

# How the Rete Algorithm Works

By [Carole-Ann Berlioz](#)

on March 14, 2011

[business rules](#) | [rete algorithm](#)



## The Rete Algorithm Series: Part 2

In part 1 of this blog series, I covered [the Rete Algorithm and its origin](#) and even [the origin of its name](#). Now as promised, I am going to explain how it works. The Rete Algorithm is one of several decision engine algorithms used in modern rule engines such as [Sparkling Logic SMARTS™](#). See our recent blog post on [Decision Engine Performance](#) to learn more about decision performance and the decision engines that drive today's automated decisions.

Explaining how the Rete algorithm works simply can be challenging, so I'll do my best to not lose our less technical audience and yet keep things interesting enough for the techies. I hope I will reach that goal. And without further ado, let me introduce the Rete algorithm!

## How it Works

[Dr. Charles Forgey](#) developed the brilliant idea to decouple the evaluation of hypothesis from execution sequencing.

You may be wondering why that idea is brilliant. Answer: When you need to assess large volumes of business rules against facts or data, the constant screening for applicability can be costly — both in terms of evaluation (and re-evaluation) and in terms of ordering the statements properly. This innovative approach for inference allows savings in both areas. Let's explore how.

Traditionally, before this algorithm and derivatives, you would need to sequence and nest the IF-THEN-ELSE rule statements in such a way that you capitalize on the test evaluations. And you would reuse them as much as possible. When you think about it, it would be akin to creating the leanest decision tree possible, in a graphical form or more likely in straight code. This “sequential” approach breaks (mostly in terms of performance) when rules need to execute as a result of the execution of other rules. This is what we call inference.

In a sequential approach, you would have to restart the complete ruleset evaluation after each rule execution (if rules are prioritized) or after a full pass through the ruleset (in absence of prioritization). Keep in mind too that prioritization might prevent you from reusing test evaluations all together in a sequential approach. If the conditions are not co-located in the logic, then you will likely have to perform those tests multiple times. Systems have become much more sophisticated than they used to be back then. Most BRMS products provide a sequential deployment option which is still pretty good when inference is not needed. The beauty of those products is that you can still manage the rules independently of each other. The sequential engine takes care of optimizing the generated spaghetti code.

## **Airline Example**

For illustration, I'll use a simple set of rules that we are mostly familiar with: airline reward miles calculations. The below decision service is responsible for processing the frequent flyer's activity:

IF award miles for last year or current year > 25,000, THEN status = Silver.

IF award miles for last year or current year > 100,000, THEN status = Gold.

IF flight is less than 500 miles, THEN award 500 miles.

IF flight is 500 miles or more, THEN award flight miles.

IF category is business or first, THEN award 50% bonus miles.

IF status is Gold and airline is not partner, THEN award 100% bonus miles.

IF status is Silver and airline is not partner, THEN award 20% bonus miles.

IF member signed up for 3-flights-for-5k-in-March and number of return flights in March 2011 = 3, THEN award 5,000 additional miles.

IF status becomes Gold, THEN award 8 upgrade certificates.

Rewards can include hotel or rental car bonuses, incentive programs, lifetime members, accelerator programs, etc. I will leave it up to the creative minds of the Airline marketers to add more promotions.

This example does not use a lot of inference but has enough to illustrate our point. If you run through the rules sequentially in the current order and a Frequent Flyer member reaches Gold status in that very transaction, you would need to start over: you would need to reprocess all the rules to update the status and award the proper bonus miles and other incentives.

Inference engines assess all the applicable rules and then fire the first applicable rule, propagate the changes and fire again. How does that work?

## Constructing the Rete Network

The Rete network is the heart of the Rete algorithm. It is made of nodes that each hold a list of objects that satisfy the associated condition. The original Rete algorithm worked out of facts but I will simplify the description to refer to objects since all commercial engines have evolved to be object-oriented nowadays.

The Rete network is constructed when you compile a rules project — only once when you start that service (or class for simpler designs) — and then shared across all invocations.

