Chasing Parallelism in Aggregating Graph Queries

Alin Deutsch ✉
University of California, San Diego, CA, USA

To Val Tannen, my Doktorvater and role model

Abstract

In practice, one frequently encounters queries that extract tabular results from graph databases by employing grouping and aggregation. This paper introduces a technique for rewriting the group-by list of graph queries in order to increase aggregation parallelism in graph engines that conform to a modern instantiation of the Bulk-Synchronous-Parallel computation model.

2012 ACM Subject Classification Information systems → Graph-based database models; Information systems → Query optimization

Keywords and phrases Graph Databases, Grouping and Aggregation, Parallel Graph Computation Models, Rewriting, Constraint-based Minimization

Digital Object Identifier 10.4230/OASIcs.Tannen.2024.5

1 Introduction

It is a long-standing tradition in both database research and practice to conceptualize data as a graph, in which vertices model the entities of the application domain and edges model their relationships. This tradition preceded the advent of the relational model, starting with the *network* database model; it co-existed with the relational model from its inception, in form of the *Entity/Relationship* model; it matured into dedicated database systems conforming to the *semistructured* model and its specializations as XML and JSON [1]; it was recently rejuvenated in form of the *property graph* data model by strong commercial demand from the industrial sector; and it culminated in the just-released GQL standard [11], the first ISO/ANSI query language standard since SQL.

An invariant throughout most of this evolution has been the need for queries that cross models by extracting data from the underlying graph and presenting it in tabular form. Research on the property graph model and this class of queries is not simply a revisit of the plethora of results developed for the semistructured model. Two recent developments raise novel research challenges:

(i) Recent industrial demand for expressing graph analytic tasks requires the output tables to be the result of grouping and aggregation, as inspired by SQL. This has led to incorporating aggregation primitives as first-class citizens into graph query languages, as opposed to treating them as an afterthought in the era of semistructured data research.

(ii) New query evaluation strategies are called for due to the recent development of novel parallel computing paradigms motivated by today's distributed and cloud infrastructures.

This paper focuses on maximizing the degree of parallelism for the aggregation task when run in a graph engine that conforms to a commercially implemented parallel computation paradigm called *Edge-Map/Vertex-Reduce (EMVR)*. The solution involves exploiting integrity constraints to rewrite the group-by list of queries in ways that expose opportunities to distribute the computation over independently working processors.

EMVR is an instantiation of Valiant's *Bulk-Synchronous Parallel (BSP)* model of computation [19]. The results presented here can be ported to other currently circulating instantiations of the BSP model for graphs.

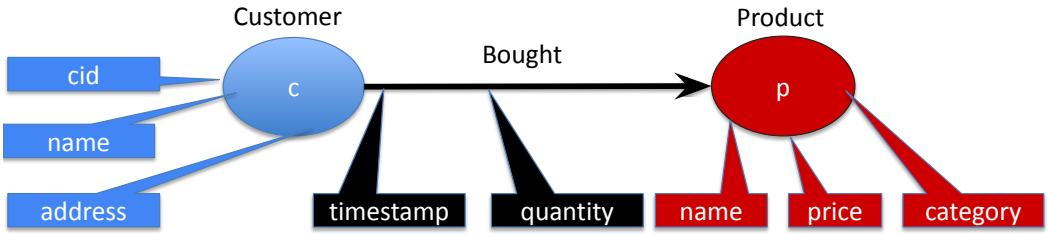

Figure 1 Modeling customers and products as vertices, and sales as edges.

The remainder of this paper is organized as follows. Section 2 describes the property graph model, and Section 3 introduces the EMVR computation paradigm. Section 4 illustrates a grouping and aggregating graph query that serves as the running example for the subsequent discussion. Section 5 illustrates a class of EMVR plans that can generically support the class of queries represented by the running example. These plans contain a parallelism-limiting bottleneck, which can be removed by exploiting knowledge of integrity constraints. Section 6 illustrates such an optimized plan for the running example. Section 7 describes our optimization approach, which is based on reducing the problem to relational minimization under integrity constraints. We conclude in Section 8.

2 The Property Graph Data Model

Per the GQL standard [11], property graphs comprise vertices and binary edges connecting them. In object-oriented fashion, vertices are uniquely determined by an internal identifier. Edges may be directed or undirected. Both vertices and edges have a set of attributes, called *properties*, which are key-value pairs with the key giving the property name. Both vertices and edges may have a type, which specifies their property names and the types of their property values. The GQL standard allows both a typed version which specifies a graph schema, and a schema-free version. Since our interest is in exploiting integrity constraints, we focus on the typed graph version here.

