Graph Views

We define a *graph view* over a graph G as the graph query Q to be executed against G. This definition is similar to the one first introduced by Zhuge and Garcia-Molina [15], but extended to allow the results of Q to also contain new vertices and edges in addition to those in G. A *materialized graph view* is a physical data object containing the results of executing Q over G.

KASKADE can support a wide range of graph views through the use of *view templates*, which are essentially inference rules, as we show in § V. Among all possible views, we identify two classes, namely *connectors* and *summarizers*, which are sufficient to capture most of the use cases that we have discussed so far. Intuitively, connectors result from operations over paths (i.e., path contractions), whereas summarizers are obtained via summarization operations (filters or aggregates) that reduce the size of the original graph.

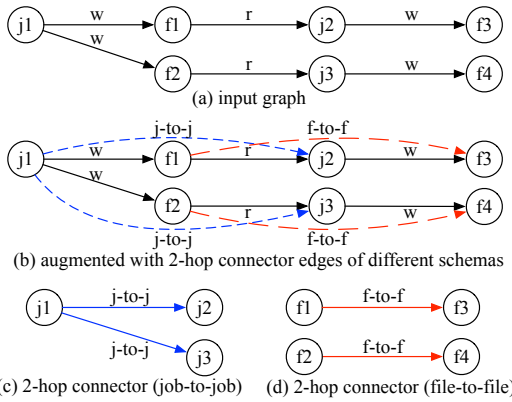

Fig. 3: Construction of different 2-hop connector graph views over a heterogeneous network (namely, a data lineage graph) with two vertex types ($N = 2$) and two edge types ($M = 2$).

As an example, Fig. 3 shows the construction of 2-hop same-vertex-type connector views over a data lineage graph (similar

TABLE I: Connectors in KASKADE

Type	Description
Same-vertex-type connector	Target vertices are all pairs of vertices with a specific vertex type.
k-hop connector	Target vertices are all vertex pairs that are connected through k-length paths.
Same-edge-type connector	Target vertices are all pairs of vertices that are connected with a path consisting of edges with a specific edge type
Source-to-sink connector	Target vertices are (source, sink) pairs, where sources are the vertices with no incoming edges and sinks are vertices with no outgoing edges.

to the graph of Fig. 1). In Fig. 3(a), the input graph contains two types of vertices, namely jobs (vertex labels with a j prefix) and files (vertex labels with an f prefix), as well as two types of edges, namely w (a job *writes to* a file) and r (a file *is read by* a job). Fig. 3(b) shows two different types of connector edges: the first contracts 2-hop paths between pairs of job vertices (depicted as blue dashed edges); the second contracts 2-hop paths between pairs of file vertices (depicted as red dashed edges). Finally, Fig. 3(c) shows the two resulting connector graph views: one containing only the *job-to-job* connector edges (left), and one with only the *file-to-file* connector edges (right).

Next, we introduce a formal definition and examples of connectors and summarizers. We consider their definition as a means to an end rather than a fundamental contribution of this work. Thus, here we only provide sufficient information over graph views for the reader to follow our techniques in Sections V and VI. As path operations tend to be the most expensive operations in graphs, our description will pivot mostly around connectors. Materializing such expensive operations can lead to significant performance improvements at query evaluation time. Moreover, the semantics of connectors is less straightforward than that of summarizers, which resemble their relational counterparts (filters and aggregates). Note, however, that the techniques described here are generic and apply to any graph views expressible as view templates.

IV. View Definitions and Examples

Next we provide formal definitions of the two main classes of views supported in KASKADE, namely connectors and summarizers, which were briefly described in § III-C. Moreover, we give various examples of the views from each category that are currently present in KASKADE's view template library.

While the examples below are general enough to capture many different types of graph structures, they are by no means an exhaustive list of graph view templates that are possible in KASKADE; as we mention in § V, KASKADE's library of view templates and constraint mining rules is readily extensible.

A. Connectors

A connector of a graph $G = (V,E)$ is a graph G' such that every edge $e' = (u,v) \in E(G')$ is obtained via contraction of a single directed path between two target vertices $u,v \in V(G)$. The vertex set $V(G')$ of the connector view is the union of all target vertices with $V(G') \subseteq V(G)$. Based on this definition, a number of specialized connector views can be defined, each of which differs in the target vertices that it considers. Table I lists examples currently supported in KASKADE.

Additionally, it is easy to compose the definition of k-hop connectors with the other connector definitions, leading to more connector types. As an example, the k-hop same-vertex-type connector is a same-vertex-type connector with the additional requirement that the target vertices should be connected through k-hop paths. Finally, connectors are useful in contexts where a query's graph pattern contains relatively long paths that can be contracted without loss of generality, or when only the endpoints of the graph pattern are projected in subsequent clauses in the query.

B. Summarizers

A summarizer of a graph $G = (V,E)$ is a graph G' such that $V(G') \subseteq V(G)$, $E(G') \subseteq E(G)$, and at least one of the following conditions are true: (i) $|V(G')| < |V(G)|$, or (ii) $|E(G')| < |E(G)|$. The summarizer view operations that KASKADE currently provides are filters that specify the type of vertices or edges that we want to preserve (inclusion filters) or remove (exclusion filters) from the original graph.[2] Also, it provides aggregator summarizers that either group a set of vertices into a supervertex, a group of edges into a superedge, or a subgraph into a supervertex. Finally, aggregator summarizers require an aggregate function for each type of property present in the domain of the aggregator operation. We provide examples of summarizer graph views currently supported in an extended preprint version at [16]. KASKADE's library of template views currently does not support aggregation of vertices of different types. The library is readily extensible, however, and aggregations for vertices or edges with different types are expressible using a higher-order aggregate function to resolve conflicts between properties with the same name, or to specify the resulting aggregate type.

Lastly, summarizers can be useful when subsets of data (e.g., entire classes of vertices and edges) can be removed from the graph without incurring any side-effects (in the case of filters), or when queries refer to logical entities that correspond to groupings of one or more entity at a finer granularity (in the case of aggregators).

V. Constraint-Based View Enumeration

Having introduced a preliminary set of graph view types in § III-C (see also § IV for more examples), we now describe KASKADE's view enumeration process, which generates candidate views for the graph expressions of a query. As discussed in § II, view enumeration is central to KASKADE (see Fig. 2), as it is used in both view selection and query rewriting, which are described in § VI. Given that the space of view candidates can be large, and that view enumeration is on the critical path of view selection and query rewriting, its efficiency is crucial.

In KASKADE we use a novel approach, which we call *constraint-based view enumeration*. This approach mines

[2]Summarizer views can also include predicates on vertex/edge properties in their definitions. Using such predicates would further reduce the size of these views, but given they are straightforward, here we focus more on predicates for vertex/edge types.

