# Tuning SPARQL queries in Neptune using explain and query hints

We heard you! Motivated by the discussions we had with our customers, we extended Neptune’s RDF/SPARQL stack by two new features that make it easy to understand and tune SPARQL queries.

First, we added [query explain](https://docs.aws.amazon.com/neptune/latest/userguide/sparql-explain.html) capabilities allowing you to plot execution plans visualizing the logical evaluation flow, also capturing dynamics during query evaluation such as the number of intermediate solutions flowing through the individual solutions – these capabilities can help you understand the “hard parts” in query evaluation and identify queries with a suboptimal processing strategy. Making these findings actionable, [query optimizer hints](https://docs.aws.amazon.com/neptune/latest/userguide/sparql-query-hints.html) can be used to manually improve such queries by hand tuning evaluation aspects such as join and evaluation order.

## Scenario and setup

In this blog post, we will illustrate both features by sample queries over the AirRoutes dataset, which contains information about airports and their connections. The RDF version of the dataset is publicly available in S3 at

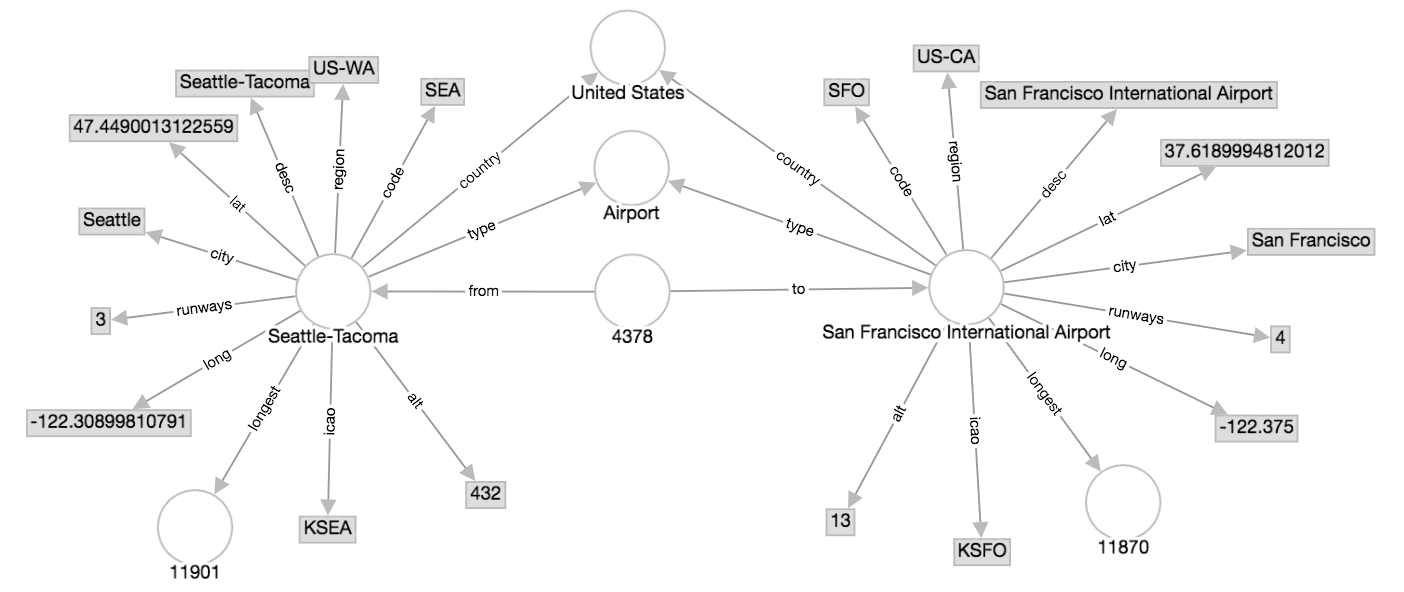
**s3://aws-neptune-customer-samples-[REGION]/bulkload-datasets/turtle/airroutes/v01 /**

, where [REGION] is the region identifier of your Neptune cluster (e.g., *us-east-1*). If you want to play with the examples in the blog post while reading, you can either load the dataset into one of your Neptune instances using [Neptune’s data loader](https://docs.aws.amazon.com/neptune/latest/userguide/bulk-load.html)[[1]](#footnote-1) or, even more conveniently, we have prepared a CloudFormation stack that sets up a small stack containing Neptune + SageMaker, running a Jupyter notebook containing all the example queries discussed in the following (please be aware that costs will apply for the infrastructure during the time the stack is running). To set up the stack:

* Log in to your AWS and visit the following region-specific link, for the region of your choice
* Choose a name for you stack, such as “SPARQLExplainBlogPost”, check the two boxes at the bottom of the page allowing CloudFormation to … Confirm with … This will set up a fresh NeptuneInstance (r4.large), a SageMaker stack containing a Jupyter notebook tied to the Neptune instance, and required infrastructure (such as a VPC, data loader role, etc.).
* Once your stack is ready, open SageMaker, choose *Notebook Instance* on the left, then click *Open Jupyter*. Select *Neptune* -> *scripts* -> *SPARQLExplainAndQueryHints.ipy*.
* Once your script has opened, you can play through the examples by executing the cells in sequence using the “play” button at the top. If you want to change examples, you may of course modify existing or create new cells, re-execute them, etc.