▶ **Example 1.** Assume we wish to model a domain revolving around customers who buy products. Individual customers are modeled as vertices of type *Customer*, with attributes *cid*, *name* and *address*. Products are modeled as vertices of type *Product*, with attributes *pname*, *price* and *category*. The fact that a customer represented by vertex c has bought a product represented by vertex p is modeled by a directed edge of type *Bought* oriented from c to p. The edge is adorned with attributes *timestamp* and *quantity* (see Figure 1). Figure 2 shows a graph modeling a history of sales. ⌟

3 The Edge-Map/Vertex-Reduce Graph Computation Model

The *Edge-Map/Vertex-Reduce (EMVR)* model of computation is an instantiation to graphs of Valiant's BSP paradigm (described below). It organizes parallel processing of the graph into *vertex functions* and *edge functions*. Each vertex/edge is thought of as an independent processor that can run a vertex/edge function in parallel with the other vertices/edges.

The computation is a sequence of steps guided by the *active vertex set*. A step can be of either vertex or edge kind. A *vertex step* maps the vertex function in parallel over all active vertices. An *edge step* maps the edge function in parallel over all edges incident to the active vertex set. Among other tasks, each step computes the next active vertex set. The next step can start only when the current step has completed (which requires a synchronization barrier).

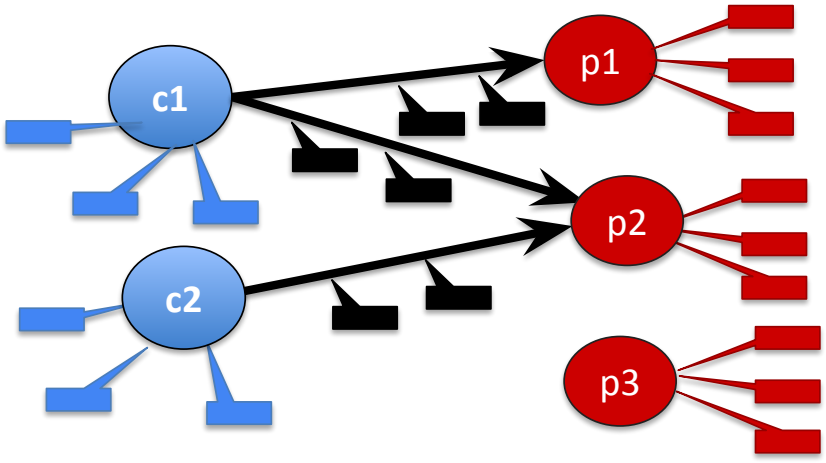

Figure 2 A graph showing multiple sales.

The *Reduce* phase of the EMVR step consists in aggregating the values generated by mapping the vertex or edge functions. In the EMVR model, aggregation is performed by *accumulators*. Accumulators are containers that store a data value, accept inputs, and aggregate these inputs into the stored data value using a binary operation $\oplus$.[1].

Since the only available EMVR implementation we are aware of is provided by the TigerGraph company, we adopt the syntax of its query language, GSQL, to denote accumulator types [7].

▶ **Example 2.** In GSQL,

$SumAccum < float >$

denotes the type of accumulators that sum up their floating point inputs. Each accumulator of this type has its stored value initialized to 0 at construction time. During operation, it adds each received input into the stored value using binary addition. ⌟

GSQL provides built-in accumulators that perform the standard aggregations one has come to expect of query languages (sum, average, min, max, logical or, logical and, etc.). It also supports user-defined accumulators, whose behavior is specified by providing the initial stored value and the binary operation $\oplus$.

There are two kinds of accumulators: global or vertex-attached. A *global accumulator* has a single instantiation for the entire computation. A *vertex-attached accumulator* has one instantiation for every vertex.

Accumulator implementations need to ingest inputs generated in parallel by the invocations of the vertex and edge functions. Race conditions are avoided by using an exclusive locking mechanism, which effectively serializes the inputs. The order in which inputs are incorporated into the stored value is nondeterministic. However, if the binary operation $\oplus$ is associative and commutative, the final result of the aggregation is well-defined. This is the case for all built-in accumulators (the average accumulator is implemented to maintain the sum and the count of its inputs, both of which have associative and commutative binary operations).

[1] The inspiration and theoretical underpinnings for thinking about database aggregation in this way go back to Val Tannen's work [12]

3.1 Predecessor BSP Models

Valiant's *Bulk-Synchronous Parallel (BSP)* model [19] includes three main components: a number of processors, each with its own local memory and ability to perform local computation; a communication environment that delivers messages from one processor to another; and a barrier synchronization mechanism. A BSP computation is a sequence of supersteps. A superstep comprises of a computation stage, where each processor performs a sequence of operations on local data, and a communication stage, where each processor sends a number of messages. The processors are synchronized between supersteps, i.e. they wait at the barrier until all processors have received their messages.

The *vertex-centric* computational model was introduced by Google's Pregel [15] system as an adaptation to graph data of the BSP model. In the vertex-centric model, each vertex plays the role of a processor that executes a user-defined program. Vertices communicate with each other by sending messages via outgoing edges, or directly to any other vertex whose identifier they know (e.g. discovered during computation).

