# Processing an AQuA Statement

To meet its goal of providing a common language to express applicability statements, it is critical that an AQuA statement always produce the same outputs when applied against the same available data. In other words, it is critical that all AQuA statements “mean” the same thing to all parties. This section provides guidance to ensure that this common understanding can be achieved. However, within this guidance, implementers have a wide range of options for how AQuA interpretation occurs. This reflects that the AQuA interpreter is a “black box” in this specification – the nature of the inputs and outputs are specified, but interpreters have a great deal of flexibility in going from the former to the latter. In particular, AQuA interpreters might perform transformations on the AQuA statement and/or the underlying data source against which the statement is applied, but the nature of these transformations is left to the discretion of the interpreter implementers.

## Step 1: Validate the AQuA statement

All interpreters MUST verify that an input AQuA statement is syntactically correct before processing. This simply involves verifying that the statement conforms to the grammar defined in this specification. If the interpreter detects a syntax error it MUST raise a SyntaxError and not proceed to Step 2 below. Interpreters MAY continue evaluation of syntax after one syntax error is found in order to more completely report all syntax errors at once.

## Step 2: Data Source Mapping

Once the AQuA statement has been verified as grammatically correct, the next step is to map the AQuA statement’s declared classes to the relevant information in the enterprise data step. The first step in this process is to extract the class declarations of the AQuA statement. Recall that the class declarations identify a list of classes, defined in AQuA domain ontologies, and assign a variable to each listed class.

The AQuA interpreter first needs to look up and make sure that there is a domain ontology that defines each of the named classes. Each class named in the class declarations MUST have an associated definition. If any class name cannot be resolved to a class definition, the AQuA interpreter MUST raise an OntologyNotFound error and halt processing of the AQuA statement. Similarly, some class definitions might include properties whose type is another class defined by reference. The interpreter will need to ensure that all such referenced classes can also be associated with a class definition. Be aware that cycles of class definitions are permitted, interpreters should be sure to avoid infinite loops during the checking process. Note that a statement’s class declarations might name the same class multiple times. This is not an error and simply means that two instances of that class are used when processing the query body of the statement.

Once all classes named in the class declarations have been associated with class definitions in one or more domain ontologies, the interpreter MUST verify that all class properties referenced in the AQuA statement have definitions in the class definition, and that the types assigned to those properties are correct within the statement’s query body. Recall that property references to class instances are made via the variables associated with each class in the class declaration. If the query body references properties that are not defined in the corresponding class, then the interpreter MUST raise an OntologyError. In the case that a property exists but it has the wrong type given its context in the AQuA query body, the interpreter MUST raise a TypeError. Note that AQuA is strongly typed, so once all class definitions are known, it will be possible to verify that the correct types are used in all parts of the AQuA statement. (It might still be possible for type conversion functions to fail due to incorrect value of inputs, e.g., trying to convert an “N” into a NUMBER or DATETIME, but the inputs and outputs to the type conversion function will at least be of the correct type.)

At this point the AQuA statement will be known to be syntactically and semantically correct and the interpreter can begin evaluating the query body.

## Step 3: Evaluate the Query Body

In order to evaluate an AQuA statement, each class named in that statement MUST be mappable to data. This mapping can be handled in a variety of ways: through dynamic lookups to populate the class properties, via association with specific fields in a database, or other mechanisms. The end result, however, is that each class named in the class declarations of the AQuA statement needs to have one or more instances created by mapping data to the class’s properties. An instance represents a mapping of some set of the class’s properties to a single value (noting that a “single value” might take the form of a single instance of a sub-class). An enterprise data source might (and in most cases probably will) have multiple instances associated with any named class. Note that it is NOT necessary for every property of the class to be mapped to data, nor is it necessary for a given property to be consistently present in every class instance. AQuA is designed to function even if there are “holes” in class instances where certain property values are unknown. (Specifically, those properties are assigned the value of UNKNOWN.)

During the process of mapping enterprise data to class instances, be aware that class definitions can create cycles of references. (E.g., Class A can have a property with type Class B, while class B can have a property whose type is Class A.) Be careful that the processing of such classes does not create infinite loops.

Each class named in the class declarations of the AQuA statement MUST have one or more instances after mapping to the enterprise data source. If any class in the class declarations has no instances after mapping, then the query body is not run and the AQuA statement returns an empty list (if a SELECT statement) or FALSE (if an EXISTS statement).

The result of the mappings between each class named in the AQuA statement class declarations and the enterprise data source is effectively a list of sets, where each set represents all the instances of a named class. Note that, if the same class is named multiple times in the class declarations, then the associated sets of instances corresponding to these classes will be duplicates of each other. Evaluation will take place against every combination of instances among these sets of instances. (In other words, take the cross product of these sets of instances, and evaluate the query body against each resulting tuple.)