Please don’t forget to shut down the CloudFormation stack to clean up deployed infrastructure resources once you are done playing with it.

The basic schema of the AirRoutes data is illustrated in the Figure below, by example of two directly connected Airports, *Seattle Tacoma* and *San Francisco International Airport*. In this case, the two airports are connected via a flight route (here identified by the node “4378”), and each airport is described by a number of properties such as code, city, latitude and longitude, the country in which the airport is located.



## A sample query

As a running example, we consider a simple SPARQL query that extracts all possible stop locations for one-stop connections between Frankfurt airport (**airport:FRA**) and Seattle airport (**airport:SEA**). The following bash script (with variables $NEPTUNE\_CLUSTER\_ENDPOINT and $NEPTUNE\_CLUSTER\_PORT pointing to the endpoint and port of a cluster in which the AirRoutes data set has been loaded) allows us to send this query to the Neptune server:

**query1.sparql**

**PREFIX prop: <http://kelvinlawrence.net/air-routes/vocab/prop#>**

**PREFIX airport: <http://kelvinlawrence.net/air-routes/data/airport#>**

**SELECT DISTINCT ?via WHERE {**

**?route1 prop:from airport:FRA .**

**?route1 prop:to ?via .**

**?route2 prop:from ?via .**

**?route2 prop:to airport:SEA .**

**}**

**curl -s** [**http://$NEPTUNE\_CLUSTER\_ENDPOINT:$NEPTUNE\_CLUSTER\_PORT/sparql**](http://$NEPTUNE_CLUSTER_ENDPOINT:$NEPTUNE_CLUSTER_PORT/sparql\) **\**

**-d "@query1.sparql" \**

**-H "Content-type: application/sparql-query" \**

**-H "Accept: text/csv"**

The query defines two airports, *?airportFrom* and *?airportTo*, which are bound to Frankfurt and Seattle airport via their respective codes. The query then checks for a route *?route1* starting from *?airportFrom* (i.e., Seattle), ending at an intermediate stop identified by variable *?via*, and a route *?route2* starting at the intermediate stop *?via* and ending at *?airportTo* (i.e., Frankfurt).

When executing the script over the AirRoutes dataset, the query will report 45 airport identifiers, reflecting all the 45 possible ways of getting from Frankfurt to Seattle with a single stop (in the URI scheme implemented in the AirRoutes dataset, the last three letters in the identifiers represent the airport codes):

via

<http://kelvinlawrence.net/air-routes/data/airport#TPE>

<http://kelvinlawrence.net/air-routes/data/airport#YVR>

<http://kelvinlawrence.net/air-routes/data/airport#KEF>

<http://kelvinlawrence.net/air-routes/data/airport#HND>

<http://kelvinlawrence.net/air-routes/data/airport#YYC>

… (40 more)

## Understanding the query’s runtime behavior

When running the query, the SPARQL query optimizer will generate a query plan based on available statistics and heuristics. One crucial task of the optimizer is choosing an effective order in which the triple patterns inside the query are evaluated; in doing so, it tries to exploit both distributions in the underlying as well as structural constraints such as the connectivity of variables of variables in the triple patterns. Typically, when evaluating join graphs, it will choose an order that differs from the order given in the query (unless this order is the same as the estimated best order). The nice thing about this automatic reordering is that, in many cases the query developer does not need to think about the best order of evaluating the query – for the case of triple patterns, any order will lead to the same result.

In some cases, however, you may want to gain more insights into the evaluation order of triple patterns (and, more generally, the execution plan) that has been chosen by the optimizer. This is where the new SPARQL explain feature comes into play, as it allows you to inspect the generated evaluation plan, to understand this order. Getting the query explain is as easy as adding an additional parameter “explain=<MODE>” to the HTTP request (in addition, we change the Accept header from text/csv to text/plain, in order to obtain a human-readable format of the explain output, rather than a CSV serialization).