Each vertex is identified by a vertex ID. It holds a state, which represents intermediate results of the computation; a list of outgoing edges; and an incoming message queue. Edges are identified by the IDs of their source and destination vertices, and they can also store state. The computation is organized in supersteps delimited by a synchronization barrier, as dictated by the BSP paradigm. During each superstep, each vertex runs the same vertex program in parallel. The vertex program is designed from the perspective of a vertex, operating on local data only: the vertex state, the received messages, and the incident edges. Based on these inputs, the vertex program can modify the vertex state, send messages to neighbors along the edges, and decide whether the vertex remains active for the next superstep. If an inactive vertex receives a message, it is reactivated. At the beginning of a computation all vertices are activated. The computation halts when all vertices are inactivated and no more messages are in transit.

3.2 Examples of BSP graph engines

A vertex-centric BSP model was first introduced in Google's Pregel [15], followed by a proliferation of open-source and commercial implementations, some running on distributed clusters, others realizing vertex communication via a shared memory. Examples include open-source implementations such as Giraph [4], GPS (Graph Processing System) [17] and Apache Spark with its Pregel API GraphX [5]. Graph engines realizing communication via a shared memory include GraphLab [13], Signal/Collect [18], and PowerGraph [8].

Surveys of the landscape can be found in [16, 20], while comparative experimental evaluations have been reported in [3, 9, 14].

More recently, TigerGraph introduced its commercial engine [7], which implements an Edge-Map/Vertex-Reduce API by exploiting thread parallelism within a server as well as distributed parallelism across computing nodes in a cluster.

4 A Grouping and Aggregating Graph Query

The following query conforms to the syntax introduced by GSQL, TigerGraph's query language. It is also close to the syntax of the recently released GQL standard [11], which is inspired by GSQL.[2]

[2] The GQL standard admits two syntactic flavors: an SQL-like one inspired by TigerGraph's GSQL, and one inspired by Neo4j's Cypher language (not shown here).

```
SELECT   c.cid, sum(b.quantity * p.price) AS revenue INTO T
FROM     (c:Customer) -[b:Bought]-> (p:Product)
WHERE    p.category = 'Toys'
GROUP BY c.cid
```

The clauses are mostly familiar from SQL.

The exception is the FROM clause, which, instead of SQL tables, specifies *graph patterns*. Round parentheses denote pattern fragments to be matched against vertices, while arrows with square parentheses are to be matched against edges (the arrow specifies the orientation for directed edges; it is omitted for undirected edges). Vertex and edge types are specified to the right of the colon delimiters, and variables to the left. In this example, the pattern matches directed edges b of type *Bought* oriented from a vertex c of type *Customer* to a vertex p of type *Product*. Each match can be thought of as a tuple whose attribute names are the three variables, and whose values are the identities of the matched vertices and edges. These tuples are collected in a bag referred to as the *binding table*. For the example sales graph in Figure 2, the binding table is

c	b	p
c1	c1 $\longrightarrow$ p1	p1
c1	c1 $\longrightarrow$ p2	p2
c2	c2 $\longrightarrow$ p2	p2

The remaining clauses manipulate the binding table analogously to SQL: the `WHERE` clause keeps only the matches whose product category is 'Toys'; the `GROUP BY` clause groups by the the customer vertex attribute *cid* (assumed here to be a key); finally, the SELECT clause aggregates the groups by summing up the product prices, to obtain the revenue per customer. The result is output into a table called `T`, as directed by the `INTO` sub-clause.

GSQL, Cypher and GQL are significantly more expressive than the example query, for instance admitting multi-hop patterns that match sequences of edges, regular expressions for specifying complex shapes of traversed paths, conjunctions of such regular expression patterns, composition of query blocks, etc. These are not discussed in this paper, which focuses on the tables produced by grouping and aggregation.

5 Supporting Tables with Accumulators

Like many real-life graph queries, our example query crosses between data models, extracting a table from the input graph data.

At first glance, this raises a challenge to supporting this class of queries in a graph engine based on the EMVR model of computation, as the latter is centered around vertices and edges and has no notion of relational tables as first-class citizens.

However, tables can be supported as syntactic sugar in the EMVR model, being implementable as accumulators (this is the case for TigerGraph's GSQL implementation). We borrow GSQL's syntax to denote the types and operations of accumulators.

For our running example query, the GSQL compiler implements table T as an accumulator of type

```
GroupByAccum<int cid, SumAccum<float> revenue>
```

which denotes accumulators that contain a set of key-value pairs, each pair corresponding to a group. In each pair, the *cid* field is the integer group key, and the *revenue* field contains the associated value, which is in turn a nested accumulator that sums up its floating point inputs (its type is denoted in GSQL as `SumAccum<float>`).

