# Sharded Joins for Scalable Incremental Graph Queries

János Maginecz, Gábor Szárnyas
Budapest University of Technology and Economics
Department of Measurement and Information Systems
Email: janos.maginecz@gmail.com, szarnyas@mit.bme.hu

***Abstract*—Model query operations form the basis of model-driven software engineering tools and model transformations. While the last decade brought considerable improvements in distributed storage technologies, known as NoSQL systems, evaluation of graph-like queries on large models requires further research. Unlike typical NoSQL workloads, model queries often include lots of join and complex filtering operations. Thus, the evaluation of such queries on continuously changing data proves to be a challenge for query engines. In this paper, we present INCQUERY-DS, a distributed graph query engine, which utilizes sharding to allow scaling for larger models.**

## I. INTRODUCTION

Model-driven software engineering (MDE) plays an important role in the development processes of critical embedded systems. With the dramatic increase in complexity that is also affecting critical embedded systems in recent years, modeling toolchains are facing scalability challenges as the size of design models constantly increases, and automated tool features become more sophisticated. Many scalability issues can be addressed by improving query performance.

Traditional query approaches reevaluate the entire query on every modification, which is expensive for large datasets. In contrast, with incremental query evaluation, the reevaluation is only calculated on parts of the model impacted by the change. This leads to a significant speedup for large, continuously changing data. The Rete algorithm [3] is an incremental algorithm that keeps the partial matches in memory. These matches are stored in nodes that are also the units of computation, i.e. each node performs a relational algebraic operation.

*Sharding* or *horizontal partitioning* of data is a technique widely used in NoSQL databases and stream processing engines [12]. However, up to our best knowledge, it has not been applied to incremental query engines. In this paper we adopt the idea of sharding to distributed query processing networks.

While sharding mitigates the problem of memory exhaustion, we should also reduce the memory consumption of our tools without performance degradation. The performance of the join operation is crucial in this area, hence we also present the performance comparison of different join algorithms.

This work was partially supported by the MONDO (EU ICT-611125) project and the MTA-BME Lendület 2015 Research Group on Cyber-Physical Systems.

## II. PRELIMINARIES

### A. Running Example: the Train Benchmark

We use the Train Benchmark [5] to present the core concepts of our approach.[1] The benchmark was designed to measure the efficiency of model queries under a real-world MDE workload. It defines a railway network composed of typical railroad items, including routes, semaphores, and switches (Figure 1).

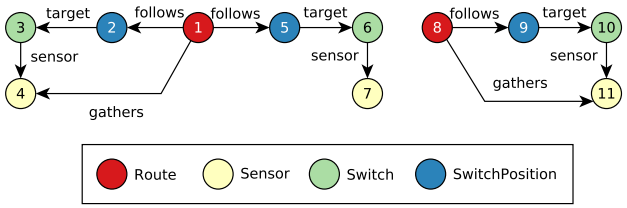


Fig. 1: Train Benchmark example model.

### B. Model Validation with Graph Queries

Engineering models can be represented as *typed graphs* with labeled vertices and edges. For example, the edges of the graph in Figure 1 can be represented with the following relations:

- follows (Route, SwitchPosition) : $(1, 2), (1, 5), (8, 9)$
- gathers (Route, Sensor) : $(1, 4), (8, 11)$
- sensor (Switch, Sensor) : $(3, 4), (6, 7), (10, 11)$
- target (SwitchPosition, Switch) : $(2, 3), (5, 6), (9, 10)$

*Well-formedness validation constraints* are often checked with *graph queries* [2]. The model is queried with graph patterns that search for *violations of the constraint* in the model. The result of a graph query is a set of *tuples*. The RouteSensor constraint requires that all sensors that are associated with a switch that belongs to a route must also be associated directly with the same route. The constrain can be translated to a graph query (in the lower left corner of Figure 2), which looks for sensors that are connected to a switch, but the sensor and the switch are not connected to the same route.

Graph queries can be formalized in relational algebra. Here we only elaborate the join and antijoin operations, as their performance has the greatest effect on query evaluation. The rest of the operations are discussed in [13].