For example, consider the following class declaration:

(ClassA VarA, ClassB VarB1, ClassB VarB2, ClassC VarC)

Assume that for a given enterprise data source, there are 3 instances of ClassA (a1, a2, and a3), 2 instances of ClassB (b1, b2), and 2 instances of ClassC (c1, c2). Based on this, we end up with a list containing four sets of instances associated with the class declaration:

1. {a1, a2, a3}
2. {b1, b2}
3. {b1, b2}
4. {c1, c2}

Note that sets 2 and 3 are identical, reflecting that ClassB is referenced twice in the class declaration.

The query would be evaluated against every tuple in the cross product of these four sets. Given that there are 3, 2, 2, and 2 members in each set, this would produce a cross product with 3x2x2x2, or 24, members. In pseudocode, this means that evaluation of the AQuA statement would involve evaluating:

QueryBody(a1, b1, b1, c1}

QueryBody(a1, b1, b1, c2}

QueryBody(a1, b1, b2, c1}

QueryBody(a1, b1, b2, c2}

QueryBody(a1, b2, b1, c1}

QueryBody(a1, b2, b1, c2}

QueryBody(a1, b2, b2, c1}

...

QueryBody(a3, b1, b2, c2}

QueryBody(a3, b2, b1, c1}

QueryBody(a3, b2, b1, c2}

QueryBody(a3, b2, b2, c1}

QueryBody(a3, b2, b2, c2}

Thus, the query body is evaluated for every possible combination of values of the named variables in the class declaration, where each variable is assigned one instance of its associated class in each evaluation of the query body.

During these evaluations, conditions might arise that prevent the statement from evaluating. These can include math errors (e.g., dividing by 0), type conversion errors (attempting to perform a type conversion when the input value cannot be converted to the required output), or other issues. In any of these cases, the AQuA interpreter MUST display or log an appropriate error (e.g., a MathException , TypeConversionException, or RuntimeError, respectively) with sufficient diagnostic information to allow a reader to understand the cause of the error. That iteration of the query body is then treated as evaluating to a FALSE and processing continues. Note that these are not fatal errors and the interpreter MUST continue processing with the next combination of instances.

## Step 4: Formulate Result

If the AQuA statement is an EXISTS statement, the result is known after every tuple of class instances derived from the enterprise data source has been evaluated, or after the first such tuple that evaluates to TRUE. The latter is possible because, once a TRUE result is discovered, the final result of the evaluation is known and all subsequent evaluations of the query body become irrelevant. If the AQuA statement is a SELECT statement, the result can only be known after the query body has been executed over every tuple of class instances from the enterprise data source.

If the AQuA statement is a SELECT statement, then the output will use the AQuA results structure. In effect, this structure creates a table (really a list of lists) where each row of the table provides the values of the class elements named in the statement’s results clause for an evaluation of the query body that was TRUE. In the example above, we saw how the query body is evaluated for each combination of instances from the class declaration. In effect, when one of those query body evaluations results in TRUE, a new row is added to the results structure containing the value of the elements in the result clause using the class instance values of that query body evaluation. Query body evaluations that result in FALSE or an error do not contribute to this result table. Duplicate rows in the table MUST NOT be hidden. In other words, if the evaluation of the query body results in TRUE for some combination of input class instances, a row MUST be added to the result table, even if that row duplicates previous rows.

If the AQuA statement is an EXISTS statement, then the output will be a Boolean value. Specifically, the AQuA statement result will be the logical OR of the results of the set of query body results. Put another way, the result of an EXISTS statement will be:

* TRUE if and only if at least one query body execution resulted in TRUE.
* UNKNOWN if and only if no query body execution resulted in TRUE, and at least one query body execution resulted in UNKNOWN.
* FALSE if and only if every query body execution resulted in FALSE.

## Errors

Below are the errors that an AQuA processor MUST be able to detect and raise. All errors are returned as a JSON Object, where the string is the name of the error and the value is the error information. Only the name of the error is standardized in this specification; the error information is left to implementers. However, implementers SHOULD be sure to include sufficient information to support debugging.

1. SyntaxError = Occurs when the input AQuA statement is not syntactically correct. This is a fatal error and precludes further processing of the AQuA statement.
2. TypeError = Occurs when the input AQuA statement fails to conform with typing requirements. (Usually happens when an ontology property is used that is the wrong type for its clause.) This is a fatal error and precludes further processing of the AQuA statement.
3. OntologyError = Occurs if the AQuA statement makes reference to a property that does not exist, or treats a property as a subclass when it is actually a value. This is a fatal error and precludes further processing of the AQuA statement.
4. OntologyNotFoundError = Occurs if the named class in the class declarations section of an AQuA statement cannot be found in the interpreter’s ontology set. This is a fatal error and precludes further processing of the AQuA statement
5. MathException = Occurs if an impossible mathematical operation is requested. (E.g., divide by 0) This is a non-fatal exception and processing of the AQuA statement should continue.
6. TypeConversionException = Evaluating the statement resulted in a failed type conversion operation. (E.g., trying to convert a letter to a number). This exception is thrown either due to a conversion error when populating an ontology instance from the Enterprise Data Source, or due to the failure of an explicit type conversion within the AQuA statement itself. This is a non-fatal exception and processing of the AQuA statement should continue.
7. ArgumentException = A function received arguments that failed to meet requirements of the function’s definition. This exception is not raised by type conversion functions, which instead raise the TypeConversionException.
8. RuntimeError = Covers all other errors encountered during processing of an AQuA statement. These could include, but are not limited to, running out of memory, permission failures, etc. These errors might or might not be fatal.

