AQL Reference Manual - Draft

Ryan Wisnesky

December 2, 2016

Contents

Chapter 1

Introduction

This reference manual for AQL exhaustively documents 1) the keywords and options of AQL, 2) the features of the AQL IDE, and 3) the subtleties that arise when using AQL practice. It is not a tutorial.

1.1 The IDE

AQL is implemented within the *functorial query language integrated development environment (FQL IDE)*. The FQL IDE contains several query languages besides AQL that were developed as part of the research program that culminated in AQL (all of the languages in the FQL IDE share a property known as *functorality*, hence the name FQL IDE). AQL is not a superset of the other languages in the FQL IDE; readers interested in the non-AQL aspects of the FQL IDE are encouraged to visit http://categoricaldata.net. By *AQL IDE*, we mean the FQL IDE set to the AQL language mode. The AQL IDE is an open-source java program that provides an AQL code editor, an AQL execution engine, and a graphical viewer for AQL programs and results.

The AQL IDE contains a multi-tabbed text file editor that supports saving, opening, copy-paste, undo-redo, RTF export, code folding, outline-view, and search-replace through right-click context menus, menu bars, and keyboard shortcuts. A variety of options are available in the options menu (text size, default file path, etc.) Below each AQL text editor is a response text area that displays error and status messages. The built-in AQL examples are loaded by selecting them from load-example combo-box in the upper-right of the IDE. To distinguish AQL examples from examples written in non-AQL languages, AQL examples are prefixed with the letter “A” (e.g., the AQL employees example is called “A Employees”). AQL programs are executed by pressing the run button in the upper-left of the IDE. While an AQL program is running its status will be continually updated in the response area. AQL programs that are running can be manually terminated using the abort menu option. When an AQL program finishes running, the AQL IDE displays a graphical viewer allowing the result of the AQL program to be visually inspected.

1.2 Changing the Memory Limit

To run the AQL IDE with more/less than the default 64mb heap, you must use a command line option such as:

```
java -Xms512m -Xmx2048m -jar fql.jar
```

1.3 Adding JDBC drivers to the classpath

To run the AQL IDE with a particular JDBC driver requires placing the driver on the class path using a “-cp” command line option **and** running java directly, for example:

```
java -cp "./mysql-connector-java-5.1.27-bin.jar:./fql.jar" catdata.ide.IDE
```

1.4 Using AQL without a GUI

The AQL IDE also supports command-line execution: an AQL program is passed to stdin, and an HTML view of the result is output on stdout:

```
java -cp "./fql.jar" catdata.aql.exp.AqlInACan.openCan
```

Chapter 2

AQL Basics

2.1 Comments

Comments in AQL are C style, either "//" or "/* */".

2.2 Kinds

A *kind* is either *typeside*, *schema*, *instance*, (schema) *mapping*, (instance) *transform*, *query*, *graph*, or *pragma*.

2.3 Identifiers

AQL identifiers are case-sensitive alpha-numeric strings that do not start with numerals, or arbitrary strings escaped with double-quotes.

2.4 Declarations

An AQL program is an ordered list of *declarations* of the form *kind name* = *expression*. Each declaration consists of a *name*, which is an identifier unique per AQL program. Each *expression* is evaluated by the AQL execution engine and the resulting artifact (a schema, instance, query, etc., according to *kind*) is bound to the declaration *name*. Before execution, the AQL engine checks that a program has an acyclic set of dependencies.

2.5 Caching

The AQL IDE will cache result artifacts to save time in subsequent executions; this behavior can be disabled per-expression through the `always_reload` option. Disabling caching is can be useful when and AQL program contains side-effecting pragmas.

2.6 Terms

An AQL program contains many *terms*. Their raw syntax is:

```
<term> ::= <identifier> | <identifier>@<identifier> | <term> . <identifier>
             | (<term> <identifier> <term>) | <identifier>(<term>, ...)
```

Intuitively, terms denote "things which can be named" in typesides (java objects, functions and types), schemas (entities, attributes, foreign keys), and instances (generators and labelled nulls). Their abstract syntax is:

```
<term> ::= <variable> | <java object>@<type> | <generator> | <labelled null>
             | <term>.<attribute> | <term>.<foreign key> | <function>(<term>, ...)
```

The syntax $p.q$ abbreviates $q(p)$. Parenthesis in AQL cannot be added (e.g., if v is a variable, then (v) is not valid syntax).