[1]The framework is available as an open-source project at https://github.com/FTSRG/trainbenchmark

- The *join* operation ($\bowtie$) is used to connect relations based on their attributes. The *natural join* operation performs the join based on mutual attributes of the relations.
  *Example:* the target $\bowtie$ follows $\bowtie$ sensor query selects the matching $\langle$SwitchPosition, Switch, Route, Sensor$\rangle$ tuples.
- The *antijoin* operation ($\triangleright$) is used to express *negative conditions*. Formally, $r \triangleright s = r \setminus \pi_R (r \bowtie s)$.
  *Example:* the sensor $\triangleright$ gathers query selects the $\langle$Switch, Sensor$\rangle$ pairs not connected to a Route.

Based on these examples, the RouteSensor query can be formalized as:

$$\text{target} \bowtie \text{follows} \bowtie \text{sensor} \triangleright \text{gathers}$$

On the example graph, the result set consist of the tuple $(5, 6, 1, 7)$. This indicates that the model is *not well-formed* w.r.t. the RouteSensor constraint, as there should be a gather edge from node 1 to node 7.

### C. Incremental Query Evaluation

Rete [13] is an algorithm for incremental query evaluation. It ensures incrementality by keeping partial matches in memory. This is essentially a *space-time tradeoff*, an approach widely used in computer science (e.g. lookup tables, caching).

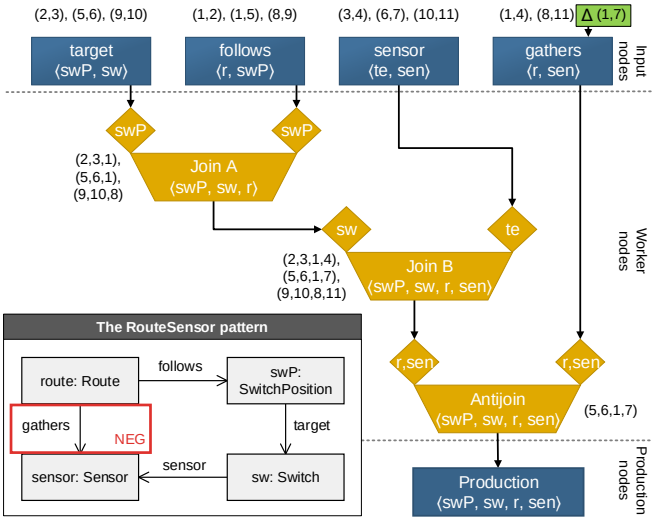


Fig. 2: The RouteSensor pattern and its Rete network.

The Rete algorithm constructs a network of processing nodes consisting of three layers (Figure 2). The partial matches (represented as tuples) are propagated through the network as messages.

- *Input nodes* store the model elements: target, follows, sensor and gathers.
- *Worker nodes* implement relational algebra operators: Join A, Join B and Antijoin.
- *Production nodes* store the results of the query.

If the engineer inserts the missing gathers edge by adding the tuple $(1, 7)$ to the gathers input node (marked with a $\Delta$ character in the figure), the Rete network only has to reevaluate the results of *Antijoin* and *Production* nodes.

## III. Related Work

EMF-IncQuery is an incremental query engine for models defined in the Eclipse Modeling Framework (EMF). It uses the Rete algorithm for incremental query evaluation [2]. As EMF-IncQuery is a single workstation tool, the memory consumption of the Rete algorithm does not allow it to scale for arbitrarily large models.

Similarly to EMF-IncQuery, IncQuery-D is based on the Rete engine but it was designed from the ground up as a distributed pattern matching system [13]. It allows for using NoSQL databases and triplestores as data sources, which means that the input of the engine can be distributed. IncQuery-D's workflow is similar to its predecessor, but it deploys the Rete nodes over a distributed system.

Drools [10] is a business rule management system that provides a *rule engine* that is capable of checking well-formedness constraints. Drools also uses the Rete algorithm as well to support incremental query evaluation. Rete-based query evaluation is used for processing Linked Data as well. INSTANS [11] uses this algorithm to perform complex event processing on streaming RDF. Diamond [9] also uses a Rete network to evaluate SPARQL queries on RDF data sets.

## IV. Overview of the Approach

This section describes the methods used to make IncQuery-DS fast and scalable.