The discrimination tree is the first part of the Rete network. It starts with Alpha nodes that are associated with Classes (in the object-oriented sense). All instances of a given class will be listed in any given Alpha node. The discrimination happens by adding conditions as single nodes attached to the Alpha node or to another parent node.

### Constructing the Rete Network — Airline Example

Let's review what that means for our Airline example:

Alpha nodes are created for each class: Frequent Flyer Account and Flight.

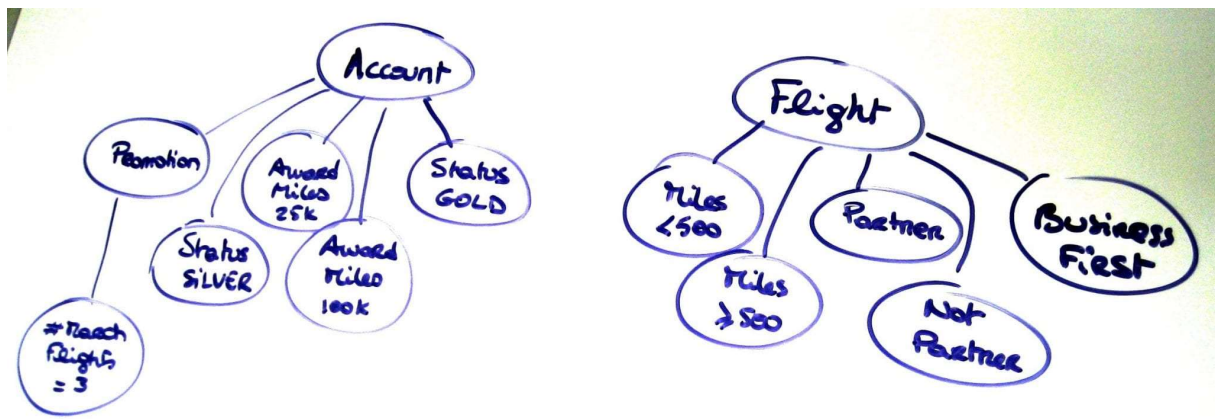


Conditions are then appended:

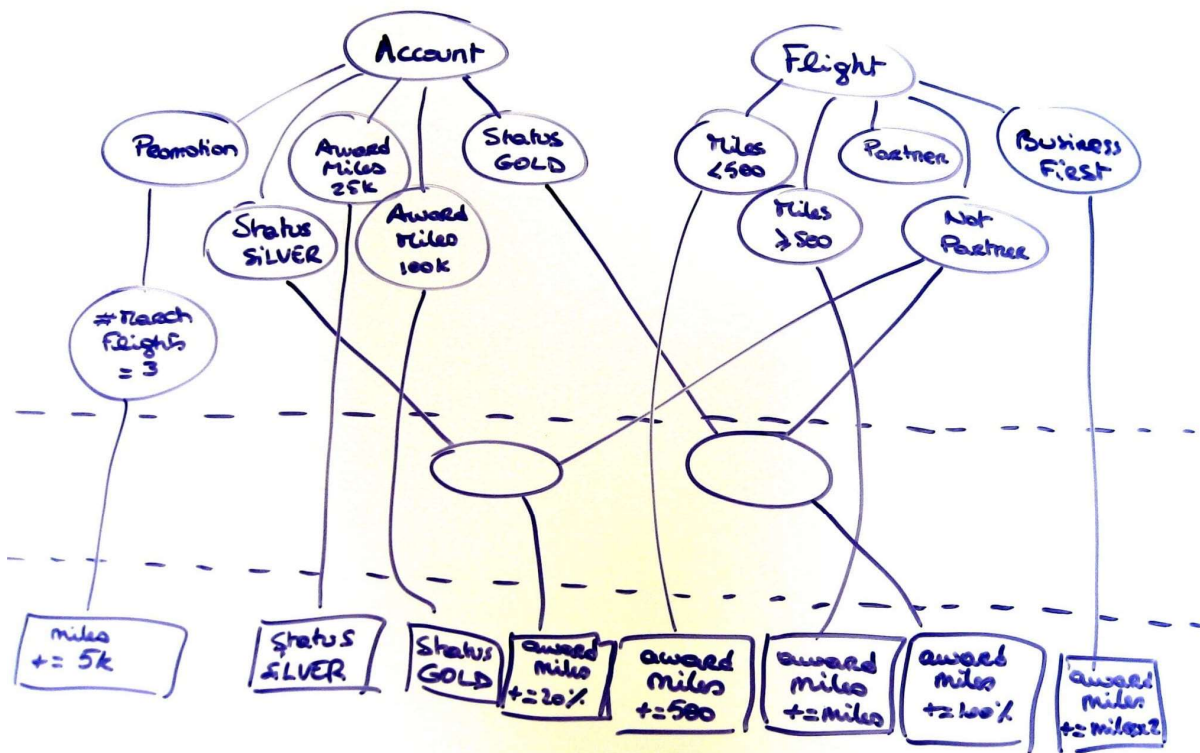
- Frequent Flyer
  - status is Gold
  - status is Silver
  - award miles for last year or current year > 25k
  - award miles for last year or current year > 100k
  - signed up for “3-flights-for-5k-in-March” promotion
    - number of flights in March = 3
- Flight
  - miles >= 500
  - miles < 500
  - airline is not a partner
  - airline is partner
  - category is business or first