2.7 Contexts

An AQL program contains many *contexts*. Let K, V be two non-terminals in a grammar. A context over K, V is a (possibly empty) list of K, V pairs:

```
<ctx>(K,V) ::= "" | K : V <ctx>
```

The syntax $a\ b\ c\ \ldots : X$ abbreviates the context $a : X\ b : X\ c : X$. Occasionally, `->` or `=` will be used in place of `:` (e.g., in mappings). With the exception of `multi_equations` in instance literals, lists (including contexts) are space-separated.

2.8 Terms in Contexts

The abstract syntax for a *term-in-context* is:

```
<type-or-entity> ::= <type> | <entity>
<term-in-ctx> ::= ctx(<variable>,<type-or-entity>) . <term>
```

Terms are subject to an intuitive typing discipline; e.g., if $f : Nat \to Nat$, then for $f(e)$ to be well-typed, it must be that $e : Nat$.

2.9 Paths

A *path* is a well-typed list of foreign keys. Paths give rise to families of terms: if $f_1, \ldots, f_k$ is a path, then for every term e, $f_k(\ldots f_1(e))$ is a term.

2.10 Equations

The abstract syntax for an *equation* is:

```
<type-or-entity> ::= <type> | <entity>
<term-in-ctx> ::= forall ctx(<variable>,<type-or-entity>) . <term> = <term>
```

The type (or entity) of each term must match. A *path equation* is an equation of terms that are paths terminating on a variable.

2.11 Sections, Literals, Imports, Options

Many AQL expressions are written using *sections*. In the example below, `imports`, `types`, `constants`, and `options` are the sections.

```
typeside Ty = literal {
  imports
    Arithmetic
  types
    Bool
  constants
    t f : Bool
  options
    timeout = 5
}
```

The type side Ty is an example of a *literal*; i.e., an expression that evaluates to itself. All AQL kinds except for pragmas have literals. Literals are also called *constants*, but they should not be confused with the `constants` declared in AQL type sides. All sections are optional, but they must appear in the order specified in this manual.

Many expressions contain import sections. For example, a typeside T can import another typeside T', whose effect is the same as if T' were textually included in T. In the above example, Ty imports the typeside for arithmetic.

Many expressions contain lists of key-value pairs that specify options in an option section. The defaults of these options many change with new releases of AQL and are displayed in the AQL pane of the AQL IDE options menu. The effect of these options is described in the rest of the manual. In the above example, the execution timeout for the type side is set to five seconds.

2.12 AQL's Typing Discipline

AQL programs are nominally typed. For example, if `schema X = Y` is a declaration, then `id X` has type `X -> X` *which is not equal to* `Y -> Y` (beause `X` and `Y` are different *names*).

2.13 Scripting with Side Effects using Pragmas

All AQL expressions with the exception of pragmas are side-effect free. Many pragmas have side effects; for example, `export_instance_jdbc` can write data to a database via JDBC. Therefore, the execution order for pragmas matters. To establish an ordering, each pragma implicitly depends on all declarations before it. Consequently, pragmas can bottleneck AQL execution (the AQL execution engine is multi-threaded). Pragmas are provided to execute arbitrary SQL (via JDBC), Java (via Nashorn javascript), and command-line commands, thereby allowing AQL programs to "script" an entire data integration / ETL flow.

2.14 Algebraic Databases are Deductive Databases

AQL databases are deductive: that is, they are defined only up to provable equality. This means, for example, that an instance may contain a row whose value is $2 + 2$ rather than 4. From AQL's perspective, $2 + 2$ and 4 are freely interchangeable. However, when exporting AQL databases to CSV or SQL, there is a difference between $2 + 2$ and 4: the former is an expression, and the latter is a constant, and they can be exported differently. To establish a true normal form (i.e., to everywhere prefer 4 over $2 + 2$) it is necessary to use java type sides.

2.15 Java (script) typesides and inline functions

AQL typesides can contain bindings of types to java classes and of function symbols to javascript code. (In this manual, we use java and javascript interchangeably, because the Java VM's Nashorn execution engine for javascript allows the use of java functions.) The AQL execution engine will invoke this javascript code during execution, for example, to reduce $2 + 2$ to 4. In order to maintain completeness of AQL's theorem prover, certain equations involving java are disallowed. This restriction can be overridden using the `allow_java_eqs_unsafe` option. Java functions need to be side-effect free.

