IncQuery-D: A Distributed Incremental Model Query Framework in the Cloud*

Gábor Szárnyas[1], Benedek Izsó[1], István Ráth[1], Dénes Harmath[4], Gábor Bergmann[1] and Dániel Varró[1,2,3]

[1] Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
H-1117 Magyar tudósok krt. 2, Budapest, Hungary
{szarnyas, izso, rath, bergmann, varro}@mit.bme.hu
[2] DIRO, Université de Montréal
[3] MSDL, Dept. of Computer Science, McGill University
[4] IncQuery Labs Ltd.
H-1113 Bocskai út 77-79, Budapest, Hungary
denes.harmath@incquerylabs.com

Abstract. Queries are the foundations of data intensive applications. In model-driven software engineering (MDE), model queries are core technologies of tools and transformations. As software models are rapidly increasing in size and complexity, traditional tools exhibit scalability issues that decrease productivity and increase costs [17]. While scalability is a hot topic in the database community and recent NoSQL efforts have partially addressed many shortcomings, this happened at the cost of sacrificing the ad-hoc query capabilities of SQL. Unfortunately, this is a critical problem for MDE applications due to their inherent workload complexity. In this paper, we aim to address both the scalability and ad-hoc querying challenges by adapting incremental graph search techniques – known from the EMF-IncQuery framework – to a distributed cloud infrastructure. We propose a novel architecture for distributed and incremental queries, and conduct experiments to demonstrate that IncQuery-D, our prototype system, can scale up from a single workstation to a cluster that can handle very large models and complex incremental queries efficiently.

1 Introduction

Nowadays, model-driven software engineering (MDE) plays an important role in the development processes of critical embedded systems. Advanced modeling tools provide support for a wide range of development tasks such as requirements and traceability management, system modeling, early design validation, automated code generation, model-based testing and other validation and verification tasks. With the dramatic increase in complexity that is also affecting

* This work was partially supported by the CERTIMOT (ERC_HU-09-01-2010-0003) and MONDO (EU ICT-611125) projects partly during the sixth author's sabbatical.

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 [17].

Many scalability issues can be addressed by improving query performance. *Incremental evaluation* of model queries aims to reduce query execution time by limiting the impact of model modifications to query result calculation. Such algorithms work by either (i) building a cache of interim query results and keeping it up-to-date as models change (e.g. EMF-IncQuery [5]) or (ii) applying impact analysis techniques and reevaluating queries only in contexts that are affected by a change [10,21]. This technique has been proven to improve performance dramatically in several scenarios (e.g. on-the-fly well-formedness validation or model synchronization), at the cost of increasing memory consumption. Unfortunately, this overhead is combined with the increase in model sizes due to in-memory representation (found in state-of-the-art frameworks such as EMF [25]). Since single-computer heaps cannot grow arbitrarily (as execution times degrade drastically due to garbage collection problems), memory consumption is the most significant scalability limitation.

An alternative approach to tackling MDE scalability issues is to make use of advances in persistence technology. As the majority of model-based tools uses a graph-oriented data model, recent results of the NoSQL and Linked Data movement [20,1,2] are straightforward candidates for adaptation to MDE purposes (as experimented e.g. in Morsa [7] or Neo4EMF [3]). Unfortunately, this idea poses difficult conceptual and technological challenges as property graph databases lack strong metamodeling support and their query features are simplistic compared to MDE needs [15]. Additionally, the underlying data representation format of semantic databases (RDF [11]) has crucial conceptual and technological differences to traditional metamodeling languages such as Ecore [25]. Additionally, while there are initial efforts to overcome the mapping issues between the MDE and Linked Data worlds [13], even the most sophisticated NoSQL storage technologies lack efficient and mature support for executing expressive queries incrementally.

We aim to address these challenges by proposing a *novel architecture for a distributed and incremental model query framework* by adapting incremental graph pattern matching techniques to a distributed cloud based infrastructure. A main contribution of our novel architecture is that the distributed storage of data is completely separated from the distributed handling of indexing and query evaluation. Therefore, caching the result sets of queries in a distributed fashion provides a way to scale out the memory intensive components of incremental query evaluation, while still providing instantaneous execution time for complex queries.

We present IncQuery-D, a prototype tool based on a distributed Rete network that can scale up from a single workstation to a cluster to handle very large models and complex queries efficiently (Sec. 3). For the experimental evaluation, we revisit a model validation benchmark (Sec. 2) from the railway systems domain and extend it to a distributed setup (Sec. 4). Furthermore, we carry out

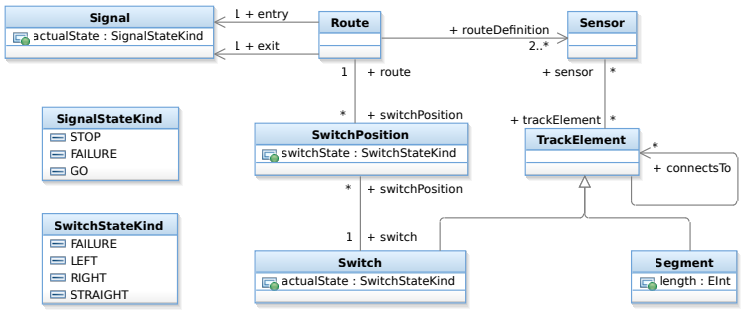


Fig. 1: The metamodel of the Train Benchmark.

detailed performance evaluation in the context of on-the-fly well-formedness validation of design models (Sec. 4) which demonstrates that our distributed incremental query layer can be significantly more efficient than the native SPARQL query technology of an RDF triple store. Finally, we discuss related work in Sec. 5 and conclude the paper in Sec. 6.