*Hint: using content negotiation you can switch between different serializations of the explain output. In this blog post, we always use text/plain, which generates an ASCII based serialization, Neptune also supports HTML based output and a CSV serialization (which can conveniently be pasted into spreadsheets for further investigation). See the* [*documentation*](https://docs.aws.amazon.com/neptune/latest/userguide/sparql-explain-examples.html) *for more details.*

**curl -s** [**http://$NEPTUNE\_CLUSTER\_ENDPOINT:$NEPTUNE\_CLUSTER\_PORT/sparql**](http://$NEPTUNE_CLUSTER_ENDPOINT:$NEPTUNE_CLUSTER_PORT/sparql\)**?explain=dynamic \**

**-d "@query1.sparql" \**

**-H "Content-type: application/sparql-query" \**

**-H "Accept: text/plain"**

**╔════╤════════╤════════╤═══════════════════╤═══════════════════════════════════════════════════╤══════════╤══════════╤═══════════╤════════╗**

**║ ID │ Out #1 │ Out #2 │ Name │ Arguments │ Mode │ Units In │ Units Out │ Ratio ║**

**╠════╪════════╪════════╪═══════════════════╪═══════════════════════════════════════════════════╪══════════╪══════════╪═══════════╪════════╣**

**║ 0 │ 1 │ - │ SolutionInjection │ solutions=[{}] │ - │ 0 │ 1 │ 0.00 ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 1 │ 2 │ - │ PipelineJoin │ pattern=distinct(?route2, prop:to, airport:SEA) │ - │ 1 │ 118 │ 118.00 ║**

**║ │ │ │ │ joinType=join │ │ │ │ ║**

**║ │ │ │ │ joinProjectionVars=[?route2] │ │ │ │ ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 2 │ 3 │ - │ PipelineJoin │ pattern=distinct(?route2, prop:from, ?via) │ - │ 118 │ 118 │ 1.00 ║**

**║ │ │ │ │ joinType=join │ │ │ │ ║**

**║ │ │ │ │ joinProjectionVars=[?route2, ?via] │ │ │ │ ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 3 │ 4 │ - │ PipelineJoin │ pattern=distinct(?route1, prop:to, ?via) │ - │ 118 │ 10030 │ 85.00 ║**

**║ │ │ │ │ joinType=join │ │ │ │ ║**

**║ │ │ │ │ joinProjectionVars=[?route1, ?via] │ │ │ │ ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 4 │ 5 │ - │ PipelineJoin │ pattern=distinct(?route1, prop:from, airport:FRA) │ - │ 10030 │ 45 │ 0.00 ║**

**║ │ │ │ │ joinType=join │ │ │ │ ║**

**║ │ │ │ │ joinProjectionVars=[?route1] │ │ │ │ ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 5 │ 6 │ - │ Distinct │ vars=[?via] │ - │ 45 │ 45 │ 1.00 ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 6 │ 7 │ - │ Projection │ vars=[?via] │ retain │ 45 │ 45 │ 1.00 ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 7 │ - │ - │ TermResolution │ vars=[?via] │ id2value │ 45 │ 45 │ 1.00 ║**

**╚════╧════════╧════════╧═══════════════════╧═══════════════════════════════════════════════════╧══════════╧══════════╧═══════════╧════════╝**

The explain mode *dynamic* indicates that we’re also interested in dynamic aspects of the evaluation, namely the number of solutions flowing through the plan at runtime (whereas “static” prints a limited view that just summarizes the structure of the plan). When re-executing the query above with the explain parameter set, we obtain the following output:

Neptune query plans can be understood as a pipeline of operators, where operators receive solutions from one or more downstream operator and forward solutions to (at most two) upstream operators. In our examples, starts with a *SolutionInjection* operator, followed by a sequence of *PipelineJoin* operations, one for each triple pattern in the query, and finally computes a distinct, projection, and a term resolution operation. The columns *Out #1* and *Out #2* describe how operators are connected to each other; they contain the IDs of the operators to which the given operator forwards its output. In our example, the operator pipeline is perfectly linear: each operator forwards to the next operator. But for more complex queries, Neptune may choose non-linear plans (an example is a *Tee* operator, which forward results into subplans representing the different parts of a SPARQL UNION).

Going into more details, here is how the plan above works in detail (you may consult the [SPARQL EXPLAIN operator reference](https://docs.aws.amazon.com/neptune/latest/userguide/sparql-explain-operators.html) for more details about operators and their arguments):

* The first step in query evaluation is a *SolutionInjection* step. In our example, this step injects a single, so-called “universal” solution {}, which is expanded to the final result throughput the evaluation process. This step does not contribute much information in this case, it is just telling us that we do not inject any static solutions (which we would do, for instance, when SPARQL queries contain an outer VALUES clause that injects variable bindings to start out with). As indicated by columns *Units In* and *Units Out*, this step contains no input and forwards the single, universal solution.
* The following four steps are so-called *PipelineJoin* operators. They reflect the sequence in which the triple patterns in the query are evaluated. Most important here is the *pattern* argument, which indicates the triple pattern that is evaluated by the join; the first operator, for instance, evaluates the triple pattern **(?route2, prop:from, airport:FRA)**, the second one to the triple pattern **(?route2, prop:from, ?via)**, and so on. In all cases, the patterns are prefixed with a **distinct**, indicating that we are interested in distinct solutions only.[[2]](#footnote-2) Comparing the sequence of *PipelineJoin*s against the order of triple patterns as specified in our query, you can see that the optimizer has changed the evaluation order. The approach chosen in the query plan is to start out with the destination, Seattle, and traverse the graph “backwards”, via the intermediate stop, to the starting airport.