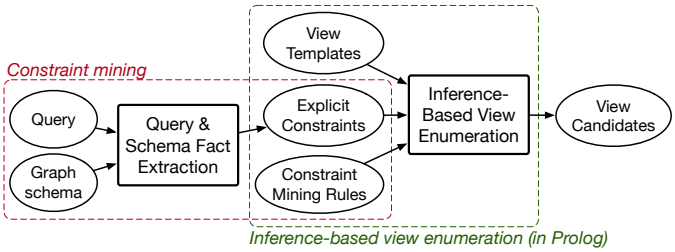

Fig. 4: Constraint-based view enumeration in KASKADE.

constraints from the query and the graph schema and injects them at view enumeration time to drastically reduce the search space of graph view candidates. Fig. 4 depicts an overview of our approach. Our view enumeration takes as input a query, a graph schema and a set of declarative view templates, and searches for instantiations of the view templates that apply to the query. KASKADE expresses view templates as inference rules, and employs an inference engine (namely, Prolog)[3] to perform view enumeration through rule evaluation. Importantly, KASKADE generates both *explicit* constraints, extracted directly from the query and schema, and *implicit* ones, generated via constraint mining rules. This constraint mining process identifies structural properties from the schema and query that allow it to significantly prune the search space for view enumeration, discarding infeasible candidates (e.g., job-to-job edges or 3-hop connectors in our provenance example).

Apart from effectively pruning the search space of candidate views, KASKADE's view enumeration provides the added benefit of not requiring the implementation of complex transformations and search algorithms—the search is performed by the inference engine automatically via our view templates and efficiently via the constraints that we mine. Moreover, view templates are readily extensible to modify the supported set of views. In our current implementation, we employ SWI-PL's inference engine [17], which gives us the flexibility to seamlessly add new view templates and constraint mining rules via Prolog rules. On the contrary, existing techniques for view enumeration in the relational setting typically decompose a query through a set of application-specific transformation rules [18] or by using the query optimizer [19], and then implement search strategies to navigate through the candidate views. Compared to our constraint-based view enumeration, these approaches require higher implementation effort and are inflexible when it comes to adding or modifying complex transformation rules.

We detail the two main parts of our constraint-based view enumeration, namely the constraint mining and the inference-based view enumeration, in Sections V-A and V-B, respectively.

A. Mining Structural Graph Constraints

KASKADE exploits information from the graph's schema and the given query in the form of constraints to prune the set of candidate views to be considered during view enumeration. It mines two types of constraints:

- **Explicit constraints** (§ V-A1) are first-order logic statements (facts) extracted from the schema and query (e.g., that files do not write files, only jobs);
- **Implicit constraints** (§ V-A2) are not present in the schema or query, but are derived by combining the explicit constraint facts with constraint mining rules. These constraints are essential in KASKADE, as they capture structural properties that otherwise cannot be inferred by simply looking at the input query or schema properties.

1) Extracting explicit constraints

The first step in our constraint-based view enumeration is to extract explicit constraints (facts) from the query and schema.

Transforming the query to facts. Our view enumeration algorithm goes over the query's `MATCH` clause, i.e., its graph pattern matching clause, and for each vertex and edge in the graph pattern, KASKADE's constraint miner emits a set of Prolog facts. In our example of the job "blast radius" query (Lst. 1), KASKADE extracts the following facts from the query:

```
queryVertex(q_f1). queryVertex(q_f2).
queryVertex(q_j1). queryVertex(q_j2).
queryVertexType(q_f1, 'File').
queryVertexType(q_f2, 'File').
queryVertexType(q_j1, 'Job').
queryVertexType(q_j2, 'Job').
queryEdge(q_j1, q_f1). queryEdge(q_f2, q_j2).
queryEdgeType(q_j1, q_f1, 'WRITES_TO').
queryEdgeType(q_f2, q_j2, 'IS_READ_BY').
queryVariableLengthPath(q_f1, q_f2, 0, 8).
```

The above set of facts contains all named vertices and edges in the query's graph pattern, along with their types, and any variable-length regular path expression (`queryVariableLengthPath( X,Y,L,U)` corresponds to a path between nodes `X` and `Y` of length between `L` and `U`). In Lst. 1, a variable-length regular path of up to 8 hops is specified between query vertices `q_f1` and `q_f2`.

Transforming the schema to facts. Similar to the extraction of query facts, our view enumeration algorithm goes over the provided graph schema and emits the corresponding Prolog rules. For our running example of the data lineage graph, there are two types of vertices (files and jobs) and two types of edges representing the producer-consumer data lineage relationship between them. Hence, the set of facts extracted about this schema is:

```
schemaVertex('Job'). schemaVertex('File').
schemaEdge('Job', 'File', 'WRITES_TO').
schemaEdge('File', 'Job', 'IS_READ_BY').
```

2) Mining implicit constraints

Although the explicit query and schema constraints that we introduced so far restrict the view enumeration to consider only views with meaningful vertices and edges (i.e., that appear in the query and schema), they still allow many infeasible views to be considered. For example, if we were to enumerate k-hop connectors between two files to match the variable-length path in the query of Lst. 1, all values of $k \geq 2$ would have to be considered. However, given the example schema, we know that only even values of k are feasible views. Similarly, since the query specifies an upper limit $u = 8$ in the number hops in

[3]While a subset of KASKADE's constraint mining rules are expressible in Datalog, Prolog provides native support (i.e., without extensions) for aggregation and negation, and we use both in summarizer view templates. KASKADE also relies on higher-order predicates (e.g., `setof`, `findall`) in constraint mining rules, which Datalog does not support.

Listing 2 Example of constraint mining rule for the graph schema.

```
% Determine whether acyclic directed k-length paths
% between two nodes X and Y are feasible over the input
% graph schema. schemaEdge are explicit constraints
% extracted from the schema.
schemaKHopPath(X,Y,K) :- schemaKHopPath(X,Y,K,[]).
schemaKHopPath(X,Y,1,_) :- schemaEdge(X,Y,_).
schemaKHopPath(X,Y,K,Trail) :-
  schemaEdge(X,Z,_), not(member(Z,Trail)),
  schemaKHopPath(Z,Y,K1,[X|Trail]), K is K1 + 1.
```

the variable-length path, we should not be enumerating k-hop connectors with $k \geq 8$.

To derive such implicit schema and query constraints that will allow us to significantly prune the search space of views to consider, KASKADE uses a library of constraint mining rules[4] for the schema and query. Lst. 2 shows an example of such a constraint mining rule for the schema. Rule `schemaKHopPath`, expressed in Prolog in the listing, infers all valid k-hop paths given the input schema. Note that the rule takes advantage of the explicit schema constraint `schemaEdge` to derive this more complex constraint from the schema, and it considers two instances of `schemaKHopPath` different if and only if all values in the resulting unification tuple are different. For example, a `schemaKHopPath(X='Job',Y='Job',K=2)` unification, i.e., a *job-to-job* 2-hop connector is different from a `schemaKHopPath(X='File',Y='File',K=2)`, i.e., a *file-to-file* 2-hop connector. For completeness, we also provide a procedural version of this constraint mining rule in an extended preprint version at [16]. When contrasted with the Prolog rule, the procedural version is not only more complex, but it also explores a larger search space. This is because the procedural function cannot be injected at view enumeration time as an additional rule together with other inference rules that further bound the search space (see § V-B).

