Vertex-centric Parallel Computation of SQL Queries

Ainur Smagulova
UC San Diego
asmagulo@cs.ucsd.edu

Alin Deutsch
UC San Diego
deutsch@cs.ucsd.edu

ABSTRACT

We present a scheme for parallel execution of SQL queries on top of any vertex-centric BSP graph processing engine. The scheme comprises a graph encoding of relational instances and a vertex program specification of our algorithm called TAG-join, which matches the theoretical communication and computation complexity of state-of-the-art join algorithms. When run on top of the vertex-centric TigerGraph database engine on a single multi-core server, TAG-join exploits thread parallelism and is competitive with (and often outperforms) reference RDBMSs on the TPC benchmarks they are traditionally tuned for. In a distributed cluster, TAG-join outperforms the popular Spark SQL engine.

CCS CONCEPTS

• **Information systems → Graph-based database models**; **Join algorithms**.

KEYWORDS

vertex-centric graph processing; BSP parallel SQL evaluation

ACM Reference Format:
Ainur Smagulova and Alin Deutsch. 2021. Vertex-centric Parallel Computation of SQL Queries. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457314

1 INTRODUCTION

We study the evaluation of SQL join queries in a parallel model of computation that we show to be extremely well-suited for this task despite the fact that it was designed for a different purpose and it has not been previously employed in this setting. We are referring to the vertex-centric flavor [38] of Valiant's bulk-synchronous parallel (BSP) model of computation [48], originally designed for processing analytic tasks over data modeled as a graph.

Our solution comprises (i) a graph encoding of relational instances which we call the *Tuple-Attribute Graph (TAG)*, and (ii) an evaluation algorithm specified as a vertex-centric program running over TAG inputs. The evaluation is centered around a novel join algorithm we call *TAG-join*.

On the theoretical front, we show that TAG-join's communication and computation complexities are competitive with those of the best-known parallel join algorithms [10, 15, 16, 32, 33, 35] while avoiding the relation reshuffling these algorithms require (for re-sorting or re-hashing) between individual join operations. TAG-join adapts techniques from the best sequential join algorithms (based on worst-case optimal bounds [42, 43, 49] and on generalized hypertree decompositions [28, 29]), matching their computation complexity as well.

On the practical front, we note that our vertex-centric SQL evaluation scheme applies to both intra-server thread parallelism and to distributed cluster parallelism. The focus in this work is to tune and evaluate how our approach exploits thread parallelism in the *"comfort zone"* of RDBMSs: running the benchmarks they are traditionally tuned for, on a multi-threaded server with large RAM and SSD memory holding all working set data in warm runs.

We note that the benefit of recent developments in both parallel and sequential join technology has only been shown in settings beyond the RDBMS comfort zone. The parallel join algorithms target scenarios of clusters with numerous processors, while engines based on worst-case optimal algorithms tend to be outperformed [1] by commercial RDBMSs operating in their comfort zone [8, 9, 26, 40].

TAG-join proves particularly well suited to data warehousing scenarios (snowflake schemas, primary-foreign key joins). Our experiments show competitive performance on the TPC-H [1] and across-the-board dominance on the TPC-DS [2] benchmark.

In a secondary investigation, we also evaluate our TAG-join implementation's ability to exploit parallelism in a distributed cluster, showing that it outperforms the popular Spark SQL engine [12].

A bonus of our approach is its applicability on top of vertex-centric platforms without having to change their internals. There are many exemplars in circulation, including open-source [3, 6, 27, 36] and commercial [21, 38]. We chose the free version of the TigerGraph engine [7, 21] for our evaluation due to its high performance.

Our work uncovers a synergistic coupling between the TAG representation of relational databases and vertex-centric parallelism that went undiscovered so far because, despite abundant prior work on querying graphs on native relational backends [24, 34, 50, 54], there were no attempts to query relations on native graph backends.

Paper organization. After reviewing the vertex-centric BSP model in §2, we present the TAG encoding of relational instances (§3), then develop TAG-join starting from two-way (§4), to acyclic (§5) and to arbitrary join (§6). We discuss extensions beyond joins in §7, report experiments in §8 and conclude in §9.

2 VERTEX-CENTRIC BSP MODEL

The vertex-centric computational model was introduced by Google's Pregel [38] system as an adaptation to graph data of Valiant's Bulk Synchronous Parallel (BSP) model of computation [48].

SIGMOD '21, June 20–25, 2021, Virtual Event, China.

ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3457314

[1] Their benefit kicks in on queries where intermediate results are much larger than the input tables. This is not the case with the primary-foreign key joins that are prevalent in OLTP and OLAP workloads since the cardinality of $R \bowtie_{R.FK=S.PK} S$ is upper bounded by that of R (every R-tuple joins with at most one S-tuple).

A BSP model 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 model adapts the BSP model to graphs, such that 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 vertex program is designed from the perspective of a vertex. The vertex program operates on local data only: the vertex state, the received messages, and the incident edges.

At the beginning of a computation all vertices are in active state, and start the computation. At the end of the superstep each vertex deactivates itself, and it will stay inactive unless it receives messages. All messages sent during superstep i are available at the beginning of superstep $i + 1$. Vertices that did not receive any messages are not activated in superstep $i + 1$, and thus do not participate in the computation. The computation terminates when there are no active vertices, i.e. no messages were sent during the previous superstep. The output of the computation is the union of values computed by multiple vertices (distributed output).

2.0.1 Examples of vertex-centric engines. A vertex-centric BSP model was first introduced in Pregel [38], followed by a proliferation of open-source and commercial implementations, some running on distributed clusters, others realizing vertex communication via a shared memory. Surveys of the landscape can be found in [39, 51], while their comparative experimental evaluation has been reported in [11, 31, 37]. These works exclude the new arrival TigerGraph [21], which exploits both thread parallelism within a server and distributed cluster parallelism.

3 TAG ENCODING OF A RELATIONAL DB

We present the Tuple-Attribute Graph (TAG) data model we use to encode a relational database as a graph. A graph is a collection of vertices and edges, where each vertex and edge has a label and can store data as a collection of (key,value) pairs (i.e. attributes). The TAG model defines two classes of vertices: *tuple* vertices, representing tuples of a relation; and *attribute* vertices, representing attribute values of a tuple. Tuple and attribute vertices function the same in the vertex-centric computational model, i.e. both can execute a user-defined program and communicate via messages.

We construct a TAG graph from a relational database as follows.

(1) For every tuple t in relation R create a tuple vertex v_t labeled R (each duplicate occurrence of t receives its own fresh tuple vertex). Store t in v_t's state.

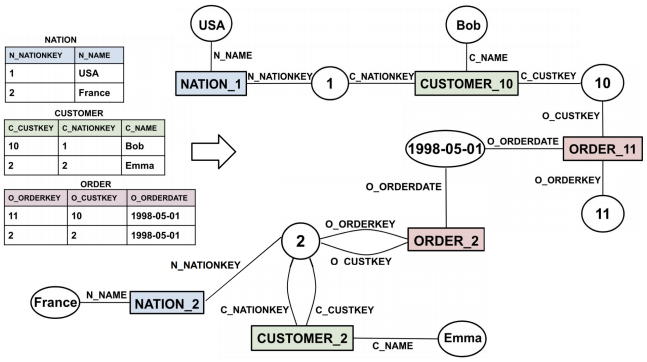

Figure 1: Encoding relational data in a TAG representation. Tuple vertices are depicted as rectangles, and attribute vertices as circles.

(2) For each attribute value a in the active domain of the database create an attribute vertex v_a. Add a label based on the domain/type of a (e.g. int, string, etc.). Create exactly one vertex per value regardless of how many times the value occurs in the database.
(3) For each occurrence of R-tuple t with an attribute named A of value a, add an edge labeled $R.A$ between v_t and v_a.

Notice that the graph is bipartite, as edges never connect tuple vertices with each other, nor attribute vertices with each other.

Example 3.1. *Figure 1 shows the example instance of relational data and its corresponding graph representation (to unclutter the figure, edge labels do not include the table names). We start with the first tuple of relation NATION, and map it to a tuple vertex with ID NATION_1 with corresponding label NATION. Then each of its attribute values maps to an attribute vertex, i.e. value 1 maps to an integer attribute vertex, and value USA maps to a string attribute vertex. We finish the transformation by creating edges from the tuple vertex to the two attribute vertices with labels that correspond to the attribute names of the tuple. We repeat the same steps for the rest of the tuples and for all relations. Tuple vertex CUSTOMER_10 also uses integer value 1 as its attribute, and thus we simply add an edge to connect it to this attribute vertex. Note how integer attribute vertex 2 is shared among three tuple vertices NATION_2, CUSTOMER_2, ORDER_2, and used as a value of five different attributes, hence the five edges with different labels are added. Two ORDER tuples are connected with each other via the date attribute value that they share.* □

The TAG representation of a relational database is query-independent and therefore can be computed offline. Moreover, the size of the graph is linear in the size of the relational database.

The most prominent feature of the TAG representation is that pairs of joining[2] tuples are explicitly connected with each other via edges to their join attribute value. For each attribute value a, the tuples that join through a can be found by simply following the outgoing edges from the attribute vertex representing a. Therefore, attribute vertices act as an indexing scheme for speeding up joins. This scheme features significant benefits over RDBMS indexing:

First, note that the TAG representation corresponds in the relational setting to indexing *all* attributes in the schema. While this is

[2]In this paper, the unqualified term "join" is shorthand for "equi-join".

prohibitively expensive in an RDBMS because of the duplication of information across indexes, notice that TAG attribute vertices are not duplicated. One can think of them as shared across indexes, in the sense that even if a value appears in an A and a B attribute, it is still represented only once (e.g. attribute vertex 2 in Figure 1).