One important thing to know about Neptune’s *PipelineJoin* is that variable bindings known from previous operators are “substituted” into triple patterns for subsequent operations. In our running example, this concretely means:

* + Operator #1: first PipelineJoin receives the universal solution as input, in which no variables are bound. In the absence of any known variable bindings, it will simply lookup all triples in the database matching the pattern **(?route2, prop:to, airport:SEA)**. As a result of this operation, variable **?route2** will be bound to all routes that have **airport:SEA** as their target.

The *Units Out* column in the explain indicates that there are 118 such routes, and the *PipelineJoin* operator will forward all these routes (represented by pairs of bindings for variables **?airportFrom** and **?route1** to the subsequent operator).

* + Operator #2: The next operator implements the triple pattern **(?route2, prop:from, ?via)**. Column *Units In* tells us that there are 118 bindings flowing in, representing the different **?route2** bindings found in the previous step. Again, the variables from these bindings will now be substituted in successively into the triple patterns, thus extracting the **?via** airports from which the given ?route2 started out. Given that a route represents a single connection between two airports, we will also have 118 solutions flowing out of this operator. In each of these solutions, the **?via** variable will be bound to the candidate intermediate airport.
  + Operators #3: evaluating the pattern **(?route1, prop:to, ?via)**, we now look for all routes **?route1** ending in airport **?via**, which was bound in the previous pattern. For the 118 **?via** airports, we find 10003 such routes, indicating that each of these airports in average has about ~85 incoming connected airports.
  + Operators #4: The last *PipelineJoin* implements the triple pattern **(?route1, prop:from, airport:FRA).** Note that in the input that is flowing into the operator, the variable **?route1** has just recently been bound. As usual, we substitute in these candidates, thus evaluating a fully bound triple pattern; while the evaluation of such a fully bound pattern does not introduce new bindings, its evaluation a fully bound pattern can be understood as “filtering” for those solutions for which the triple pattern exists in the database. In this case, we filter the set of candidate bindings to retain those that actually started in Frankfurt (up to this point, we were just exploring the graph backwards, to retrieve all airports from which we can reach Seattle with one intermediate stop). Unsurprisingly, by applying this filter constraint, the operator brings our intermediate solution size significantly, from 10030 *Units In* to only 45 *Units Out*.
* Once all four *PipelineJoin*s have been evaluated, we compute a *Distinct* (operator number 5) and *Projection* (operator number 6) over the designated output variable **?via**. The *Distinct* operator essentially removes all other variables from the solutions (as they have been collected throughout the PipelineJoin sequence) and only retains the distinct values found for the **?via** variable.
* For the sake of performance, throughout the evaluation process Neptune operates on internal identifiers for the terms (such as URIs and string literals). The final *TermResolution* operator performs a mapping from these internal identifiers to their lexicographical form. The result of this final transformation is then serialized into the requested serialization format and streamed to the client.

One interesting aspect that becomes immediately apparent from the query plan is that the **Distinct** operator effectively performs no work – it has 45 solutions flowing in and the same number of solutions flowing out. This is because, in the AirRoutes dataset, two connected airports are always connected by a single route only, which means that our query will produce DISTINCT results by design, even without an explicit DISTINCT operator in the SPARQL. This is quite useful information: we may want to remove the DISTINCT operator in our SPARQL, to save the time and required memory for its computation. So let’s look at the explain output for the query without DISTINCT:

**query2.sparql**

**PREFIX prop: <http://kelvinlawrence.net/air-routes/vocab/prop#>**

**PREFIX airport: <http://kelvinlawrence.net/air-routes/data/airport#>**

**SELECT ?via WHERE {**

**?route1 prop:from airport:FRA .**

**?route1 prop:to ?via .**

**?route2 prop:from ?via .**

**?route2 prop:to airport:SEA .**

**}**

As expected, the plan now omits the *Distinct* operator and thus gets by with less work:

**curl -s** [**http://$NEPTUNE\_CLUSTER\_ENDPOINT:$NEPTUNE\_CLUSTER\_PORT/sparql**](http://$NEPTUNE_CLUSTER_ENDPOINT:$NEPTUNE_CLUSTER_PORT/sparql\)**?explain=dynamic \**

**-d "@query2.sparql" \**

**-H "Content-type: application/sparql-query" \**

**-H "Accept: text/csv"**

**╔════╤════════╤════════╤═══════════════════╤═══════════════════════════════════════════════════╤══════════╤══════════╤═══════════╤════════╗**

**║ ID │ Out #1 │ Out #2 │ Name │ Arguments │ Mode │ Units In │ Units Out │ Ratio ║**

**╠════╪════════╪════════╪═══════════════════╪═══════════════════════════════════════════════════╪══════════╪══════════╪═══════════╪════════╣**

**║ 0 │ 1 │ - │ SolutionInjection │ solutions=[{}] │ - │ 0 │ 1 │ 0.00 ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 1 │ 2 │ - │ PipelineJoin │ pattern=distinct(?route2, prop:to, airport:SEA) │ - │ 1 │ 118 │ 118.00 ║**