This group-by accumulator takes as input key-value pairs (k, v), aggregating them into its contents as follows. Identify the group of k and its associated *revenue* accumulator r (if no such group exists yet, create one, initializing r's contents to 0). Insert the value v into r; since r is of type `SumAccum<float>`, this will add v to the current stored value of r.

An EMVR plan for the example query is shown below.

```
construct global accumulator @@T of type
GroupByAccum<int cid, SumAccum<float> revenue>

define F(edge b of type 'Bought' from c to p) {
    if p.category = 'Toys' then
       @@T += (c.cid -> b.quantity * p.price)
    end
}

for each edge e of type 'Bought' do in parallel
    F(e)
end
```

In the above pseudocode, we borrow the GSQL syntax for signalling that an identifier denotes a global accumulator: prepend two @ characters to its name. Thus, @@T is a global accumulator called T. Recall that globality means that there is only one instance of @@T for the entire query.

We also borrow from GSQL the syntax for inserting a key-value pair (k, v) into a group-by accumulator @@*acc*: `@@acc += (k -> v)`. Notice the function F, which is applied to individual edges. F inserts the revenue from the individual sale modeled by the b edge into the nested *revenue* accumulator associated to customer c's *cid* attribute ($c.cid$): `@@T += (c.cid -> b.quantity * p.price)`.

The for loop maps F over all relevant edges in parallel. The order in which the various invocations of F write into @@T is not determined, depending on how they interleave at runtime. Nevertheless, the semantics of the plan is well-defined, as the result is race-free and interleaving-invariant. Race freedom is ensured via an exclusive locking mechanism that all invocations of F use to write into @@T. Interleaving invariance follows from the fact that @@T aggregates product prices using addition, which is commutative and associative.

When the plan's execution completes, the result is centralized in the global accumulator @@T. GSQL allows subsequent query blocks to refer to the table T, in which case their plan accesses the contents of accumulator @@T instead.

6 Changing Grouping Criteria to Increase Parallelism

While the EMVR plan in Section 5 shows that tables can be supported using accumulators, we observe that this plan features only limited potential for parallel evaluation. Despite the function F being mapped in parallel over all relevant edges, each write operation to global accumulator @@T involves the acquisition of an exclusive lock to avoid race conditions. This effectively serializes the invocations of F, turning the operation of writing to the global accumulator into a bottleneck.

Parallelism can be dramatically unlocked by observing that, since attribute *cid* is a key for *Customer* vertices, each group corresponds to precisely one such vertex. This enables the removal of the global accumulator bottleneck @@T by replacing it with many vertex-attached accumulators, one located at each *Customer* vertex. By writing only into its own

accumulator, each vertex can work on its own group independently of the other vertices, yielding an eminently parallelizable plan. We detail this alternate plan below.

```
attach to each 'Customer' vertex its own
accumulator of type SumAccum<float>, called @revenue

define F_opt(edge b of type 'Bought' from c to p) {
    if p.category = 'Toys' then
       c.@revenue += b.quantity * p.price
    end
}

for each edge e of type 'Bought' do in parallel
    F_opt(e)
end
```

Following GSQL syntax, the single leading @ indicates that the @*revenue* accumulators are vertex-attached. The type of these accumulators, `SumAccum<float>`, states that they each take floating-point inputs and add them to their stored value.

Also following GSQL syntax, *c*.@*revenue* denotes the @*revenue* accumulator attached to vertex *c*.

Notice that the edge function F_{opt} writes into *c*.@*revenue*, the accumulator of *Customer* vertex *c*. Only the writes due to edges adjacent to *c* compete for access to this accumulator and require serialization. Thus, each @*revenue* accumulator serializes only as many writes as the degree of *c*, which is likely a much smaller quantity than the number of writes serialized by the global accumulator @@T, namely the total count of *Bought* edges in the graph.

Upon completion of the plan's execution, there is no centralized data structure holding table T. This table is virtual; its contents are distributed across *Customer* vertices. The components of each virtual tuple $t \in T$ resides a the corresponding vertex *c*, with *t.cid* stored in *c.cid* and *t.revenue* stored in *c*.@*revenue*.

7 Reduction to Relational Minimization under Constraints

We present an optimization technique that reasons about the structure of the graph to automatically rewrite global-accumulator-based plans (like the one in Section 5) into vertex-accumulator-based plans (like the one in Section 6) whenever possible.

The crux of the reasoning consists in deciding whether the original group-by list can be replaced with a single vertex variable while preserving the partition of the binding table into groups. If this is possible, then the output table can be virtualized and distributed as illustrated in Section 6. In not possible, groups do not correspond to individual vertices and therefore need to be hosted somewhere else; in the EMVR model, the only alternative is a global accumulator (as illustrated in Section 5). The global-accumulator-based plans are always am available choice, and in the general case may be unavoidable. However, they restrict parallelism, so we seek alternatives whenever possible.