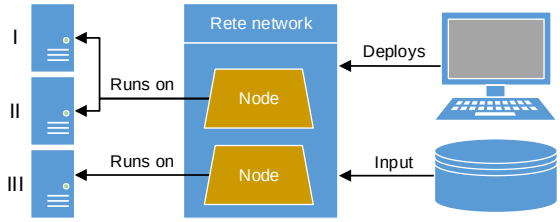


Fig. 3: The architecture of IncQuery-DS.

### A. Sharded Rete Algorithm

As a Rete node stores partial matches of the graph, its memory consumption is proportional to the size of the model. This causes *memory exhaustion* for large models. However, it is possible to split a Rete node to multiple node shards. This way a logical node can be distributed across multiple computers, splitting the memory requirements between the shards (Figure 3).

Previous work [14] only focused on distributing the Rete nodes between the machines in the cloud, but did not shard individual Rete nodes in the network. This implies that each Rete node needs to fit in the memory of a single computer, which limits the scalability of the system. For example, in the network for RouteSensor most of the memory is consumed by a single node (Join B), so distributing the Rete network does not allow it to scale for arbitrarily large models. However, sharding allows us to distribute the content of a single Rete network on multiple machines.

To achieve high performance, the computations of a Rete node must be performed based on the contents of a single shard (thus avoiding the communication overhead between the shards). For the join/antijoin nodes this requires tuples with the same join key, from both inputs, to be sent to the same shards.

A sharded layout is shown in Figure 4. Node Join B is split into two shards, Shard 0 and Shard 1, allocated on Host II and Host III. The tuples of Join A, $\{(2,\underline{3},1),(5,\underline{6},1),(9,\underline{10},8)\}$, joined against the tuples of the sensor input node, $\{(\underline{3},4),(\underline{6},7),(\underline{10},11)\}$. The join keys are their second (sw) and first (te) attributes, respectively.

For distributing the tuples, we use two hash functions. First, we map the join key to a number. Consider the simple hash function $h(\langle k_1, k_2, \ldots, k_n\rangle) = 37\sum_i k_i \mod 16$. This produces the following hash values for the keys:

$$h(\langle 3\rangle) = 15, \quad h(\langle 6\rangle) = 14, \quad h(\langle 10\rangle) = 2$$

To shard the tuples, we use another hash function, which simply uses modulo $s$, where $s$ is the number of shards: $g(x) = x \mod s$. Here, $s = 2$, hence

$$g(h(\langle 3\rangle)) = 1, \quad g(h(\langle 6\rangle)) = 0, \quad g(h(\langle 10\rangle)) = 0$$

Based on the hash values, the tuples with the join key 6 and 10 are processed by Shard 0, while the tuples with the join key 3 are processed by shard 1.

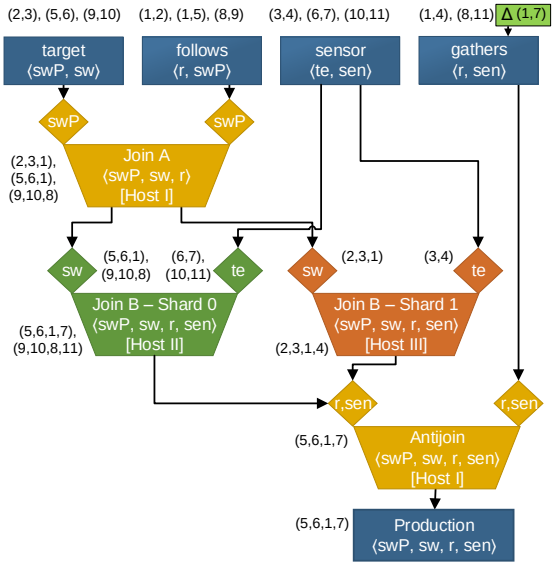


Fig. 4: Layout of the sharded Rete network for RouteSensor.

INCQUERY-DS makes the degree of sharding for each Rete node a separate decision, i.e. some nodes might have many shards, while others may remain unsharded. This greatly affects the performance of the network. Efficient node allocation is out of the scope of this paper, but is discussed in [8].

### B. Join node optimization

As mentioned in our previous report [7], different join algorithms and their underlying data structures have a significant effect on the query performance. To elaborate this, we compare the *incremental performance* of the *hash join* and the *sort merge join* algorithm [4]. We implemented both algorithms with both the standard Scala library data structures[2] and a third-party collection framework, GS-Collections[3] (developed by Goldman Sachs Group, Inc).

