M Ű E G Y E T E M 1 7 8 2

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Scalable Incremental Graph Query Evaluation

Scientific Students' Association Report

Author:

János Maginecz

Advisors:

Gábor Szárnyas, PhD Student
Dr. István Ráth, Research Fellow
Dr. Gábor Bergmann, Research Fellow

2015

Contents

Kivonat

Az adatbázisokban tárolt adatmennyiség folyamatos növekedésének következtében a teljes lekérdezések folyamatos újraértékelése költségessé válhat. Ezeknek a lekérdezéseknek a válaszideje kiemelten fontos interaktív használat esetén.

Egy módszer a válaszidő csökkentésére az inkremetális lekérdezések használata. Ennek megvalósítására használható például a Rete algoritmus, amely a lekérdezést egy ún. háló segítségével értékeli ki. Az algoritmus egy kezdeti inicializáció után a lekérdezéseket csak az adatváltozások mentén értékeli újra, ezzel elkerülve a teljes lekérdezés kiértékelését. Ehhez azonban szükséges, hogy a háló állapotát folyamatosan a memóriában tartsuk. Emiatt a memóriaigény miatt az igazán nagy adathalmazok felett végzett lekérdezésekhez a hálót több számítógépre kell elosztanunk.

A dolgozat célja egy skálázható Rete háló tervezése, ami képes a gráflekérdezést megvalósító logikai csomópontokat több gépre tördelni.

Abstract

As the size of the data in databases increasing, reevaluating entire queries every time a line is modified becomes costly. The low response time of these queries is crucial for user experience, when the database is used interactively.

A possibility for the reduction of the response times is the use of incremental query evaluation algorithms. An implemetation is the Rete algorithm, that defines a so called network for the evaluation of the network. After the initialization of the network, the reevaluation is only done on the parts of the database that were changed, avoiding the reevaluation of entire queries. This however comes at the price of having to keep the state of the whole network in memory. Due to the memory requirements, very large data sets require the distribution of the network.

The goal of this report is to design a truly scalable distributed Rete network, that is able to split the memory requirement of the Rete nodes amongst multiple machines, by sharding them.

Chapter 1

Introduction

1.1 Problem statement

Keeping the response times low is essential for user experience during interactive use. One way of achieving this is using Rete networks, and calculating the new result only on the changed parts of the model. This however comes at the price of having to store the state of the network.

EMF-IncQuery is a framework that is based on this algorithm, but its scalability is limited to a single workstation. IncQuery-D remedies this by making it possible to put each node on a different machine, making the evaluation of queries possible for even larger models.

1.2 Contribution

There are numerous solutions that intend to tackle scalability issues by using incremental query evaluation. While EMF-IncQuery is limited to run in a single JVM, IncQuery-D operates on a set of nodes in the cluster. However, even IncQuery-D is limited in not being able to handle queries where the memory consumption of a single node is greater than what is available on a single machine.

By splitting logical nodes into node shards, we split the memory requirements as well. This report details the theoretical background and design decisions of the IncQuery-DS prototype, a Rete network that is able to shard the work of each node to multiple computers.

1.3 Structure of the Report

Chapter 2 provides a summary of the tools and concepts used for the construction of IncQuery-DS and the tools used for comparison. The design of IncQuery-DS is detailed in Chapter 3. Chapter 4 analyzes the performance of IncQuery-DS. Chapter 5 presents the related work. Chapter 6 summarizes the report, and describes the future plans for IncQuery-DS.

Chapter 2

Background

This chapter lays the foundations of this report by describing the concepts and technologies required for IncQuery-DS, and presents an example of the models and constraints we are using in this report.

2.1 Incremental Query Evaluation

Using incremental query evaluation, the response time of the queries for small modifications can be reduced significantly. Incremental query evaluation means, that the results of the query are saved during the first evaluation, and maintained, when the model is updated. The Rete algorithm [7, 23] is a method of evaluating queries using a network of nodes that each do a part of the evaluation. Its main difference compared to other incremental query evaluation algorithms is that it keeps the interim results of the nodes in memory. This is essentially a *space-time tradeoff*, an approach widely used in computer science (e.g. lookup tables, caching, and sorting networks). After an initial phase of processing the entire dataset, the network is able the update the results of its queries using only the changes to the data and previous results, without rereading the entire database.

The network consists of three parts (Figure 2.1):

- *Input nodes* store the model elements (objects, attributes, references), and send the appropriate changes to the *worker nodes* in the form of messages containing the data. The changes are sent in the form of tuples. A tuples is an ordered list of elements, e.g. $\langle a, b, c \rangle$.
- *Worker nodes* perform a transformation on the data they receive from their parent nodes, and send the result to their children. The output of the transformation may be based on the previous data. This is were the state information is in the network, this is why Rete uses memory space.
- *Production nodes* get the messages after they propagated through the network. These are the nodes that clients can get the results of the queries from.

As the nodes may use significant amount of memory, distributing them over many machines can increase the size of the models the network is able to process.

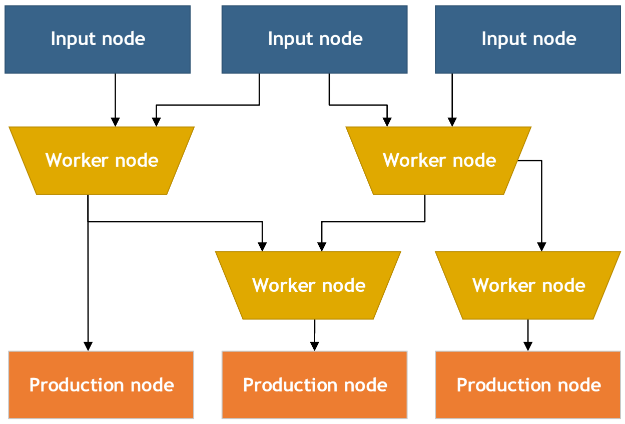


Figure 2.1. *The structure of the Rete propagation network*

2.1.1 Worker Nodes

This section describes the worker node types required to check SwitchSensor and Route-Sensor.

Trimmer Node

The trimmer node is equivalent to the projection operation in relational algebra. It has a selection mask, which defines the attributes to be selected in the ouput.

Applying a selection mask to a tuple returns a tuple only containing the values that were indexed by the selection mask. The indexes for the selections start from 0 in this report.

For example if the selection mask is $\langle 1, 2 \rangle$ and the tuple is $\langle a, b, c \rangle$, the result of the trimming is $\langle b, c \rangle$ (Figure 2.2).

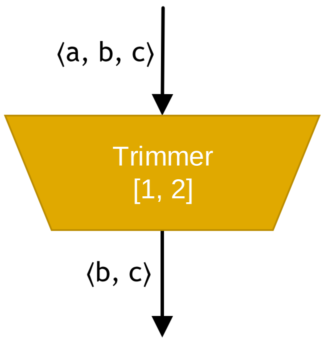


Figure 2.2. *An example of the antijoin node*

Join Node

The join node has two input slots, primary and secondary, and two selection masks for them. We call two tuples matching, if the tuples resulting from the application of the appropriate selection masks are equal. The join operation is defined as the subset of the Cartesian product of the primary and secondary tuples where the tuples match.

For example with the input of primary: $\langle a, b, c\rangle$, $\langle d, b, c\rangle$, secondary: $\langle e, a\rangle$, a primary selection mask $\langle 0\rangle$, and a secondary selection mask $\langle 1\rangle$, the output is $\langle a, b, c, e\rangle$. After the initialization a positive update, $\langle e, a\rangle$, that matches one of the previous tuples is received on the secondary slot and the node forwards the joined tuple, $\langle d, b, c, f\rangle$ (Figure 2.3).

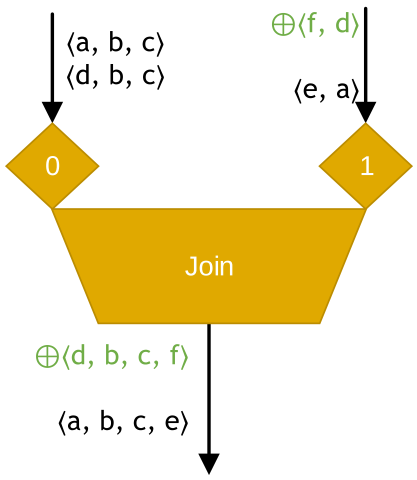


Figure 2.3. *An example of the antijoin node*

Because the node has to store the tuples from both slots, it uses a significant amount of memory.

The node in IncQuery-DS uses hash join to do this.