2.16 Consistency

It is possible for an AQL instance to be inconsistent; i.e., for $1 = 2$ to be provable. If this behavior is not desired, consistency can be enforced using the `require_consistency` option. This option is overly conservative: it may disallow instances that are consistent.

2.17 Labelled Nulls

Nulls in AQL are labelled; i.e., two null values need not be equal. In addition, there is no way to test if a value is a labelled null. Non-labelled nulls can be defined using optional types in typesides. For example:

```
types
  Nat NullableNat Bool
functions
  null : NullableNat
  in : Nat -> NullableNat
  out : NullableNat -> Nat
  isNull : NullableNat -> Bool
equations
   forall x. out (in x) = x
```

2.18 Schema, Mapping, Query, Instance, Transform Inference

The AQL IDE provides code-assistance functions. For example, if S and T are schemas, then if `literal : S -> T` is highlighted, the AQL editor is right-clicked, and "infer mapping" is selected in the resulting pop-up menu, then the IDE will add the skeleton of a mapping declaration. In particular, it will list all of the entities, attributes, and foreign keys for S and will describe how they can be mapped into T; and similarly for queries, transforms, etc. In addition, by pressing control-shift-space after a kind name (e.g., typeside), the AQL editor will pop up a template containing the sections for a corresponding literal.

2.19 Theorem Proving

The AQL engine uses automated theorem proving technology to ensure that AQL programs cannot fail at runtime (modulo javascript errors), or materialize instances that do not satisfy their data integrity constraints. To provide this guarantee, every AQL typeside, schema, and instance must be a decidable equational theory. Not all equational theories are decidable; moreover, automated theorem proving methods can be slow or further incomplete. The AQL engine provides a number of theorem provers which by default it uses automatically as appropriate; however, specific theorem provers can be chosen by using the `prover` option. In addition, schema mappings and transforms are required to preserve provable equality; this behavior can be disabled with the `dont_validate_unsafe` option. To find out which prover is being used, click the "info" button in the "DP" pane in the viewer. The DP pane also allows users to decide the equality of arbitrary terms by entering them as text (useful for debugging AQL programs).

2.20 Provenance / Lineage of generated IDs

The AQL engine generates fresh IDs during execution. These IDs are exported via JDBC and CSV and are used by the engine internally. The AQL engine maintains a "lineage" for these IDs that is displayed in the AQL viewer. This lineage is invariably more meaningful to humans than the generated IDs. For example, rather than display "fresh ID 847", the viewer might display "bill.manager". It is important to remember that these lineages are not canonical, are provided for convenience only, and the real data is the generated IDs, not the lineages.

Chapter 3

Kind `graph`

AQL graphs are directed multi-graphs with labels on nodes and edges.

3.1 `empty`

The empty graph with no nodes and no edges.

3.2 `literal`

3.2.1 `imports`

A list of names of `graph`s.

3.2.2 `nodes`

A list of identifiers.

3.2.3 `edges`

A `ctx(<edge>, <node> -> <node>)`.

Chapter 4

Kind `typeside`

Type sides contain types, function symbols (called constants when 0-ary), and equations, as well java bindings for certain types.

4.1 `empty`

The empty `typeside` with no types or java types.

4.2 `typeSideOf <schema>`

The `typeside` that a `schema` is on.

4.3 `literal : <typeside>`

4.3.1 `imports`

A list of names of `typesides`.

4.3.2 `types`

A list of identifiers.

4.3.3 `constants`

A `ctx(<constant>, <type>)`.

4.3.4 `functions`

A `ctx(<function>, <type>, ... -> <type>)`.

4.3.5 `equations`

A list of equations, i.e., a list of `forall ctx(<variable>,<type>) . <term> = <term>`.

4.3.6 `java_types`

A `ctx(<java_type>, [java class name])`.

4.3.7 `java_constants`

A `ctx(<java_type>, [javascript code])`. During parsing, the `input[0]` of `javascript code` is passed a string, and should return a java object of type `java_type`.

4.3.8 `java_functions`

A `ctx(<java_function>, <java_type> , ... -> <java_type> = [javascript code])`. `javascript code` receives its arguments as `input[0] , ...`.

4.3.9 `options`

Allowable options are `timeout`, `prover` and related options, and `allow_java_eqs_unsafe`.

Chapter 5

Kind `schema`

All schemas are "on" a typeside, and contain entities, foreign keys, attributes, and equations.