2 Preliminaries

2.1 Motivating Example: a DSL for Railways System Design

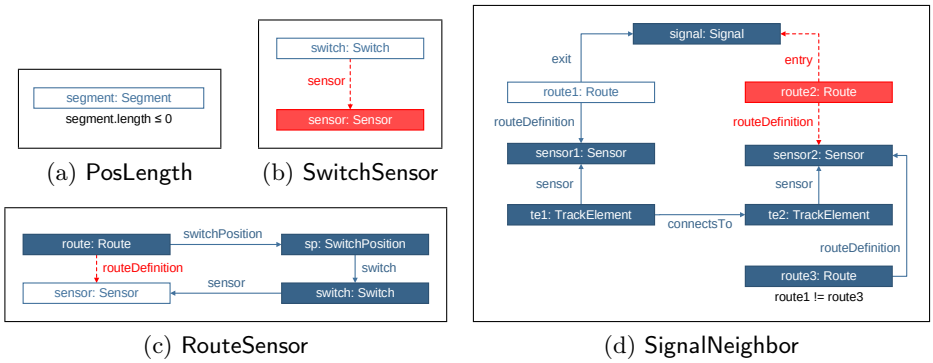
In this paper, we use the Train Benchmark [15,29] to present our core ideas and evaluate the feasibility of the approach. The Train Benchmark is used in the MONDO EU FP7 [27] project to compare query evaluation performance of various MDE tools and it is publicly available[5]. It is built around the railroad system defined in the MOGENTES EU FP7 [26] project. The system defines a network composed of typical railroad items, including signals, segments, switches and sensors. The complete EMF metamodel is shown in Fig. 1.

2.2 Queries

The Train Benchmark defines four queries which have similar characteristics to the workload of a typical MDE application. The queries look for violations of *well-formedness constraints* in the model. The violations are defined by graph patterns. The graphical representation of the patterns is shown in Fig. 2. Opaque blue rectangles and solid arrows mark positive constraints, while red rectangles and dashed arrows represent negative application conditions (NACs). The result of the query (also referred as the *match set*) is marked with transparent blue rectangles. Additional constraints (e.g. arithmetic comparisons) are shown in the figure in text.

The queries contain a mix of join, antijoin and filtering operations. The two simpler queries involve at most 2 objects (PosLength and SwitchSensor), while the

[5] `https://opensourceprojects.eu/p/mondo/wiki/TrainBenchmark`.



(a) PosLength (b) SwitchSensor

(c) RouteSensor (d) SignalNeighbor

Fig. 2: Graphical representation of the patterns in the Train Benchmark.

other two queries involve 4–8 objects and multiple join operations (RouteSensor and SignalNeighbor).

For the sake of conciseness, we only discuss the RouteSensor query in detail. 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. Therefore, the query (Fig. 2c) looks for sensors that are connected to a switch, but the sensor and the switch are not connected to the same route. This query checks for the absence of circles, so the efficiency of both the join and the antijoin operations is tested.

```
1  pattern routeSensor(Sen : Sensor) = {
2    Route(R);
3    SwitchPosition(Sp);
4    Switch(Sw);
5    Route.switchPosition(R, Sp);
6    SwitchPosition.switch(Sp, Sw);
7    Trackelement.sensor(Sw, Sen);
8    neg find noRouteDefinition(Sen, R);
9  }
10 pattern noRouteDefinition(Sen, R) {
11   routeDefinition(R, Sen);
12 }
```

Fig. 3: The RouteSensor query in IncQuery Pattern Language.

The textual representation of the RouteSensor query, defined in IncQuery Pattern Language, is shown in Fig. 3. This query binds each variable (Sen, Sw, Sp, R) to the appropriate type. It defines the three edges as relationships between the variables and defines the negative application condition as a negative pattern (neg find).

2.3 Transformations

The Train Benchmark defines a quick fix model transformation for each query. The graphical representation of the transformations is shown in Fig. 4. The insertions are shown in green with a «new» caption, while deletions are marked with a red cross and a «del» caption. In general, the goal of these transformations is to remove a subset of the invalid elements from the model. For example, in the case of the RouteSensor query, randomly selected invalid sensors are disconnected from their switch, which means that the constraint is no longer violated (Fig. 4c).

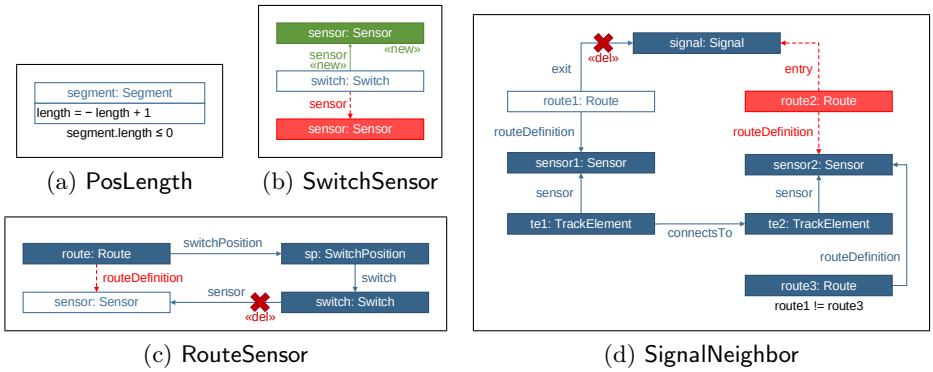


Fig. 4: Graphical representation of the transformations in the Train Benchmark.