Antijoin Node

The antijoin node also has primary and secondary input slots and selection masks, but instead of searching for matching tuples, its output is the tuples from the primary slot that do not match the tuples from the secondary slot after applying the selection mask. This node also stores the tuples coming from both slots and it also uses a hash-based algorithm for finding matching tuples.

For example with the input of primary: $\langle a, b, c\rangle$, $\langle d, b, c\rangle$, secondary: $\langle e, a\rangle$, a primary selection mask $\langle 0\rangle$, and a secondary selection mask$\langle 1\rangle$, the output is $\langle d, b, c\rangle$. After the initialization, a positive update, $\langle f, d\rangle$, is received on the secondary slot and the previous output is negated ((Figure 2.4)).

The node in IncQuery-DS uses hash-based algorithm to do this.

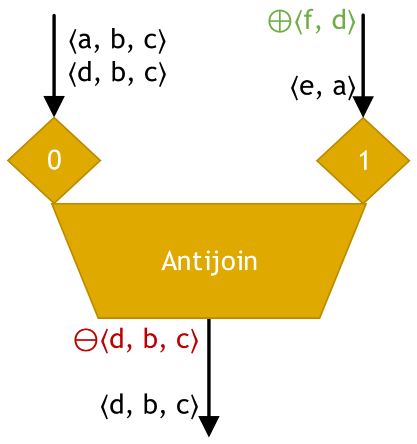


Figure 2.4. *An example of the antijoin node*

2.1.2 Building Blocks for Distributed Systems

Distributed computing is the practice of using multiple machines as parts of the same software system. An example of this is the actor model [22]. Maintaining a shared state with concurrent access can lead to issues. The main point of the actor model is to eliminate the shared state entirely.

An actor system consist of actors that serve as the unit of computation. The system can span across computers and networks. The programming model allows the system to utilize an arbitrary number of actors, which are ran by the scheduler. This is usually done using threads or thread pools. Actors only work reactively, when something communicates with them.

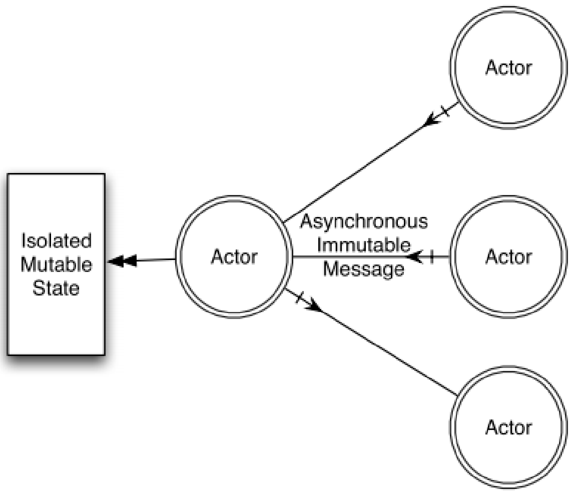


Figure 2.5. *Overview of the actor model [22]*

The communication with the actor system and between the actors is done via asynchronous messages (Figure 2.5). To do this, every actor needs a unique address, so others know where to send messages. To temporarily store these messages before they are processed, a mailbox is defined for each actor. When an actor processes a message, it can create actors, send messages to other actors, or change the way it will react to messages in the future (e.g. store state). The actors process the received messages one at a time, though not necessarily in the order they were received in. The messages are immutable, so actors can store messages without worrying about shared state.

Scala

Scala is a JVM-based language that was designed from the ground up to be scalable [18]. The Scala language uses a combination of object-oriented and functional concepts to ensure this.

The functional constructs and immutability make it easier to reason about complex, concurrent systems, while the object-oriented constructs help with the structuring.

For the reasons above, we chose Scala for the language of IncQuery-DS.

Akka

Akka is an actor framework implementation for building concurrent, distributed and resilient applications. Akka is written in Scala [25].

Actors run on actor systems, that can be run by different Java Virtual Machines (JVMs) while remaining functionally transparent. This means that after an initial configuration sending messages to actors on remote systems is as simple as sending them to local actors, assuming the messages are serializable.

2.2 Incremental Query Systems

This section presents incremental query systems. Figure 2.6 shows an overview of the described tools.

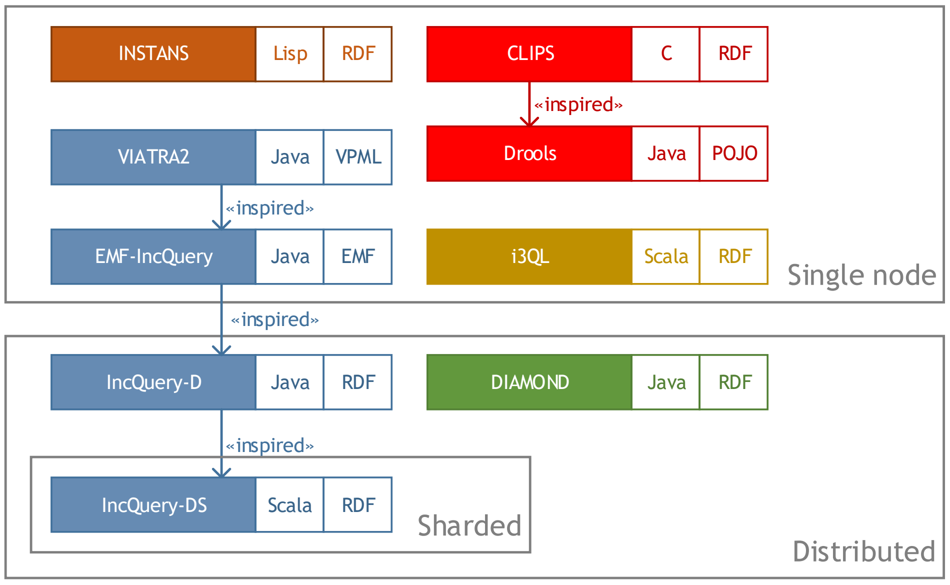


Figure 2.6. *Overview of the Rete-based tools*

2.2.1 EMF-IncQuery

EMF-IncQuery is an incremental query engine designed to check the well-formedness of EMF models. It uses the Rete algorithm for query evaluation [23]. It integrates with Eclipse-based modeling tools, making it convenient for well-formedness constraint evaluation. It is incremental, so the small changes made to the model are evaluated fast.

It first constructs a Rete network, then initializes it by filling the Rete network with the data. After the initialization, only the changes need to be propagated through the network.

EMF-IncQuery is limited to run on a single machine (Figure 2.7), and as it uses the Rete algorithm, that requires many resources to be persisted in memory, the model sizes it can handle is limited.

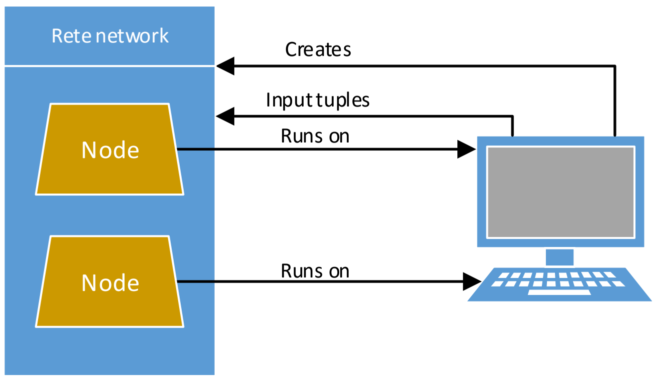


Figure 2.7. *The architecture of* EMF-IncQuery

2.2.2 IncQuery-D

Similarly to EMF-IncQuery, IncQuery-D is based on the Rete engine but was designed from the ground up as a distributed pattern matching system [23]. It also allows for using NoSQL databases and triplestores as data sources, which means that even the input of the engine can be distributed.

Because of the asynchronous nature of the distribution, a termination protocol is needed to get consistent results from the production nodes.

The workflow is similar to its predecessor, but it deploys the nodes over a distributed actor system. It also defines a middleware component to handle sending notifications to the worker nodes. It also provides an Eclipse-based editor for the allocation of the Rete nodes.

2.2.3 Drools

Drools 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 to support incremental query evaluation.