It turns out that we need not devise a customized algorithm for this optimization task. Instead, we can reduce it to the well-studied problem of minimization of relational queries under integrity constraints, for which there exist sound and complete algorithms [6, 10]. The technically interesting exercise consists in defining the reduction, which we present below.

7.1 Encoding vertex types as relations

We encode each vertex type VT as a homonymous relational table, whose columns correspond to VT's attributes. We also prepend a column intended to hold the object id of the vertex.

▶ **Example 3.** In our example, assuming that the $Customer$ vertex type has attributes cid, $name$ and $address$, it is encoded as a table

$$Customer(id, cid, name, address).$$

Similarly, assuming that the $Product$ vertex type has attributes pno, $pname$, $price$ and $category$, we encode it as table

$$Product(id, pno, pname, price, category).$$ ⌟

7.2 Encoding edge types as relations

We encode each edge type ET as a homonymous relational table whose columns correspond to ET's attributes. We also prepend two columns, intended to hold the ids of the edge's source and target vertices.

▶ **Example 4.** In our example, assuming that the $Bought$ edge type has attributes $timestamp$ and $quantity$, it is encoded as a table

$$Bought(c, p, timestamp, quantity).$$ ⌟

7.3 Encoding the group-by list as a relational query

The list of group-by attributes is the one we seek to rewrite. Since the problem we reduce to is query minimization under constraints, we encode this list as a relational query whose body contains the relational encoding of the type information for the variables of the original query's graph pattern.

▶ **Example 5.** We show the encoding of the group-by list of our running example query as a relational conjunctive query:

$$\begin{aligned} G() \quad \leftarrow \quad & GroupBy(cid), Customer(c, cid, name, address), \\ & Bought(c, p, timestamp, quantity), Product(p, pno, pname, price, category). \end{aligned}$$

Notice the use of the predicate $GroupBy$ to label the elements of the group-by list. We seek to rewrite this to

$$\begin{aligned} G() \quad \leftarrow \quad & GroupBy(c), Customer(c, cid, name, address), \\ & Bought(c, p, timestamp, quantity), Product(p, pno, pname, price, category) \end{aligned}$$

which groups by the vertex id instead of the cid attribute. ⌟

Note that the encoding queries in Example 5 are boolean (they have no distinguished attributes). A natural encoding alternative that we initially considered would have chosen to place the group-by list elements as distinguished variables in the query head ($G(cid)$ and $G(c)$ in the example). However, this would have potentially required the rewriting to change the type of the query output: for instance, assume alphanumerical strings for the cid attribute, and unsigned integers for the vertex id. More gravely, even if we ignore typing information, there are cases when the original group-by list has a different length than the rewritten one, requiring a change in the arity of the query head. Standard query minimization would no longer apply in this case, as it always preserves the arity of the query.

▶ **Example 6.** Assuming that persons are identified by their name, date of birth, and place of birth, and that the original query seeks to group by persons, the alternate encoding scheme we considered would yield a query like

$$G(name, dob, place) \leftarrow Person(p, name, dob, place).$$

We seek to rewrite the group-by list to use the vertex id instead. In relationally encoded form, this would amount to obtaining

$$G(p) \leftarrow Person(p, name, dob, place)$$

which would technically fall outside the purview of standard query minimization. ⌟

The reason why this more natural encoding disables standard query minimization is fundamental: minimization preserves equivalence to the original query, and the concept of equivalence between queries with different output types/arity seems meaningless. It is conceivable (and we were tempted!) to define a more relaxed version of query equivalence (and therefore minimization) that changes the arity of the query head and it would be possible to adapt state-of-the-art algorithms [6] accordingly. The benefit of the alternate encoding introduced here is that it reuses the standard theory and algorithms with no change.

▶ **Example 7.** Revisiting Example 6, our actually adopted encoding scheme yields

$$G() \leftarrow GroupBy(name), GroupBy(dob), GroupBy(place), Person(p, name, dob, place)$$

for the original group-by list, and

$$G() \leftarrow GroupBy(p), Person(p, name, dob, place)$$

for its rewritten form.

Notice how this encoding into boolean queries avoids the issue of arity and/or type discrepancy between original and rewritten query output. ⌟

7.4 Encoding the vertex id using relational dependencies

The property graph data model regards vertices through an object-oriented lens: each vertex is an object with an id that uniquely identifies it. This fact is crucially exploited in rewriting the group-by list and therefore must be captured in the relational encoding.

The consequence for the encoding is that the id column acts as a key for the table modeling the vertex type. The standard way to express key constraints in a relational setting is to resort to *functional dependencies* [2].