Second, an attribute vertex can lookup the tuples joining through it in time linear in their number by simply following the appropriate edges. In contrast, even when an appropriate RDBMS index exists (which cannot be taken for granted), the RDBMS index lookup time depends, albeit only logarithmically, on the size of the involved input relations even if the lookup result size is small.

Third, attribute vertices are cheaper to build, and in the presence of changing data they are less challenging to maintain than traditional RDBMS indexes, as they do not require any reorganization of the graph. It suffices to locally insert/delete vertex attributes and their incident edges.

Finally, since the set of edges is disjointly partitioned by the attribute vertices they are incident on, the TAG model is particularly conducive to parallel join processing in which attribute vertices perform the tuple lookup in parallel. The vertex-centric BSP model of computation suggests itself as a natural candidate because it enables precisely such parallel computation across vertices and messaging along edges. In the remainder of the paper, we exploit this fact by developing a vertex-centric BSP join algorithm.

3.0.1 Related Encodings for Attribute-centric Indexing. Paper [47] addresses mapping of a relational instance to a property graph, connecting vertices based on key-foreign key relationships only. In contrast, the TAG encoding supports arbitrary equi-join conditions. Moreover, [47] focuses on the data modeling aspect only and does not address query evaluation (let alone the vertex-centric kind).

TAG encoding's attribute vertices generalize the value-driven indexing of [25] from RDF triples to arbitrary tuples. Their indexing role is also related in spirit to indexing Nested Relational data [20]. Both works propose secondary indexing structures, with the requisite space and time overhead for creation and maintenance. These are avoided in TAG encoding, where attribute vertices are not redundant indexes, but the original data. Moreover, neither of [20, 25] considers parallel evaluation of joins in general, and in particular the vertex-centric computational model.

4 VERTEX-CENTRIC TWO-WAY JOIN

We begin with computing a natural join between two binary relations: $R(A, B) \bowtie S(B, C)$, using a vertex-centric algorithm over their TAG representation. For presentation simplicity we consider natural join queries as examples throughout the paper, however note that any equi-join query can be transformed into a natural join by appropriate renaming of attributes. Furthermore, the generalization to n-ary relations is straightforward.

Our approach is based on Yannakakis' algorithm for acyclic queries [53], that first performs a semi-join reduction on all relations to eliminate dangling tuples, and then joins the relations in arbitrary order. The semi-join is an essential query processing technique that can reduce the cost of a join query evaluation, especially in a distributed environment [18].

The semi-join of $R(A, B)$ with $S(B, C)$, denoted $R \ltimes S$, retrieves precisely those R-tuples that join with at least one S-tuple. To implement a reduction of R, it suffices to obtain a duplicate-free projection of the join column $S.B$: $R \ltimes S = R \ltimes \pi_B(S)$.

Per Yannakakis's algorithm, in order to compute a two-way join we first reduce the sizes of R and S via a series of semi-joins, $J_1 := R \ltimes S$ and $J_2 := S \ltimes R$. Now that the tuples that do not contribute to the output (a.k.a. 'dangling tuples') are removed, the algorithm constructs the join of the reduced relations, $J_1 \bowtie J_2$.

Our algorithm follows a similar idea of splitting the computation into: (1) a *reduction* phase to eliminate vertices and edges such that the surviving ones correspond to the TAG representation of the reduced relations; and (2) a *collection* phase that traverses the reduced TAG subgraph to construct the final join output.

4.1 Join on a Single Attribute

By design of the TAG model, the semi-join reduction can be naturally mapped to a vertex-centric computational model. Each value b of an attribute B is mapped to a TAG attribute vertex v_b whose outgoing edges connect it to precisely the (vertex representations of) R- and S-tuples that join through v_b. Thus, for a reduction to be performed, each attribute vertex needs to check its outgoing edges, confirming that it connects to at least one tuple vertex of each label R and S, and then signal those tuple vertices that they are part of the final output by sending a message. Note that all attribute vertices can do this in parallel, independently of each other, since their individual computation and messaging only requires access to the locally stored vertex data. Also note that the attribute vertex need not 'cross the edge' to inspect the proper labeling of the tuple vertex at the other end: this information can be encoded in the edge label itself, by qualifying the attribute name with the relation name it belongs to (see Figure 2).

4.1.1 Algorithm. We sketch a vertex-centric two-way join algorithm through the following example. Consider the TAG instance depicted in Figure 2. The computation starts by activating attribute vertices corresponding to the join attribute B.

Superstep 1: Each attribute vertex checks whether it can serve as *join value*, i.e. a value in the intersection of columns R.B and S.B. For a vertex v to be a join value it needs to have outgoing edges with labels R.B and S.B. If so, v messages the target tuple vertices via these edges. Otherwise, v just deactivates itself. As depicted on Figure 2(a), vertex $b1$ figures out that it is a join value, and sends its ID to the target tuple vertices (3 R vertices and 3 S vertices). Vertices b_2 and b_3 deactivate themselves without sending any message.

Superstep 2: Tuple vertices are activated by the incoming messages. Each tuple vertex t marks the edges along which it has received messages. Then t sends its non-join attribute value back to the join attribute vertex via the marked edges, and deactivates itself. In Figure 2(b), the R tuple vertices $R1, R2, R3$ send their A values and the S tuple vertices $S1, S2, S3$ send their C values to vertex $b1$ via marked edges (shown in bold).

Superstep 3: Each active attribute vertex v constructs a join result by combining the values received from both sides with their own value (the operation is really a Cartesian product). Next, v stores this result locally (or possibly outputs it to the client application). The computation completes when all vertices are deactivated. See Figure 2(c), which shows the locally stored join result.

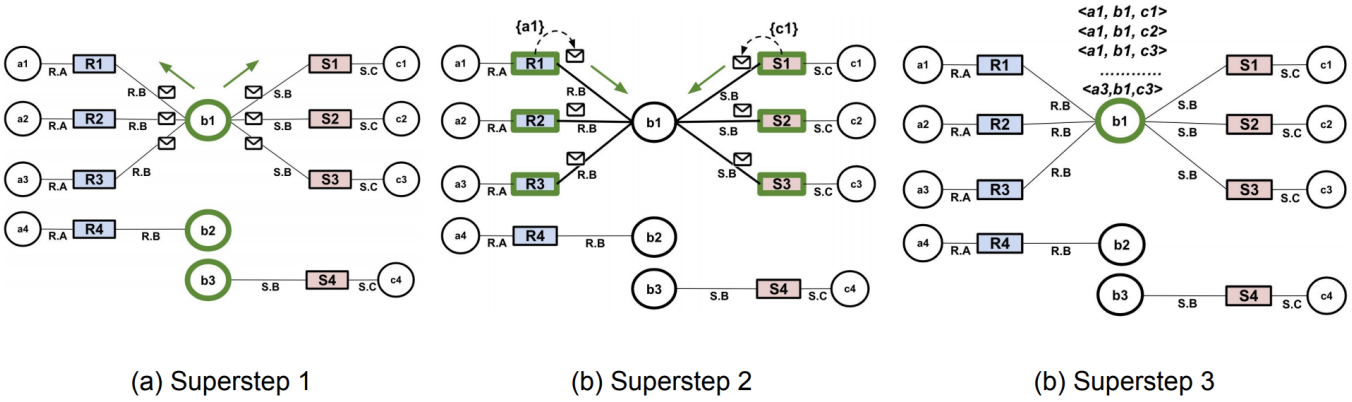

Figure 2: Example two-way join of relations R and S. Borders of active vertices are highlighted in lighter shade (green).

Superstep (1) is a reduction phase that eliminates tuple vertices with no contribution to the join, while (2) and (3) are part of a collection phase whose purpose is to collect, via messages, data from R-tuples and S-tuples that join and construct the output tuples.

In Superstep (3), the A-attribute and C-attribute values received at a B-attribute vertex correspond to the *factorized representation* of the join result [44], i.e. the latter can be obtained losslessly as their Cartesian product. If the distributed and factorized representation of the join is required, then Superstep (3) is skipped.

4.1.2 Cost Analysis. Let $|R|$ and $|S|$ denote the sizes (in tuple count) of R and S respectively. Then the total input size $IN = |R| + |S|$. Let $OUT = |R \bowtie S|$ denote the size of the join result in the standard unfactorized bag-of-tuples representation.

In superstep (1), all B-attribute vertices are active (their number is upper bounded by IN). They send messages along the edges with labels R.B and S.B, but only when the recipient tuples contribute to the join. Since the value of the join attribute disjointly partitions R, S, and also $R \bowtie S$, the total message count over all attribute vertices is $|R \ltimes S| + |S \ltimes R|$, which is upper bounded by both IN and OUT, and therefore by $min(IN, OUT)$. Even for the worst-case instance, where an attribute vertex is connected to all R and S tuple vertices, the communication cost does not exceed $min(IN, OUT)$. During the computation phase each attribute vertex with label B (e.g. $b1, b2, b3$) iterates over its outgoing edges in order to figure out whether it joins tuples from both R and S. The total computation cost summed up over all vertices is upper bounded by the total number of edges and therefore by IN.

In Superstep (2), only tuple vertices that contribute to the final output send messages, for a total number of messages $|R \ltimes S|$ (sent by R-labeled tuple vertices) + $|S \ltimes R|$ (sent by S-labeled tuple vertices). This is again upper bounded by $min(IN, OUT)$. Since the computation at tuple vertices is constant-time, the overall computation in this superstep is also upper bounded by $min(IN, OUT)$.