Similar constraint mining rules can be defined for the query. The extended preprint version at [16] shows examples of such rules, e.g., to bound the length of considered k-hop connectors (in case such limits are present in the query's graph pattern), or to ensure that a node is the source or sink in a connector.

These constraints play a crucial role in limiting the search space for valid views. As we show in § V-B, by injecting such rules at view enumeration time, KASKADE can automatically derive implicit knowledge from the schema and query to significantly reduce the search space of considered views. Interestingly, KASKADE can derive this knowledge only on demand, in that the constraint mining rules get fired only when required and do not blindly derive all possible facts from the query and schema.

Importantly, the combination of both schema and query constraints is what makes it possible for our view enumeration approach to reducing the search space of possible rewritings. As an example, consider the process of enumerating valid k-hop connector views. Without any query constraints, the number of such views equals the number of k-length paths over the *schema graph*, which has M *schema edges*. While there exists no closed formula for this combination, when the schema graph has one or more cycles (e.g., a schema edge that connects a schema vertex to itself), at least M^k k-hop schema paths can be enumerated. This space is what the `schemaKHopPath` schema constraint mining rule would search over, were it not used in conjunction with query constraint mining rules on view template definitions. KASKADE's *constraint-based view inference* algorithm enumerates a significantly smaller number of views for input queries, as the additional constraints enable its inference engine to efficiently search by early-stopping on branches that do not yield feasible rewritings.

[4]The collection of constraint mining rules KASKADE provides is readily extensible, and users can supply additional constraint mining rules if desired.

Listing 3 Example view template definitions for connectors.

```
% k-hop connector between nodes X and Y.
kHopConnector(X, Y, XTYPE, YTYPE, K) :-
  % query constraints
  queryVertexType(X, XTYPE), queryVertexType(Y, YTYPE),
  queryKHopPath(X, Y, K),
  % schema constraints
  schemaKHopPath(XTYPE, TYPE, K).

% k-hop connector where all vertices are of the same type.
kHopConnectorSameVertexType(X, Y, VTYPE, K) :-
  kHopConnector(X, Y, VTYPE, VTYPE, K).

% Variable-length connector where all vertices are of
% the same type.
connectorSameVertexType(X, Y, VTYPE) :-
  % query constraints
  queryVertexType(X, VTYPE), queryVertexType(Y, VTYPE),
  queryPath(X, Y),
  % schema constraints
  schemaPath(X, Y).

% Source-to-sink variable-length connector.
sourceToSinkConnector(X, Y) :-
  % query constraints
  queryVertexSource(X), queryVertexSink(Y), queryPath(X, Y),
  % schema constraints
  schemaPath(X, Y).
```

B. Inference-based View Enumeration

As shown in Fig. 4, KASKADE's view enumeration takes as input (i) a query, (ii) a graph schema, and (iii) a set of view templates. As described in § V-A, the query and schema are used to mine explicit and implicit constraints. Both the view templates and the mined constraints are rules that are passed to an inference engine to generate candidate views via rule evaluation. Hence, we call this process *inference-based view enumeration*. The view templates drive the search for candidate views, whereas the constraints restrict the search space of considered views. This process outputs a collection of instantiated candidate view templates, which are later used either by the workload analyzer module (see Fig. 2) when determining which views to materialize during view selection (§ VI-B), or by the query rewriter to rewrite the query using the materialized view (§ VI-C) that will lead to its most efficient evaluation.

Lst. 3 shows examples of template definitions for connector views. Each view template is defined as a Prolog rule and corresponds to a type of connector view. For instance, `kHopConnector(X,Y,XTYPE,YTYPE,K)` corresponds to a connector of length `K` between nodes `X` and `Y`, with types `XTYPE` and `YTYPE`, respectively. An example instantiation of this template is `kHopConnector(X,Y,'Job','Job',2)`, which corresponds to a con-

crete *job-to-job* 2-hop connector view. This view can be translated to an equivalent Cypher query, which will be used either to materialize the view or to rewrite the query using this view, as we explain in § VI. Other templates in the listing are used to produce connectors between nodes of the same type or source-to-sink connectors. Additional view templates can be defined in a similar fashion, such as the ones for summarizer views that we provide in an extended preprint version at [16].

Note that the body of the view template is defined using the query and schema constraints, either the explicit ones (e.g., `queryVertexType`) coming directly from the query or schema, or via the constraint mining rules (e.g., `queryKHopPath`, `schemaKHopPath`), as discussed in § V-A. For example, `kHopConnector`'s body includes two explicit constraints to check the type of the two nodes participating in the connector, a query constraint mining rule that will check whether there is a valid k-hop path between the two nodes in the query, and a schema constraint mining rule that will do the same check on the schema.

Having gathered all query and schema facts, KASKADE's view enumeration algorithm performs the actual view candidate generation. In particular, it calls the inference engine for every view template. As an example, assuming an upper bound of $k = 10$ — in Lst. 1 we have a variable-length path of at most 8 hops between 2 *File* vertices, and each of these two vertices is an endpoint for another edge — the following are valid instantiations of the `kHopConnector(X,Y,XTYPE,YTYPE,K)` view template for query vertices `q_j1` and `q_j2` (the only vertices projected out of the `MATCH` clause):

```
(X='q_j1', Y='q_j2', XTYPE='Job', XTYPE='Job', K=2)
(X='q_j1', Y='q_j2', XTYPE='Job', XTYPE='Job', K=4)
(X='q_j1', Y='q_j2', XTYPE='Job', XTYPE='Job', K=6)
(X='q_j1', Y='q_j2', XTYPE='Job', XTYPE='Job', K=8)
(X='q_j1', Y='q_j2', XTYPE='Job', XTYPE='Job', K=10)
```

Or, in other words, the unification `(X='q_j1', Y='q_j2', XTYPE='Job', XTYPE='Job', K=2)` for the view template `kHopConnector(X,Y,XTYPE,YTYPE,K)` implies that a view where each edge contracts a path of length $k = 2$ between two nodes of type *Job* is feasible for the query in Lst. 1.

Similarly, KASKADE generates candidates for the remaining templates of Listings 3 and the additional example templates at [16]. For each candidate it generates, KASKADE's inference engine also outputs a rewriting of the given query that uses this candidate view, which is crucial for the view-based query rewriting, as we show in § VI-C. Finally, each candidate view incurs different costs (see § VI-A), and not every view is necessarily materialized or chosen for a rewrite.

VI. VIEW OPERATIONS

In this section, we present the main two operations that KASKADE supports: selecting views for materialization given a set of queries (§ VI-B) and view-based rewriting of a query given a set of pre-materialized views (§ VI-C). To do this, KASKADE uses a cost model to estimate the size and cost of views, which we describe next (§ VI-A).