5.1 `empty : <typeside>`

The empty `schema` with no entities, foreign keys, or attributes, on a particular `typeside`.

5.2 `schemaOf <instance>`

The `schema` of an `instance`.

5.3 `dom <mapping>`

The source of a schema mapping.

5.4 `cod <mapping>`

The target of a schema mapping.

5.5 `src <query>`

The source of a query.

5.6 `dst <query>`

The target of a query.

5.7 `literal : <typeside>`

5.7.1 `imports`

A list of names of kind `typeside`.

5.7.2 `entities`

A list of identifiers.

5.7.3 `foreign_keys`

A `ctx(<foreign_key>, <entity> -> <entity>)`.

5.7.4 `path_equations`

A list of path equations, where each path starts and ends at an entity.

5.7.5 `attributes`

A `ctx(<function>, <entity> -> <type>)`.

5.7.6 `observation_equations`

A list of `forall <variable> : <entity> . <term> = <term>` , where the terms are of the same type (not entity).

5.7.7 `options`

Allowable options are `timeout`, `prover` and related options, and `allow_java_eqs_unsafe`.

Chapter 6

Kind `mapping`

A schema mapping $F : S \to T$ takes entities in S to entities in T, foreign keys in S to paths in T, and attributes in S to terms in T, in a way that is functorial: if $f : s \to s'$ is a symbol in S, then it must be that $F(f) : F(s) \to F(s')$. In addition, if $\forall \Gamma . e = e'$ is provable in S, then $\forall F(\Gamma) . F(e) = F(e')$ must be provable in T.

6.1 `id <schema>`

The identity schema mapping on a particular schema.

6.2 `compose <mapping> <mapping>`

Composes two mappings: if `f : A -> B` and `g : B -> C`, then `compose f g : A -> C`.

6.3 `literal : <schema1> -> <schema2>`

6.3.1 `imports`

A list of names of kind `mapping`.

6.3.2 `entities`

A `ctx(<entity1>, <entity2>)`.

6.3.3 `foreign_keys`

A `ctx(<foreign_key1>, <path2>)`.

6.3.4 `attributes`

A `ctx(<attribute1>, lambda x:<entity>. <term> : <type> )`, where `type` is the type of `attribute1` and x occurs free in `term`. Any variable can be used in place of x.

6.3.5 `options`

Allowable options are `timeout` and `dont_validate_unsafe`, which disables checking that equations preserved.

Chapter 7

Kind `query`

A query $S \to T$ assigns to each entity en in S a ("frozen") instance on T, which we will write as $[en]$, and to each foreign key $fk : e \to e'$ a transform $[fk] : [e'] \to [e]$ (note the reversal), and to each attribute $att : e \to \tau$ a term of type τ in context $[e]$.

7.1 `id <schema>`

The identity query on a schema.

7.2 `compose <mapping> <mapping>`

Composes two queries: if `f : A -> B` and `g : B -> C`, then `compose f g : A -> C`.

7.3 `literal : <schema> -> <schema>`

7.3.1 `imports`

A list of names of queries.

7.3.2 `entities`

A `ctx(<entity>, <instance>, ctx(<attribute>, <term>))`. The instance uses keywords `from` and `where` instead of `generators` and `equations`. Note that only only generators, not labelled nulls, can be used in the instance. Associated with each entity is a context mapping attributes to terms (the `return` clause).

7.3.3 `foreign_keys`

A `ctx(<foreign key>, <transform>)`.

7.3.4 `options`

Allowed are `timeout` and `dont_validate_eqs`. Note that each instance and transform in the query can have its own options.

Chapter 8

Kind `instance`

8.1 `empty : <schema>`

The empty instance with no generators or labelled nulls on a particular schema.

8.2 `dom <transform>`

The source of a transform.

8.3 `cod <transform>`

The target of a transform.

8.4 `distinct <instance>`

Returns an instance that equates all (entity) generators that are observationally equivalent (i.e., are provably equal on all attributes and foreign keys) in the input instance.

8.5 `eval <query> <instance>`

Evaluates a query on an instance. If $Q : S \to T$, and I is an instance on S, then *eval* Q I is an instance on T.

8.6 `coeval <query> <instance>`

Co-evaluates a query on an instance. If $Q : S \to T$, and J is an instance on T, then *coeval* Q J is an instance on S.