# Special Considerations

This section reflects on some particular aspects of the AQuA language and how it is to be utilized.

## Critical Values

Not all class properties will always be populated in all instances extracted from an enterprise data source. When class properties lack a corresponding value in the data source, their value is left as UNKNOWN. Sometimes certain comparisons will only be meaningful if certain critical values are known, and the presence of UNKNOWN values render the rest of the query meaningless. In these situations, it may be desirable to “set” the result of the query to a certain value when one of those critical values has a value of UNKNOWN. This can be done using the IsUnknown Boolean function and the rules of three-value logic.

Authors should identify any properties where it is critical that the value be known for the query to be meaningful. The IsUnknown function can then be used to test that all of these values are known. If any of these tests fails, (i.e., IsUnknown returns TRUE) then certain three-value logic can be triggered. The specific logic to be used depends on the type of output the author desires in the case of unknown critical values. Briefly:

If the author desires that an unknown critical value cause the whole query body to evaluate to TRUE, they can take advantage of the fact that a value of TRUE logically ORed with any other value results in TRUE.

For each critical value, create an ORed list of IsUnknown tests of those values. (E.g., for critical values A, B, C, etc., create (...((IsUnknown(A) OR IsUnknown(B)) OR IsUnknown(C)) OR ... )

Logically OR this statement with the rest of the query body logic.

If any of the IsUnknown tests in the first clause evaluate to TRUE, the whole statement will evaluate to TRUE.

If the author desires that an unknown critical value cause the whole query body to evaluate to FALSE, they can take advantage of the fact that a value of FALSE logically ANDed with any other value results in FALSE.

For each critical value, create an ORed list of IsUnknown tests of those values and negate (NOT) its result. (E.g., for critical values A, B, C, etc., create (NOT (...((IsUnknown(A) OR IsUnknown(B)) OR IsUnknown(C)) OR ... )

Logically AND this statement with the rest of the query body logic.

If any of the IsUnknown tests in the first clause evaluate to TRUE, the whole statement will evaluate to FALSE.

These represent fairly simple cases, where a single unknown critical value dictates a specific result of the query body. Different logical constructions can be used for more complex combinations. For example, one might wish to return TRUE if \*both\* A and B are UNKNOWN, but process as normal if one or fewer are UNKNOWN.

It is also possible to craft statements where different results are returned if different values are UNKNOWN (e.g., return TRUE if A is UNKNOWN and return FALSE if B is UNKNOWN). However, since a query body can have only a single result, authors will need to make sure that the correct result is generated in the case that multiple values are unknown.

The AQuA language does not include a mechanism to deliberately raise an error if certain critical values are unknown nor is it possible to deliberately return a query result of UNKNOWN.

## UNKNOWN Under Translation

Within AQuA, a value of UNKNOWN comes about either directly because some class instance property could not be populated from an enterprise data source, or indirectly due to an operation, function, or comparison where one of the operands is UNKNOWN. Ultimately, however, every UNKNOWN value can be traced to one or more properties whose values could not be populated from the enterprise data source. In short, a value of UNKNOWN means “a property value could not be populated”.

Other query languages also employ three-value logic to deal with gaps in knowledge. For example, SQL supports the three values of TRUE, FALSE, and NULL. In this case, NULL and UNKNOWN serve similar roles. However, care needs to be taken when translating enterprise data and AQuA statements to avoid changing meanings. An endpoint might have certain configurations deliberately set to NULL, where NULL has a specific system meaning. For example, a configuration file that includes “cryptographic\_algorithm = NULL” might be an application’s way of indicating that no cryptographic algorithm should be used and communications should be unencrypted. In this case “NULL” does not mean that the value of the given parameter is unknown.

When mapping an AQuA statement into a query language or when mapping enterprise data into class instances, UNKNOWN MUST only be used to represent situations where data is unavailable. In the example above, the value of the cryptographic\_algorithm configuration item might be represented by the string “NULL”, but should not be given a value of UNKNOWN. The bottom line is that, while NULL and UNKNOWN are sometimes used equivalently, they are not always equally appropriate and care must be taken to ensure that meaning is not lost under translation.