**║ │ │ │ │ joinType=join │ │ │ │ ║**

**║ │ │ │ │ joinProjectionVars=[?route2] │ │ │ │ ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 2 │ 3 │ - │ PipelineJoin │ pattern=distinct(?route2, prop:from, ?via) │ - │ 118 │ 118 │ 1.00 ║**

**║ │ │ │ │ joinType=join │ │ │ │ ║**

**║ │ │ │ │ joinProjectionVars=[?route2, ?via] │ │ │ │ ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 3 │ 4 │ - │ PipelineJoin │ pattern=distinct(?route1, prop:to, ?via) │ - │ 118 │ 10030 │ 85.00 ║**

**║ │ │ │ │ joinType=join │ │ │ │ ║**

**║ │ │ │ │ joinProjectionVars=[?route1, ?via] │ │ │ │ ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 4 │ 5 │ - │ PipelineJoin │ pattern=distinct(?route1, prop:from, airport:FRA) │ - │ 10030 │ 45 │ 0.00 ║**

**║ │ │ │ │ joinType=join │ │ │ │ ║**

**║ │ │ │ │ joinProjectionVars=[?route1] │ │ │ │ ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 5 │ 6 │ - │ Projection │ vars=[?via] │ retain │ 45 │ 45 │ 1.00 ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 6 │ - │ - │ TermResolution │ vars=[?via] │ id2value │ 45 │ 45 │ 1.00 ║**

**╚════╧════════╧════════╧═══════════════════╧═══════════════════════════════════════════════════╧══════════╧══════════╧═══════════╧════════╝**

## Changing evaluation order with SPARQL query hints

Now that we understand how our SPARQL query is being evaluated, let us take the next step and experiment with different execution strategies. To do so, we make use of [SPARQL query hints](https://docs.aws.amazon.com/neptune/latest/userguide/sparql-query-hints.html), which allow us to provide the optimizer with information about the desired join order and evaluation strategy used for query execution.

As a first attempt, let us fix the join order using the *joinOrder* query hint, where – instead of the optimizer chosen behavior discussed previously, namely starting out with the destination airport Seattle and walking “backwards” to Frankfurt – we choose a join order that starts out at Frankfurt airport and performs a “forward” graph traversal to Seattle. Here is the adjusted query:

**query3.sparql**

**PREFIX prop: <http://kelvinlawrence.net/air-routes/vocab/prop#>**

**PREFIX airport: <http://kelvinlawrence.net/air-routes/data/airport#>**

**PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#>**

**SELECT ?via WHERE {**

**hint:Query hint:joinOrder "Ordered" .**

**?route1 prop:from airport:FRA .**

**?route1 prop:to ?via .**

**?route2 prop:from ?via .**

**?route2 prop:to airport:SEA .**

**}**

Query hints are specified as “magic” triple patterns, inlined in the original query: the first line in the SELECT clause specifies the query hint, where the first component, *hint:Query*, specifies the scope (in this case saying that we want to disable automatic join reordering globally, everywhere in the query), the second component, *hint:joinOrder*, specifies the type of the hint (saying that this is a hint to change the join reordering behavior of the optimizer), and the last component, keyword *Ordered*, defines that the join sequence in the query has been *manually ordered*, thus effectively disabling automatic join reordering. More details on query hints and available options can be found in [Neptune’s query hint documentation](https://docs.aws.amazon.com/neptune/latest/userguide/sparql-query-hints.html). Running the query without explain will give us the same 45 results as before; in fact, for simple triple patterns the chosen join evaluation order does never have an impact on the query result. But what we’re mostly interested in is the execution plan obtained by getting the explain for the new version of the query:

**curl -s** [**http://$NEPTUNE\_CLUSTER\_ENDPOINT:$NEPTUNE\_CLUSTER\_PORT/sparql**](http://$NEPTUNE_CLUSTER_ENDPOINT:$NEPTUNE_CLUSTER_PORT/sparql\)**?explain=dynamic \**

**-d "@query3.sparql" \**

**-H "Content-type: application/sparql-query" \**

**-H "Accept: text/csv"**

**╔════╤════════╤════════╤═══════════════════╤═══════════════════════════════════════════════════╤══════════╤══════════╤═══════════╤════════╗**

**║ ID │ Out #1 │ Out #2 │ Name │ Arguments │ Mode │ Units In │ Units Out │ Ratio ║**

**╠════╪════════╪════════╪═══════════════════╪═══════════════════════════════════════════════════╪══════════╪══════════╪═══════════╪════════╣**

**║ 0 │ 1 │ - │ SolutionInjection │ solutions=[{}] │ - │ 0 │ 1 │ 0.00 ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 1 │ 2 │ - │ PipelineJoin │ pattern=distinct(?route1, prop:from, airport:FRA) │ - │ 1 │ 297 │ 297.00 ║**