etc.

Each node represents an additional test to the series of conditions applied upstream. If you follow a path from top to bottom, you should be able to read the complete set of conditions THAT APPLY TO ONE GIVEN CLASS OF OBJECTS.



Finally the nodes are connected across classes. This is where we can combine the conditions for a non-partner flight taken by a Gold member. We call those “joints” as we will combine the list of objects that verify conditions on one branch with the list of objects that verify the conditions on another branch.



On the performance side, you may have been warned in the past not to combine too many patterns in a single rule. It is clear looking under the hood that the product of all possible accounts by all possible flights could result into a combinatorial explosion. The more discrimination upfront, the better obviously.

## All Paths Lead to Actions

The path eventually ends with the action part of the rules. The content of the actions is irrelevant for the Rete network. you could replace the labels here with the name of the rule (I did not name the rules in our example so I displayed the actual actions). The rules can be reconstituted by following the incoming paths. All nodes are AND-ed. This is an interesting

point. ORs are typically not natively supported: rules are duplicated for each OR-ed flavor. If you were to add a rule to grant the same bonus for SILVER or GOLD customers traveling to Hawaii then you would just have nodes connected to the existing GOLD and SILVER nodes as if they were written as separate rules. In terms of performance or maintenance, it does not matter though since the Rete algorithm handles the supposed duplication as efficiently as any manual code would.

It is a great time to emphasize the great savings you would get when the number of rules increases. Let's say that you want to introduce a new promotion for GOLD customers to/from specific destinations or for reached milestones (50k and 75k award miles). The nodes that test for GOLD status do not need to be created or duplicated. The test as well as the execution of the test is leveraged and reused transparently regardless of the order in which the rules need to be executed.