▶ **Example 8.** We can express the fact that the id of Customer vertices uniquely determines each vertex, in particular the values of the vertex attributes, using the functional dependency

$$Customer(c, cid_1, name_1, address_1) \wedge Customer(c, cid_2, name_2, address_2)$$
$$\longrightarrow cid_1 = cid_2 \wedge name_1 = name_2 \wedge address_1 = address_2.$$

⌟

Functional dependencies like the one shown in Example 8 are useful in optimizing the body of the query reflecting from the graph patterns in the `FROM` clause and the conditions in the `WHERE` clause. This kind of optimization is beyond the scope of this paper, as it can be tackled by standard rewriting techniques such as the ones in [6, 10]. Here, we focus on optimizing the group-by list. Unfortunately, this optimization requires a kind of rewriting that standard algorithms for minimization under constraints cannot achieve using functional dependencies alone.

Intuitively, the reason is that the rewriting needs to drop certain atoms from the query body and introduce new ones instead: in Example 7, atoms $GroupBy(name)$, $GroupBy(dob)$, $GroupBy(place)$ need to be dropped, and atom $GroupBy(p)$ added; in Example 5, atom $GroupBy(cid)$ needs to be dropped in favor of adding atom $GroupBy(c)$. These operations require as justification a class of relational dependencies known as *tuple-generating dependencies (TGDs)* [2]. We therefore add to the relational encoding appropriate TGDs.

▶ **Example 9.** The fact that the id of Customer vertices uniquely determines each vertex, in particular the values of the vertex attributes, implies that, once the query groups by the vertex id, it can just as well group by any of the vertex attributes without changing the contents of the groups. We capture this with the following TGD:

$$\begin{aligned} Customer(c, cid, name, address) \wedge GroupBy(c) \quad &\rightarrow \quad GroupBy(cid) \\ &\wedge \quad GroupBy(name) \\ &\wedge \quad GroupBy(address) \end{aligned}$$ ⌟

7.5 Encoding vertex key constraints as relational dependencies

Another crucial ingredient on which the reasoning about the group-by list is based is the knowledge about vertex keys, i.e. attribute sets whose values uniquely determine the vertex. Vertex keys are analogous to their relational counterpart, the relational key integrity constraint. And yet, for our purposes the encoding of vertex key constraints requires TGDs, in contrast to the standard representation of relational keys using functional dependencies. The motivation is analogous to the one discussed for encoding knowledge about vertex ids.

▶ **Example 10.** In our running example, the *cid* attribute value determines the $Customer$ vertex. Consequently, *cid* is a key for the encoding relational table, and this information is typically captured by the functional dependency

$$Customer(c_1, cid, name_1, address_1) \wedge Customer(c_2, cid, name_2, address_2) \longrightarrow c_1 = c_2$$

which states that the Cid attribute determines the identity of the $Customer$ vertices. As per our previous discussion, this functional dependency does not suffice for rewriting the group-by list, as it needs to drive the introduction and dropping of $GroupBy$ atoms. What is needed is a tuple-generating dependency such as

$$Customer(c, cid, name, address) \wedge GroupBy(cid) \longrightarrow GroupBy(c).$$

Intuitively, this TGD states that, if the query already groups by the *cid* attribute, it might just as well group by the vertex id without changing the contents of the resulting groups. ⌟

7.6 Putting it all together: rewriting the group-by list

Our technique for rewriting the group-by list involves the following steps:

1. Encode the group-by list as a relational conjunctive query Q using the types inferred for the variables in the original graph pattern.
2. Encode the vertex id and vertex key information for each vertex type using relational dependencies; call their set Σ.
3. Feed Q and Σ to an algorithm for minimization under constraints (such as the Provenance-Aware Chase&Backchase (PACB) [6, 10]). The algorithm is invoked as a black box, which returns a set of rewritings of Q.
4. Inspect the rewritings, keeping those that contain a single $GroupBy$ atom, the argument of which is a variable that binds to vertex identifiers.

Any rewriting found in Step 4 admits a plan in which each group is computed in parallel at its corresponding vertex, and in which the per-group aggregation is implemented by vertex-attached accumulators. In case of multiple qualifying rewritings, the selection can be informed by cost information (cost-based selection is beyond the scope of this paper).

▶ **Example 11.** For the running example, we obtain the first query from Example 5 and the set Σ comprising the dependencies from Examples 8, 9, and 10. The PACB algorithm run on this query with Σ yields two minimal rewritings, corresponding to both queries shown in Example 5. They are both minimal, but only the second query satisfies the criteria of Step 4 above. From this rewriting, we can directly generate a plan that aggregates groups in vertex-attached accumulators at the *Customer* vertices.

Though understanding the inner workings of the PACB algorithm is not necessary since the latter is used as a black box, we sketch them for this example in order for the reader to better appreciate the need to design the encoding in its current form.