**║ │ │ │ │ joinType=join │ │ │ │ ║**

**║ │ │ │ │ joinProjectionVars=[?route1] │ │ │ │ ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 2 │ 3 │ - │ PipelineJoin │ pattern=distinct(?route1, prop:to, ?via) │ - │ 297 │ 297 │ 1.00 ║**

**║ │ │ │ │ joinType=join │ │ │ │ ║**

**║ │ │ │ │ joinProjectionVars=[?route1, ?via] │ │ │ │ ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 3 │ 4 │ - │ PipelineJoin │ pattern=distinct(?route2, prop:from, ?via) │ - │ 297 │ 23182 │ 78.05 ║**

**║ │ │ │ │ joinType=join │ │ │ │ ║**

**║ │ │ │ │ joinProjectionVars=[?route2, ?via] │ │ │ │ ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 4 │ 5 │ - │ PipelineJoin │ pattern=distinct(?route2, prop:to, airport:SEA) │ - │ 23182 │ 45 │ 0.00 ║**

**║ │ │ │ │ joinType=join │ │ │ │ ║**

**║ │ │ │ │ joinProjectionVars=[?route2] │ │ │ │ ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 5 │ 6 │ - │ Projection │ vars=[?via] │ retain │ 45 │ 45 │ 1.00 ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 6 │ - │ - │ TermResolution │ vars=[?via] │ id2value │ 45 │ 45 │ 1.00 ║**

**╚════╧════════╧════════╧═══════════════════╧═══════════════════════════════════════════════════╧══════════╧══════════╧═══════════╧════════╝**

Structurally, the query plan is identical to the previous one, the only aspect that changed is the order of the *PipelineJoin*s, which (as requested via the query hint) now exactly corresponds to the order given in the input query. When comparing the *Units In* and *Units Out* during query evaluation, we can observe that there are now significantly more solutions flowing through the plan (up to ~23k, whereas before the peak was at 10k). The overall solutions flowing through the plan typically are a strong indicator for the work that is actually being performed (which usually correlates with the execution time/latency).

Our finding thus implies that forcing the evaluation engine to start out with Frankfurt and walk “forward” is not a good idea. In fact, when measuring answer latency, you will find that the latter query has a higher latency. Informally speaking, the reason is that the Frankfurt airport is larger than Seattle airport, so exploring all outgoing connections from Frankfurt increases the search space. Looking at such statistics for query planning, Neptune’s optimizer made a good choice here in deciding for the “backwards” evaluation strategy.

## Changing the evaluation strategy with SPARQL query hints

Let us finally explore yet another strategy to evaluate the query: instead of the two pipelined plans that we have investigated previously, which evaluate the query from start-to-end airport and end-to-start airport respectively, the third alternative is switching to a “bushy” query plan where we (a) independently extract the outgoing connected airports from Frankfurt, (b) independently extract the incoming connected airports from Seattle, and (c) join together the two extracted sets of airports.

Using query hints, this idea could be implemented using the following query:

**query4.sparql**

**PREFIX prop: <http://kelvinlawrence.net/air-routes/vocab/prop#>**

**PREFIX airport: <http://kelvinlawrence.net/air-routes/data/airport#>**

**PREFIX hint: <http://aws.amazon.com/neptune/vocab/v01/QueryHints#>**

**SELECT ?via WHERE {**

**{**

**SELECT ?via WHERE {**

**hint:SubQuery hint:evaluationStrategy "BottomUp" .**

**?route1 prop:from airport:FRA .**

**?route1 prop:to ?via .**

**}**

**}**

**{**

**SELECT ?via WHERE {**

**hint:SubQuery hint:evaluationStrategy "BottomUp" .**

**?route2 prop:to airport:SEA .**

**?route2 prop:from ?via .**

**}**

**}**

**}**

The idea is to create two subqueries, and enforce these queries to follow a “BottomUp” evaluation strategy using the *evaluationStrategy* query hint. Running the query without explain enabled will give us the same 45 results as before. The query plan revealed when executing it with *explain=dynamic* now looks significantly different though:

**curl -s** [**http://$NEPTUNE\_CLUSTER\_ENDPOINT:$NEPTUNE\_CLUSTER\_PORT/sparql**](http://$NEPTUNE_CLUSTER_ENDPOINT:$NEPTUNE_CLUSTER_PORT/sparql\)**?explain=dynamic \**

**-d "@query4.sparql" \**

**-H "Content-type: application/sparql-query" \**

**-H "Accept: text/csv"**

**╔════╤════════╤════════╤═══════════════════╤═══════════════════════════════════════════╤══════════╤══════════╤═══════════╤════════╗**

**║ ID │ Out #1 │ Out #2 │ Name │ Arguments │ Mode │ Units In │ Units Out │ Ratio ║**

**╠════╪════════╪════════╪═══════════════════╪═══════════════════════════════════════════╪══════════╪══════════╪═══════════╪════════╣**

**║ 0 │ 1 │ - │ SolutionInjection │ solutions=[{}] │ - │ 0 │ 1 │ 0.00 ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 1 │ 2 │ - │ NamedSubquery │ subQuery=subQuery1 │ - │ 1 │ 1 │ 1.00 ║**