## **Rete Cycle: Evaluate**

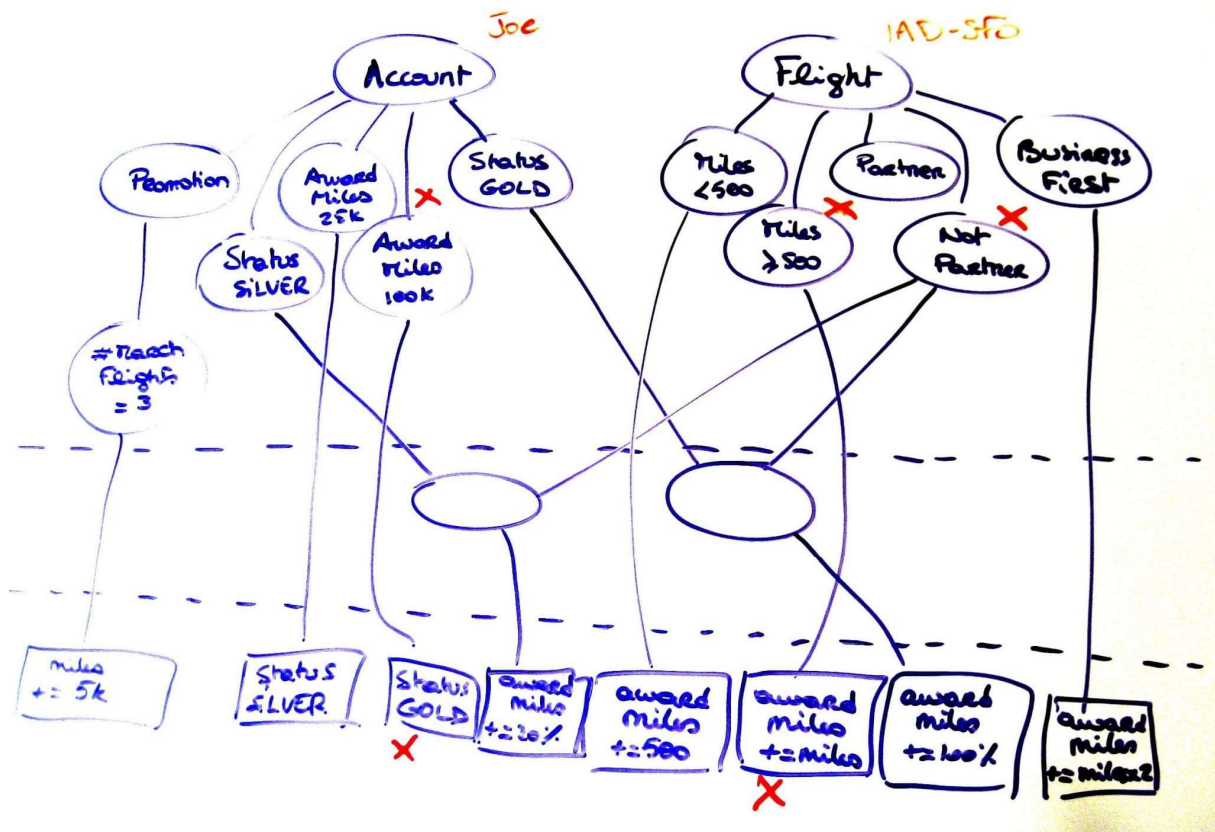
Let's look now at what happens at runtime when you invoke that service with customer data. Your design will dictate whether those transactions are applied "real-time" (or on-demand) or whether you want to execute a batch of transactions at once, at night when you have access to your Mainframe window for example. Rules can actually be used either way and in many projects, they are actually used both ways by two different systems. For an Insurance underwriting project, you might deploy the same rules online in your self-service website as well as your older batch application that processes the applications coming from your brokers.

Let's assume that we are processing transactions one by one for simplicity.

The evaluation phase consist in running the data through the Rete network to identify the applicable rules, for which all conditions are satisfied. The nodes in the network will hold lists of objects that satisfy the conditions in the incoming path.

## **Rete Cycle: Evaluate — Airline Example**

In our airline example, Joe flies from Washington DC (IAD) to San Francisco (SFO). Flight accounts for 2,419 miles I recall. Starting with already 150k award miles, Joe should bank double miles (4,838) for his GOLD status. How does Rete propagates those facts to make the same decision?



The Account Alpha node references Joe as a Frequent Flyer. Because of his 150k award miles, he also qualifies for the “> 100k” condition. He is referenced in this node too. My handwriting did not allow me to write down Joe in the small real estate so I just highlighted the node with a X. In reality, Joe would be referenced specifically as we need to keep track of who qualifies in case we process multiple objects at once. For this exercise, I also assume that service does not store the GOLD status and computes it every time so the associated node still has an empty list.