## V. EVALUATION

### A. Benchmark environment

The benchmarks were executed on virtual machines with the following setup: 2 cores of an Intel Xeon E5420 processor running at 2.50 GHz, 8 GBs of memory, Ubuntu 14.04 LTS operating system, Oracle JDK 8 runtime with 4 GBs of heap memory, and Gigabit Ethernet network.

### B. Benchmark phases

We use the "Repair" scenario of the Train Benchmark. In this scenario, the model is loaded and validated. Next, a subset of the model is transformed and revalidated (Figure 5). This aims to simulate the workload of a user applying quick fixes to the model. The *memory consumption* and *execution time* is recorded for each phase.

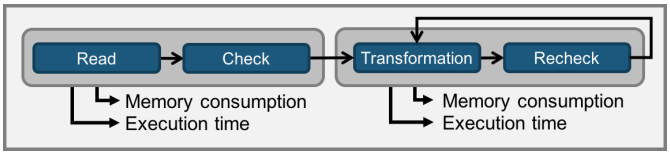


Fig. 5: Phases of the Repair Scenario.

### C. Benchmark goals

We benchmarked various aspects of the system.

*1) Scalability:* To measure the scalability improvements provided by the sharded join algorithm, we executed a benchmark on three machines in four settings:

- As a non-incremental baseline, we used Jena [1], a state-of-the-art RDF-based SPARQL in-memory query engine.
- To compare the scalability of the various degrees of distribution the benchmark measured 3 variants of INCQUERY-DS.
  - The Local variant acts as an incremental baseline, allocating all nodes on a single machine.
  - The Distributed variant allocates each node on separate computer, but does not utilize sharding.
  - The Sharded variant also allocates two nodes on separate computers, but the third node is split into two shards, allocated on different machines.

The transformation change set is indicated with a Δ character in Figure 4. This figure also shows the allocation of the worker nodes in the Sharded variant.

*2) Join Algorithm Performance:* We compared the performance of join algorithms and collection frameworks. This benchmark was executed on a single machine and only used Join A.

[2]http://docs.scala-lang.org/overviews/collections/overview.html
[3]https://github.com/goldmansachs/gs-collections

### D. Benchmark results

Figure 6 and Figure 7 show the results of the benchmarks. In both figures, the $x$-axis shows the number of triples in the model, while the $y$-axis shows the time required for the run. Both axes use a logarithmic scale.

*1) Scalability:* Figure 6 shows the results for repeated evaluations of the RouteSensor query. For large models (5M+ triples), Jena is two orders of magnitude slower than the incremental variants. Compared to the Local variant, the network overhead of Sharded INCQUERY-DS is apparent, but the response time in the "Transformation and Recheck" phase is still within the subsecond range. The Sharded variant handles models twice as large as the Distributed variant. The Distributed variant fails on the largest model, because Join B runs out of memory.

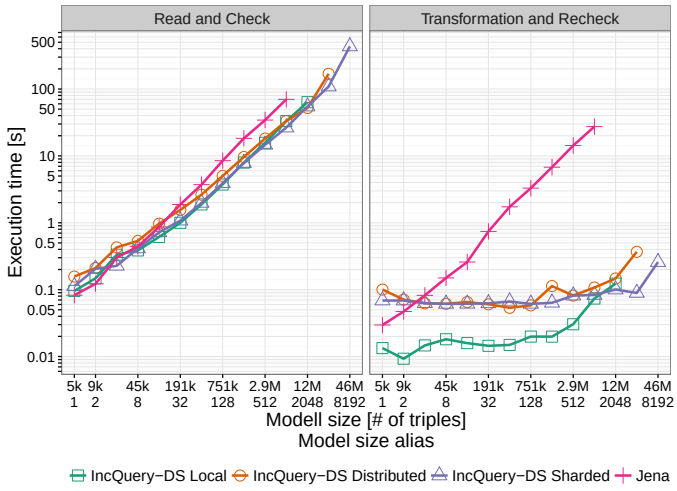


Fig. 6: Scalability of the RouteSensor query with hash join.

*2) Join Algorithm Performance:* Figure 7 displays the execution times of the different join algorithms with different underlying data structures. The "Check" phase times do not differ significantly, but the implementation using sort merge join are characteristically slower than the hash joins in the "Transformation and Recheck" phase, since the merge join algorithm has to iterate over relevant parts of the data. Comparing the GS and Scala hash joins, we can conclude that the GS variant provides a modest improvement in both scenarios.