8.6.1 `options`

Allowed options are `timeout` and `prover` and related options.

8.7 `delta <mapping> <instance>`

If $F : S \to T$ is a mapping and J is an T-instance, then $\Delta_F J$ is an S-instance.

8.8 `sigma <mapping> <instance>`

If $F : S \to T$ is a mapping and I is an S-instance, then $\Sigma_F I$ is a T-instance.

8.9 `colimit <graph> : <schema>`

Let G be a graph and S a schema.

8.9.1 `nodes`

A list of names of `instance`s on schema S.

8.9.2 `edges`

A `ctx(<name>, <transform>)`, where the transforms are typed according to the `node` assignments.

8.9.3 `options`

Allowable options are `timeout` and `static_typing`, which when disabled causes AQL to type check the colimit at runtime rather than compile time. This will reduce the number of transforms required to compute any particular colimit, at the cost of potential runtime failure.

8.10 `import_jdbc [jdbcclass] [jdbcuri] : <schema>`

8.10.1 `imports`

A list of instance names.

8.10.1.1 entities

A `ctx(<entity> | <attribute> | <foreign key> | <type> , [sql code])`. The SQL code for each entity and type should return a one column table, and the SQL code for each attribute and foreign key should return a two column table.

8.10.2 `options`

Allowed are `timeout`, `prover` and related options, `always_reload`, and `require_consistency`.

8.11 `import_csv [directoryname] : schema`

The directory name should contain one file named *en.csv* per entity *en*, and optionally, one file named *t.csv* per type *t*. The file for *en* should be a CSV file with a header; the fields of the header should be an ID column name (specified using options), as well as any attributes and foreign keys whose domain is *en*. For types, the file should have a header and one column. Records can contain nulls (using the string specified in options).

8.11.1 `imports`

A list of instance names.

8.11.2 `options`

Allowed are `timeout`, `prover` and related options, `always_reload`, `csv` related options, `id_column_name`, and and `require_consistency`.

8.12 `literal : <schema>`

An instance is a collection of generators at entities and labelled nulls at types, as well as ground equations.

8.12.1 `imports`

A list of instance names.

8.12.2 `generators`

A `ctx(<generator> | <labelled null>, <entity> | <type>)`.

8.12.3 `equations`

A list of ground (non quantified) equations.

8.12.4 `multi_equations`

A `ctx(<attribute> | <foreign key>, <generator> <labelled null>, ...` . For example,

```
multi\_equations
name -> {person1 bill, person2 alice}
```

is equivalent to

```
equations
person1.name = bill
person2.name = alice
```

The key-value pairs in multi-equations must be comma separated (necessary for readability and error correction).

8.12.5 `options`

Allowable options are `timeout`, `prover` and related options, and `require_consistency`.

Chapter 9

Kind `transform`

9.1 `id <instance>`

The identity transform on an instance.

9.2 `compose <transform> <transform>`

Composes two transforms: if `f: A -> B` and `g: B -> C`, then `compose f g: A -> C`.

9.3 `distinct <transform>`

If $t : I \to J$ is a transform, then $\textit{distinct}\ t : \textit{distinct}\ I \to \textit{distinct}\ J$ is a transform.

9.4 `delta <mapping> <transform>`

If $t : I \to J$ is a transform and F a well-typed mapping, then $\Delta_F t : \Delta_F(I) \to \Delta_F(J)$ is a transform.

9.5 `sigma <mapping> <transform>`

If $t : I \to J$ is a transform and F a well-typed mapping, then $\Sigma_F t : \Sigma_F(I) \to \Sigma_F(J)$ is a transform.

9.5.1 options1

Passed to $\Sigma_F(I)$.

9.5.2 options2

Passed to $\Sigma_F(J)$.

9.6 `eval <query> <transform>`

If $t : I \to J$ is a transform, and Q a well-typed query, then $\textit{eval}_Q t : \textit{eval}_Q(I) \to \textit{eval}_Q(J)$ is a transform.

9.7 `coeval <query> <transform>`

If $t : I \to J$ is a transform, and Q a well-typed query then $\textit{coeval}_Q t : \textit{eval}_Q(I) \to \textit{eval}_Q(J)$ is a transform.

9.7.1 options1

Passed to $coeval_Q(I)$.

9.7.2 options2

Passed to $coeval_Q(J)$.