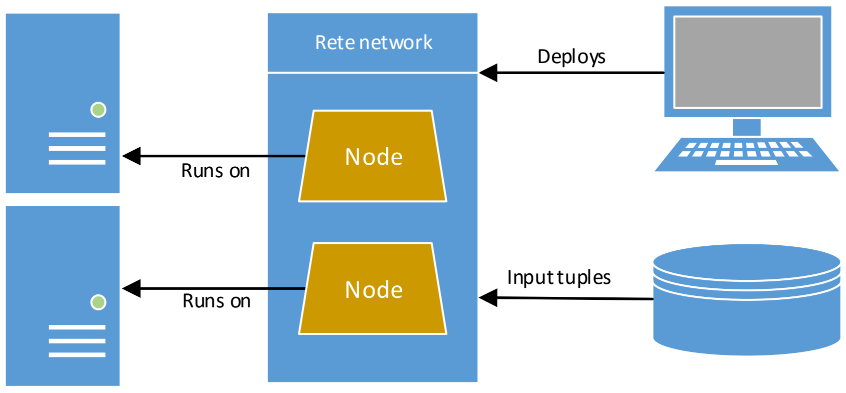


Figure 2.8. *The architecture of* INCQUERY-D

2.2.4 INSTANS

Rete-based caching approaches have been proposed for the processing of Linked Data (bearing the closest similarity of our approach). INSTANS [19] uses this algorithm to perform complex event processing (formulated in SPARQL) on RDF data, gathered from distributed sensors.

2.2.5 Diamond

Diamond uses a distributed Rete network to evaluate SPARQL queries on Linked Data [16].

2.2.6 i3QL

i3QL provides a declarative SQL-like syntax for incremental query evaluation using the DRed algorithm [10].

2.3 Model-Driven Engineering (MDE)

Model-driven engineering is a software development methodology, that attempts to reduce the complexity of systems, by using models that focus on the essentials of the system [12]. Instead of only using models as documentation, they are the main artifact of the development process. The models are often used to generate the source code of certain components or their interfaces.

MDE advocates the combination of domain-specific solutions, general purpose languages, domain specific languages and tools for automated model management. Studies show that using model driven engineering can increase productivity by a factor of 10, and it also improves software quality by enhancing maintainability, consistency and traceability.

2.3.1 Metamodeling

Metamodels define the main concepts and the relations of the domain specific modeling languages, and provide the bases for the instance models [4]. We can use predefined domains of data types supported by the metamodeling language to extend these concepts.

2.3.2 Well-Formedness Validation

Catching errors early saves development time and money. To do this as early as possible, during the model-driven development of safety-critical systems, a set of well-formedness constraints is continually evaluated [8]. Using a large amount of imperative model traversal code is complicated and hard to maintain. The rapidly increasing complexity of the models and the sophistication of validation constrains make the evaluation of these rules challenging for currently available toolchains. Defining these constraints as model queries provides a concise and efficient way to achieve the same results.

Most standards define these rules in a plain text format, but they can be translated to model queries, that can be efficiently evaluated by automatic tools.

2.3.3 Model Queries

Models can be representad as *graphs*. We can define error patterns *must not* occur in the model, for example assigning two engines or no wheels to a car.

To find these errors we can specify the queries in the form of *graph patterns*, and perform *pattern matching*. The result of this is the *match set* that consist of the elements that satisfy the pattern (i.e. violate the well-formedness constraint). When the model is edited by the user, the model is valid with regards to a particular set of well-formedness constraints if the queries return an empty result set.

As model-driven engineering becomes more and more prevalent, it gets applied to increasingly larger models. Continuously reevaluating all queries every time a small modification is made to the model may lead to long response times, which degrades user experience.

2.3.4 The Train Benchmark

To evaluate the performance of IncQuery-DS we need a standard model and a set of well-formedness constraints for it. This report uses the Train Benchmark [11] to do this.

The Train Benchmark was designed to measure the efficiency of model queries under a real-world MDE workload by simulating a user's interaction with the model [23]. After an initial load, an arbitrary number of small modifications are made to the model.

The Train Benchmark was initially published as part of the MONDO project [17].

The model revolves around a railway system (Figure 2.9).

Figure 2.10 depicts an instance model of the railway.

The well-formedness constraints we evaluate IncQuery-DS against are SwitchSensor and RouteSensor.

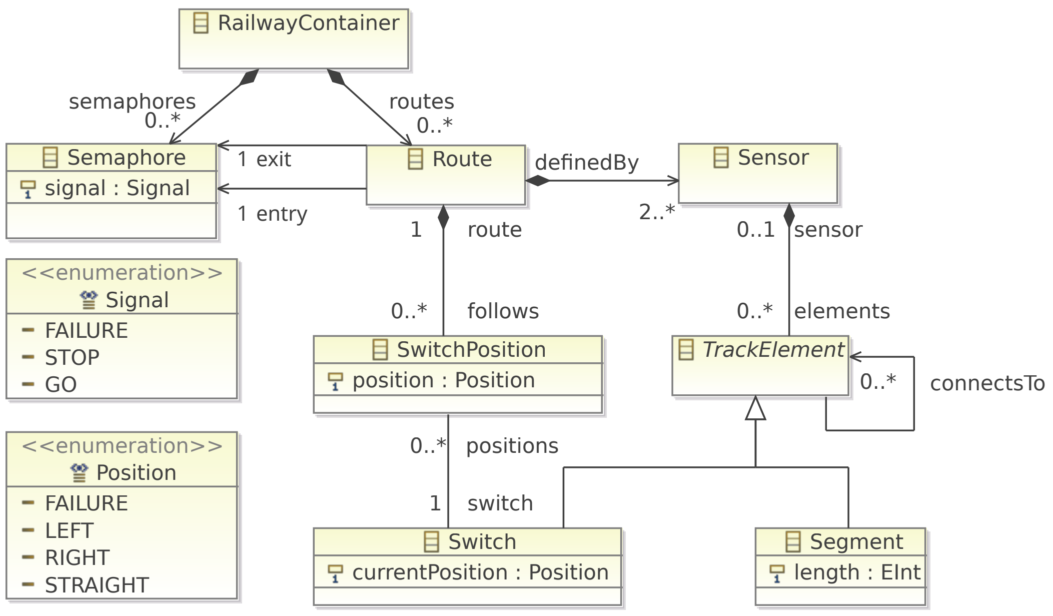


Figure 2.9. *The metamodel of the Train Benchmark*

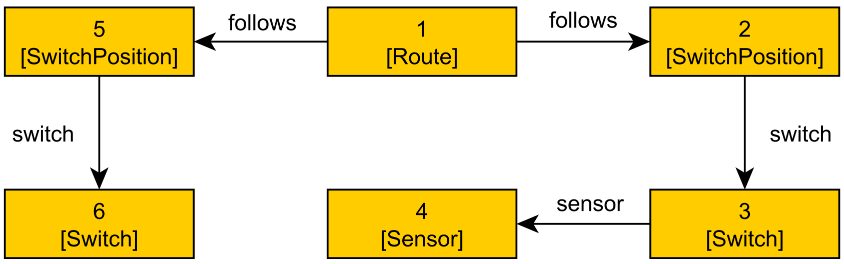


Figure 2.10. *Example model*

SwitchSensor

The SwitchSensor constraint requires every switch to have at least one sensor connected to it. On Figure 2.10 this constraint is violated by node {6}, but not by node {3}. The error pattern can be seen on Figure 2.11.

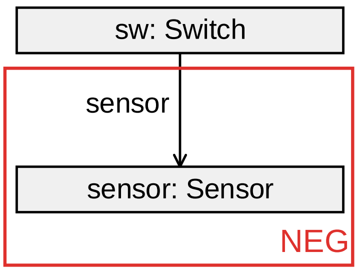


Figure 2.11. *The error pattern of SwitchSensor [8]*

Figure 2.12 is the Rete network that checks the SwitchSensor constraint.

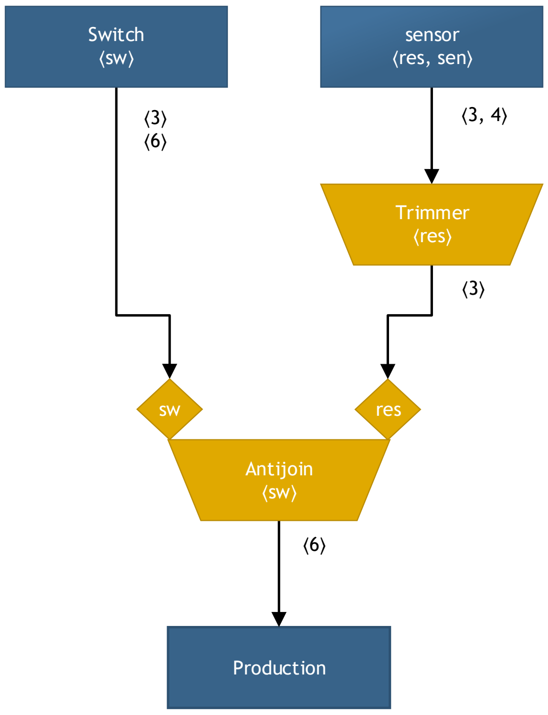


Figure 2.12. *The Rete network of the SwitchSensor constraint*

RouteSensor

The RouteSensor constraint requires all sensors that are associated with a switch that belongs to a route, to be associated directly with the same route. On Figure 2.10 this constraint is violated by the combination of node {1}, node {2}, node {3}, and node {4}, because nodes {1} and {4} are not directly linked. The error pattern can be seen on Figure 2.13.