**║ │ │ │ │ solutionSet=solutionSet3 │ │ │ │ ║**

**║ │ │ │ │ joinVars=[] │ │ │ │ ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 2 │ 3 │ - │ NamedSubquery │ subQuery=subQuery2 │ - │ 1 │ 1 │ 1.00 ║**

**║ │ │ │ │ solutionSet=solutionSet2 │ │ │ │ ║**

**║ │ │ │ │ joinVars=[?via] │ │ │ │ ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 3 │ 4 │ - │ HashIndexJoin │ solutionSet=solutionSet3 │ - │ 1 │ 297 │ 297.00 ║**

**║ │ │ │ │ joinType=join │ │ │ │ ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 4 │ 5 │ - │ HashIndexBuild │ solutionSet=solutionSet1 │ - │ 297 │ 297 │ 1.00 ║**

**║ │ │ │ │ joinVars=[?via] │ │ │ │ ║**

**║ │ │ │ │ sourceType=pipeline │ │ │ │ ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 5 │ 6 │ - │ MergeJoin │ solutionSets=[solutionSet1, solutionSet2] │ - │ 0 │ 45 │ 0.00 ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 6 │ 7 │ - │ Projection │ vars=[?via] │ retain │ 45 │ 45 │ 1.00 ║**

**╟────┼────────┼────────┼───────────────────┼───────────────────────────────────────────┼──────────┼──────────┼───────────┼────────╢**

**║ 7 │ - │ - │ TermResolution │ vars=[?via] │ id2value │ 45 │ 45 │ 1.00 ║**

**╚════╧════════╧════════╧═══════════════════╧═══════════════════════════════════════════╧══════════╧══════════╧═══════════╧════════╝**

**subQuery2**

**╔════╤════════╤════════╤══════════════╤═════════════════════════════════════════════════╤════════╤══════════╤═══════════╤════════╗**

**║ ID │ Out #1 │ Out #2 │ Name │ Arguments │ Mode │ Units In │ Units Out │ Ratio ║**

**╠════╪════════╪════════╪══════════════╪═════════════════════════════════════════════════╪════════╪══════════╪═══════════╪════════╣**

**║ 0 │ 1 │ - │ Distinct │ vars=[?via] │ - │ 1 │ 1 │ 1.00 ║**

**╟────┼────────┼────────┼──────────────┼─────────────────────────────────────────────────┼────────┼──────────┼───────────┼────────╢**

**║ 1 │ 2 │ - │ PipelineJoin │ pattern=distinct(?route2, prop:to, airport:SEA) │ - │ 1 │ 118 │ 118.00 ║**

**║ │ │ │ │ joinType=join │ │ │ │ ║**

**║ │ │ │ │ joinProjectionVars=[?route2] │ │ │ │ ║**

**╟────┼────────┼────────┼──────────────┼─────────────────────────────────────────────────┼────────┼──────────┼───────────┼────────╢**

**║ 2 │ 3 │ - │ PipelineJoin │ pattern=distinct(?route2, prop:from, ?via) │ - │ 118 │ 118 │ 1.00 ║**

**║ │ │ │ │ joinType=join │ │ │ │ ║**

**║ │ │ │ │ joinProjectionVars=[?route2, ?via] │ │ │ │ ║**

**╟────┼────────┼────────┼──────────────┼─────────────────────────────────────────────────┼────────┼──────────┼───────────┼────────╢**

**║ 3 │ - │ - │ Projection │ vars=[?via] │ retain │ 118 │ 118 │ 1.00 ║**

**╚════╧════════╧════════╧══════════════╧═════════════════════════════════════════════════╧════════╧══════════╧═══════════╧════════╝**

**subQuery1**

**╔════╤════════╤════════╤══════════════╤═══════════════════════════════════════════════════╤════════╤══════════╤═══════════╤════════╗**

**║ ID │ Out #1 │ Out #2 │ Name │ Arguments │ Mode │ Units In │ Units Out │ Ratio ║**

**╠════╪════════╪════════╪══════════════╪═══════════════════════════════════════════════════╪════════╪══════════╪═══════════╪════════╣**

**║ 0 │ 1 │ - │ Distinct │ vars=[?via] │ - │ 1 │ 1 │ 1.00 ║**

**╟────┼────────┼────────┼──────────────┼───────────────────────────────────────────────────┼────────┼──────────┼───────────┼────────╢**

**║ 1 │ 2 │ - │ PipelineJoin │ pattern=distinct(?route1, prop:from, airport:FRA) │ - │ 1 │ 297 │ 297.00 ║**

**║ │ │ │ │ joinType=join │ │ │ │ ║**

**║ │ │ │ │ joinProjectionVars=[?route1] │ │ │ │ ║**

**╟────┼────────┼────────┼──────────────┼───────────────────────────────────────────────────┼────────┼──────────┼───────────┼────────╢**