A. View Size Estimation and Cost Model

While some techniques for relational cost-based query optimization may at times target filters and aggregates, most efforts in this area have primarily focused on join cardinality estimation [20]. This is in part because joins tend to dominate query costs, but also because estimating join cardinalities is a harder problem than, e.g., estimating the cardinality of filters. Furthermore, KASKADE can leverage existing techniques for cardinality estimation of filters and aggregates in relational query optimization for summarizers.

Therefore, here we detail our cost model contribution as it relates to connector views, which can be seen as the graph counterpart of relational joins. We first describe how we estimate the size of connector views, which we then use to define our cost model.

Graph data properties During initial data loading and subsequent data updates, KASKADE maintains the following graph properties: (i) *vertex cardinality* for each vertex type of the raw graph; and (ii) *coarse-grained out-degree distribution summary statistics*, i.e., the 50th, 90th, and 95th out-degree for each vertex type of the raw graph. KASKADE uses these properties to estimate the size of a view.

View size estimation. Estimating the size of a view is essential for the KASKADE's view operations. First, it allows us to determine if a view will fit in a space budget during view selection. Moreover, our cost components (see below) use view size estimates to assess the benefit of a view both in view selection and view-based query rewriting.

We estimate the size of a view as the number of edges that it has when materialized since the number of edges usually dominates the number of vertices in real-world graphs. Next, we observe that the number of edges in a k-hop connector over a graph G equals the number of k-length simple paths in G. Therefore, for a directed graph G that is homogeneous (i.e., has only one type of vertex, and one type of edge), we define the following estimator for its number of k-length paths:

$$\hat{E}(G,k,\alpha) = n \cdot deg_{\alpha}^{k} \tag{1}$$

where $n = |V|$ is the number of vertices in G, and deg_{α} is the α-th percentile out-degree of vertices in G ($0 < \alpha \leq 100$). For a directed graph G that is heterogeneous (i.e., has more than one type of vertex and/or more than one type of edge), an estimator for its number of k-length paths is as follows:

$$\hat{E}(G,k,\alpha) = \sum_{t \in T_G} n_t \cdot (deg_{\alpha}(n_i))^k \tag{2}$$

where T_G is the set of types of vertices in G that are edge sources (i.e., are the domain of at least one type of edge), n_t is the number of vertices of type $t \in T_G$, and $deg_{\alpha}(n_t)$ is the maximum out-degree of vertices of type $t \in T_G$.

Observe that if $\alpha = 100$, then deg_{α} is the maximum out-degree and the estimators above are upper bounds on the number of k-length paths in the graph. This is because there are n possible starting vertices, each vertex in the path has at most deg_{100} (or $deg_{100}(n_i)$) neighbors to choose from for its successor vertex in the path, and such a choice has to be made k times in a k-length path. In practice, we found that

$\alpha = 100$ gives a very loose upper bound, whereas $50 \leq \alpha \leq 95$ gives a much more accurate estimate depending on the degree distribution of the graph. We present experiments on the accuracy of our estimator in § VII.

View creation cost. The creation cost of a graph view refers to any computational and I/O costs incurred when computing and materializing the views' results. Since the primitives required for computing and materializing the results of the graph views that we are interested in are relatively simple, the I/O cost dominates computational costs, and thus the latter is omitted from our cost model. Hence, the view creation cost is directly proportional to $\hat{E}(G,k,\alpha)$.

Query evaluation cost. The cost of evaluating a query q, denoted $EvalCost(q)$, is required both in the view selection and the query rewriting process. KASKADE relies on an existing cost model for graph database queries as a proxy for the cost to compute a given query using the raw graph. In particular, it leverages Neo4j's [10] cost-based optimizer, which establishes a reasonable ordering between all vertex scans without indexes, scans from indexes, and range scans. For future work, we plan to incorporate our findings from graph view size estimation to further improve the query evaluation cost model.

B. View Selection

Given a query workload, KASKADE's view selection process determines the most effective views to materialize for answering the workload under the space budget that KASKADE allocates for materializing views. This operation is performed by the workload analyzer component, in conjunction with the view enumerator (see § V). The goal of KASKADE's view selection algorithm is to select the views that lead to the most significant performance gains relative to their cost, while respecting the space budget.

To this end, we formulate the view selection algorithm as a 0-1 knapsack problem, where the size of the knapsack is the space budget dedicated to view materialization.[5] The items that we want to fit in the knapsack are the candidate views generated by the view enumerator (§ V). The weight of each item is the view's estimated size (see § VI-A), while the value of each item is the performance improvement achieved by using that view divided by the view's creation cost (to penalize views that are expensive to materialize). We define the performance improvement of a view v for a query q in the query workload Q as q's evaluation cost divided by the cost of evaluating the rewritten version of q that uses v. The performance improvement of v for Q is the sum of v's improvement for each query in Q (which is zero for the queries for which v is not applicable). Note that we can extend the above formulation by adding weights to the value of each query to reflect its relative importance (e.g., based on the query's frequency to prioritize frequent queries, or on the query's estimated execution time to prioritize expensive queries).

The views that the above process selects for materialization are instantiations of Prolog view templates output by the view enumeration (see § V). KASKADE's workload analyzer translates those views to Cypher and executes them against the graph to perform the actual materialization. As a byproduct of this process, each combination of query q and materialized view v is accompanied by a rewriting of q using v. In other words, a pair of candidate view and query rewriting are both inferred *and* costed together during inference-based view enumeration. This does not imply, however, that all possible candidate views and rewriting pairs are enumerated; rather, only the rewritings that are feasible given the explicit and implicit mined constraints. Further, we also note that iteratively rewriting the query as part of view enumeration is a commonly accepted approach, with significant advantages over query rewriting done only after enumeration [21], [22]. This is crucial in the view-based query rewriting, described next. The rewriter component converts the rewriting in Prolog to Cypher, so that KASKADE can run it on its graph execution engine.

C. View-Based Query Rewriting

Given a query and a set of materialized views, view-based rewriting is the process of finding the rewriting of the query that leads to the highest reduction of its evaluation cost by using (a subset of) the views. In KASKADE, the query rewriter module (see Fig. 2) performs this operation.

When a query q arrives in the system, the query rewriter invokes the view enumerator, which generates the possible view candidates for q, pruning those that it has not materialized. Among the views that are output by the enumerator, the query rewriter selects the one that, when used to rewrite q, leads to the smallest evaluation cost for q. As discussed in § V, the view enumerator outputs the rewriting of each query based on the candidate views for that query. If this information is saved from the view selection step (which is true in our implementation), we can leverage it to choose the most efficient view-based rewriting of the query without having to invoke the view enumeration again for that query. As we mentioned in § V, KASKADE currently supports rewritings that rely on a single view. Combining multiple views in a single rewriting is left as future work.