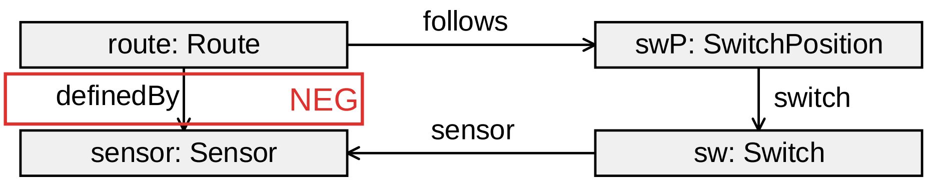


Figure 2.13. *The error pattern of RouteSensor [8]*

Figure 2.14 is the Rete network that checks the RouteSensor constraint.

2.4 Distributed Data Representation

This section describes the different ways of distributing information over machines with a standard data set. To demonstrate the concepts, we use the data set in Table 2.1 as an example.

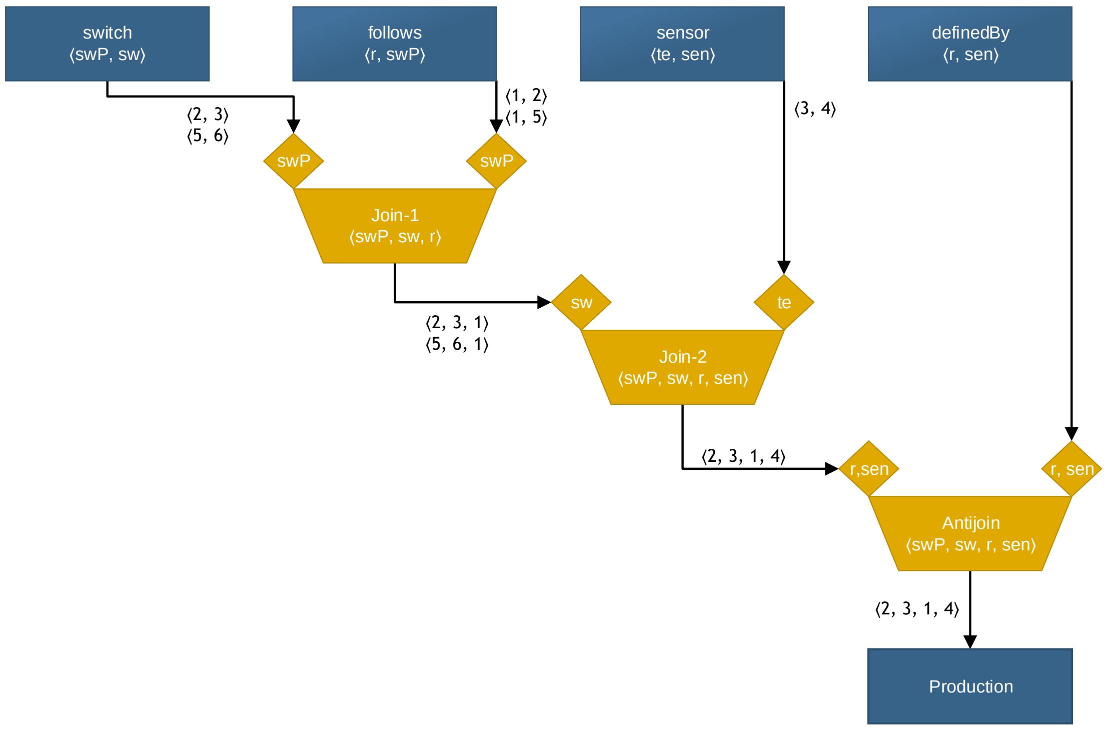


Figure 2.14. *The Rete network of the RouteSensor constraint*

ID	length	connectsTo
1	30	2
2	60	3
3	15	4
4	25	1

Table 2.1. *Example data*

One way of distribution is to store complete rows together on each machine (Table 2.2). This requires coordination between the machines, to ensure that the ordering of the attributes is the same on every database. If the order would be mixed, the clients receiving results would not be able to correctly decode them.

Server 1			Server 2		
ID	length	connectsTo	ID	length	connectsTo
1	30	2	2	60	3
3	15	4	4	25	1

Table 2.2. *Storing rows together*

Another way is to keep each column on a different machine, making them responsible for one kind of information (Table 2.3). The benefit of this is that if we run a query we only need to communicate with machines that store columns needed for our queries. Storing data this way has the added overhead of having to store the ID on each machine. This still requires coordination, the location of each column has to be tracked.

We can combine the benefits of the previous methods by storing Subject-Predicate-Object triples, where the subject identifies the entity, the predicate is the column of the infor-

Server 1		Server 2	
ID	length	ID	connectsTo
1	30	1	2
2	60	2	3
3	15	3	4
4	25	4	1

Table 2.3. *Storing columns together*

mation and the object is the actual value (Table 2.4). Descriptions of the entities can be stored on different machines, and each machine can store multiple properties. Although this approach results in a easier data distribution, it combines the costs of the previous approaches as well.

To globally identify the subjects and the predicates, the RDF standard [27] suggests that they are defined using Uniform Resource Identifiers (URIs). Objects can be either literals, like an integer, or URIs. For the purposes of this example, we will use a short prefix, '`:`'. By convention this prefix should reflect our organization: `http://www.semanticweb.org/ontologies/2015/trainbenchmark#`.

Server 1			Server 2		
Subject	Predicate	Object	Subject	Predicate	Object
:_1	:connectsTo	:_2	:_1	:length	30
:_2	:length	60	:_2	:connectsTo	:_3
:_3	:length	15	:_3	:connectsTo	:_4
:_4	:connectsTo	:_1	:_4	:length	25

Table 2.4. *Storing triples*

Directed graphs with named edges can be represented with triples, where a triple is defined for every edge using the source node as the subject, the edge as the predicate, and the target node is the object (Figure 2.15). The types and attributes of the objects can also be defined the same way.

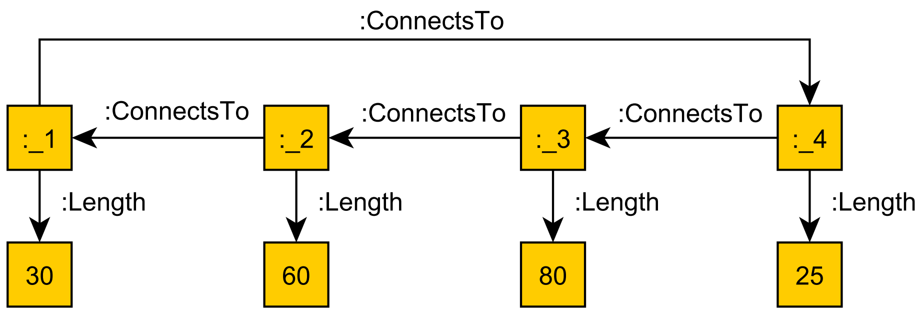


Figure 2.15. *The example instance model represented as a graph*

Resource Descriptor Framework (RDF)

RDF is a standard model used for data interchange on the web [6]. It stores the model in triples, which makes it easy to distribute the model over different machines.

We can assign classes to resources by using the `rdf:type` property. It is possible for a resource to have multiple types, because RDF also allows multiple inheritance for objects.

Turtle Format

Turtle [28], the Terse RDF Triple Language is a concrete textual representation for RDF. It provides a compact way to store the triples. Writing the URIs every time they are referenced can result in an excessive use of storage when all resources are in the same namespace, as we have seen in Table 2.4. To avoid this verbosity, the Turtle format allows us to define short prefixes (Listing 2.1).

```
@prefix : <http://www.semanticweb.org/ontologies/2015/trainbenchmark#>
:_1 :length "30"^^xsd:int .
:_2 :length "30"^^xsd:int .
:_3 :length "60"^^xsd:int .
:_4 :length "25"^^xsd:int ;
    :connectsTo :_1 .
:_1 :connectsTo :_2 .
:_2 :connectsTo :_3 .
:_3 :connectsTo :_4 .
```

Listing 2.1. *Turtle example*

Chapter 3

Overview of the Proposed Solution

This chapter describes the design of IncQuery-DS.