In Superstep (3) each join vertex combines the received messages to construct output tuples. Each attribute vertex of value b receives the A attribute values from its R-tuple neighbors, the C attribute values from its S-tuple neighbors, and computes $\sigma_{B=b}(R \bowtie S)$ locally as a Cartesian product. The total computation cost across all active vertices is OUT because the output tuple sets are disjoint across join vertices. If the output is left distributed over the vertices (the standard convention in distributed join algorithms [10, 16] is to leave the result distributed over processors, which in our setting are the vertices) no further messages are sent and the communication cost is 0. Even if the output is instead sent to a client application, the additional communication cost totalled over all vertices is OUT.

In summary, if we desire the collection of the join output in unfactorized representation, we require the total communication $O(OUT + min(IN, OUT))$ and the total computation in $O(IN + OUT + min(IN, OUT)) \subseteq O(IN + OUT)$. To leave the unfactorized join output distributed across vertices requires communication cost in $O(min(IN, OUT)$ and computation cost in $O(IN + OUT + min(IN, OUT)) \subseteq O(IN + OUT)$. To collect the factorized join output, the algorithm requires communication cost in $O(min(IN, OUT))$ and computation cost in $O(IN+min(IN, OUT)) \subseteq O(IN)$. Finally, leaving the factorized join output distributed across vertices requires communication cost in $O(min(IN, OUT))$ and computation cost in $O(IN + min(IN, OUT)) \subseteq O(IN)$.

A lax upper bound that covers all cases is therefore $O(IN+OUT)$ for both computation and communication cost.

Dependence on the number of hardware processors. The vertex-centric model of computation makes the conceptual assumption that each vertex is a processor. This is of course just an abstraction as the vertex processors are virtual, several of them being simulated by the same hardware processor in practice. Our complexity analysis is carried out on the abstract model, hence it overestimates the communication cost by counting each message as inter-processor when a large number of messages are actually intra-processor and do not tax the bandwidth of the interconnect.

4.1.3 Comparison to other algorithms. Regarding both total communication and computation, our algorithm has the same lax upper bound $O(IN + OUT)$ as the computation upper bound of the classical sequential Yannakakis' algorithm [53], with the advantage of parallelism due to the vertex-centric nature.

State-of-the-art parallel join algorithms are mainly based on two techniques: hashing and sorting. These algorithms are usually based on the MPC [16] and Map-Reduce [10] computational models in a cluster setting.

The parallel hash-join algorithm [16] is the most common approach used in practice, where tuples are distributed by hashing on the join attribute value. It achieves the same total communication complexity of $O(IN+OUT)$, assuming the unfactorized result from each processor is collected in a centralized location. In the scenario of a distributed, factorized representation of the join result, parallel hash-join requires communication $O(IN)$ while our vertex-centric join requires communication in $O(min(IN, OUT))$, which is better when the join is selective (the computation cost is the same, $O(IN)$).

The parallel sort-join algorithm [32] is an MPC-based algorithm designed to handle arbitrarily skewed data, and measures the communication complexity per processor. We do not consider processor load balance in the scope of this paper, leaving it for future work. For apples-to-apples comparison, we derive from [32] the total communication cost as $O(IN+\sqrt{p \cdot OUT}+OUT)$, where p is the number of processors and the last term describes the cost of streaming the final output to a centralized location. A skew resilient generalized version of a parallel hash join is also presented in [15, 35], and achieves the same total communication cost as parallel sort-join. Depending on the size of the output and how skewed the input is, parallel sort-join will require more total communication. Otherwise (skew-free input), it achieves the same total communication cost as our vertex-centric algorithm. The parallel sort-join requires the input to be sorted on the join attribute via a reshuffling phase (which we do not require). Such sorting incurs a communication cost of $O(IN)$, but requires additional supersteps, degrading the parallel sort-join's performance on ad-hoc queries that join on different attributes and in scenarios where the data is not read-only.

In summary, using the vertex-centric BSP model we can compute a single-attribute two-way join over the TAG representation of the input relations matching the communication complexity of the best-known parallel algorithms (and even improving on it for the distributed factorized output scenario), while saving the query-dependent reshuffling they each require (for hashing or sorting).

4.2 Join on Multiple Attributes

A necessary building block to generalize to arbitrary queries is to consider join conditions on multiple attributes. We reuse the algorithm described in §4.1 with a small adjustment in the reduction phase. We reduce a join on two attributes X_1 and X_2 to a join on a single attribute by having tuple attributes send their value of the X_2 attribute to the X_1-attribute vertices. Each X_1-attribute vertex intersects these values and messages back only to the tuple vertices whose X_2 value is in the intersection. The two-attribute join generalizes to a multi-attribute join on $X_1, X_2, \ldots, X_n$, by sending a message with $X_2, \ldots, X_n$ attribute values to the coordinating X_1-attribute vertex. See the extended version [46] for the algorithm details, examples and cost analysis. The communication complexity remains in $O(IN + OUT)$.

5 ACYCLIC MULTI-WAY JOINS

We extend the two-way join algorithm to multi-way joins, as long as they are acyclic. For presentation simplicity, our treatment is confined to single-attribute joins, with the understanding that these can be generalized to multi-attribute joins as described in §4.2.

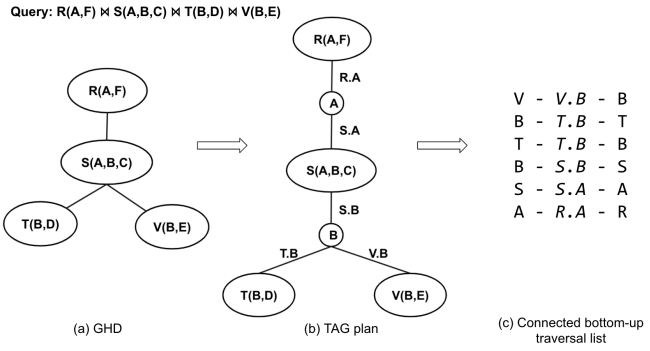

Figure 3: Example of a query plan translation

5.1 TAG Traversal Plan

We use a generalized hypertree decomposition (GHD) of the join query as a basis to obtain a TAG traversal plan (TAG plan), which in turn will help drive the vertex program.

We refresh the notion of GHD, referring to [28, 29] for a more detailed treatment. A *GHD* is a tree decomposition of a query, such that each node of the tree (referred to as a 'bag') is assigned a set of attributes $\mathcal{A}$ and a set of relation names $\mathcal{R}$. For every bag, the schema of each $R \in \mathcal{R}$ is included in $\mathcal{A}$. Moreover, every relation mentioned in the query is assigned to some bag. Finally, for any attribute A, all bags A occurs in form a connected subtree. A GHD where each bag is labeled by a single relation is called a *join tree*. Recall that a query is *acyclic* if and only if it has a join tree [17].

TAG plan. The TAG plan is itself a tree structure, constructed as follows from the join tree:

(1) For each bag, create a node labeled with the same relation.
(2) For each join attribute A (A occurs in at least two bags), create a node labeled A if it does not exist already.
(3) For each join attribute A, let $\mathcal{B}_A$ denote the set of all bags containing A. For each bag $b \in \mathcal{B}_A$, denote with R_b the relation labeling b. Add an edge that connects the TAG plan node corresponding to A with the TAG plan node corresponding to b. Label the edge with $R_b.A$.

We call the elements of the TAG plan *nodes*, to avoid confusion with elements of the TAG representation, which we call *vertices*.

Example 5.1. *Figure 3(b) shows the TAG plan constructed from the join tree in Figure 3(a).* □

Note that the example query only computes joins, thus it suffices to create plan nodes only for the join attributes. However, depending on the input query other attributes may be necessary for the computation (e.g. *GROUP BY* attributes and attributes occurring in the *WHERE* clause). In such cases we create plan nodes for these attributes as well.

We next translate a TAG plan into a list L of edge labels. Intuitively, L is used to drive a vertex-centric program as follows: at superstep i, the active vertices send messages along their outgoing edges labeled $L(i)$. We detail the list-driven vertex-centric program in Algorithm 1 below. First we explain how the list is obtained.

Connected Bottom-up Traversal. The list corresponds to what we call a *connected bottom-up traversal* of the TAG plan. The traversal is bottom-up in that it starts from the rightmost leaf and makes its way towards the root, eventually visiting the entire plan, while first visiting all subtrees of a node n before moving to n's parent. Moreover, the traversal is *connected* in the sense that each traversal step must start from the node reached in the previous step. It should also be noted that reversing the list corresponds to a top-down (preorder) traversal of the TAG plan.

Example 5.2. *See Figure 3 (c) for the list of labels obtained from the TAG plan in Figure 3 (b). Notice that it corresponds to the connected bottom-up traversal of the plan starting from the rightmost leaf (labeled V). For convenience, we show to the left and right of each edge label the source, respectively destination of the traversal step.* □

5.2 Vertex-Centric Algorithm

Vertex Program. The join algorithm performs a vertex-centric analogy to Yannakakis' semijoin reduction technique in the following sense. All vertices execute in parallel a vertex program comprising two phases: an initial reduction phase followed by a collection phase. The role of the reduction phase is to mark precisely the edges that connect the tuple and attribute vertices that contribute to the join. The collection phase then traverses the marked subgraph to collect the actual join result.