The PACB starts from query

$$\begin{aligned} G() \quad \leftarrow \quad & GroupBy(cid), Customer(c, cid, name, address), \\ & Bought(c, p, timestamp, quantity), Product(p, pno, pname, price, category). \end{aligned}$$

It chases this query with the vertex-key-encoding dependency

$$Customer(c, cid, n, a) \wedge GroupBy(cid) \longrightarrow GroupBy(c)$$

from Example 10, to obtain

$$\begin{aligned} G() \quad \leftarrow \quad & GroupBy(cid), Customer(c, cid, name, address), \\ & Bought(c, p, timestamp, quantity), Product(p, pno, pname, price, category), \\ & \boxed{\text{GroupBy(c)}}. \end{aligned}$$

Note the addition of the $GroupBy(c)$ atom. This query encodes a group-by list that includes *both* the attribute *cid* and the identifier c of *Customer* vertices.

A further chase step applies, with the vertex-id-encoding dependency

$$\begin{aligned} Customer(c, cid, name, address) \wedge GroupBy(c) \quad & \rightarrow \quad GroupBy(cid) \\ & \wedge \quad GroupBy(name) \\ & \wedge \quad GroupBy(address) \end{aligned}$$

from Example 9, yielding

$$\begin{aligned} G() \quad \leftarrow \quad & GroupBy(cid), Customer(c, cid, name, address), \\ & Bought(c, p, timestamp, quantity), Product(p, pno, pname, price, category) \\ & \boxed{\text{GroupBy(c)}}, \\ & \boxed{\text{GroupBy(name), GroupBy(address)}}. \end{aligned}$$

Notice the addition of the last two *Groupby* atoms. No further chase steps apply.

Intuitively, the algorithm has at this point inferred that, given that the original query was grouping by the *cid* attribute of *Customer* vertices, it is safe to add the vertex identity and all other *Customer* vertex attributes to the group-by list as this won't change the group contents.

The PACB algorithm now seeks minimal rewritings of the original query among the subqueries of the chase result (see [6] for details on how this search is directed to avoid exhaustively inspecting all subsets of atoms in the chase result). Notice that both queries

from Example 5 are found this way. The first query coincides with the original user query (unsurprisingly, as its singleton group-by list is necessarily minimal). More interestingly, it turns out that the second query is an equivalent rewriting (the PACB checks equivalence by chasing, and this time the TGD from Example 9 is instrumental).

Also note that the second query satisfies the requirement of having a single *GroupBy* atom, whose variable c binds to vertex identifiers (namely those of $Customer$ vertices). It is therefore the one returned by Step 4 of our optimization technique. ⌟

Let's call a group-by list *Unary Vertex-Grouping (UVG)* if it contains a single variable, which binds to vertex identifiers. We say that a query is UVG if its group-by list is UVG. UVG queries admit plans based on vertex-attached accumulators, thus avoiding the aggregation bottleneck caused by the use of global accumulator to simulate the output table. Our technique is guaranteed sound and complete for finding UVG rewritings.

▶ **Theorem 12.** *The technique described here is guaranteed to find precisely all UVG rewritings of a given query's group-by list under a given set of vertex key constraints.*

This follows from the fact that all TGDs involved in the relational encoding are so-called *full*, for which the chase procedure is guaranteed to terminate [2]. In addition, it follows from the soundness and completeness of the PACB algorithm whenever the chase terminates [6, 10].

7.7 Exploiting Additional Classes of Constraints

Since our approach is based on a reduction to relational minimization under constraints, as long as the constraints in the graph schema admit encoding as dependencies for which the chase terminates, the PACB remains a sound and complete minimization procedure. We can exploit such constraints for free, allowing them to synergistically interact with each other to uncover further UVG rewritings.

One class of frequently encountered constraints pertains to the cardinality of edges when viewed as relationships between the connected nodes. Many-to-one and one-to-one constraints can also be exploited to uncover UVG rewritings of the group=by list.

▶ **Example 13.** Assume that the graph in our running example provides information on the customer's city via a $LivesIn$ edge.

Consider the following variation of our running example query, in which we wish to list the customer *cid*, the *name* of the city they live in, and the revenue from the sales to this customer.

```
SELECT   c.cid, cty.name, sum(b.quantity * p.price) AS revenue INTO T
FROM     (cty:City) <-[:LivesIn]- (c:Customer) -[b:Bought]-> (p:Product)
GROUP BY c.cid, cty.name
```

Assume that the graph schema declares, as above, that the attribute *cid* is a key for $Customer$ vertices. Additionally, it states that attribute *name* is a key for $City$ vertices, and that the $LivesIn$ edge is many-to-one (customers live in at most one city). In that case the above query can be equivalently rewritten to

```
SELECT   c.cid, cty.name, sum(b.quantity * p.price) AS revenue INTO T
FROM     (cty:City) <-[LivesIn]- (c:Customer) -[b:Bought]-> (p:Product)
GROUP BY c
```

whose new group-by list makes explicit the fact that each group is determined by a $Customer$ vertex. Table T can once again be virtualized and distributed across $Customer$ vertices by a plan based on vertex-attached accumulators.