Lst. 4 shows the rewritten version of our example query of Lst. 1 that uses a 2-hop connector ("job-to-job") graph view.

Listing 4 Job blast radius query rewritten over a 2-hop connector (*job-to-job*) graph view.

```
SELECT A.pipelineName, AVG(T_CPU) FROM (
  SELECT A, SUM(B.CPU) AS T_CPU FROM (
    MATCH (q_j1:Job)-[:2_HOP-JOB_TO_JOB*1..4]->(q_j2:Job)
    RETURN q_j1 as A, q_j2 as B
  ) GROUP BY A, B
) GROUP BY A.pipelineName
```

VII. EXPERIMENTAL EVALUATION

In this section, we experimentally confirm that by leveraging graph views (e.g., summarizers and connectors) we can: (i) accurately estimate graph view sizes—§ VII-D; (ii) effectively reduce the graph size our queries operate on—§ VII-E; and (iii) improve query performance—§ VII-F. We start by providing details on KASKADE's current implementation, and follow by introducing our datasets and query workload.

[5] The space budget is typically a percentage of the machine's main memory size, given that we are using a main memory execution engine. Our formulation can be easily extended to support multiple levels of memory hierarchy.

TABLE II: Networks used for evaluation: `prov` and `dblp` are heterogeneous, while `roadnet-usa` and `soc-livejournal` are homogeneous and have one edge type.

Short Name	Type	$\|V\|$	$\|E\|$
prov (raw)	Data lineage	3.2B	16.4B
prov (summarized)	Data lineage	7M	34M
dblp-net	Publications [25]	5.1M	24.7M
soc-livejournal	Social network [26]	4.8M	68.9M
roadnet-usa	Road network [6]	23.9M	28.8M

A. Implementation

We have implemented KASKADE's components (see Fig. 2) as follows. The *view enumerator* component (§ V) uses SWI-Prolog [17] as its inference engine. We wrote all constraint mining rules, query constraint mining rules, and view templates using SWI-Prolog syntax [23]. We wrote the knapsack problem formulation (§ VI-B) in Python 2.7, using the branch-and-bound knapsack solver from Google OR tools combinatory optimization library [24]. As shown in Fig. 2, KASKADE uses Neo4J (version 3.2.2) for storage of raw graphs, materialized graph views, and query execution of graph pattern matching queries. All other components were written in Java.

For the experimental results below, KASKADE extracted schema constraints only once for each workload, as these do not change throughout the workload. Furthermore, KASKADE extracted query constraints as part of view inference only the first time we entered the query into the system. This process introduces a few milliseconds to the total query runtime and is amortized for multiple runs of the same query.

B. Datasets

In our evaluation of different aspects of KASKADE, we use a combination of publicly-available heterogeneous and homogeneous networks and a large provenance graph from Microsoft (heterogeneous network). Table II lists the datasets and their raw sizes. We also provide their degree distributions in an extended preprint version of this work at [16].

For our size reduction evaluation in Section VII-E, we focus on gains provided by different graph views over the two heterogeneous networks: `dblp-net` and a data lineage graph from Microsoft. The `dblp-net` graph, which is publicly available at [25], contains 5.1M vertices (authors, articles, and venues) with 24.7M with 2.2G on-disk footprint. For the second heterogeneous network, we captured a provenance graph modeling one of Microsoft's large production clusters for a week. This raw graph contains 3.2B vertices modeling jobs, files, machines, and tasks, and 16.4B edges representing relationships among these entities, such as job-read-file, or task-to-task data transfers. The on-disk footprint of this data is in the order of 10+ TBs.

After showing size reduction achieved by summarizers and connectors in Section VII-E, for our query runtime experiments (§ VII-F), we consider already summarized versions of our heterogeneous networks, i.e., of `dblp-net` and provenance graph. The summarized graph over `dblp-net` contains only authors and publications ("article", "publication", and "inproc" vertex types), totaling 3.2M vertices and 19.3M edges, which requires 1.3G on disk. The summarized graph over the raw provenance graph contains only jobs and files and their relationships, which make up its 7M vertices and 34M edges. The on-disk footprint of this data is 4.8 GBs. This allows us to compare runtimes of our queries on the Neo4j 3.2.2 graph engine, running on a 128 GB of RAM and 28 Intel Xeon 2.40GHz cores, 4 x 1TB SSD Ubuntu box. We chose to use Neo4j for storage of materialized views and query execution because it is the most widely used graph database engine as of writing, but our techniques are graph query engine-agnostic.

C. Queries

We use the following 8 queries in our evaluation of query runtimes. Queries Q1 through Q3 are motivated by telemetry use cases at Microsoft, and are defined as follows. The first query retrieves the job blast radius, up to 8 hops away, of all job vertices in the graph, together with their average CPU consumption property. Query Q2 retrieves the ancestors of a job (i.e., all vertices in its *backward data lineage*) up to 4 hops away, for all job vertices in the graph. Conversely, Q3 does the equivalent operation for *forward data lineage* for all vertices in the graph, also capped at 4 hops from the source vertex. Both Q2 and Q3 are also adapted for the other 3 graph datasets: on `dblp`, the source vertex type is "author" instead of "job", and on homogeneous networks `roadnet-usa` and `soc-livejournal` all vertices are included.

Next, queries Q4 through Q7 capture graph operation primitives which are commonly required for tasks in dependency driven analytics [3]. The first, query Q4 ("path lengths"), computes a weighted distance from a source vertex to all other vertices in its forward k-hop neighborhood, limited to 4 hops. For each vertex in the 4-hop neighborhood, it performs an aggregation operation (max) over a data property (edge timestamp) of all edges in the 4-hop path. Queries Q5 and Q6 both measure the overall size of the graph (edge count and vertex count, respectively).

Finally, Q7 ("community detection") and Q8 ("largest community") are representative of common graph analytics tasks. The former runs a 25 passes iterative version of community detection algorithm via label-propagation, updating a "community" property on all vertices and edges in the graph, while Q8 uses the community label produced by Q7 to retrieve the largest community in terms of graph size as measured by number of "job" vertices in each community. The label propagation algorithm used by Q7 is part of the APOC collection of graph algorithm UDFs for Neo4j [27].

For query runtimes experiments in § VII-F, we use the equivalent rewriting of each of these queries over a 2-hop connector. Specifically, queries Q1 through Q4 go over half of the original number of hops, and queries Q7 and Q8 run around half as many iterations of label propagation. These rewritings are equivalent and produce the same results as queries Q1 through Q4 over the original graph, and similar groupings of "job" nodes in the resulting communities. Queries Q5 and Q6 need not be modified, as they only count the number of elements in the dataset (edges and vertices, respectively).