Before detailing the logic implemented by the program, we note that its structure conforms to that of the classical vertex-centric BSP program. Each iteration of the while loop starting at line 5 in Algorithm 1 implements a superstep. In each superstep, all active vertices compute in parallel, carrying out a computation and a communication stage. In the computation stage, each vertex processes its incoming messages (lines 8-9). In the communication stage, each vertex sends messages to its neighbors via the edges whose label is dictated by variable *currentLabel* (lines 11-13, 15-18). At the end of the superstep, vertices wait at a synchronization barrier for all messages to be received (line 24). Only message recipients are activated for the next superstep (line 25). Once all vertices finish processing the current traversal step, the edge label indicating the next traversal step is popped from the stack (line 5) and a new superstep is carried out. The reason why the input *labels* must correspond to a connected traversal becomes apparent now: the vertex-centric model requires each superstep to be carried out by the vertices activated by the previous superstep.

Reduction Phase. The vertex program takes as input *startLabel*, the label of the rightmost leaf in the TAG plan T, and a list of edge labels (given in the stack *labels*), corresponding to the connected bottom-up traversal of T. The program starts by activating the tuple vertices labeled with *startLabel* (line 1). Next, it iteratively performs supersteps, one for every edge label in the stack.

The intuition behind the reduction phase is the following. Recall that our TAG graph is bipartite: it consists of two kinds of vertices (attribute and tuple) and edges always go between the two vertex kinds. Therefore, the active vertex set alternates between containing exclusively attribute vertices and exclusively tuple vertices. At every step, the active vertex set can be regarded as a distributed relation (with the set of attribute vertices corresponding to a single-column table). A superstep that starts from a set S

Algorithm 1: Vertex Program

```
   Input: startLabel: a vertex label
   Input: labels: a stack of edge labels
1  ActiveVertexSet ← {all vertices labeled startLabel};
2  direction ← UP ; // bottom-up traversal first
3  Reduction Phase:
4  while labels not empty do
5      currentLabel ← labels.pop();
6      reverseOrder.push(currentLabel);
7      foreach v ∈ ActiveVertexSet do in parallel
8          for each id ∈ v.incomingMsgQueue do
9              insert id into v.markedEdges;
10         if direction = UP then
11             for each e ∈ outEdges(v) do
12                 if e.label = currentLabel then
13                     send message v.id to e's target;
14         if direction = DOWN then
15             for each e ∈ outEdges(v) do
16                 t ← target of e;
17                 if e.label = currentLabel AND
                    t ∈ v.markedEdges then
18                     send message v.id to t;
19                     update v.markedEdges;
20     if labels is empty AND direction = UP then
21         labels ← reverseOrder;
22         reverseOrder.clear();
23         direction ← DOWN;
       // synchronization barrier
24     wait for all vertices to receive their messages;
25     ActiveVertexSet ← {all message recipients};
26 Collection Phase:
27 labels ← reverseOrder // bottom-up order again
28 while labels not empty do
29     currentLabel ← labels.pop();
30     foreach v ∈ ActiveVertexSet do in parallel
31         v.value ← join all tables in v.incomingMsgQueue;
32         if v is a tuple vertex then
33             if this is first superstep in collection phase then
34                 v.value ← {v.data};
35             else
36                 v.value ← v.value ⋈ {v.data};
37         for each e ∈ outEdges(v) do
38             t ← target of e;
39             if e.label = currentLabel AND
                t ∈ v.markedEdges then
40                 send message v.value to t;
41         if labels not empty then
42             Output v.value ; // computation is done
       // synchronization barrier
43     wait for all vertices to receive their messages;
44     ActiveVertexSet ← {all message recipients};
```

of active tuple vertices labeled R and activates next the attribute vertices reachable via edges labeled $R.A$ corresponds to taking the duplicate-eliminating projection on A of the R-tuples in S, $\pi_A(S)$, in the sense that the values in this column correspond precisely to the newly activated attribute vertices. Conversely, a superstep that starts from set S of active attribute vertices and next activates their neighbors reachable via edges labeled $R.A$ activates precisely the tuple vertices corresponding to the semijoin of table R with an A column of values from S, $R \ltimes \pi_A(S)$. The order in which the supersteps are performed by the reduction phase leads to a sequence of column projection and semijoins operations that corresponds to a Yannakakis-style reducer program.

The reduction phase starts with a connected bottom-up pass due to the *direction* variable being initialized to UP (line 2). At each superstep, vertices send their ID to their neighbors along edges labeled by the current step (lines 12-13). Each vertex v identifies its incoming join-relevant edges e by their source ID (received as the message payload). Vertex v marks e in its local state $v.markedEdges$ (line 9). The effect of the reduction pass is formalized by Lemma 5.1.

Lemma 5.1. *Consider a list L of edge labels and the execution of a vertex program driven by L. Let $T.A$ be the label at position i in the concatenation of L with its reverse. Denote with R_i the distributed relation corresponding to the vertex set activated by superstep i. If i is odd, then $R_i = \pi_A(R_{i-1})$. If i is even, then $R_i = T \ltimes R_{i-1}$.* □

Example 5.3. *Recall the edge label list shown in Figure 3(c). With the notation from Lemma 5.1, the list induces the following sequence of operations: $R_0 := V, R_1 := \pi_B(R_0), R_2 := T \ltimes R1, R3 := \pi_B(R_2), R_4 := S \ltimes R_3, R_5 := \pi_A(R_4), R_6 := R \ltimes R_5$. Notice that this sequence is a full reducer for table R, i.e. the last relation computed, R_6, contains precisely the R-tuples that participate in the multi-way join specified by the join tree in Figure 3(a).* □

As usual in full reducer programs, the bottom-up pass only reduces fully the root relation of the join tree (in Example 5.3 that would be table R, whose full reduction is computed as R_6). The other relations do not yet reflect the reduction of their ancestors in the join tree. To remedy this, one needs to apply further semijoin reduction operations in an order given by a top-down traversal of the join tree. In our vertex-centric adaptation, the vertex program switches to top-down mode (lines 20-23). It uses the *reverseOrder* stack to reverse the order of input steps, thus obtaining a top-down order (line 6 and 21). The resulting top-down pass continues to apply reduction steps just like the bottom-up pass.

The only difference between the top-down and the bottom-up reduction is that when sending messages, the former does not only check the edge label but it also makes sure to only signal via those edges that have been marked during the bottom-up pass (line 17). This ensures that the top-down pass never visits vertices that were already reduced away by the bottom-up pass. The set of marked edges is updated further during the top-down pass to only include edges that are on the join result path (line 19). Marked edges then guide the collection phase (line 39), ensuring that it visits only the marked subgraph corresponding to the fully reduced relations.

Example 5.4. *Continuing Example 5.3, the vertex program switches into the DOWN pass, being driven by the reversed edge label list $R.A, S.A, S.B, T.B, T.B, V.B$, which determines the sequence of operations $R_7 := \pi_A(R_6), R_8 := S \ltimes R_7, R_9 := \pi_B(R_8), R_{10} := T \ltimes R_9, R_{11} := \pi_B(R_{10}), R_{12} := V \ltimes R_{11}$ which fully reduces all tables.* □

Collection Phase. For the collection phase we reverse the order of steps once more, obtaining a bottom-up pass that starts from the vertices activated last by the reduction phase. During this phase, messages hold tables that correspond to intermediate results of the desired join. In the computation stage, each vertex v computes a value $v.value$ that corresponds to the join of the tables it receives via messages (line 31)[3]. If the vertex is a tuple vertex, it also joins the computed value with the tuple it represents. This tuple is stored in the $data$ attribute (lines 32-36). In the communication stage, vertices propagate the computed value further up via marked edges (lines 37-40). When the root of a plan is reached, the computation completes and each active vertex outputs its computed value (line 42). The join result is the union of values output by vertices.

5.2.1 Cost Analysis. We are interested in the *data complexity* of our algorithm, which treats the query size as a constant. The computation of a vertex in the reduction phase involves iterating over its incoming message queue and performing constant-time computation on each message. Each active vertex sends messages only via its outgoing edges, so at each superstep the total number of messages is bounded by the number of out-edges in the graph, while the size of each message is constant (we treat the size of vertex IDs as fixed by the architecture, therefore the message size is fixed). Since the size of the TAG graph is linear in the size of the input database, the total computation and communication of each superstep of the reduction phase is $O(IN)$. Note that the number of supersteps is independent of the data, being linear in the query size. Therefore, the total computation and communication complexity of the reduction phase remain $O(IN)$.

The collection phase involves the traversal of the reduced subgraph, which has size linear in the size OUT of the join result. Each vertex constructs tuples by joining its own data with the intermediate results received. These tuples are sent as messages along edges. There is no redundant tuple construction within and across vertices, so the overall computation and communication is in $O(OUT)$.

Combining the complexity of the reduction and collection stages, we obtain $O(IN + OUT)$ for the overall communication and computation costs of our vertex-centric acyclic join.

5.2.2 Comparison to other algorithms. Reference [35] describes a hash-based parallel version of Yannakakis' algorithm in a distributed setting (the MPC model) with the same communication complexity of $O(IN + OUT)$. The main difference from our algorithm is that data needs to be reshuffled (re-distributed among processors) in a query-dependent way for each join operation. In our vertex-centric join, the graph representation of the input database is never reshuffled, regardless of the query.

[3]This value is a joined partial table and can be size-biased, its construction potentially creating load imbalance across vertices. This effect can be mitigated by keeping join results in factorized representation as long as possible (as discussed in our complexity analyses), but a full solution involves load balancing. Since we are targeting solutions on top of vertex-centric engines, we implicitly inherit their load balancing scheme and do not attempt to control it in this work. We believe that our encouraging experimental results render our approach interesting even before incorporating customized load balancing for vertex-centric SQL evaluation, which we leave for future work.