3.1 title

3.2 Sharded Rete Algorithm

Distributing nodes is not always sufficient. The previous works [24] only focused on allocating the individual Rete on separate machines, but did not shard the nodes in the network. For example, the network for SwitchSensor (Figure 2.12) only has one worker node with significant memory usage, so distributing it holds little advantage over the single machine layout.

It is possible to assign multiple actors to a logical Rete node and split it to multiple node shards. This way a logical node can be distributed across multiple computers, splitting the memory requirements between the shards (Figure 3.1). Special care has to be taken for stateful nodes, where the output depends on the previous inputs.

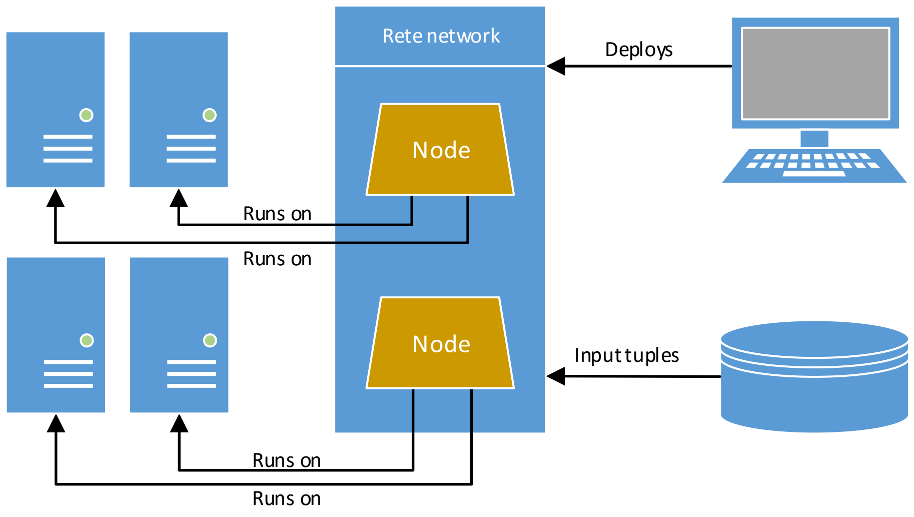


Figure 3.1. *The architecture of* IncQuery-DS

To minimize the processing time of the tuples, the computation must be performed using only the local contents of the shard (avoiding the communication overhead between the shards). For the join and antijoin nodes, this requires the matching tuples to be sent to the same shardsFigure 3.2.

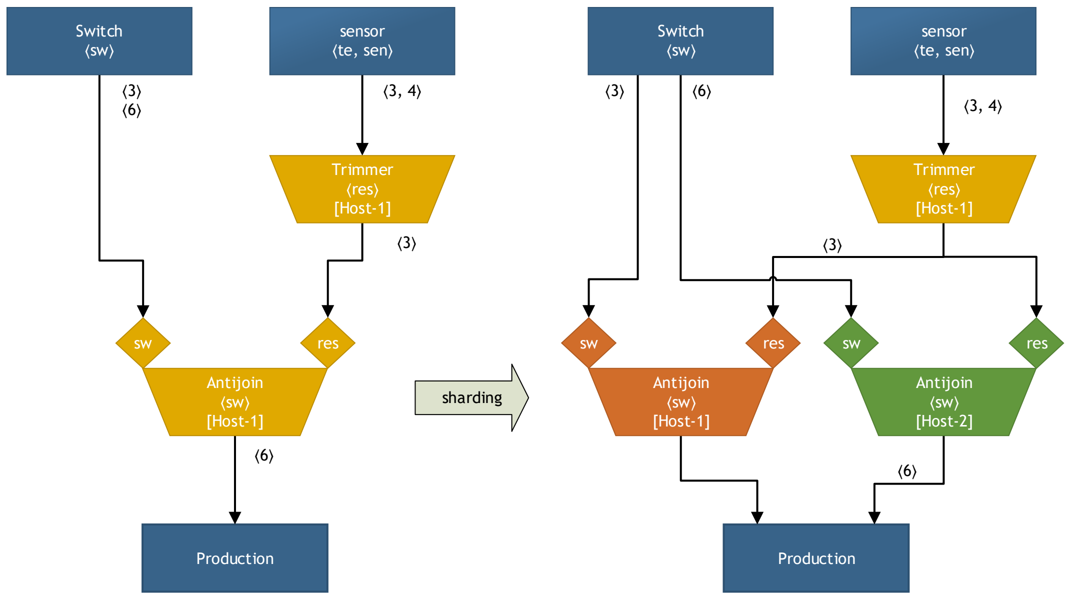


Figure 3.2. *Sharding of the SwitchSensor network*

3.2.1 Hash-Based Work Distribution

One way of splitting work between shards is to calculate a hash value for each tuple, and divide that number by the number of shards the node sends data to. Assuming the hash function is close to uniform, this method sends approximately the same amount of tuples to the shards.

To ensure that join and anti-join nodes work correctly, the hash function used by the parent node must assign equal values to matching tuples.

3.2.2 Message Fragmentation

Sending every tuple separately incurs significant overhead both in terms of object allocation and the size of the objects sent over the network. To alleviate this overhead, IncQuery-DS is capable of combining an arbitrary number of tuples into a change set message.

Setting this message size too large may cause problems due to the packet constraints of the operating system.

3.2.3 Termination

To retrieve consistent results from the production node after a transaction is finished, we must ensure that the change sets of the transaction propagated through every path of the network, and no changes from other transactions reach the production node. Due to the asynchronous nature of the algorithm, if we would not use a termination protocol, we would not know when to return results from the production node.

IncQuery-D uses a termination protocol that involves sending messages through every possible path. Every time a node is sharded into n nodes, the number of required messages is multiplied by n. This means that the algorithm scales exponentially with the number of shards, so for our implementation, a different termination protocol is used.

The protocol used in IncQuery-DS is as follows: Every physical node (unsharded nodes and node shards) receives the number of terminator messages it should expect during initialization. This number should be equal to the number of parent nodes. The terminatior message is only forwarded if the number of termination messages received for the current transaction reaches the set number (transaction termination point). This happens exactly once during each termination in every node.

The process can be seen in Figure 3.3, where Antijoin-A and Antijoin-B are the shards of the Antijoin in the previously shown Rete network (Figure 2.12). Antijoin-A and the trimmer already received all terminators, so they sent a terminator to their children. Antijoin-B and the production node only received 1 terminator and are waiting for 1 more. Because the production node has not received the expected terminators, it has not sent back results to the client.

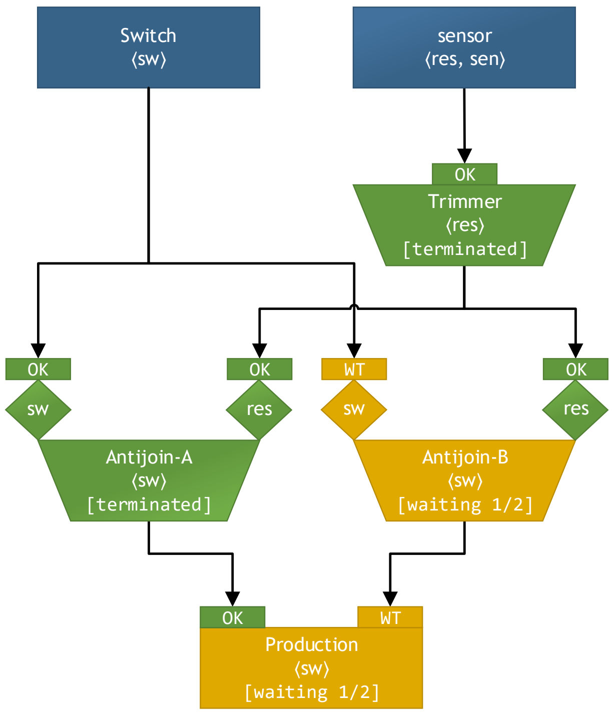


Figure 3.3. *Termination protocol*

When the termination is started, termination messages are sent to the Rete nodes directly connected to the input node (first nodes), and put a pause message in the message queue of each of these nodes. When this message is processed by the node, it suspends the processing of messages not related to the current termination. This ensures that messages from other transactions do not get to the production node before the results for the current transaction are returned.

Eventually the termination messages go through all paths and the production node returns the results and unpauses all paused nodes.

Proof of Correctness

No data reaches the production node from other transactions: The change sets queued after the pause messages are not processed until the production node unpauses the nodes, so the tuples stay there until after the production node yields results.

Every change set from the current transaction reaches the production node before it returns the results:

Starting with the production node, and structurally traversing the network backwards, we can prove that our assumption is correct.