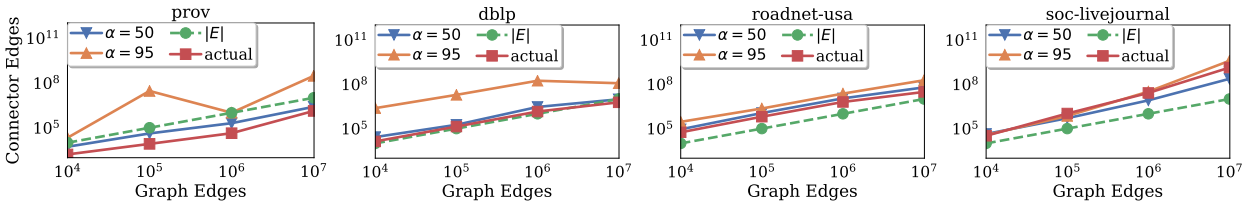

Fig. 5: Estimated, actual, and original graph sizes for 2-hop connector views over different datasets. Here we show estimates for two upper bound variations derived from summary statistics over the graph's degree distribution detailed in § VI-A. We also plot the original graph size (x-axis, and dashed $|E|$ series). Plots are in log-log scale.

D. View Size Estimation

In this section, we evaluate the accuracy of KASKADE's view size estimators (§ VI-A). Fig. 5 shows our results for different heuristics estimators on the size of a 2-hop connector view materialized over the first n edges of each public graph dataset. We focus on size estimates for 2-hop connectors since, similar to cardinality estimation for joins, the larger the k, the less accurate our estimator. We do not report results for view size estimates for summarizers, as for these KASKADE can leverage traditional cardinality estimation based on predicate selectivity for filters, as well as multi-dimensional histograms for predicate selectivity of group-by operators [20].

We found that k-hop connectors in homogeneous networks are usually larger than the original graph in real-world networks. Further, we find that there is no direct relationship between the size of a graph schema, and the expected size of a k-hop view. This is because k-length paths can exist between any two vertices in a homogeneous graph (smaller schema), as opposed to only between specific types of vertices in heterogeneous networks (larger schema), such as Microsoft's provenance graph. Note that the $\alpha = 50$ line does a good job of approximating the size of the graph as the number of edges grows. Also, for networks with a degree distribution close to a power-law, such as `soc-livejournal`, the estimator that relies on 95th percentile out-degree ($\alpha = 95$) provides an upper bound, while the one that uses the median out-degree ($\alpha = 50$) of the network provides a lower bound. On other networks that lack a power-law degree distribution, such as `road-net-usa`, the median out-degree estimator better approximates an upper bound on the size of the k-hop connector.

In practice, KASKADE relies on the estimator with $\alpha = 95$ as it provides an upper bound for most real-world graphs that we have observed. Note that the estimator with $\alpha = 95$ for `prov` decreases when the original graph increases in size from $100K$ to $1M$ edges, increasing again at $10M$ edges. This is due to a decrease in the percentage of "job" vertices with a large out-degree, shifting the 95th percentile to a lower value. This percentile remains the same at $10M$ edges, while the 95th out-degree for "file" vertices increases at both $1M$ and $10M$ edges.

E. Size Reduction

This experiment shows how by applying summarizers and connectors over heterogeneous graphs we can reduce the effective graph size for one or more queries. Figure 6 shows that for co-authorship queries over the `dblp`, and query Q1 over the provenance graph, the schema-level summarizer yields up to three orders of magnitude reduction. The connector yields another two orders of magnitude data reduction by summarizing the job-file-job paths in the provenance graph, and one order of magnitude by summarizing the author-publication-author paths in the `dblp` graph.

Besides the expected performance advantage since queries operate on less data, such a drastic data reduction allows us to benefit from single-machine in-memory technologies (such as Neo4j) instead of slower distributed on-disk alternatives for query evaluation, in the case of the provenance graph. While this is practically very important, we do not claim this as part of our performance advantage, and all experiments are shown as relative speed-ups against a baseline on this reduced provenance graph on the same underlying graph engine.

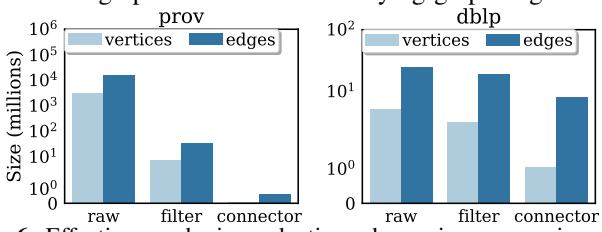

Fig. 6: Effective graph size reduction when using summarizer and 2-hop connector views over `prov` and `dblp` heterogeneous networks (y-axis is in log scale).

F. Query Runtimes

This experiment measures the difference in total query runtime for queries when executed from scratch over the filtered graph versus over an equivalent connector view on the filtered graph. Figure 7 shows our results, with runtimes averaged over 10 runs, plotted in log scale on the y-axis. Because the amount of data required to answer the rewritten query is up to orders of magnitude smaller (§ VII-E) in the heterogeneous networks, virtually every query over the `prov` and `dblp` graphs benefit from executing over the connector view. Specifically, Q2 and Q3 have the least performance gains (less than 2 times faster), while Q4 and Q8 obtain the largest runtime speedups (13 and 50 times faster, respectively). This is expected: Q2 ("ancestors") and Q3 ("descendants") explore a k-hop ego-centric neighborhood in both the filtered graph and in the connector view that is of the same order of magnitude. Q4 ("path lengths") and Q8 ("community detection"), on the other hand, are queries that directly benefit from the significant size reduction of the input graph. In particular, the maximum number of paths in a graph can be exponential on the number of vertices and edges, which affects both the count of path lengths that Q4 performs, as well as the label-propagation updates that Q8 requires. Lastly, KASKADE creates views through graph

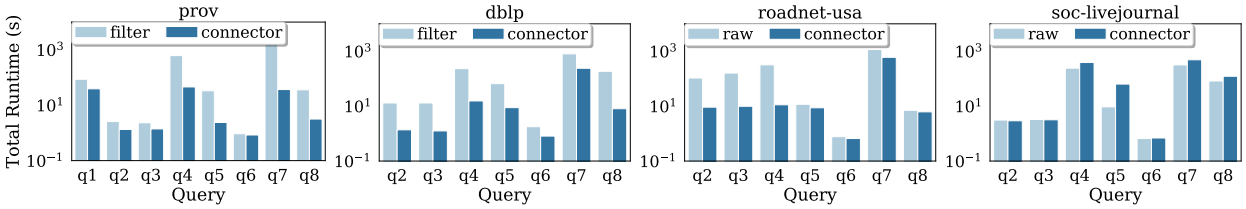

Fig. 7: Total query execution runtimes over the graph after applying a summarizer view ("filter"), and rewritten over a 2-hop connector view. Connectors are *job-to-job* (`prov`), *author-to-author* (`dblp`), and *vertex-to-vertex* for homogeneous networks `roadnet-usa` and `soc-livejournal` (*y*-axis is in log scale).