Similarly, the IAD-SFO flight is referenced in the list of all flights in the transaction. The flight is more than 500 miles and on the airline. The associated lists reference the flight accordingly.

At that point in time we do not have any joint to worry about since the account status is not known yet to qualify for GOLD.

2 rules are satisfied. All facts have been propagated. We are ready for execution.

## Rete Cycle: Execute

The rules for which all conditions are satisfied are said to be active in the agenda. The agenda contains the list of all rules that should be executed, along with the list of objects that are responsible for the conditions to be true. I insist on the word “SHOULD”. The agenda will sort them according to their priorities (and other conflict resolution methods). If the execution of a rules invalidates a rule with a lower priority, then this second rule will logically never execute. This would have been the case if Joe had started with 99k award miles – he would have qualified for the SILVER bonus until we bank the IAD-SFO miles and granted him GOLD status. Joe can qualify for SILVER or GOLD bonus miles but certainly not both.



Priorities are a touchy subject in the Rules community as some purists are set against its use.

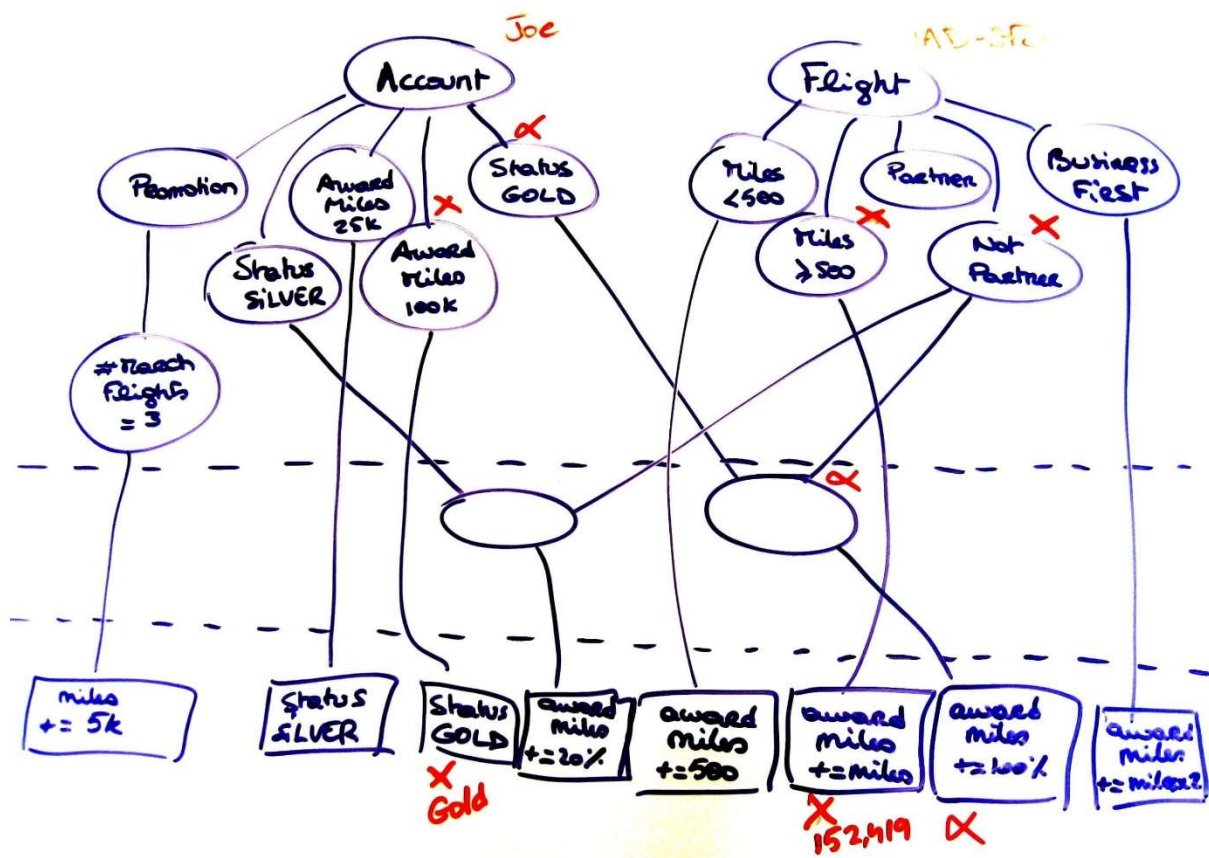
The agenda lists the 2 rules (GOLD status and 500+ Flight Miles). The first one is executed. In our airline example, order does not matter for those 2 rules. Let's assume that flight miles are awarded first. Joe gets 152,419 miles in his account so far.