3 A Distributed Incremental Model Query Framework

The queries and transformations introduced in Sec. 2 represent a typical workload profile for state-of-the-art modeling tools [15]. With current MDE technologies, such workloads can be acceptably executed for models up to several hundred thousand model elements [29], however when using larger models consisting of multiple million elements (a commonplace in complex domains such as AUTOSAR [5]), the performance of current tools is often not acceptable [17]. Incremental techniques can provide a solution, however they require additional (memory) resources.

The primary goal of our approach is to provide an architecture that can make use of the distributed cloud infrastructure to scale out memory-intensive incremental query evaluation techniques. As a core contribution, we propose a three-tiered architecture. To maximize the flexibility and performance of the system, model persistence, indexing and incremental query evaluation are delegated to three independently distributable asychronous components. Consistency is ensured by synchronized construction, change propagation and termination protocols.

3.1 Architecture

In the following, we introduce the architecture of IncQuery-D (see Fig. 5), a scalable distributed incremental graph pattern matcher. The architecture consists of three layers: (i) the *storage layer*, (ii) the *distributed indexer* with the *model access adapter* and (iii) the *distributed query evaluation network*.

Storage. For the storage layer, the most important issue from an incremental query evaluation perspective is that the *indexers* of the system should be filled as quickly as possible. This favors database technologies where model sharding can be performed appropriately (i.e. with balanced shards in terms of type-instance

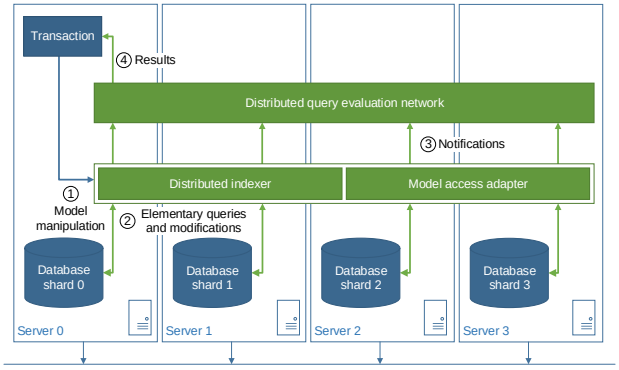


Fig. 5: The architecture of IncQuery-D, an incremental query framework (deployed in a sample four-node cluster configuration).

relationships), and elementary queries can be executed efficiently. Our framework can be adapted to fundamentally different storage back-ends, including triple stores, graph databases and relational database managements systems.

Model access adapter. In contrast to a traditional setup where the distributed model repository is accessed on a per-node basis by a model manipulation transaction, IncQuery-D provides a model access adapter that offers three core services:

1. The primary task is to provide a *surrogate key mechanism* so that each model element in the entire distributed repository can be uniquely identified and located within storage shards.
2. The model access adapter provides a *graph-like data manipulation API* (① in Fig. 5) to the user. The model access adapter translates the operations issued by the user to the query language of the backend and forwards it to the underlying data storage.
3. *Change notifications* are required by incremental query evaluation, thus model changes are captured and their effects are propagated in the form of *notification objects* (③ in Fig. 5). The notifications generate *update messages* that keep the state of the query evaluation network consistent with the model. While relational databases usually provide *triggers* for generating notifications, most triplestores and graph databases lack this feature. Due to the lack of general support, notifications are controlled by the model access adapter by providing a façade for all model manipulation operations.

Distributed indexer. Indexing is a common technique for decreasing the execution time of database queries. In MDE, *model indexing* has a key role in high

performance model queries. As MDE primarily uses a metamodeling infrastructure, all queries utilize some sort of type attribute. Typical elementary queries include retrieving all vertices of a certain type (e.g. get all vertices of the type Route), or retrieving all edges of a certain type/label (e.g. get all edges of label sensor).

To support efficient query processing, IncQuery-D maintains type-instance indexes so that all instances of a given type (both vertices and edges) can be enumerated quickly. These indexers form the bottom layer of the distributed query evaluation network. During initialization, these indexers are filled from the database backend (② in Fig. 5).

The architecture of IncQuery-D facilitates the use of a *distributed indexer* which stores the index on multiple servers. A distributed indexer inherently provides some protection from exceeding memory limits.

Distributed query evaluation network. IncQuery-D constructs a distributed and asynchronous network of communicating nodes that are capable of producing the results set of the defined queries (④ in Fig. 5). Our prime candidate for this layer is the *Rete algorithm*, however, the architecture is capable of incorporating other incremental (e.g. TREAT [18]) and search-based query evaluation algorithms as well. In the upcoming section, we provide further details on this critical component of the architecture.