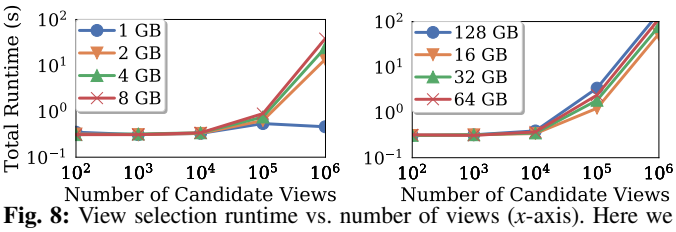

Fig. 8: View selection runtime vs. number of views (*x*-axis). Here we show runtimes for eight different space budgets using the knapsack formulation detailed in VI-B. Plots are in log-log scale.

transformations that are engine-agnostic, hence these gains should be available in other systems as well.

For the homogeneous networks, we look at the total query runtimes over the raw graph and over a *vertex-to-vertex* materialized 2-hop connector, which may be up to orders of magnitude larger than the raw graph (§ VII-D), as these networks have only one type of edge. Despite these differences in total sizes, a non-intuitive result is that the total query runtime is still linearly correlated with the order of magnitude increase in size for the network with a power-law degree distribution (`soc-livejournal`), while it decreases for path-related queries in the case of `roadnet-usa`. This is due to a combination of factors, including the larger fraction of long paths in `roadnet-usa`. Lastly, while the decision on which views to materialize heavily depends on the estimated view size (e.g., size budget constraints, and proxy for view creation cost), these 2-hop connector views are unlikely to be materialized for the two homogeneous networks, due to the large view sizes predicted by our cost model (Figure 5).

G. View Selection Runtimes

This experiment measures the runtime of our view selection algorithm, which identifies the most useful views to materialize subject to a space budget. As described in VI-B, our algorithm is formulated as a $0-1$ knapsack problem. Figure 8 shows our results for different sets of uniformly distributed view weights and values, with runtimes averaged over 10 runs, plotted in log-log scale. Aside from the 1 GB budget case (where the solver finishes more quickly due to more views having weight larger than the budget) all runtimes scale superlinearly on number of views. We believe this is largely due to a core component in our selection algorithm being currently single-threaded [24]. Nonetheless, our results with the single-threaded implementation show that even for a large set of candidate views (e.g., 10^5), KASKADE can select the most useful views for materialization in less than a few seconds. It is possible to further reduce the runtime by splitting the knapsack problem into multiple parallel subproblems, or to parallelize the solver implementation, which we leave as future work.

VIII. Related Work

Views and language. Materialized views have been widely used in the relational setting to improve query runtime by amortizing and hiding computation costs [28]. This inspires our work, but the topological nature of graph views makes them new and different. The notion of graph views and algorithms for their maintenance was first introduced in[15] by Zhuge and Garcia-Molina in 1998, but since then there has been little attention in this area. With KASKADE, we aim at providing a practical approach that can be used to deal with large-scale graphs. It addresses various problems related to graph views, including view enumeration, selection, as well as view-based query rewriting. In this paper, we focus on extracting views for the graph traversal portion of our queries, since these are the most crucial for performance. An interesting avenue for future work is to address a combined analysis of the relational and graph query fragments, related to what [29] proposes for a more restricted set of query types or [30] does for OLAP on graphs scenarios. The challenge is to identify and overcome the limits of relational query containment and view rewriting techniques for our hybrid query language. A related challenge is to support maintenance of our graph views in the case of updates, which we leave as future work. Fan et al. [31] provide algorithms to generate views for speeding up fixed-sized subgraph queries, whereas KASKADE targets traversal-type queries that can contain an arbitrary number of vertices/edges. They do not provide a system for selecting the best views to generate based on a budget constraint. Le et al. [32] present algorithms for rewriting queries on SPARQL views, but lack view selection. Katsifodimos et al. [33] present techniques for view selection to improve performance of XML queries, which are limited to trees due to XML's structure.

RDF and XML. The Semantic Web literature has explored the storage and inference retrieval of RDF and OWL data extensively [34]. While most efforts focused on indexing and storage of RDF triples, there has also been work on maintenance algorithms for aggregate queries over RDF data [35]. While relevant, this approach ignores the view selection and rewriting problems we consider here, and it has limited applicability outside RDF. RDFViewS [21] addresses the problem of view selection in RDF databases. However, the considered query and view language support pattern matching (which are translatable to relational queries) and not arbitrary path traversals, which are crucial in the graph applications we consider. Similar

caveats are present in prior work on XML rewriting [36], [37], including lack of a cost model for view selection.

GFDs. Graph Functional Dependencies (GFDs) work aims at discovering parts of a graph that violate topological constraints [38], [39], [40], [41], [42]. Our focus with graph query optimization is different: materialize smaller graphs to speed up query execution. We see their work as complementary to ours, in that dependencies discovered by their approach may be provided in the form of schema constraints inference rules in KASKADE. A future work direction is to translate instantiations constraint mining rules (§ V-A) into GFDs.

Graph summarization and compression. Summarization finds smaller graphs that are representative of the original graph to speed up graph algorithms or queries, for graph visualization, or to remove noise [43]. Most related is summarization to speed up graph computations for certain queries (lossless or lossy) [44], [45], [46], [47], [48]. As far as we know, prior work in this area has not explored the use of connectors and summarizers as part of a general system to speed up graph queries. Rudolf et al. [49] describe summarization templates in SAP HANA, which can be used to produce what we call graph views. However, their paper lacks a system that determines what views to materialize, nor uses the views to speed up graph queries. There has been significant work on lossless graph compression to reduce space usage or improve performance of graph algorithms (see, e.g., [50], [51], [52]). This is complementary to our work on graph views, and compression could be applied to reduce their memory footprint.

IX. Conclusions

We presented KASKADE, a graph query optimization framework that employs materialization to efficiently evaluate queries over graphs. Many application repeatedly run similar queries over the same graph, and many production graphs have structural properties that restrict the types of vertices and edges in them. These facts motivate KASKADE to automatically derive graph views using a new constraint-based view enumeration technique, and a novel cost model that accurately estimates the size of graph views. We show that queries rewritten over some of these views can provide up to 50 times faster response times. Finally, the rewriting techniques we have proposed are engine-agnostic (i.e., only use fundamental graph transformations that typically yield smaller graphs), thus applicable to other graph systems.

Acknowledgements

We thank the reviewers for their thoughtful feedback on this paper, which led to significant improvements. This research was supported by DOE Early Career Award #DE-SC0018947, NSF CAREER Award #CCF-1845763, Alfred P. Sloan UCEM PhD Fellowship, and Microsoft Research PhD Fellowship.