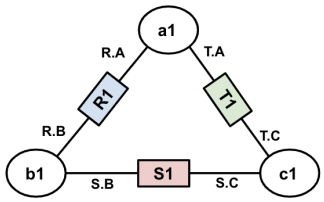

Figure 4: TAG instance for a triangle query in Example 6.1

The generalization of parallel sort-join algorithm to any acyclic join is presented in [33], and has a total communication cost of $O(IN + \sqrt{IN \cdot OUT})$. This outperforms our algorithm and parallel hash-join when the join output blows up to be lager than the input, but it is worse for selective joins and it is equivalent when the query involves only PK-FK joins.

Both parallel sort-join and parallel hash-join rely on query-dependent reshuffling (re-sorting and re-hashing) of the input, which again impairs the applicability to ad-hoc queries that join on different attributes and to scenarios where the data is not read-only.

6 ARBITRARY (CYCLIC) JOINS

In this section, we extend our algorithm to arbitrary (cyclic) joins. We begin with the famous triangle query in §6.1 and we extend the algorithm to arbitrary joins in §6.2.

6.1 Triangle Query

We begin with the triangle query $R(A,B) \bowtie S(B,C) \bowtie T(C,A)$. First, we describe a first-cut vertex-centric triangle algorithm and show that it is optimal for a class of queries with primary-foreign key (PK-FK) join conditions. We then improve upon it to achieve the complexity proportional to the worst-case output size defined by the renowned AGM bound [13]. This is a tight bound that estimates query output size based on input relations cardinalities and structural property of a query called fractional edge cover (ρ^*). For a full review of AGM and fractional edge cover we refer to [13, 30]. The AGM bound for triangle query is $O(IN^{\frac{3}{2}})$, while the traditional RDBMS plans with binary joins can run in time $O(IN^2)$ on some instances. The AGM bound has led to the development of the class of worst-case optimal (centralized, sequential) join algorithms such as NPRR [42], Leapfrog Triejoin [49] and Generic-Join [43]. Our parallel vertex-centric algorithm's communication and computation complexities match the worst-case computational upper bound of these algorithms for triangle queries.

6.1.1 Triangle Query for PK-FK Joins. The main idea of the vertex-centric triangle algorithm is to start the computation from A-attribute vertices, and send their ID values in both directions via paths that lead to C-attribute vertices such that if a cycle exists the values travelling from both sides meet at the final destination. Note that values propagated through the left side have a longer path to cross and pass through B-attribute vertices. Any order of the traversal can be chosen, e.g. start from B-attribute vertices and propagate B values in both directions to A-attribute vertices.

Example 6.1. *We illustrate the algorithm using the TAG instance in Figure 4. Attribute vertex $a1$ sends its ID via edges R.A (left side traversal) and T.A (right side traversal). Note that if vertex $a1$ has no incident R.A- or T.A-edge, it deactivates itself. On the right side the value $a1$ travels along the edge T.C to arrive at attribute vertex $c1$. On the left side the value $a1$ arrives at the $c1$ vertex via the edge list R.B, S.B, S.C. The $c1$ vertex intersects messages received from the left side with the messages received from the right side, as described in §4.2 (empty intersection indicates that a vertex is not part of any triangle). Since the result of the intersection is $a1$, vertex $c1$ continues into the collection phase. To construct the output, $c1$ sends messages back following the trail of the $a1$ value to activate tuple vertices $S1$ and $T1$. Those tuples then send their values to $c1$, which then combines them and outputs a triangle $\{(a1, b1, c1)\}$.* □

PK-FK Optimality. In the triangle query when a value reaches an attribute vertex it needs to be propagated further via its outgoing edges, i.e. every message gets replicated a number of times. The replication rate is defined by the number of outgoing edges which is linear in the size of the input IN. The number of messages that an attribute vertex can receive is also upper bounded by IN. In the worst case the number of messages that need to be sent can blow up to $O(IN^2)$, which exceeds the worst-case AGM bound. However, the replication rate is not an issue with PK-FK joins, since the size of the PK-FK join result cannot exceed the size of the foreign key relation, which is at most IN. Assume the primary key of each input relation is its first attribute, e.g attribute A is the primary key of relation R. The number of A values that $b1$ can receive is at most $|R|$, but since $b1$ itself is a primary key value of S tuple it can have at most one outgoing edge with label $S.B$. Thus, the number of messages sent by $b1$ is at most $|R|$. The same analysis applies to the $c1$ vertex.

If all joins in the query are PK-FK joins, the triangle query can be evaluated in the vertex-centric model with optimal communication and computation cost $O(IN + OUT)$, just like acyclic joins.

6.1.2 Worst-case Optimal Triangle Query Algorithm. In order to match the worst-case optimal guarantees by keeping the complexity within the AGM bound we improve upon the algorithm above. We employ the strategy of the NPRR algorithm. NPRR splits the values of attribute A in relation R into heavy and light. A value a is heavy if it occurs more times than a defined threshold value θ in relation R. Specifically, if $|\sigma_{A=a}R| > \theta$ then (a,b) $\in R_{heavy}$, otherwise (a,b) $\in R_{light}$. As a result, the original triangle query can be decomposed as follows [41]:

$$(R_{heavy} \bowtie S) \ltimes T)) \cup ((R_{light} \bowtie T) \ltimes S) \quad (1)$$

We solve the triangle query separately for heavy and light cases and then union the results. We apply the vertex-centric triangle algorithm as described above, for simplicity we refer to it as vanilla triangle. The triangle algorithm proceeds as follows:

(1) **Initialization:** Activate R-tuple vertices and navigate to A-attribute vertices via edge $R.A$. Each A-attribute vertex checks whether it's heavy or light.
(2) **Heavy** A-attribute vertices execute vanilla triangle algorithm (as described in Example 6.1).
(3) **Light** A-attribute vertices send "wake-up" messages to B-attribute vertices via their R-tuple vertex. Activated B-attribute vertices then execute vanilla triangle algorithm to propagate their ID values to C-attribute vertices.

Note that the number of outgoing edges with label $R.A$ indicate how many tuples in R contain the current attribute vertex value, hence it becomes trivial for each A-attribute vertex to check whether it's heavy or light. The threshold value θ helps to bound the number of messages and not to exceed the AGM bound. In the heavy case, the replication of messages happens when B-attribute vertices send received heavy a values via $S.B$-edges. This corresponds to the term $R_{heavy} \bowtie S$ in equation (1). The number of messages that can be received by B-attribute vertices is upper bounded by the total number of heavy a values in the R relation, which is at most $\frac{|R|}{\theta}$. Each message is sent via outgoing edge $S.B$. Since the total number of $S.B$ edges is $|S|$, the communication cost is $\frac{|R|}{\theta} \cdot |S|$ messages. In the light case, the worst replication happens when A-attribute vertices send b values via edge $T.A$, i.e. term $R_{light} \bowtie T$ in equation (1). The number of b values that A-attribute vertices can receive is at most θ, since all of these b vertex values are connected to light a vertices. Each b value is sent via at most $|T|$ edges. This results in $\theta \cdot |T|$ total messages for the light case computation. Each vertex' computation is linear to the received message count. Given $\theta = \sqrt{IN}$, where IN defines the sizes of input relations, the complexity of the reduction phase is proportional to the AGM bound, i.e. $O(IN^{\frac{3}{2}})$. Taking into account that the size of the actual output is upper bounded by the AGM, the overall communication and computation cost of our triangle algorithm in a vertex-centric BSP model is $O(IN^{\frac{3}{2}})$.

Besides being sequential algorithms, both NPRR and Leapfrog Triejoin require expensive precomputation to build index structures for each input relation based on some attribute order. Our parallel triangle algorithm does not rely on any additional index structures to achieve the worst-case optimal guarantees.

6.2 Arbitrary Equi-Join Queries

We extend our algorithm to arbitrary equi-joins by generalizing from triangle to n-cycles, then combining the acyclic and cyclic join strategies. We relegate details to [46].

7 BEYOND EQUI-JOINS

TAG-join is compatible with the efficient evaluation of other algebraic operations. Small edits to the vertex program (Algorithm 1) allow the seamless interleaving of join with other algebraic operations, supporting classical optimizations such as pushing selections, projections and aggregations before the join.

Selection. Pushing selections before joins translates in our setting to vertices checking the selection condition as early as possible. Conditions involving a single attribute are checked in parallel by the corresponding attribute vertices during the reduction phase (by adding the selection to line 12 in Algorithm 1). Attribute vertices that fail the selection deactivate themselves, reducing overall computation and communication. Conditions involving multiple attributes are also applied in parallel by attribute vertices but need to wait for the earliest round of the collection phase where collected intermediate tuples contain the relevant attributes. This is achieved by adding the selection to line 31 in Algorithm 1, together with a check of the current label (set in line 21) to identify the round.

Projection. Pushing projections early is beneficial for the collection phase. Although it does not reduce the number of sent messages, it affects their size by reducing the number of attributes of the comprised tuples. Projection pushing is implemented by application to the local joins in lines 34 and 36 of Algorithm 1. Which columns can be projected away depends on the round, which in turn is given by the current label (set in line 21).

Aggregation. The aggregation scheme is inspired by aggregation over hypertree decompositions in factorized databases [14, 44], since our TAG plan is based on a GHD (recall §5.1). The scheme includes the classic optimization technique of pushing grouping and aggregation before the join [52]. Aggregates are computed during the collection phase as soon as the intermediate tuples contain the relevant attributes, by using a modification to line 31 in Algorithm 1 and by checking the current label to identify the aggregation rounds. We distinguish three types of aggregation:

- Local aggregation (LA) corresponds to SQL queries with GROUP BY on one attribute, or multiple attributes where one attribute functionally determines the others.
- Scalar aggregation computes a single tuple of scalar values. Vertices need to send their computed aggregates to a global 'aggregation' vertex whose ID is known to all.
- Global aggregation (GA) uses a multi-attribute GROUP BY clause, such that the attributes do not determine each other. This also requires vertices to communicate with a global 'aggregation' vertex to output the final aggregation.