3.2 The Rete Algorithm in a Distributed Environment

Numerous algorithms were proposed for the purpose of incremental query evaluation. The Rete algorithm was originally proposed for rule-based expert systems [8] and later improved and adapted for EMF models in [4]. Our current paper discusses how to adapt the Rete algorithm in a distributed environment.

Data representation and structure. The Rete algorithm uses *tuples* to represent the vertices (along with their properties), edges and subgraphs in the graph. The algorithm defines an asynchronous network of communicating nodes (see Fig. 7).

The network consists of three types of nodes. *Input nodes* are responsible for indexing the model by type, i.e. they store the appropriate tuples for the vertices and edges. They are also responsible for producing the update messages and propagating them to the *worker nodes*. *Worker nodes* perform a transformation on the output of their parent node(s) and propagate the results. Partial query results are represented in tuples and stored in the memory of the worker node thus allowing for incremental query reevaluation. *Production nodes* are terminators that provide an interface for fetching the results of the query and the changes introduced by the latest transformation.

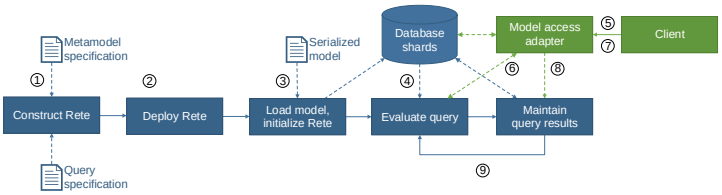


Fig. 6: The operational workflow of the distributed Rete algorithm.

Construction. The system constructs the Rete network from the layout derived from the query specification. The construction algorithm may apply various optimization techniques, e.g. reusing existing Rete nodes, known as *node sharing* [4]. An efficient Rete construction is discussed in detail in [31], and it is out of scope for the current paper.

In a distributed environment, the construction of the Rete network introduces additional challenges. First, the system must keep track of the resources available in the server cluster and maintain the mapping between the Rete nodes and the servers accordingly. Second, the Rete nodes need to be aware of the current infrastructure mapping so they can send their messages to the appropriate servers. In our system, the Rete nodes are remotely instantiated by the coordinator node. The coordinator node then sends the infrastructure mapping of the Rete network to all nodes. This way, each node is capable of subscribing to the update messages of its parent node(s). The coordinator also starts the operations in the network, such as loading the model, initiating transformations and retrieving the query results.

Operation. The operational workflow of IncQuery-D is shown in Fig. 6. Based on the *metamodel* and the *query specification*, IncQuery-D first constructs a Rete network ① and deploys it ②. In the next step, it loads the model ③ and traverses it to initialize the indexers of the Rete network. The Rete network evaluates the query by processing the incoming tuples ④. Because both the Rete indexers and the database shards are distributed across the cluster, loading the model and initializing the Rete network needs network communication. The client is able to retrieve the results ⑤–⑥, modify the model and reevaluate the query again ⑦–⑨.

The modifications are propagated in the form of *update messages* (also known as *deltas*). Creating new graph elements (vertices or edges) results in *positive update messages*, while removing graph elements results in *negative update messages*. The operation of the network is illustrated on the instance graph depicted in the lower left corner of Fig. 7. This graph violates the well-formedness constaint defined by the RouteSensor query, hence the tuple $\langle 3, 4, 2, 1 \rangle$ appears in the result set of the query. The figure also shows the Rete network containing partial matches of the original graph.

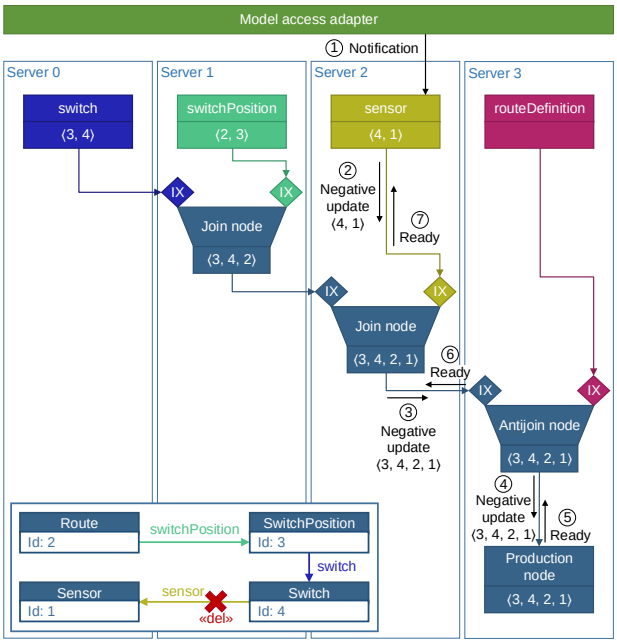


Fig. 7: A transformation sequence on a distributed Rete network.

To resolve the violation, we apply the quick fix transformation defined in the Train Benchmark and delete the sensor edge between vertices 4 and 1. When the edge is deleted, the sensor type indexer (an input node) receives a notification from the model access adapter ① and sends a *negative update* ② with the tuple $\langle 4, 1 \rangle$. The subsequent join node processes the update messages and propagates a negative update ③ with the tuple $\langle 3, 4, 2, 1 \rangle$. The antijoin node also propagates a negative update message with the same tuple ④. This is received by the production node, which initiates the *termination protocol* ⑤–⑦. After the termination protocol finishes, the indexer signals the client about the successful update. The client is now able to retrieve the results from the production node. The client may choose to retrieve only the *change set*, i.e. only the tuples that have been added or deleted since the last modification.