- The production node only yields results when it receives every terminator from its parent nodes to it.
- A node only sends the terminator to its children once the transaction termination point is reached, meaning that every change set before them was processed and forwarded by the node, as the message queueing configuration makes the nodes process the received messages in the order they were received in.

The production node will eventually reach the transaction termination point:

- The first nodes reach the transaction termination point, because the termination messages are queued at the start of the termination reach the node after the change sets are processed. At this point the node sends the termination messages to its children.
- Every worker node will eventually send a termination messages, because the the number of its parent nodes(that terminate starting from the first worker nodes) equals the expected terminator count received during initialization.

3.3 Intelligent Sharding

The system described in this chapter makes the degree of sharding for each Rete node a separate decision - some nodes might have 100 shards, while others may remain unsharded. The decision affects the performance of the network greatly.

Sharding the network of SwitchSensor is trivial, as we can shard the only stateful node on as many machines as we have. If there are multiple stateful nodes, the problem becomes more challenging as their memory requirements may differ greatly. In this case, splitting every node is not the most efficient way of allocation. For example, in the network of RouteSensor, the Join-2 node consumes 3 times as much memory as Join-1 and Antijoin combined (Table 3.1). The data was obtained using the YourKit profiler [29].

Efficient node allocation is out of the scope of this report. It has been discussed extensively in [14].

Model size	64	128	256
Join-1	3.98	7.54	15.63
Join-2	56.23	107.71	220.79
Antijoin	12.43	23.80	48.83

Table 3.1. *RouteSensor per-node memory consumption [MB]*

3.3.1 Non-Uniform Work Distribution

IncQuery-DS is capable of supporting non-uniform work distribution, i.e. sending different amounts of tuples to the nod shards, so 1 node might receive 20% of the tuples while the other receives 80% of them. Using this method, fitting the nodes into memory across multiple machines becomes significantly easier.

Chapter 4

Evaluation

4.1 Benchmark Scenarios

The Train Benchmark (Section 2.3.4) defines a number of benchmark scenarios. The scenarios we compare the tools with are Batch and Repair [11].

4.1.1 Batch

In the Batch scenario a model is loaded from text-based storage (Turtle), then the model is validated.

4.1.2 Repair

In the Repair scenario, after an initial read and validation, a certain percentage of the model is modified and revalidated (Figure 4.1). This aims to simulate the workload of a user applying quick fixes to the model.

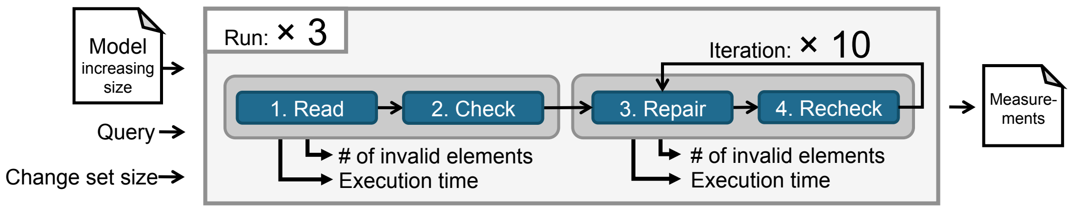


Figure 4.1. *Phases of the Repair Scenario*

4.2 Scaling on Multiple Machines

The main goal of the benchmarks is to confirm that node sharding allows the Rete network to scale for larger models than the unsharded variant.

4.2.1 Benchmark Setup

The benchmarks were run on 4 machines with the following setup:

- 2 cores of an Intel Xeon processor E5420 running at 2.50 GHz
- 8 Gigabytes of memory
- Ubuntu 14.04 LTS operating system
- Oracle JDK 8 runtime with 4 Gigabytes of heap memory
- Gigabit Ethernet network

The virtual machines were allocated on the cloud system of the Department of Measurements and Information Systems running Virtual Computing Lab.

4.2.2 Benchmark Results

This section presents the results of the benchmarks. The x axis shows the number of triples in the model and the y axis shows the time required for the run. Both axes use a logarithmic scale.

The benchmark included the following tools and configurations:

- Drools (Section 2.2.3)
- EMF-IncQuery (Section 2.2.1)
- Jena [2]
- Incquery-DS Local
- Incquery-DS Distributed
- Incquery-DS Distributed + Sharded

We included Jena, an RDF based SPARQL in-memory query engine, as a baseline for non-incremental solutions.

In the Local variant of IncQuery-DS, the worker nodes were allocated on a single machine.

Figure 4.2 shows the initial validation of the SwitchSensor constraint for the Repair scenario. The Sharded variant of IncQuery-DS split the Antijoin node of the Rete network (Figure 2.12) into 4 shards, each running on a different host machine.

Figure 4.3 shows the revalidation of the SwitchSensor constraint.

Table 4.1 shows the memory used by each antijoin node in the sharded SwitchSensor network.

Model size	64	128	256
Antijoin Shard-A	13.69	26.22	53.74
Antijoin Shard-B	13.59	26.14	53.52
Antijoin Shard-C	13.57	26.08	53.44
Antijoin Shard-D	13.56	25.79	52.94

Table 4.1. *SwitchSensor sharded memory usage [MB]*

Figure 4.5 shows the validation of the RouteSensor constraint.