Local aggregation benefits the most from vertex-centric computation: aggregation within each group is computable by the attribute vertex representing the group key, in parallel to the other groups.

Outer Joins. These can be naturally computed as a small tweak of TAG-join algorithm (details in the extended version [46]).

Correlated Subqueries. We employ a navigational execution strategy with a forward lookup [23] used in traditional RDBMSs, i.e. start with the outer query, and in a tuple-at-time fashion invoke the nested subquery. Our vertex-centric approach naturally supports parallelization: tuple vertices matching the outer query are activated in parallel and their subquery calls execute in parallel.

8 EXPERIMENTS

Our vertex-centric SQL evaluation scheme applies to both intra-server thread-based and to distributed cluster parallelism.

The bulk of our experiments (§8.2,§8.3, §8.4, §8.5) evaluates how our approach enables thread parallelism in the *comfort zone* of high-end RDBMSs: running the benchmarks they are traditionally tuned for, on a multi-threaded server with large RAM and SSD memory holding all working set data in warm runs.

We also carry out preliminary experiments evaluating the ability to exploit parallelism in a distributed cluster, where we compare our approach against Spark SQL (§8.6).

We detail our performance comparisons below but first we summarize the experimental results.

Table 1 shows the aggregate runtimes (i.e. summed over all queries) for the TPC-H and TPC-DS query workloads in single-server mode. For each benchmark we performed three sets of experiments with varying data sizes. In aggregate, TAG-join outperforms all relational systems (5x-30x speedup) on TPC-DS queries (see

Table 1: Aggregate time (in seconds) in single-server mode

	TPC-H (TPC-DS)		
	SF-30	SF-50	SF-75
psql	358.9 (7275.7)	592.6 (8789.3)	796.2 (10094.3)
rdbX	78.6 (842.6)	88.8 (1357.7)	104.1 (1888.6)
rdbX_im	32.5 (1065.7)	52.2 (1317.1)	74.5 (1611.2)
rdbY	59.2 (787.1)	80.4 (1328.2)	132.7 (2117.7)
rdbY_non	70.7 (2349.6)	87.6 (2780.8)	139.8 (3230.9)
spark_sql	248.9 (906.4)	391.5 (1282.2)	572.9 (1955.2)
TAG_tg	53.4 (149.1)	85.1 (246.3)	121.3 (357.3)

parenthesised numbers). On TPC-H, it is much faster than PostgreSQL and Spark SQL and competitive with all others except for RDBMS-X IM (whose speedup does not exceed 1.6x). As the drill-down into our measurements shows, TAG-join excels particularly at computing local-aggregation queries and, regardless of aggregation style, PK-FK join queries and queries with selective joins, outperforming even RDBMS-X IM on these query classes. Figure 5 shows that TAG-join outperforms Spark SQL in both aggregate runtime and network communication in our distributed experiments.

8.1 Experiment Setup

8.1.1 Datasets and Queries. We evaluate our approach using two standard relational database benchmarks TPC-H [1] and TPC-DS [2]. The query workload of the TPC-H benchmark consists of 22 queries: 20 among them with 2 to 8 joins, the remaining 2 queries are single table scans. All but 1 are acyclic, the exception is a five-way cycle query. The TPC-DS benchmark offers a more realistic and challenging query workload that contains 99 acyclic queries in total [45], of which we evaluate 84 queries in our experiments. We discarded 15 queries that contain functions that are not supported by the vertex-centric platform we used (and are not in the scope of this work), such as ranking function (rank over()), extracting a substring, and computing standard deviation. We also exclude a scenario containing iterative queries. The 84 TPC-DS queries in our experiments feature between 3 to 12 joins each, including joining multiple fact tables and multiple dimension tables, and joins between large dimension tables (so not all joins are PK-FK). The queries are of varying complexities, containing aggregation and grouping and correlated subqueries. All queries are run without the ORDER BY and LIMIT clauses as we left top-k processing outside the scope of this paper.

8.1.2 The Vertex-Centric System We Used. We implemented our approach on top of TigerGraph [21], a graph database system that supports native graph storage and a vertex-centric BSP computational model that takes advantage of both thread and distributed cluster parallelism. TigerGraph offers a graph query language that allows developers to express vertex-centric programs concisely. The mapping from the query to the vertex program is straightforward, well-defined [21, 22], and is not where the optimizations kick in (we checked with the TigerGraph engineers). This ensures that the queries we wrote execute our intended vertex-centric programs faithfully, without compiling/optimizing them away [4].

We used the free TigerGraph 3.0 Enterprise Edition in experiments and ran it in main-memory mode. We denote our implementation with **TAG_tg** in figures and tables below.

8.1.3 The Comparison Systems. We compared our TAG-join implementation against popular relational database systems:

PostgreSQL 12.3 is a popular open-source relational database implemented as a row store (**psql** in tables).

RDBMS-X is a leading commercial RDBMS with row store support (release version from 2018). RDBMS-X offers an In-Memory feature that uses an in-memory column store format. This format accelerates query processing by enabling faster data scans, aggregation and joins. We ran the queries with two settings: traditional row store format (**rdbX**) and dual format with in-memory column store enabled (**rdbX_im**).

RDBMS-Y is a commercial RDBMS with row store support (release version from 2019). We run RDBMS-Y with two settings: clustered (**rdbY**) and non-clustered primary key (**rdbY_non**).

Spark SQL 3.0.1 [12] is a module of Apache Spark [5] supporting relational (and JSON) data processing. For the purpose of our experiments, we include it in the collective term "relational engines".

For all relational engines we tuned memory parameters (e.g. buffer cache size) to ensure that the entire database as well as intermediate query results can fit into memory (we monitored disk usage during the experiments, confirming that all warm runs performed no disk access). We enabled parallel execution of queries for all of the RDBMSs without forcing a specific degree of parallelism, letting their optimizer decide on how many workers to use during execution. The exact configuration parameters for all systems are listed in the extended version of the paper [46].

8.1.4 Hardware. The single-server experiments ran on an AWS EC2 r4.8xlarge instance, with 2.3 GHz Intel Xeon E5-2686 v4 processor with 32 vCPU count (number of supported parallel threads), 244 GB of memory and 500GB SSD drive, running Ubuntu 16.04. For RDBMS-X, we used Linux 7.6. We ran distributed experiments on an Amazon EC2 cluster of 6 machines, each with an 8-core 2.50GHz Intel XeonPlatinum 8259CL processor and 2 threads per core (i.e. 16 vCPUcount), 64 GB of memory and 200GB SSD drive. All machines ran Ubuntu 18.04.

8.1.5 Methodology. The queries are evaluated on datasets obtained with the benchmark generators, at scale factors 30 (30GB), 50 (50GB) and 75 (75GB). Each dataset is supplied with primary and foreign key indexes. Each query is executed 11 times, the first run to warm up the cache and the remaining 10 runs to compute the average runtime. We impose a timeout of 30 minutes per execution.

[4]Some systems offer vertex-centric APIs which are implemented internally via relational-style joins (e.g. [9, 19]), thus being unsuited for our experiment. In contrast, TigerGraph features native implementation for the vertex-centric primitives we describe here. Inspection of the queries we published with the extended version of the paper reveals that they each are expressed as a sequence of separate one-edge hops, i.e. vertex-centric steps where edge sources send messages to edge targets.

8.2 Data Loading Results

The datasets are generated by the tools provided with the benchmark suites, and are loaded into databases using their respective commands for bulk data load. For relational systems, primary and foreign key indexes are built on the loaded data, as prescribed by the TPC benchmark protocol. All RDBMSs we deployed organize indexes in a B-tree structure by default.

We measured both loading time (including index creation time for the RDBMSs) and loaded data size (including index size for RDBMSs). Recall that no indexes are constructed for the TAG representation since the attribute vertices act as indexes implicitly. The exact measurements for all systems and datasets are published in the extended version.

The gist of the experiment is that we observed similar loaded data size for all systems (within 10% of each other except for the more wasteful Postgres) and similar loading times (also within roughly 10% of each other, except for RDBMS-X which takes double the time of the others on TPC-DS). This testifies to the absence of time and space overhead for loading data as a graph vs into an RDBMS.

8.3 Single-Server TPC-H Results

Table 1 shows the aggregate run times of TPC-H queries over datasets of different scale factors. In aggregate, TAG-join approach is 6.5x faster than PostgreSQL, 4.7x faster than Spark SQL, and shows competitive performance with RDBMS-Y (both versions) and RDBMS-X. The exception is RDBMS-X with In-Memory column store, which outperforms TAG-join by 1.6x in aggregate.

A more nuanced picture emerges when we drill down to individual queries: TAG-join is competitive with or outperforms both RDBMS-X row store and in-memory column store on local aggregation (LA) queries (recall §7), including those with nested correlated subqueries (where TAG-join shines). TAG-join is outperformed by RDBMS-X in-memory column store on low-selectivity global aggregation (GA) queries (§7) requiring full data scans and involving relatively few joins (in the most extreme case, query q1 aggregates a single-table scan with no joins at all). For this class of queries, TAG-join's benefit is limited since execution cost is dominated by full scan aggregation, for which the compressed in-memory column store format is tuned. Full results at individual query level for all three scale factors are shown and discussed in the extended version [46].