Termination protocol. Due to the asynchronous propagation of changes in Rete, the system must also implement a *termination protocol* to ensure that the query results can be retrieved consistently with the model state after a given transaction (i.e. by signaling when the update propagation has been terminated).

The protocol works by adding a stack to the update message propagated through the network. The stack registers each Rete node the message passes through. After the message reaches a production node, the termination protocol starts. Based on the content of the stack, acknowledgement messages (Ready)

are propagated back along the network. When all relevant input nodes (where the original update message(s) started from) receive the acknowledge messages, the termination protocol finishes. The operation of the termination protocol can be observed in Fig. 7 (messages ⑤–⑦).

4 Evaluation

To evaluate the feasibility of the IncQuery-D approach, we created a distributed benchmark environment. We implemented a prototype of IncQuery-D and compared its performance to a state-of-the-art non-incremental SPARQL query engine of a (distributed) RDF store.

4.1 Benchmark Scenario

In order to measure the efficiency of model queries and manipulation operations over the distributed architecture, we adapted the Train Benchmark [15,29] (briefly introduced in Sec. 2.1) to a distributed environment. The main goal of the Train Benchmark is to measure the query reevaluation times in systems operating on a graph-like data set. The benchmark targets a "real-world" MDE workload by running a specific set of queries (Sec. 2.2) and transformations on the model (Sec. 2.3). In this workload profile, the system runs either a single query or a single transformation at a time, as quickly as possible.

To assess scalability, the benchmark uses instance models of growing sizes, each model containing twice as many model elements as the previous one. Scalability is also evaluated against queries of different complexity. For a successful run, the tested tool is expected to evaluate the query and return the *identifiers* of the model elements in the result set.

Execution phases. The benchmark transaction sequence consists of four distinct phases. The serialization of the model is loaded into the database (load); a well-formedness query is executed on the model (initial validation); some elements are programmatically modified (transformation) and the query is reevaluated (revalidation).

Instance models. We developed a generator that creates instance models. The instance models are generated pseudorandomly, with pre-defined structural constraints and a regular fan-out structure (i.e. the in-degree and out-degree of the vertices follow a uniform distribution) [15].

Transformations. In the transformation phase, the benchmark runs quick fix transformations (Sec. 2.3) on 10% of the invalid elements (the result set of the initial validation phase), except for the SignalNeighbor query, where 1/3 of the invalid elements are modified. The transformations run in a single logical transaction, implemented with multiple physical transactions.

				PosLength (2)		RouteSensor (4)		SignalNeighbor (8)		SwitchSensor (2)	
Problem size	Triples	Nodes	Edges	RSS	MS	RSS	MS	RSS	MS	RSS	MS
1	23k	6k	17k	470	47	94	9	3	1	19	1
4	86k	23k	63k	1769	176	348	31	6	2	91	9
16	334k	88k	245k	6893	689	1301	126	19	6	326	29
64	1M	361k	1M	28239	2823	5324	511	69	19	1287	119
256	5M	1M	3M	110739	11073	21097	1996	254	74	5109	485
1024	21M	5M	15M	443458	44345	84107	8024	983	287	20716	1977
4096	85M	22M	63M	1769402	176940	336507	32051	–	–	81410	7730

RSS: result set size MS: modification size

Fig. 8: Metrics of the instance models and queries.

Metrics. To quantify the complexity of the benchmark test cases, we use a set of metrics that have been shown to correspond well to performance [15]. The values for the test cases are shown in Fig. 8. The problem size numbers take the values of 2^n in the range from 1 to 4096. For space considerations, only every other problem size is listed. The complexity of an instance model is best described by the *number of its triples*, equal to the sum of its nodes and edges. The queries are quantified by the *number of their variables* (shown in parentheses) and their *result set size* (RSS). The transformations are characterized by the number of model elements modified (*modification size*, MS).

4.2 Benchmark Architecture

Benchmark executor. The benchmark is controlled by a distinguished node of the system, called the *executor*. The executor delegates the operations (e.g. loading the model) to the distributed system. The queries and the model manipulation operations are handled by the underlying database management system which runs them distributedly and waits for the distributed operation to finish, effectively creating a synchronization point after each transaction.

Methodology. We defined two benchmark setups. (1) As a *non-incremental baseline*, we used an open-source distributed triplestore and SPARQL query system, 4store. (2) We deployed IncQuery-D with 4store as a backend database. It is important to mention that the benchmark is strongly centralized: the *coordinator* node of IncQuery-D runs on the same server as the benchmark *executor*.

The benchmark executor software used the framework of the Train Benchmark to collect data about the results of the benchmark. These were not only used for performance benchmarking but also to ensure the functional equivalence of the systems under benchmark.

The precise execution semantics for each phase are defined as follows. (1) The load phase includes loading the model from the disk (serialized as RDF/XML), persisting it in the database backend, and, in the case of IncQuery-D, initializing the Rete network. (2) The execution time of the initial validation phase is the time required for the first complete evaluation of the query. (3) The transformation phase starts with the selection of the invalid model elements and is finished after the modifications are persisted in the database backend. In the

case of IncQuery-D, the transformation is only finished after the Rete network has processed the changes and is in a consistent state. (4) The revalidation phase re-runs the query of the initial validation phase, and retrieves the updated results.