References

[1] A. Ching *et al.*, "One Trillion Edges: Graph Processing at Facebook-Scale," in *PVLDB 2015*, pp. 1804–1815.
[2] A. Halevy *et al.*, "Goods: Organizing Google's Datasets," in *SIGMOD*, 2016.
[3] R. Mavlyutov *et al.*, "Dependency-Driven Analytics: A Compass for Uncharted Data Oceans," in *CIDR 2017*.
[4] "DBpedia." https://wiki.dbpedia.org
[5] F. M. Suchanek, G. Kasneci, and G. Weikum, "Yago: a core of semantic knowledge," in *WWW 2007*, pp. 697–706.
[6] "Network Repository." http://networkrepository.com
[7] D. Yan *et al.*, "Big Graph Analytics Platforms," *Foundations and Trends in Databases*, vol. 7, pp. 1–195, 2017.
[8] C. Curino *et al.*, "Hydra: a federated resource manager for data-center scale analytics," in *NSDI 2019*.
[9] N. Francis *et al.*, "Cypher: An Evolving Query Language for Property Graphs," in *SIGMOD 2018*, pp. 1433–1445.
[10] "Neo4j Graph Database." http://neo4j.org
[11] "Property Graphs." www.w3.org/2013/socialweb/papers/Property_Graphs.pdf
[12] "AgensGraph Hybrid Graph Database." http://www.bitnine.net
[13] "Graph processing with SQL Server." https://docs.microsoft.com/en-us/sql/relational-databases/graphs/sql-graph-overview
[14] O. van Rest *et al.*, "PGQL: A Property Graph Query Language," in *GRADES 2016*, pp. 7:1–7:6.
[15] Y. Zhuge and H. Garcia-Molina, "Graph structured views and their incremental maintenance," in *ICDE 1998*, pp. 116–125.
[16] J. M. F. da Trindade *et al.*, "Kaskade: Graph Views for Efficient Graph Analytics," *CoRR 2019*, vol. abs/1906.05162, Preprint, http://arxiv.org/abs/1906.05162.
[17] "SWI-Prolog Engine." http://www.swi-prolog.org
[18] T. K. Sellis, "Multiple-Query Optimization," *ACM Trans. Database Syst.*, vol. 13, pp. 23–52, 1988.
[19] P. Roy *et al.*, "Efficient and Extensible Algorithms for Multi Query Optimization," in *SIGMOD 2000*, pp. 249–260.
[20] A. Deshpande, Z. Ives, and V. Raman, "Adaptive Query Processing," *Foundations and Trends in Databases*, pp. 1–140, 2007.
[21] F. Goasdoué *et al.*, "View Selection in Semantic Web Databases," in *VLDB 2011*, pp. 97–108.
[22] D. Theodoratos, S. Ligoudistianos, and T. Sellis, "View selection for designing the global data warehouse," *Data Knowl. Eng.*, vol. 39, pp. 219–240, 2001.
[23] "SWI-Prolog Syntax." http://www.swi-prolog.org/man/syntax.html
[24] "Google OR Tools." https://developers.google.com/optimization/bin/knapsack
[25] "GraphDBLP." https://github.com/fabiomercorio/GraphDBLP
[26] "SNAP LiveJournal dataset." http://snap.stanford.edu/data/soc-LiveJournal1.html
[27] "APOC Neo4j." https://neo4j-contrib.github.io/neo4j-apoc-procedures/#algorithms
[28] A. Gupta and I. S. Mumick, in *Materialized Views*, 1999, ch. Maintenance of Materialized Views: Problems, Techniques, and Applications, pp. 145–157.
[29] C. Lin *et al.*, "Fast In-memory SQL Analytics on Typed Graphs," in *VLDB 2016*, pp. 265–276.
[30] C. Chen *et al.*, "Graph OLAP: Towards online analytical processing on graphs," in *ICDM 2008*, pp. 103–112.
[31] W. Fan, X. Wang, and Y. Wu, "Answering Pattern Queries Using Views," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, pp. 326–341, Feb 2016.
[32] W. Le *et al.*, "Rewriting queries on SPARQL views," in *WWW 2011*, pp. 655–664.
[33] A. Katsifodimos, I. Manolescu, and V. Vassalos, "Materialized view selection for XQuery workloads," in *SIGMOD 2012*, pp. 565–576.
[34] G. Antoniou and F. Van Harmelen, *A semantic web primer*. MIT press, 2004.
[35] E. Hung, Y. Deng, and V. S. Subrahmanian, "RDF aggregate queries and views," in *ICDE 2005*.
[36] N. Onose *et al.*, "Rewriting Nested XML Queries Using Nested Views," in *SIGMOD 2006*, pp. 443–454.
[37] W. Fan *et al.*, "Rewriting Regular XPath Queries on XML Views," in *ICDE 2007*, pp. 666–675.
[38] W. Fan, Y. Wu, and J. Xu, "Functional Dependencies for Graphs," in *SIGMOD 2016*, pp. 1843–1857.
[39] W. Fan *et al.*, "Discovering Graph Functional Dependencies," in *SIGMOD 2018*, pp. 427–439.
[40] W. Fan, X. Liu, and Y. Cao, "Parallel reasoning of graph functional dependencies," in *ICDE*, apr 2018, pp. 593–604.
[41] W. Fan, "Dependencies for graphs: Challenges and opportunities," *J. Data and Information Quality*, vol. 11, pp. 5:1–5:12, Feb. 2019.
[42] W. Fan and P. Lu, "Dependencies for graphs," *ACM Trans. Database Syst.*, vol. 44, pp. 5:1–5:40, Feb. 2019.
[43] Y. Liu *et al.*, "Graph Summarization Methods and Applications: A Survey," *ACM Comput. Surv.*, vol. 51, pp. 62:1–62:34, Jun. 2018.
[44] W. Fan *et al.*, "Query Preserving Graph Compression," in *SIGMOD 2012*, pp. 157–168.
[45] S. Navlakha, R. Rastogi, and N. Shrivastava, "Graph Summarization with Bounded Error," in *SIGMOD 2008*, pp. 419–432.
[46] K. Xirogiannopoulos and A. Deshpande, "Extracting and Analyzing Hidden Graphs from Relational Databases," in *SIGMOD 2017*, pp. 897–912.
[47] N. Tang, Q. Chen, and P. Mitra, "Graph Stream Summarization: From Big Bang to Big Crunch," in *SIGMOD 2016*, pp. 1481–1496.
[48] C. Chen *et al.*, "Mining Graph Patterns Efficiently via Randomized Summaries," in *VLDB 2009*, pp. 742–753.
[49] M. Rudolf *et al.*, "SynopSys: Large Graph Analytics in the SAP HANA Database Through Summarization," in *GRADES 2013*, pp. 16:1–16:6.
[50] P. Boldi and S. Vigna, "The webgraph framework I: compression techniques," in *WWW 2004*, pp. 595–602.
[51] J. Shun, L. Dhulipala, and G. E. Blelloch, "Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+," in *DCC 2015*, pp. 403–412.
[52] A. Maccioni and D. J. Abadi, "Scalable Pattern Matching over Compressed Graphs via Dedensification," in *KDD 2016*, pp. 1755–1764.