The rewritten group-by list is obtainable by feeding the PACB the dependencies from Example 11, accompanied by the encoding of the key constraint for $City$ vertices,

$$City(cty, name, population) \wedge GroupBy(name) \longrightarrow GroupBy(cty)$$

and the encoding as a TGD of the many-to-one constraint on the $LivesIn$ edge:

$$LivesIn(cust, cty) \wedge GroupBy(cust) \longrightarrow GroupBy(cty).$$

Intuitively, the latter dependency states that, once the query groups by $Customer$ vertex id, it can just as well group by the $City$ vertex id without changing the groups, because the $Customer$ vertex determines the $City$ vertex. ⌟

In general, we capture a many-to-one constraint on an edge of type ET with the TGD

$$ET(src, tgt, \ldots) \wedge GroupBy(src) \longrightarrow GroupBy(tgt),$$

where the dots stand for the edge attributes.

One-to-one constraints on an edge are captured by two TGDs, corresponding to the many-to-one TGDs in each direction.

We can uncover further rewriting opportunities by exploiting any additional class of constraints, as long as they are expressible as relational dependencies, and as long as the chase with the resulting set of dependencies terminates.

8 Conclusion

For graph queries that return tables by performing grouping and aggregation in an engine conforming to the edge-map/vertex-reduce paradigm, the grouping criteria determine the degree of parallelism of the aggregation task. By exploiting the integrity constraints in the graph schema, the user query's group-by list can be equivalently rewritten to expose the maximally available degree of aggregation parallelism. This applies more generally to all models in the class of vertex-centric computation models. There is no need to devise novel algorithms for rewriting the group-by list under integrity constraints - it suffices to devise a novel encoding scheme that reduces the problem to relational query minimization under dependencies. This enables us to leverage the sound and complete PACB minimization algorithm introduced in Peter Buneman's Festschrift [6].

References

1 Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

2 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. URL: `http://webdam.inria.fr/Alice/`.

3 Khaled Ammar and M. Tamer Özsu. Experimental analysis of distributed graph systems. *Proc. VLDB Endow.*, 11(10):1151–1164, June 2018. `doi:10.14778/3231751.3231764`.

4 Apache. Apache giraph, 2020. URL: `https://giraph.apache.org/`.

5 Apache. Apache spark graphx, 2020. URL: `https://spark.apache.org/graphx/`.

6 Alin Deutsch and Richard Hull. Provenance-directed chase&backchase. In Val Tannen, Limsoon Wong, Leonid Libkin, Wenfei Fan, Wang-Chiew Tan, and Michael P. Fourman, editors, *In Search of Elegance in the Theory and Practice of Computation - Essays Dedicated to Peter Buneman*, volume 8000 of *Lecture Notes in Computer Science*, pages 227–236. Springer, 2013. doi:10.1007/978-3-642-41660-6_11.

7 Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. Tigergraph: A native MPP graph database. *CoRR*, abs/1901.08248, 2019. arXiv:1901.08248.

8 Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, October 2012. USENIX Association. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez.

9 Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Özsu, Xingfang Wang, and Tianqi Jin. An experimental comparison of pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 7:1047–1058, August 2014. doi:10.14778/2732977.2732980.

10 Ioana Ileana, Bogdan Cautis, Alin Deutsch, and Yannis Katsis. Complete yet practical search for minimal query reformulations under constraints. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1015–1026. ACM, 2014. doi:10.1145/2588555.2593683.

11 ISO. GQL, 2024. URL: https://www.iso.org/standard/76120.html.

12 S. Kazem Lellahi and Val Tannen. A calculus for collections and aggregates. In Eugenio Moggi and Giuseppe Rosolini, editors, *Category Theory and Computer Science, 7th International Conference, CTCS '97, Santa Margherita Ligure, Italy, September 4-6, 1997, Proceedings*, volume 1290 of *Lecture Notes in Computer Science*, pages 261–280. Springer, 1997. doi:10.1007/BFB0026993.

13 Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, UAI'10, pages 340–349, Arlington, Virginia, USA, 2010. AUAI Press.

14 Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. Large-scale distributed graph computing systems: An experimental evaluation. *Proceedings of the VLDB Endowment*, 8, November 2014.

15 Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1807167.1807184.

16 Robert McCune, Tim Weninger, and Gregory Madey. Thinking like a vertex: a survey of vertex-centric frameworks for distributed graph processing. *ACM Computing Surveys*, 48, July 2015. doi:10.1145/2818185.

17 Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Scientific and Statistical Database Management*. Stanford InfoLab, July 2013. URL: http://ilpubs.stanford.edu:8090/1039/.

18 Philip Stutz, Abraham Bernstein, and William Cohen. Signal/collect: Graph algorithms for the (semantic) web. *Lecture Notes in Computer Science (LNCS)*, 6496:764–780, October 2010. doi:10.1007/978-3-642-17746-0_48.

19 Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990. doi:10.1145/79173.79181.

20 Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. Big graph analytics platforms. *Foundations and Trends in Databases*, 7:1–195, January 2017. doi:10.1561/1900000056.