The execution time includes the time required for the defined operation, the computation and I/O operations of the servers in the cluster and the network communication (to both directions). The execution times were determined using the System.nanoTime() Java method.

Environment. We used 4store [12] (version 1.1.5) as our storage backend. The servers ran the Ubuntu 12.10 64-bit operating system with Oracle Java 7. For the implementation of the distributed Rete network, we used Akka [28] (version 2.1.4), a distributed, asynchronous messaging system.

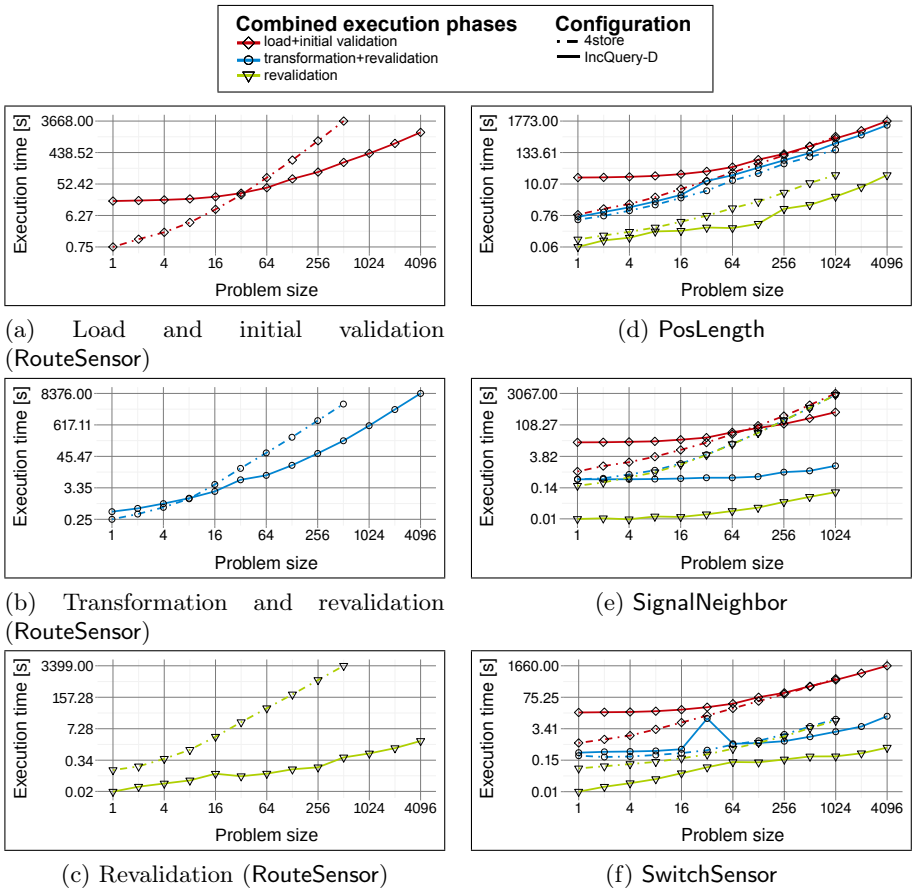
The system was deployed on the private cloud that runs on the Apache VCL (Virtual Computing Lab) platform. We reserved four virtual machines on separate host machines, with each using a quad-core Intel Xeon L5420 CPU running at 2.5 GHz and having 16 GB of RAM. The host machines were connected to a dedicated gigabit Ethernet network.

4.3 Results

The benchmark results of our experiments are shown in Fig. 9. On each plot, the x axis shows the problem size, i.e. the size of the instance model, while the y axis shows the execution time of a certain phase, measured in seconds. Both axes use logarithmic scale.

First, we discuss the results for RouteSensor, a query of medium complexity. Fig. 9a presents the combined execution time for the load and initial validation phases. The execution time is a low order polynomial of the model size for both the standalone 4store and the IncQuery-D system. The results show that despite the initial overhead of the Rete network initialization, IncQuery-D has a significant advantage starting from medium-sized models (with approximately 1 million triples). Fig. 9b shows the execution time for the sum of the transformation and revalidation phases. The results show that the Rete maintenance overhead imposed by IncQuery-D on model manipulation operations is low, and overall the model transformation phase when using IncQuery-D is considerably faster for models larger than a few hundred thousand triples. Fig. 9c focuses on the revalidation phase. The performance of IncQuery-D is characteristically different from that of the SPARQL engine of 4store. Even for models with tens of millions of tuples, IncQuery-D provides close to instantaneous query re-evaluation.

Fig. 9d–9f are presented to compare the results for the PosLength, the SignalNeighbor and the SwitchSensor queries, respectively. The PosLength query uses only a few variables but has a large result set. The SignalNeighbor query includes many variables but has a small match set. The SwitchSensor query uses a few variables and has a medium-sized result set.