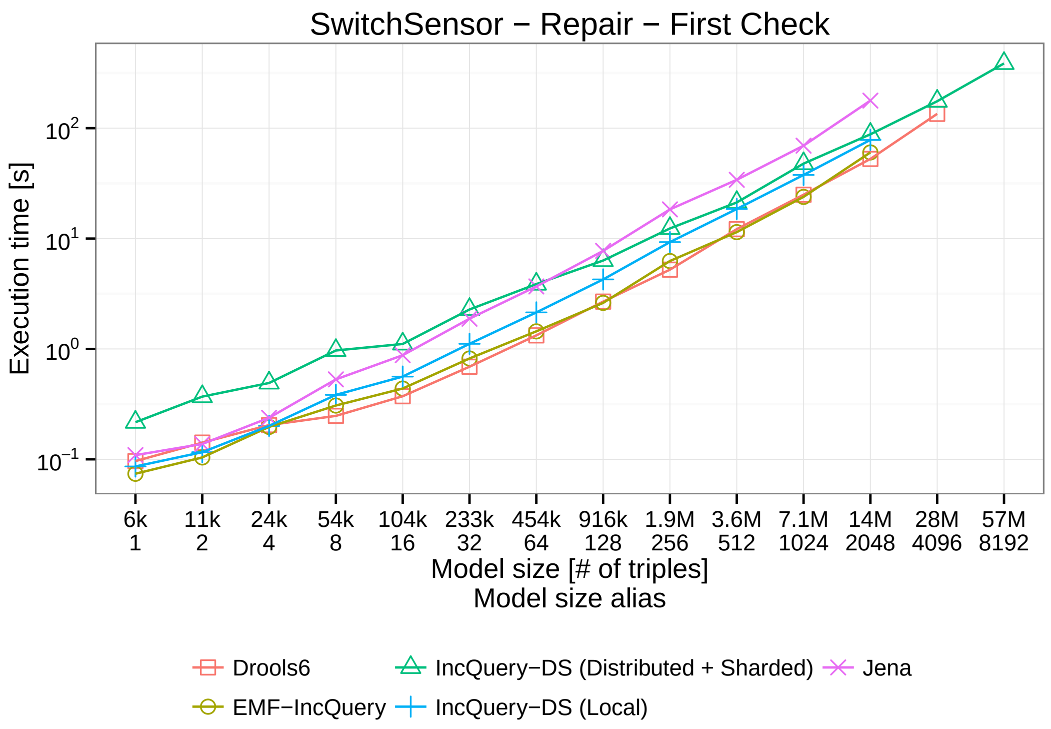


Figure 4.2. *SwitchSensor first check in the Repair scenario*

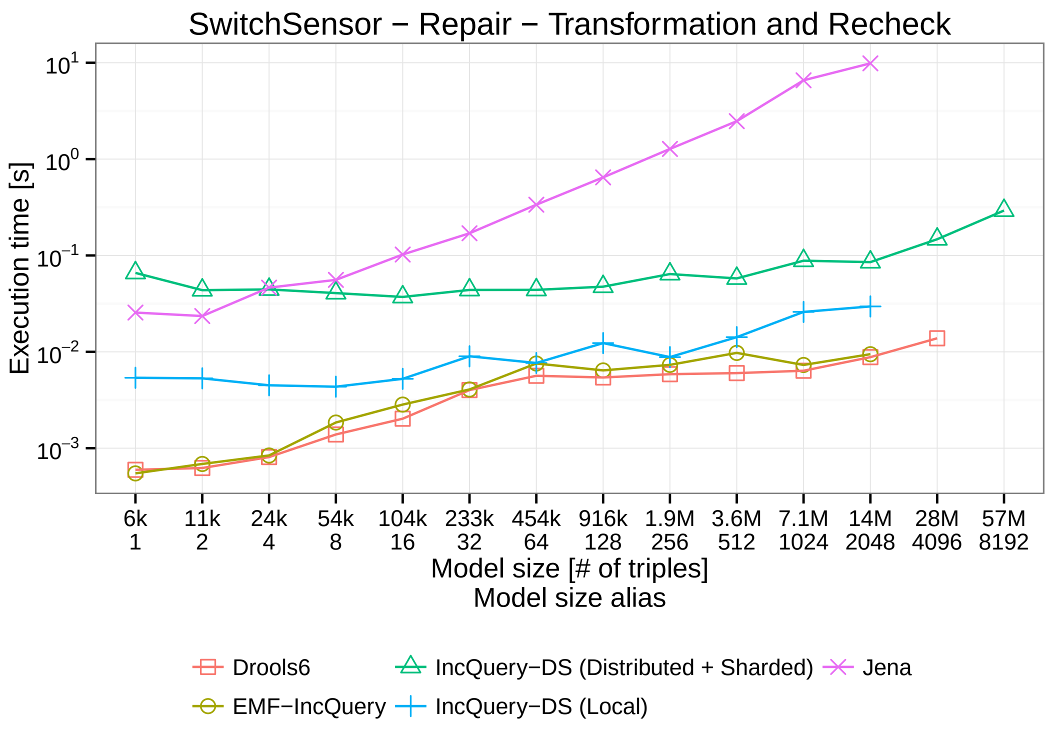
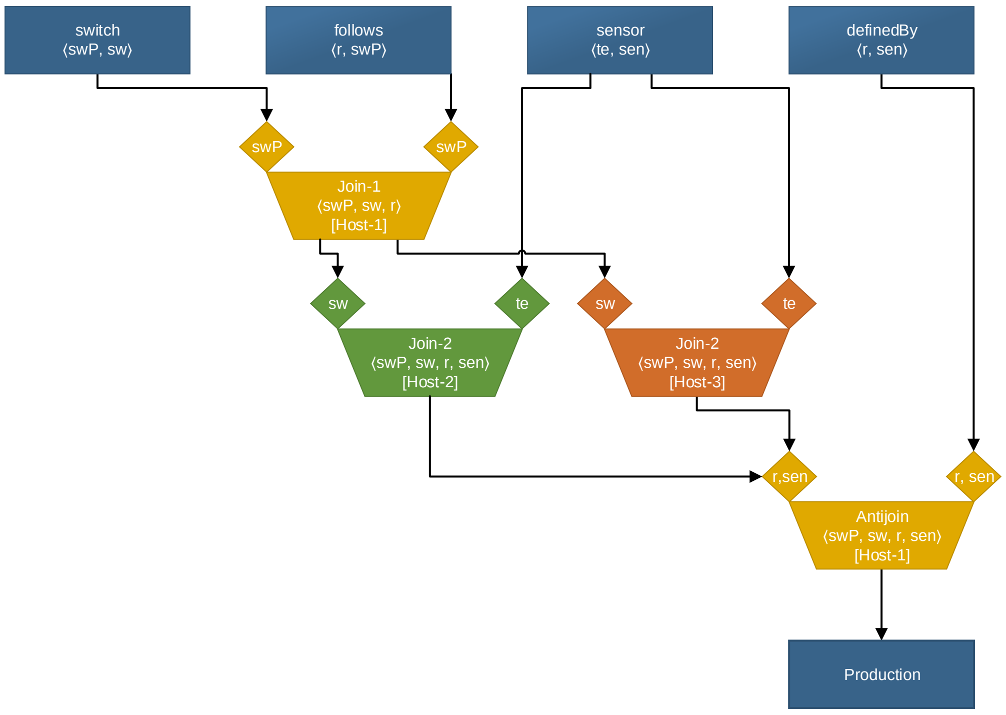


Figure 4.3. *SwitchSensor first check in the Repair scenario*



switch ⟨swP, sw⟩
follows ⟨r, swP⟩
sensor ⟨te, sen⟩
definedBy ⟨r, sen⟩
swP
swP
Join-1 ⟨swP, sw, r⟩ [Host-1]
sw
te
sw
te
Join-2 ⟨swP, sw, r, sen⟩ [Host-2]
Join-2 ⟨swP, sw, r, sen⟩ [Host-3]
r,sen
r, sen
Antijoin ⟨swP, sw, r, sen⟩ [Host-1]
Production

Figure 4.4. *Node allocation of the Sharded RouteSensor network*

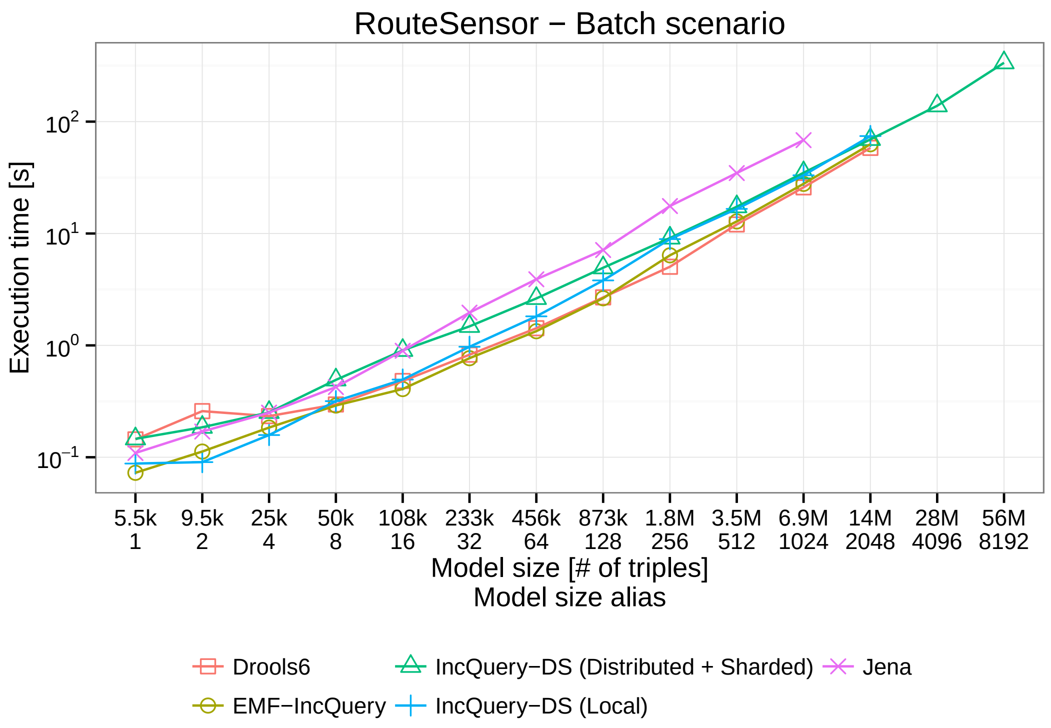


Figure 4.5. *RouteSensor Batch scenario*

Due to the differences in the memory requirement of the nodes (Table 3.1), Join-2 was split into 2 shards running on different host machines, and the Antijoin and Join-1 are allocated the same machine (Figure 4.4).

Table 4.2 shows the memory used by each node in the sharded RouteSensor network.

Model size	64	128	256
Join-1	3.98	7.54	15.63
Join-2 Shard-A	29.07	55.55	114.15
Join-2 Shard-B	27.188	52.23	106.79
Antijoin	11.459	21.92	44.97

Table 4.2. RouteSensor *sharded memory usage [MB]*

Model size	2048	4096	8192
Local	82.91	N/A	N/A
Distributed	61.04	222.94	N/A
Distributed+Sharded	69.34	138.59	335.80

Table 4.3. *RouteSensor Distributed compared to Split*

4.2.3 Analysis

Figure 4.2 shows that IncQuery-DS is slower on a single machine than EMF-IncQuery, but the Sharded variant can handle models 4 times as large, albeit using 3 additional machines. Because the memory usage is concentrated in a single node, the Distributed variant does not scale further than the Local variant.

In Table 4.1 we can see that we managed to partition the tuples uniformly amongst the nodes, so the hash based work distribution proved to be successful.

Figure 4.3 depicts the motivation for using incremental evaluation. Jena is 2 orders of magnitude slower than the incremental tools for large models. The network overhead of the Sharded variant of IncQuery-DS is also apparent compared to the Local variant, but the response time is still within the subsecond range. The number of modifications to the model scales linearly with the model size, so the increase in the response times is expected.