Table I shows the memory consumption of each algorithm-implementation pair on the Join A node. The memory consumption of the different algorithms does not differ significantly, but the GS implementations use consistently less memory in every observation.

| Join node \ Model size alias | 512 | 1024 | 2048 |
|---|---|---|---|
| GS-HashJoiner | 30.0 | 54.5 | 103.7 |
| Scala-HashJoiner | 36.5 | 68.5 | 131.6 |
| GS-MergeJoiner | 34.0 | 63.1 | 120.5 |
| Scala-MergeJoiner | 37.3 | 69.9 | 134.0 |

TABLE I: Memory usage of the Join A node [MB]

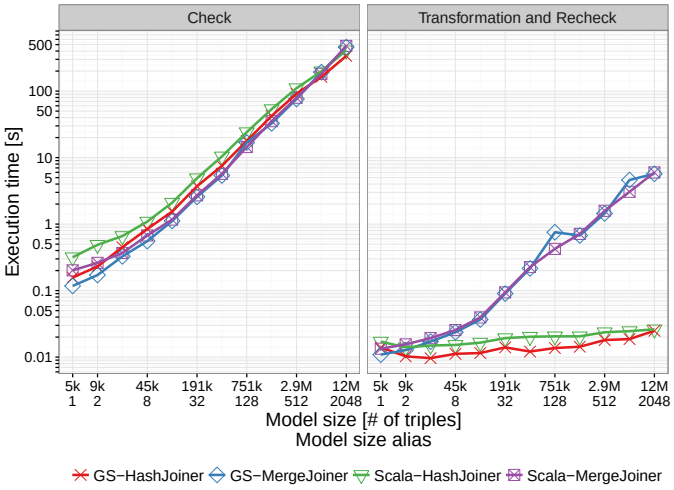


Fig. 7: Comparison of different join algorithms.

## VI. CONCLUSION AND FUTURE WORK

In this paper we presented and evaluated a truly scalable incremental query evaluation framework prototype. The results imply that the approach provides high performance for use cases requiring incremental query evaluation, while scaling well for large models.

As future work, we plan to integrate INCQUERY-DS with existing stream processing frameworks, e.g. Kafka [6].


## ACKNOWLEDGEMENTS

We want to thank Gábor Bergmann and István Ráth for their continuous support.


## REFERENCES


[1] Apache Software Foundation. Apache Jena. https://jena.apache.org/.

[2] G. Bergmann, Á. Horváth, I. Ráth, D. Varró, A. Balogh, Z. Balogh, and A. Ökrös. Incremental evaluation of model queries over EMF models. In *Model Driven Engineering Languages and Systems - 13th International Conference*, pages 76–90. Springer, 2010.

[3] C. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligences*, 19(1):17–37, 1982.

[4] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[5] B. Izsó, G. Szárnyas, and I. Ráth. Train Benchmark. Technical report, Budapest University of Technology and Economics, 2014.

[6] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, 2011.

[7] J. Maginecz. Scalable incremental graph query evaluation. Student Report, Budapest University of Technology and Economics, 2015.

[8] J. Makai, G. Szárnyas, Á. Horváth, I. Ráth, and D. Varró. Optimization of Incremental Queries in the Cloud. In *CloudMDE*, 2015.

[9] Miranker, Daniel P. et al. Diamond: A SPARQL query engine, for linked data based on the Rete match. *AImWD*, 2012.

[10] Red Hat. Drools. http://www.drools.org/.

[11] M. Rinne. SPARQL update for complex event processing. In *ISWC'12*, volume 7650 of *LNCS*, pages 453–456. 2012.

[12] M. Stonebraker. SQL databases v. NoSQL databases. *Communications of the ACM*, 53(4):10–11, 2010.

[13] G. Szárnyas. Superscalable Modeling. Master's thesis, Budapest University of Technology and Economics, Budapest, 2013.

[14] G. Szárnyas, B. Izsó, I. Ráth, D. Harmath, G. Bergmann, and D. Varró. IncQuery-D: A Distributed Incremental Model Query Framework in the Cloud. In *ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems*, pages 653–669. Springer, 2014.