(a) Load and initial validation (RouteSensor)

(d) PosLength

(b) Transformation and revalidation (RouteSensor)

(e) SignalNeighbor

(c) Revalidation (RouteSensor)

(f) SwitchSensor

Fig. 9: Benchmark results

The large result set of the PosLength query (Fig. 9d) is a challenge for incremental query evaluation systems, however, IncQuery-D still provides reasonably fast load, transformation and query evaluation times, while outperforming 4store on the revalidation time. The results for the SignalNeighbor query (Fig. 9e) show IncQuery-D has a characteristic advantage on both the transformation and the revalidation times. The SwitchSensor query also shows a clear advantage of IncQuery-D for transformation and revalidation.

Summary of observations. Based on the results, we can conclude the following observations. As expected, due to the overhead of the Rete construction, the *non-incremental approach* is often faster for small models. However, even for medium-sized models (with a couple of million triples), the Rete construction overhead already pays off for the initial validation. After the Rete network is initialized, IncQuery-D provides significantly improved transformation and

revalidation times, with the revalidation times being consistently orders of magnitude faster due to the different characteristics of their execution time.

In summary, these observations show that IncQuery-D is not just capable of processing models with over 10 million elements (pushing the limits well beyond the capabilities of single-workstation modeling tools), but also, it provides close to instantaneous query evaluation times even for very complex queries.

Threats to validity. To minimize *internal threats to validity*, we turned off the caching mechanisms of the operating system to force rereading the serialized model from the disk. Additionally, to avoid the propagation of the warmup effect of the Java Virtual Machine between the runs, each test case was started independently in separate JVM.

As our cloud infrastructure was subject to minimal concurrent load during the measurements, we aimed to minize the distortion due to load transients by running the benchmark three times and taking the *minimum value* for each phase into consideration. We did experience a certain deviation of execution times for smaller models (Fig. 9f). However, for larger models (our most important target), the transient effects do not influence validity of the benchmark results.

Regarding *external validity*, we used a benchmark that is a faithful representation of a workload profile of a modeling tool for large-scale models [15,29]. The queries both for 4store and IncQuery-D were validated by domain experts. We aimed to minimize the potential bias introduced by the additional degrees of freedom inherent in distributed systems, e.g. by a randomized manual allocation of the processing nodes of Rete network in the cloud. We plan to conduct a more detailed investigation of these effects as future work.

5 Related Work

A wide range of special languages have been developed to support *graph-based* querying over EMF [25] for a single-machine environment. OCL is a declarative constraint and query language that can be evaluated with the local-search based [6] engine. To address scalability issues, impact analysis tools [10,21] have been developed as extensions.

Outside the Eclipse ecosystem, the Resource Description Framework (RDF [11]) is developed to support the description of instances of the semantic web, assuming sparse, ever-growing and incomplete data stored as triples and queried using the SPARQL [33] graph pattern language. Property graphs [23] provide a more general way to describe graphs by annotating vertices and edges with key-value properties. They can be stored in graph databases like Neo4j [20] which provides the Cypher [24] query language.

Even though big data storages (like document databases, column family stores or MapReduce based databases) provide fast object persistence and retrieval, query engines realized directly on these data structures do not provide dedicated support for incremental query evaluation or efficient evaluation of query primitives (like join). This inspired Morsa [7] and Neo4EMF [3] to use

MongoDB and Neo4j, respectively, as a scalable NoSQL persistence backend for EMF persistence, extended with caching and dynamic loading capabilities. The commercial Virtuoso binds relational and RDF domains into one universal database, supporting SQL and SPARQL querying, and distributed query evaluation. While Morsa and Virtuoso use disk-based backend, Trinity.RDF [34] is a closed source, pure in-memory solution, which executes a highly optimized local-search based algorithm on top of the Trinity distributed key-value store with low response time. However, the effect of data updating on query performance is currently not investigated.

Rete-based caching approaches have been proposed to process Linked Data (bearing the closest similarity of our approach). INSTANS [22] uses this algorithm to perform complex event processing (formulated in SPARQL) on RDF data, gathered from distributed sensors. Diamond [19] evaluates SPARQL queries on Linked Data, where the main challenge is the efficient traversal of data, but our distributed indexing technique is still unique wrt. these approaches.

The TrainBenchmark framework was introduced in [29], where the domain and scenario were defined together with four queries, and an instance model generator. In [15], we extended the approach by characterizing models and queries with metrics, and introducing 30 new queries, and a new instance model generator. There are numerous graph and model transformation benchmarks [32,9] presented also at GRABATS and TTC tool contests, but only [16,30] focuses specifically on query performance for large models.

The conceptual foundations of our approach are based on EMF-IncQuery [5], a tool that evaluates graph patterns over EMF models using Rete. With respect to an earlier prototype [14], the main contributions of the current paper are (i) a novel architecture that introduces a separate distributed indexer component in addition to the distributed data store and distributed query evaluation network (which is key distinguishing feature compared to similar tools [19,22,34]) and (ii) the detailed performance evaluation and analysis of the system with respect to a state-of-the-art distributed RDF/SPARQL engine. Up to our best knowledge, IncQuery-D is the first approach to support *distributed incremental query evaluation* in an MDE context.

6 Conclusion

We presented IncQuery-D, a novel approach to adapt distributed incremental query techniques to large and complex model-driven software engineering scenarios. Our proposal is based on a distributed Rete network that is decoupled from a sharded graph database by a distributed model indexer and model access adapter. We presented a detailed performance evaluation in the context of quick-fix software design model transformations combined with on-the-fly well-formedness validation. The results are promising as they show nearly instantaneous complex query re-evaluation well beyond 10^7 model elements.