Figure 4.5 shows the execution times of RouteSensor contraint. We can see that EMF-IncQuery and the Local IncQuery-DS variant scale to the size 2048 model, so from this figure and Figure 4.2 we can conclude, that IncQuery-DS has approximately the same memory requirements for equivalent models as EMF-IncQuery. This figure also shows, that using 4 machines instead of 1 increases the size of the model IncQuery-DS is capable of evaluating by a factor of 4. The sharding and allocation was done manually as an approximation of a naive allocation algorithm. By using a sufficiently intelligent algorithm, the distribution could be further optimized.

Table 4.2 shows that the data was divided uniformly between the sharded nodes.

As Table 4.3 shows, the Sharded variant handles models twice as large compared to the Distributed one. The Local variant does not have sufficient memory to evaluate the size 4096 model (shown as N/A). The Distributed variant does not finish the processing of the size 8192 model (shown as N/A), because the system running the Join-2 node runs out of memory.

Although the results of the other queries in the Train Benchmark are not shown here, the prototype is capable of supporting them.

4.2.4 Threats to Validity

As the measurements were conducted in a cloud-based environment, there may have been transient workloads in effect. To mitigate the effect of these workloads, every measurement was run 3 times, and the figures show the minimum of the resulting execution times.

To confirm that IncQuery-DS yields the correct results, the output was thoroughly tested on various models.

To ensure the efficient operation of other tools, the implementations of the other tools (Drools, EMF-IncQuery, Jena) were reviewed by domain experts.

Chapter 5

Related work

This chapter discusses the frameworks related to distributed data processing, and graph query evaluation.

5.1 Distributed Data Processing Frameworks

The MapReduce paradigm [5] defines an abstraction for expressing simple computations while hiding the details of parallelization, data distribution and fault tolerance.

The Apache Hadoop framework [20] is an open-source implementation of a distributed file system and a tool capable of analysis and transformation of large datasets based on the MapReduce paradigm.

YARN (Yet Another Resource Negotiator) [26] introduces a way of decoupling the programming model from the resource allocation. This decoupling allows frameworks other than Hadoop to efficiently share a computational cluster.

Hadoop falls short when a working set of data is reused across multiple parallel operations. Spark [30] aims to mitigate this, while retaining the scalability and fault tolerance of MapReduce. Spark introduces an abstraction for the collection of objects partitioned across a set of machines, that can be rebuilt if a partition is lost.

5.2 Distributed Graph Frameworks

The Pregel [15] framework for large scale graph processing utilizes the BSP (Bulk Synchronous Parallel) model. Unfortunately this approach is not suitable for achieving immediate response times [13]. An open-source implementation is Apache Giraph [1], that is built on top of Apache Hadoop.

GraphX [9] is a graph processing framework built on top of Apache Spark. GraphX recasts graph-specific optimizations as distributed join optimizations and materialized view maintenance.

5.3 Incremental Graph Query Engines

Various incremental graph query engines were mentioned in Section 2.2. In addition, paper [21] introduces a Rete-based distributed query engine, but the paper does not present any benchmark results on the performance of the solution and the implementation is not publicly available.

Chapter 6

Conclusion

This chapter summarizes the report and presents ideas for future elaboration.

In this report we designed and evaluated a truly scalable incremental query evaluation framework prototype. Although the performance was only measured on MDE workloads the approach should perform well in different environments for similar workloads.

6.1 Scientific Accomplishments

- I designed the architecture of a distributed Rete-based query engine capable of node sharding.
- I modified the termination algorithm of IncQuery-D to avoid exponential scaling due to the sharding.

6.2 Practical Accomplishments

- I developed a distributed Rete-based query engine, that makes it possible to shard the worker nodes to multiple machines, using the Akka framework.
- I tested my code with unit and integration tests. The Scala code including tests amounts to around 2300 lines.[1]
- I integrated my tool with the Train Benchmark framework. The integration took 1100 lines of Java code including tests.[2]
- I carried out benchmarks to analyze the performance of the query engine and confirm that its scalability is beyond previous solutions.

6.3 Limitations and Future Work

In the current implementation of IncQuery-DS the nodes are allocated manually. This will be mitigated in the near future as IncQuery-DS will be integrated to IncQuery-D. This will benefit both systems, as it allows IncQuery-D to shard nodes, and relieves

[1] `https://github.com/FTSRG/incquery-ds`

[2] `https://github.com/FTSRG/trainbenchmark`

IncQuery-DS of having to maintain a plethora of tools required for Rete network creation and management.

Using more advanced algorithms inside the nodes, for example sort-merge join, may increase both the capacity and the performance depending on the hardware [3].

The performance of the algorithms may be different in various phases of the evaluation, e.g. it may be beneficial to use a different join algorithm for the initialization and the maintenance of the Rete network.

Acknowledgements

I would like to extend my deepest gratitude towards *Gábor Szárnyas*, *Dr. István Ráth*, and *Dr. Gábor Bergmann* for their advice and support.

I would also like to thank Cseh Dávid and the Department of Measurements and Information Systems for providing me with the virtual machines required for the evaluation of IncQuery-DS.

List of Figures

Bibliography

[1] Apache. Giraph(October 12, 2015). `http://giraph.apache.org/`.

[2] Apache Software Foundation. Apache Jena(October 11, 2015). `https://jena.apache.org/`.

[3] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proc. VLDB Endow.*, 7(1):85–96, September 2013.

[4] Stein Dániel. Incremental Static Analysis of Large Source Code Repositories. Bachelor's thesis, Budapest University of Technology and Economics, Budapest, 2014.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[6] Dean Allenmang and James Hendler. *Semantic Web for the Working Ontologist.* Morgan Kaufmann, 2011.

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

[8] Szárnyas Gábor, Semeráth Oszkár, Ráth István, and Varró Dániel. The TTC 2015 Train Benchmark Case for Incremental Model Validation. In *Transformation Tool Contest*, 07/2015 2015.

[9] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of OSDI*, pages 599–613, 2014.

[10] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally (extended abstract). In *In: Proc. of the International Conf. on Management of Data, ACM*, pages 157–166, 1993.

[11] Benedek Izsó, Gábor Szárnyas, and István Ráth. Train Benchmark. Technical report, Budapest University of Technology and Economics, 2014.

[12] Dimitrios S. Kolovos, Louis M. Rose, Nicholas Matragkas, Richard F. Paige, Esther Guerra, Jesús Sánchez Cuadrado, Juan De Lara, István Ráth, Dániel Varró, Massimo Tisi, and Jordi 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.

[13] Christian Krause, Matthias Tichy, and Holger Giese. Implementing Graph Transformations in the Bulk Synchronous Parallel Model. In Stefania Gnesi and Arend Rensink, editors, *Fundamental Approaches to Software Engineering*, volume 8411 of *Lecture Notes in Computer Science*, pages 325–339. Springer Berlin Heidelberg, 2014.

[14] József Makai, Gábor Szárnyas, Ákos Horváth, István Ráth, and Dániel Varró. Optimization of Incremental Queries in the Cloud. In *CloudMDE*, 2015.

[15] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

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

[17] MONDO Project. MONDO Project(October 15. `http://www.mondo-project.org/`.

[18] Odersky, Martin and Spoon, Lex and Venners, Bill. *Programming in Scala: A Comprehensive Step-by-step Guide.* Artima Incorporation, 1st edition, 2008.

[19] Mikko Rinne. SPARQL update for complex event processing. In *ISWC'12*, volume 7650 of *LNCS*. 2012.

[20] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[21] Yan Shvartzshnaider, Maximilian Ott, and David Levy. Publish/Subscribe on Top of DHT Using RETE Algorithm. In *FIS*, pages 20–29. Springer, 2010.

[22] Subramaniam, Venkat. *Programming Concurrency on the Jvm : Mastering Synchronization, Stm, and Actors.* Pragmatic Bookshelf, Dallas, 2011.

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

[24] Gábor Szárnyas, Benedek Izsó, István Ráth, Dénes Harmath, Gábor Bergmann, and Dániel 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, MODELS 2014*, Valencia, Spain, 2014. Springer, Springer. Acceptance rate: 26%.

[25] Typesafe Inc. Akka(October 10, 2015). `http://akka.io/`.

[26] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[27] W3C. RDF - Semantic Web Standards(October 10, 2015). `http://www.w3.org/RDF/`.

[28] W3C. Turtle - Terse RDF Triple Language(October 6, 2015). `http://www.w3.org/TeamSubmission/turtle`.

[29] Yourkit. Java Profiler - .NET Profiler - The profilers for Java and .NET professionals. `https://www.yourkit.com/`.

[30] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.