**║ 2 │ 3 │ - │ PipelineJoin │ pattern=distinct(?route1, prop:to, ?via) │ - │ 297 │ 297 │ 1.00 ║**

**║ │ │ │ │ joinType=join │ │ │ │ ║**

**║ │ │ │ │ joinProjectionVars=[?route1, ?via] │ │ │ │ ║**

**╟────┼────────┼────────┼──────────────┼───────────────────────────────────────────────────┼────────┼──────────┼───────────┼────────╢**

**║ 3 │ - │ - │ Projection │ vars=[?via] │ retain │ 297 │ 297 │ 1.00 ║**

**╚════╧════════╧════════╧══════════════╧═══════════════════════════════════════════════════╧════════╧══════════╧═══════════╧════════╝**

The main query now invokes two subqueries, indicated by the *NamedSubquery* operator, each representing one of the SELECT subqueries in the query plan. These subqueries are executed independently, stand-alone; and each of them follows the previously seen pattern of (in this case, only two) chained *PipelineJoin*s. However, given that these two subqueries perform only a one-step expansion, namely all the outgoing and incoming routes from Frankfurt and to Seattle, respectively, their intermediate result are quite low (below 300 intermediate solutions, whereas before we had >10k intermediate solutions). The subquery results are stored in so-calledsolution sets, *solutionSet3* and *solutionSet3* in our example. These two solution sets are then reused in the plan of the main query. Without going into the technical details, what happens is the following:

* After having computed and stored the two subquery results, the HashIndexJoin (Operator #3) joins solutionSet3 (obtained from the first subquery) against the universal solution. A join against the universal solution is leaving the solutions unchanged, so this essentially spools the result of subQuery3 back into the operator pipeline.
* The *HashIndexBuild* (Operator #4) now spools this solution set into a new hash index, called *solutionSet1*. While containing the same solutions as solutionSet3, the new solutionSet1 differs in one aspect: it contains **?via** as a join variable.
* Finally, the *MergeJoin* (Operator #5) joins solutionSet1 (representing the result of the first subquery joined with our universal solution) with solutionSet2 (representing the result of the second subquery). The join is exploiting the join variable **?via**, which is shared between solutionSet1 and solutionSet2.
* What follows in Operator #6 and Operator #7 is the projection and term resolution, the same pattern that we already discussed in previous plans.

Looking at the overall number of solutions flowing through, it becomes quickly apparent that our new plan is favorable in terms of overall work performed. In fact, when running the queries, you will observe that the new plan runs faster.

However, here’s an important trade-off: the new plan is what we call a “blocking” query plan, i.e. it is no longer fully pipelined: Neptune will compute the two results for (a) and (b) independently, store them in hash indices, and perform a join over these indices. These hash index builds are blocking, where the intermediate solution sets are fully materialized in memory. As a consequence, memory consumption for query evaluation may be higher and time-to-**first**-result may generally be worse. Having non-pipelined plans can be particularly problematic when intermediate solution sets grow really large or in the presence of LIMIT clauses, as they enforce the engine to compute all the intermediate results. For this reason, query planning in Neptune has a bias towards producing pipelined plans. Nevertheless, in this this case (depending on your workload, including aspects such as the chosen instance type, memory requirements for parallel queries, etc.), enforcing the “bushy” plan via query hints may actually give you a nice performance boost!

## Conclusion

Explain and query hints are a great tool to understand and tune Neptune’s runtime behavior. They can be useful to simply understand the behavior of queries (e.g. why queries are actually challenging to evaluate), when you seek for writing improved versions of queries (e.g. by removing redundant operators, such as the DISTINCT in the example above), identifying time-space tradeoffs, or watching out for scenarios where a hand-tuned plan may improve upon the optimizer generated plan. In particular when you are building higher-level graph applications on top of Neptune based on a set of SPARQL query templates, understanding and tuning the evaluation behavior of your templates via explain and query hints can leverage additional performance potentials and – ultimately – improve the user experience of your application’s end users.

If you dive deeper into Neptune’s explain (and query hints), you will notice that the examples only scratched the surface – our [official explain documentation](https://docs.aws.amazon.com/neptune/latest/userguide/sparql-explain.html) contains an exhaustive [reference for all the operators and possible arguments](https://docs.aws.amazon.com/neptune/latest/userguide/sparql-explain-operators.html).

Do you have questions, comments, or did you try out explain and query hints in your use case? We would be happy to hear about your experiences with these new features and appreciate any feedback!

1. The format of the data is ntriples. Given that the data set is publicly accessible, you can use any valid IAM role for data loading. [↑](#footnote-ref-1)
2. This implements SPARQL’s default graph semantics – while it is not relevant in our example, the same matching triple could exist in multiple named graphs, in which case we would only extract one solution (as implied by the distinct). See the documentation on [how Neptune treats named vs. default graphs](https://docs.aws.amazon.com/neptune/latest/userguide/sparql-api-reference.html) for more information. [↑](#footnote-ref-2)