References

1. OpenLink Software: Virtuoso Universal Server. `http://virtuoso.openlinksw.com/`.
2. Sesame: RDF API and Query Engine. `http://www.openrdf.org/`.
3. Atlanmod research team. NEO4EMF. `http://neo4emf.com/`, Oct. 2013.
4. G. Bergmann. *Incremental Model Queries in Model-Driven Design*. Ph.D. dissertation, Budapest University of Technology and Economics, Budapest, 10/2013 2013.
5. Bergmann, Gábor et al. Incremental Evaluation of Model Queries over EMF Models. In *MODELS*, volume 6394 of *LNCS*. Springer, 2010.
6. Eclipse MDT Project. Eclispe OCL website, 2011. `http://eclipse.org/modeling/mdt/?project=ocl`.
7. J. Espinazo Pagan, J. Sanchez Cuadrado, and J. García Molina. Morsa: A scalable approach for persisting and accessing large models. In J. Whittle, T. Clark, and T. Kühne, editors, *Model Driven Engineering Languages and Systems*, volume 6981 of *Lecture Notes in Computer Science*, pages 77–92. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-24485-8_7.
8. C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligences*, 19(1):17–37, 1982.
9. R. Geiß and M. Kroll. On improvements of the Varro benchmark for graph transformation tools. Technical Report 2007-7, Universität Karlsruhe, IPD Goos, 12 2007. ISSN 1432-7864.
10. T. Goldschmidt and A. Uhl. Efficient OCL impact analysis, 2011.
11. R. C. W. Group. Resource Description Framework (RDF). `http://www.w3.org/RDF/`, 2004.
12. S. Harris, N. Lamb, and N. Shadbolt. 4store: The design and implementation of a clustered RDF store. In *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, 2009.
13. G. Hillairet, F. Bertrand, J. Y. Lafaye, et al. Bridging emf applications and rdf data sources. In *Proceedings of the 4th International Workshop on Semantic Web Enabled Software Engineering, SWESE*, 2008.
14. B. Izsó, G. Szárnyas, I. Ráth, and D. Varró. Incquery-d: Incremental graph search in the cloud. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, pages 4:1–4:4, New York, NY, USA, 2013. ACM.
15. B. Izsó, Z. Szatmári, G. Bergmann, Á. Horváth, and I. Ráth. Towards precise metrics for predicting graph query performance. In *2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 412–431, Silicon Valley, CA, USA, 11/2013 2013. IEEE, IEEE.
16. F. Jouault, J.-S. Sottet, et al. An AmmA/ATL solution for the grabats 2009 reverse engineering case study. *5th International Workshop on Graph-Based Tools, Grabats*, 2009.
17. D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi, and J. Cabot. A research roadmap towards achieving scalability in model driven engineering. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, pages 2:1–2:10, New York, NY, USA, 2013. ACM.
18. D. P. Miranker and B. J. Lofaso. The Organization and Performance of a TREAT-Based Production System Compiler. *IEEE Trans. on Knowl. and Data Eng.*, 3(1):3–10, Mar. 1991.

19. Miranker, Daniel P et al. Diamond: A SPARQL query engine, for linked data based on the Rete match. *AImWD*, 2012.
20. Neo Technology. Neo4j. `http://neo4j.org/`, 2013.
21. A. Reder and A. Egyed. Incremental consistency checking for complex design rules and larger model changes. In *MODELS'12*. Springer-Verlag, 2012.
22. M. Rinne. SPARQL update for complex event processing. In *ISWC'12*, volume 7650 of *LNCS*. 2012.
23. M. A. Rodriguez and P. Neubauer. Constructions from dots and lines. *CoRR*, abs/1006.2361, 2010.
24. A. Taylor and A. Jones. Cypher Query Lang., 2012.
25. The Eclipse Project. Eclipse Modeling Framework. `http://www.eclipse.org/emf/`.
26. The MOGENTES project. Model-Based Generation of Tests for Dependable Embedded Systems. `http://www.mogentes.eu/`.
27. The MONDO project. Scalable Modelling and Model Management on the Cloud. `http://www.mondo-project.org/`.
28. Typesafe, Inc. Akka documentation. `http://akka.io/`, 2013.
29. Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró. EMF-IncQuery: an integrated development environment for live model queries. *Science of Computer Programming*, 2014. Accepted.
30. Z. Ujhelyi, Á. Horváth, D. Varró, N. I. Csiszár, G. Szőke, L. Vidács, and R. Ferenc. Anti-pattern Detection with Model Queries: A Comparison of Approaches. In *IEEE CSMR-WCRE 2014 Software Evolution Week*. IEEE, IEEE, 02/2014 2014.
31. G. Varró and F. Deckwerth. A rete network construction algorithm for incremental pattern matching. In K. Duddy and G. Kappel, editors, *ICMT*, volume 7909 of *Lecture Notes in Computer Science*, pages 125–140. Springer, 2013.
32. G. Varró, A. Schürr, and D. Varró. Benchmarking for graph transformation. In *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 05)*, pages 79–88, Dallas, Texas, USA, September 2005. IEEE Press.
33. W3C. SPARQL Query Language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`.
34. K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 265–276. VLDB Endowment, 2013.