8.4 Single-Server TPC-DS Results

Table 1 shows the aggregate run times of TPC-DS queries at different scale factors (the parenthesised numbers). In aggregate, TAG-join on TigerGraph (TAG_tg) performs the best on TPC-DS queries, followed by RDBMS-X in-memory column store (rdbX_im), RDBMS-X row-store (rdbX), RDBMS-Y with clustered PK (rdbY) and without (rdbY_non), then Spark SQL and PostgreSQL (psql).

The breakdown in Table 2 shows that TAG-join outperforms the relational engines on the majority of the TPC-DS queries. Runtimes of all individual queries are shown in the extended version.

Most of the TPC-DS queries involve aggregation, including local, global and scalar. In Table 3 we present aggregate times of queries across all systems by breaking down the queries into 4 groups based on the type of the aggregation.

Table 2: Number of TPC-DS queries where TAG-join approach outperforms, shows competitive or worse performance against each of the relational systems at SF-75. Total number of queries is 84.

#queries	outperforms	competitive	worse
psql	84	-	-
rdbX	74	4	4
rdbX_im	64	3	17
rdbY	53	22	9
rdbY_non	64	12	8
spark_sql	73	5	6

Table 3: Runtime (in seconds) of TPC-DS workload at SF-75 broken down by aggregation type

SF-75	No agg	LA	GA	Scalar GA
psql	0.58	1913.7	7788.433	391.592
rdbX	2.42	72.3	1375.739	438.137
rdbX_im	1.71	43.9	1279.353	286.183
rdbY	0.52	42.8	1771.947	302.384
rdbY_non	0.32	138.3	2736.952	355.361
spark_sql	13.6	160.4	1549.5	231.5
TAG_tg	0.16	8.37	231.044	117.734

Queries without Aggregation. TAG-join achieves 1.5-27x speedup over relational systems on individual queries that do not use any aggregation function, i.e. select-project-join queries. There are only 3 queries of this type.

LA Queries. TAG-join does very well on queries involving local aggregation, as observed in the TPC-H results and confirmed on the 15 TPC-DS queries whose aggregation is local. TAG-join consistently outperforms PostgreSQL with an aggregate speedup of two orders of magnitude, and the other relational engines with an aggregate speedup of 5-16x. The highest speedups on individual LA queries are observed on queries that use a WITH clause in order to union results of subquery blocks, where each block joins at least 3 dimensions with a different fact table (e.g. q56, q33, q60).

GA and Scalar GA Queries . TAG-join does very well on most of the queries with global aggregation, showing 5-30x speedup in aggregate. There are 66 queries (out of 84) with GA or scalar GA. Most of these queries have relatively selective filter conditions, reducing the number of active vertices that need to aggregate their values into the same global structure, and thus achieving competitive performance with relational systems. Examples of such queries with GA are queries q22, q45, q69 and q74. Note that these queries include roll-up aggregations (e.g. q18) to compute subtotal aggregate values of each group by key. This functionality is not offered out-of-the-box by the TigerGraph engine, but it can be simulated using multiple global structures with different keys. TAG-join performs well on queries with global scalar aggregation (e.g. q32 and q94), achieving 2-3x speedup in aggregate over most of the relational engines, except for RDBMS-X IM, which outperforms TAG-join on 17 GA or Scalar GA queries out of the total 84 TPC-DS queries.

Table 4: Peak RAM usage percentage at SF-75.

%	psql	rdbX	rdbX_im	rdbY	spark_sql	TAG_tg
TPC-H	65.9	57.1	51.2	55.1	57.4	53.8
TPC-DS	61.7	49.8	43.5	54.3	68.1	52.9

8.5 Memory Usage

Table 4 shows the peak RAM usage percentage (full measurements and methodology are detailed in [46]). For RDBMS-Y the numbers are shown only for clustered PK, being the same for non-clustered PK. We only show results for SF-75, but the numbers are proportional for SF-30 and SF-50. Notice that TAG_tg' memory performance is similar to RDBMS-Y and RDBMS-X row store. RDBMS-X IM does better, but not by a game-changing margin: 9.4% on TPC-DS (where TAG_tg is faster though) and a negligible 2.6% on TPC-H.

8.6 Distributed Experiments

In a cluster setting, we compared TAG-join implementation on TigerGraph 3.1 against Spark SQL 3.0.1.

8.6.1 Experiment Setup. We used the TPC workloads described in §8.1.1 and the cluster described in §8.1.4.

We operated TigerGraph in distributed query mode, in which the vertex program is executed in parallel on all machines and the result is gathered at the machine where the given query is started. No additional tuning was carried out for TigerGraph. We used TigerGraph's default automatic partitioning of the input graph among the machines. Since we observed strong performance as is, we did not try to tune partitioning, which we leave for future work.

A Spark application consists of a driver process and a number of executor processes distributed across machines in a cluster. Executor processes are launched by the driver process, read data from distributed files systems (S3 bucket in our experiments) and execute the given query. Each executor process can run multiple tasks in parallel. Through extensive tuning experiments, we found the best-performing configuration to assign 3 executors per machine, with 5 cores and 16GB of memory per executor (almost 2x faster than one executor per machine). We used Parquet [4], a compressed columnar file format, as a data source, for which Spark supports column pruning and filter predicate pushdown. Spark SQL also offers in-memory caching, which we enabled in our experiments.

8.6.2 Results. Figure 5 shows the aggregate runtimes of TPC-H and TPC-DS queries. On TPC-H queries TAG_tg is in aggregate 2x faster than Spark SQL, and 1.46x faster on TPC-DS queries. Similarly to the single-server experiments, the best performance is observed on queries without aggregation, with local aggregation and correlated subqueries.

On TPC-H queries, TAG_tg is faster than Spark SQL on 17 queries and competitive on 3 queries (out of 22 queries total). For example, on LA queries such as q3, q4, q5 and q10 the speedup ranges from 1.6x to 4x. The biggest speedup of 17x over Spark SQL is observed on q17, which contains a correlated a subquery. TAG_tg performs

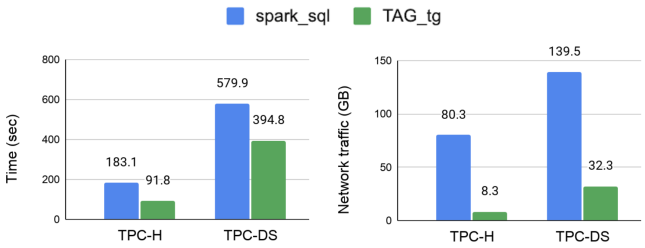

Figure 5: Summary of distributed experiments

well on most queries with GA or scalar GA, except for q6 and q13 where Spark SQL is faster by 1.4-2.5x. Individual runtimes of all TPC-H queries are shown in [46].

Out of 84 TPC-DS queries, TAG_tg is either competitive or outperforms Spark SQL on 64 queries. On queries without aggregation the speedup is 3.4-5.5x, while on queries with LA TAG_tg achieves up to 7.6x speedup. TAG_tg performs well on most of the queries with either GA or scalar GA. Spark SQL is only faster on 20 queries from this class, because in order to compute the final result all active vertices need to write into a single global accumulator, which becomes the parallelism bottleneck. Individual runtimes of all TPC-DS queries are shown in [46].

We track network usage on each machine in the cluster using the *sar* tool to record the total number of bytes received and transmitted. Figure 5 shows the total network traffic per benchmark execution. Spark SQL incurs 9x more traffic on TPC-H and 4x more on TPC-DS because it uses broadcast or shuffle joins, which require runtime replication of data across partitions.

9 CONCLUSION AND FUTURE WORK

We have shown that the TAG encoding and our TAG-join algorithm combine to unlock the potential of vertex-centric SQL evaluation to exploit both intra- and inter-machine parallelism. By running full TPC SQL queries we have proven that our vertex-centric approach is compatible with executing RA operations beyond joins. The observed performance constitutes very promising evidence for the relevance of vertex-centric approaches to SQL evaluation.

From the SQL user's perspective, our experiments show that in single-server data warehousing settings, vertex-centric evaluation can clearly outperform even leading commercial engines like RDBMS-X IM column store. In TPC-H workloads, comparison to RDBMS-X IM depends on the kind of aggregation performed, while our approach is competitive with or superior to the other relational engines. In a distributed cluster, our TAG-join implementation outperforms Spark SQL on both TPC benchmarks.

Our main focus has been on join evaluation and we have only scratched the surface of inter-operator optimizations, confining ourselves to those inspired by the relational setting (like pushing selection, projection and aggregations before joins). We plan to explore optimizations specific to the vertex-centric model.

If the value domain is continuous and the database is constantly being updated, the TAG encoding would prescribe creating a new attribute vertex for virtually each incoming value, which is impractical. Applying our vertex-centric paradigm to this scenario is an open problem which constitutes an appealing future work avenue.