## The Results

After the execution of the first rule, facts are propagated again. Flights have not changed so none of that subtree is re-evaluated. Joe's account has changed so the conditions related to award miles are re-assessed, with no change in our example. The GOLD status rule remains active in the agenda.

We are ready to execute the following rule: GOLD Status which is the only rule left in the agenda.

As a result, the GOLD Status node is updated for Joe, which leads to the GOLD status and not partner node to update as well, which leads to the 100% GOLD bonus rule to become activated in the agenda.



In the end, Joe gets the proper credit of 154,839 in his award miles account.

## Rete Cycle: Repeat & Reuse

As you add more rules that apply to GOLD or SILVER customers, the RETE network will grow always reusing the same nodes as much as possible — which is the part that takes time to do properly by hand if you are still coding it manually. Those spaghetti in the “Joint” part of the algorithm are a nightmare to maintain properly and can cost you significant debug time. This trivial example can become quickly messy when you consider the many different ways you can qualify for status with eligible segments rather than eligible miles, with accelerators, with lifetime miles, as well as the incentive promotions that you might get by default and those that you need to sign up for. Reusing those conditions — and their execution — regardless of when and where they show up is a performance boost that largely compensates for the initial overhead when inference is needed.

When you don’t need inference, you only go through one cycle of propagation. That’s debatable of course.

## Conclusion

In my opinion, Charles opened the door for an algorithmic solution to properly ensuring the enforcement of many rules in an uncertain order. In other words, he created an approach where very large volumes of rules could be assessed in a short time. This approach is much faster than any other algorithms when rules need to be executed repeatedly on a given data set. However, Non-Rete vendors have marketed the term “Rete Wall” to refer to the potential performance limits of the Rete algorithm. Forgy has since released further performance improvements: Rete II, Rete III and lately in Rete-NT. These updates have also changed the initial “ideal” characteristics of a Rete project to better support much larger data sets.

It is somewhat surprising that no new algorithm break-through (other than Charles’s) were made since then. The basis for Business Rules execution is vastly based on Rete, or sequential, but hardly anything else. It reminds me of the sharks and their evolution. I remember learning, in my life science days, that sharks have not evolved at all in a very long time. This is likely because they are already perfectly adapted to their environment.

Editor’s Note: This post was originally published in March 2011 and has been updated for accuracy and comprehensiveness.

[\*PART THREE: Evolution of the Rete Algorithm\*](#)

Learn more about [Decision Management](#) and [Sparkling Logic’s SMARTS™ Data-Powered Decision Manager](#)

- [NinjaHoldings Revolutionizing FinTech Solutions for the Underbanked through SMARTS™](#)
- [How to Keep Your Pricing Engine Running Smooth](#)
- [How to Write Business Rules](#)
- [Future-Proof Business Rules Management](#)
- [Rete-NT Inference Rules Engine — A Closer Look](#)



## **ABOUT US**

Sparkling Logic Inc. is a Silicon Valley-based company dedicated to helping organizations automate and optimize key decisions in daily business operations and customer interactions in a low-code, no-code environment. Our core product, SMARTS™ Data-Powered Decision Manager, is an all-in-one decision management platform designed for business analysts to quickly automate and continuously optimize complex operational decisions. Learn more by requesting a live demo or free trial today.