9.8 `unit <mapping> <instance>`

If $F : S \to T$ is a schema mapping and I is an S instance, then $unit_F(I) : I \to \Delta_F(\Sigma_F(I))$.

9.8.1 options

Passed to $\Sigma_F(I)$.

9.9 `counit <mapping> <instance>`

If $F : S \to T$ is a schema mapping and J is an T instance, then $counit_F(J) : \Sigma_F(\Delta_F(J)) \to J$.

9.9.1 options

Passed to $\Sigma_F(\Delta_F(J))$.

9.10 `unit_query <query> <instance>`

If $Q : S \to T$ is a query and J is a T instance, then $unit_Q(I) : eval_Q(coeval_Q(J)) \to J$.

9.10.1 options

Passed to $coeval_Q(IJ)$.

9.11 `counit_query <query> <instance>`

If $Q : S \to T$ is a query and I is an S instance, then $counit_Q(I) : coeval_Q(eval_Q(I)) \to I$.

9.11.1 options

Passed to $coeval_Q(eval_Q(I))$.

9.12 `import_jdbc [jdbcclass] [jdbcuri] : <instance> -> <instance>`

Imports a transform using JDBC. Expects a section assigning each entity *en* and (optionally) type *ty* to a string containing sql. The first column is interpreted as the source and the second column as the target.

9.12.1 options

Allowed are `timeout`, `always_reload`.

9.13 `import_csv [filename] : <instance> -> <instance>`

Imports a transform from CSV. The filename must be a CSV file without a header and with two columns (representing the source and target).

9.13.1 options

Allowable options are `timeout` and `always_reload` and `csv` related options.

9.14 `literal : <instance> -> <instance>`

9.14.1 `imports`

A list of names of transforms.

9.14.2 `generators`

A `ctx(<generator> | <labelled null>, <term>)` that preserves types. Note that the `term` must be ground (not in a context).

9.14.3 `options`

Allowed are `transform` and `dont_validate_unsafe`.

Chapter 10

Kind `pragma`

10.1 `exec_cmdline`

Expects a list of strings inside a section. Each string will be executed by the Java VM as an external process. The return value of each command is displayed in the viewer.

10.1.1 `options`

Allowable options are `timeout` and `always_reload`.

10.2 `exec_javascript`

Expects a list of strings inside a section. Each string will be executed by the Java VM's Nashorn javascript execution engine. The return value of each command is displayed in the viewer.

10.2.1 `options`

Allowable options are `timeout` and `always_reload`.

10.3 `exec_sql [jdbcclass] [jdbcuri]`

Expects a list of (single) SQL commands inside a section. Each string will be executed by JDBC. The return value of each command is displayed in the viewer.

10.3.1 `options`

Allowable options are `timeout` and `always_reload`.

10.4 `export_instance_csv [jdbcclass] [jdbcuri] <instance>`

Exports an instance as CSV. The directory name will contain one file named *en.csv* per entity *en*, and if non-empty, one file named *t.csv* per type *t*. The file for *en* will be a CSV file with a header; the fields of the header will be an ID column name (specified using options), as well as any attributes and foreign keys whose domain is *en*. For types, the file will have a header and one column. AQL rows that are not constants will be exported using the null string specified in options.

10.4.1 `options`

Allowed are `timeout`, `always_reload`, `csv` related options, and `id_column_name`.

10.5 `export_transform_csv [filename] <transform>`

Exports a transform as CSV. The filename will be a CSV file without a header and with two columns (src, tgt). AQL rows that are not constants will be exported using the null string specified in options.

10.5.1 options

Allowable options are `timeout` and `always_reload` and `csv` related options.

10.6 `export_instance_jdbc [jdbcclass] [jdbcuri] [prefix] <instance>`

Exports an instance using JDBC. There will be a table $prefixen$ for each entity en and a table $prefixty$ for each type ty. The columns will be the entities and foreign keys whose domain is en, and an ID column whose name is set in options. Every column will be a VARCHAR type whose length is set in the options. AQL rows that are not constants will be exported as NULL.

10.6.1 options

Allowable options are `timeout` and `always_reload` and `id_column_name` and `varchar_length`.

10.7 `export_transform_jdbc [jdbcclass] [jdbcuri] [prefix] <transform>`

Exports a transform using JDBC. There will be a two-column table $prefixen$ for each entity en and a two-column table $prefixty$ for each type ty. The two columns will be *srcidcol* and *dstidcol*, where *idcol* is the ID column name set in the options. Every column will be a VARCHAR type whose length is set in the options. AQL rows that are not constants will be exported as NULL.