REFERENCES

[1] 2018. *TPC-H Benchmark.* http://www.tpc.org/tpch
[2] 2019. *TPC-DS Benchmark.* http://www.tpc.org/tpcds
[3] 2020. *Apache Giraph.* https://giraph.apache.org/
[4] 2020. *Apache Parquet.* https://parquet.apache.org/
[5] 2020. *Apache Spark.* https://spark.apache.org
[6] 2020. *Apache Spark GraphX.* https://spark.apache.org/graphx/
[7] 2020. *TigerGraph.* https://www.tigergraph.com/
[8] C. Aberger, A. Lamb, K. Olukotun, and C. Re. 2018. LevelHeaded: A Unified Engine for Business Intelligence and Linear Algebra Querying. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 449–460.
[9] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4, Article 20 (Oct. 2017), 44 pages. https://doi.org/10.1145/3129246
[10] F. N. Afrati and J. D. Ullman. 2011. Optimizing Multiway Joins in a Map-Reduce Environment. *IEEE Transactions on Knowledge and Data Engineering* 23, 9 (2011), 1282–1298.
[11] Khaled Ammar and M. Tamer Özsu. 2018. Experimental Analysis of Distributed Graph Systems. *Proc. VLDB Endow.* 11, 10 (June 2018), 1151–1164. https://doi.org/10.14778/3231751.3231764
[12] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1383–1394. https://doi.org/10.1145/2723372.2742797
[13] Albert Atserias, Martin Grohe, and Dániel Marx. 2008. Size Bounds and Query Plans for Relational Joins. In *Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS '08)*. IEEE Computer Society, USA, 739–748. https://doi.org/10.1109/FOCS.2008.43
[14] Nurzhan Bakibayev, Tomáš Kočiský, Dan Olteanu, and Jakub Závodný. 2013. Aggregation and Ordering in Factorised Databases. *Proc. VLDB Endow.* 6, 14 (Sept. 2013), 1990–2001. https://doi.org/10.14778/2556549.2556579
[15] Paul Beame, Paraschos Koutris, and Dan Suciu. 2014. Skew in Parallel Query Processing. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Snowbird, Utah, USA) *(PODS '14)*. Association for Computing Machinery, New York, NY, USA, 212–223. https://doi.org/10.1145/2594538.2594558
[16] Paul Beame, Paraschos Koutris, and Dan Suciu. 2017. Communication Steps for Parallel Query Processing. *J. ACM* 64, 6, Article 40 (Oct. 2017), 58 pages. https://doi.org/10.1145/3125644
[17] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. 1983. On the Desirability of Acyclic Database Schemes. *J. ACM* 30, 3 (July 1983), 479–513. https://doi.org/10.1145/2402.322389
[18] Philip A. Bernstein and Dah-Ming W. Chiu. 1981. Using Semi-Joins to Solve Relational Queries. *J. ACM* 28, 1 (Jan. 1981), 25–40. https://doi.org/10.1145/322234.322238
[19] Yingyi Bu, Vinayak R. Borkar, Jianfeng Jia, Michael J. Carey, and Tyson Condie. 2014. Pregelix: Big(ger) Graph Analytics on a Dataflow Engine. *Proc. VLDB Endow.* 8, 2 (2014), 161–172. https://doi.org/10.14778/2735471.2735477
[20] Anand Deshpande and D. V. Gucht. 1988. An Implementation for Nested Relational Databases. In *VLDB*.
[21] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2019. TigerGraph: A Native MPP Graph Database. *CoRR* abs/1901.08248 (2019). arXiv:1901.08248 http://arxiv.org/abs/1901.08248
[22] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2020. Aggregation Support for Modern Graph Analytics in TigerGraph. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 377–392. https://doi.org/10.1145/3318464.3386144
[23] Mostafa Elhemali, César A. Galindo-Legaria, Torsten Grabs, and Milind M. Joshi. 2007. Execution Strategies for SQL Subqueries. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data* (Beijing, China) *(SIGMOD '07)*. Association for Computing Machinery, New York, NY, USA, 993–1004. https://doi.org/10.1145/1247480.1247598
[24] Jing Fan, Adalbert Gerald Soosai Raj, and J. M. Patel. 2015. The Case Against Specialized Graph Analytics Engines. In *CIDR*.
[25] George H.L. Fletcher and Peter W. Beck. 2009. Scalable Indexing of RDF Graphs for Efficient Join Processing. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management* (Hong Kong, China) *(CIKM '09)*. Association for Computing Machinery, New York, NY, USA, 1513–1516. https://doi.org/10.1145/1645953.1646159
[26] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting Worst-Case Optimal Joins in Relational Database Systems. *Proc. VLDB Endow.* 13, 12 (July 2020), 1891–1904. https://doi.org/10.14778/3407790.3407797
[27] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 17–30. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez
[28] G. Gottlob, M. Grohe, N. Musliu, M. Samer, and Francesco Scarcello. 2005. Hypertree Decompositions: Structure, Algorithms, and Applications. In *WG*.
[29] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 1999. Hypertree Decompositions and Tractable Queries. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania*. ACM Press, 21–32. https://doi.org/10.1145/303976.303979
[30] Martin Grohe and Dániel Marx. 2014. Constraint Solving via Fractional Edge Covers. *ACM Trans. Algorithms* 11, 1, Article 4 (Aug. 2014), 20 pages. https://doi.org/10.1145/2636918
[31] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Özsu, Xingfang Wang, and Tianqi Jin. 2014. An experimental comparison of Pregel-like graph processing systems. *Proceedings of the VLDB Endowment* 7 (08 2014), 1047–1058. https://doi.org/10.14778/2732977.2732980
[32] Xiao Hu, Yufei Tao, and Ke Yi. 2017. Output-Optimal Parallel Algorithms for Similarity Joins. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Chicago, Illinois, USA) *(PODS '17)*. Association for Computing Machinery, New York, NY, USA, 79–90. https://doi.org/10.1145/3034786.3056110
[33] Xiao Hu and Ke Yi. 2019. Instance and Output Optimal Parallel Algorithms for Acyclic Joins. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Amsterdam, Netherlands) *(PODS '19)*. Association for Computing Machinery, New York, NY, USA, 450–463. https://doi.org/10.1145/3294052.3319698
[34] Alekh Jindal, Praynaa Rawlani, Eugene Wu, Samuel Madden, Amol Deshpande, and Mike Stonebraker. 2014. Vertexica: Your Relational Friend for Graph Analytics! *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 1669–1672. https://doi.org/10.14778/2733004.2733057
[35] Paraschos Koutris, Semih Salihoglu, and Dan Suciu. 2018. Algorithmic Aspects of Parallel Data Processing. *Foundations and Trends® in Databases* 8, 4 (2018), 239–370. https://doi.org/10.1561/1900000055
[36] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. 2010. GraphLab: A New Framework for Parallel Machine Learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence* (Catalina Island, CA) *(UAI'10)*. AUAI Press, Arlington, Virginia, USA, 340–349.
[37] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. 2014. Large-Scale Distributed Graph Computing Systems: An Experimental Evaluation. *Proceedings of the VLDB Endowment* 8.
[38] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184
[39] Robert McCune, Tim Weninger, and Gregory Madey. 2015. Thinking Like a Vertex: a Survey of Vertex-Centric Frameworks for Distributed Graph Processing. *Comput. Surveys* 48 (07 2015). https://doi.org/10.1145/2818185
[40] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proc. VLDB Endow.* 12, 11 (July 2019), 1692–1704. https://doi.org/10.14778/3342263.3342643
[41] Hung Q. Ngo. 2018. Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems. arXiv:1803.09930 [cs.DB]
[42] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-Case Optimal Join Algorithms. *J. ACM* 65, 3, Article 16 (March 2018), 40 pages. https://doi.org/10.1145/3180143
[43] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Rec.* 42, 4 (Feb. 2014), 5–16. https://doi.org/10.1145/2590989.2590991
[44] Dan Olteanu and Jakub Závodný. 2015. Size Bounds for Factorised Representations of Query Results. *ACM Trans. Database Syst.* 40, 1, Article 2 (March 2015), 44 pages. https://doi.org/10.1145/2656335
[45] Meikel Poess, Raghunath Othayoth Nambiar, and David Walrath. 2007. Why You Should Run TPC-DS: A Workload Analysis. In *Proceedings of the 33rd International Conference on Very Large Data Bases* (Vienna, Austria) *(VLDB '07)*. VLDB Endowment, 1138–1149.
[46] Ainur Smagulova and Alin Deutsch. 2021. Vertex-centric Parallel Computation of SQL Queries. arXiv:2103.14120 [cs.DB]
[47] Radu Stoica, George Fletcher, and Juan F. Sequeda. 2019. On directly mapping relational databases to property graphs. In *Alberto Mendelzon Workshop on Foundations of Data Management (AMW2019) (CEUR Workshop Proceedings)*, Aidan Hogan and Tova Milo (Eds.). CEUR-WS.org. http://ceur-ws.org/Vol-2369 13th Alberto Mendelzon International Workshop on Foundations of Data Management,

AMW 2019 ; Conference date: 03-06-2019 Through 07-06-2019.

[48] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111. https://doi.org/10.1145/79173.79181

[49] Todd L. Veldhuizen. 2012. Leapfrog Triejoin: a worst-case optimal join algorithm. arXiv:1210.0481 [cs.DB]

[50] Adam Welc, Raghavan Raman, Zhe Wu, Sungpack Hong, Hassan Chafi, and Jay Banerjee. 2013. Graph Analysis: Do We Have to Reinvent the Wheel?. In *First International Workshop on Graph Data Management Experiences and Systems* (New York, New York) *(GRADES '13)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2484425.2484432

[51] Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. 2017. Big Graph Analytics Platforms. *Foundations and Trends in Databases* 7 (01 2017), 1–195. https://doi.org/10.1561/1900000056

[52] Weipeng P. Yan and Per-Åke Larson. 1995. Eager Aggregation and Lazy Aggregation. In *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB '95)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 345–357.

[53] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7* (Cannes, France) *(VLDB '81)*. VLDB Endowment, 82–94.

[54] Kangfei Zhao and Jeffrey Xu Yu. 2017. All-in-One: Graph Processing in RDBMSs Revisited. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1165–1180. https://doi.org/10.1145/3035918.3035943