10.7.1 options

Allowed are `timeout`, `always_reload`, `id_column_name` and `varchar_length`.

Chapter 11

options

11.1 timeout = number (seconds)

Causes execution to halt after `number` seconds.

11.2 require_consistency = boolean

When enabled, requires AQL instances to be consistent (e.g., to not prove 1=2). (This is checked at runtime.)

11.3 static_typing = boolean

When disabled, relaxes AQL's nominal typing discipline for colimit instances.

11.4 allow_java_eqs_unsafe = boolean

When enabled, allows arbitrary equations involving java typeside symbols.

11.5 dont_validate_unsafe = boolean

When enabled, mappings and transforms are not checked to be equality-preserving.

11.6 always_reload = boolean

When enabled, pragmas (which can have side effects, like loading data from CSV files) are always executed and are not cached between runs of the AQL program.

11.7 CSV options

11.7.1 csv_charset = string

Sets the java character set to be used for CSV import/export. Should be one of US-ASCII, ISO-8859-1, UTF-8, UTF-16BE, UTF-16LE, UTF-16.

11.7.2 csv_line_delim_string = string

Sets the line delimiter for use in CSV import/export.

11.7.3 `csv_field_delim_char = char`

Sets the field delimiter for use in CSV import/export.

11.7.4 `csv_escape_char = char`

Sets the escape character for use in CSV import/export.

11.7.5 `csv_quote_char = char`

Sets the quote character for use in CSV import/export.

11.7.6 `csv_format = string`

Sets the Apache Commons CSV format to use. Should be one of DEFAULT, EXCEL, MYSQL, RFC4180, TDF.

11.7.7 `csv_null_string = string`

Sets the string that indicates null fields for use in CSV import/export.

11.8 `id_column_name = string`

Specifies the name of the ID columns for CSV and JDBC import/export.

11.9 `varchar_length = number`

Specifies the length of the VARCHAR fields to use for JDBC export.

11.10 `dont_verify_is_appropriate_for_prover_unsafe = boolean`

Many provers require that their input equational theories have a certain form (e.g., be unary). When this option is enabled, this (possibly expensive) condition will not be checked.

11.11 `prover = string`

Specifies which theorem prover to use. The prover string should come from the list below. Only the completion method has options. Note that these theorem proving methods are not "java aware"; to use java typesides, instances "wrap" these provers with java simplification.

11.11.1 `auto`

The auto theorem proving method attempts the free, congruence, monoidal, and program methods, in that order.

11.11.2 `fail`

Applies to all theories. Always fails with an exception.

11.11.3 `free`

Applies only to theories without equations. Returns the syntactic equality of two terms.

11.11.4 `saturated`

Applies only to ground (variable-free) theories with a complete set of equations (e.g., as results from importing a CSV file).

11.11.5 `congruence`

Applies only to ground (variable-free theories). Uses the classical congruence-closure with union-find algorithm.

11.11.6 `monoidal`

Applies only to theories where all symbols are 0-ary or 1-ary. Uses length-reducing Knuth-Bendix completion specialized to semi-Thue systems.

11.11.7 `program`

Applies only to weakly orthogonal theories. Interprets all equations $p = q$ as rewrite rules $p \to q$ and symbolically evaluates terms to normal forms (or diverges).

11.11.8 `completion`

Applies unfailing (ordered) Knuth-Bendix completion specialized to lexicographic path ordering. If no completion precedence is given, attempts to infer a precedence using constraint-satisfaction techniques.

11.11.8.1 `completion_precedence = list of strings`

Defines the precedence to be used for Knuth-Bendix completion. The list `a b c` indicates that $a < b < c$. Every symbol in a typeside or schema or instance must appear exactly once in this list.

11.11.8.2 `completion_sort` = boolean

Sorts the list of critical pairs in Knuth-Bendix completion by length, processing shorter pairs first (but still fairly).

11.11.8.3 `completion_compose = boolean`

Uses the “compose” inference rule in Knuth-Bendix completion.

11.11.8.4 `completion_filter_subsumed = boolean`

Filters out equations that are substitution instances of other equations in Knuth-Bendix completion.

11.11.8.5 `completion_syntactic_ac = boolean`

Enables special support for associative and commutative operators in Knuth-Bendix completion.