Aalto University
School of Science
and Technology

SPARQL to SQL Translation Based on an Intermediate Query Language

Sami Kiminki, Jussi Knuuttila and Vesa Hirvisalo

Department of Computer Science and Engineering
Aalto University, School of Science and Technology
sami.kiminki@aalto.fi

November 8th, 2010

Introduction

The setup

- RDF graph stored into an SQL database
- RDF graph is queried by SPARQL queries
  - For efficient evaluation, translate SPARQL into SQL
  - To reduce round-trips and to allow more SQL DB optimization, should be $1 \times$ SPARQL $\rightarrow 1 \times$ SQL translation
- We want schema flexibility
  - We know from benchmarks that one schema does not fit all
- We want query optimization
  - It's not that we don't trust the databases
  - But sometimes we can do better

Examples are translated using Type-ARQuE 0.2

Translation Overview

The familiar approach

- SPARQL to SPARQL algebra
- Simplify, normalize
- SPARQL algebra to SQL

Our approach

- SPARQL to IL
- Simplify, normalize, transform
- Optimize
- Specialize
- IL to SQL

AQL (Abstract Query Language) is our IL (Intermediate language)

Why Intermediate Query Language and not SPARQL algebra?

- SPARQL algebra defines the SPARQL semantics
  - But it is not designed specifically for SPARQL-to-SQL translation
- Intermediate query language can be designed specifically for SPARQL-to-SQL
  - May operate on lower-level and simpler semantics
  - Additional translate-time information may be easily attached
  - More powerful transformations can be used for, *e.g.*, optimizations
  - To emphasize: focus on a single task
- Side note
  - Similar shift has happened in computer program compilers (syntax-directed to IR-based)

Contents of the Rest of the Talk

- AQL Semantics
  - Basics, joins, expressions
- Type inference
- SPARQL to AQL translation
  - Variable binding
- AQL transformations / lowering
  - Nested join flattening (postponed after conclusions)
  - Triple component access resolution
- AQL to SQL
- Conclusions

For reference:
Type-ARQuE translation passes

SPARQL front-end	SPARQL parse to AST
	AST normalize
	Variable binding
	Generate AQL
General preparation	Normalization passes
	Type inference
	Empty type sets to `nulls`
	Nested join flattening
	Comparison optimization
	Function variant selection
Specialization	Property value requirer
	Triple access resolution
	Expression optimization
	Function variant selection
	Typecast injection
AQL to SQL	SQL access collection
	SQL emit

Contents of the Rest of the Talk

- AQL Semantics
  - Basics, joins, expressions
- Type inference
- SPARQL to AQL translation
  - Variable binding
- AQL transformations / lowering
  - Nested join flattening (postponed after conclusions)
  - Triple component access resolution
- AQL to SQL
- Conclusions

For reference:
Type-ARQuE translation passes

SPARQL front-end	SPARQL parse to AST
	AST normalize
	Variable binding
	Generate AQL
General preparation	Normalization passes
	Type inference
	Empty type sets to `nulls`
	Nested join flattening
	Comparison optimization
	Function variant selection
Specialization	Property value requirer
	Triple access resolution
	Expression optimization
	Function variant selection
	Typecast injection
AQL to SQL	SQL access collection
	SQL emit

Contents of the Rest of the Talk

- AQL Semantics
  - Basics, joins, expressions
- Type inference
- SPARQL to AQL translation
  - Variable binding
- AQL transformations / lowering
  - Nested join flattening (postponed after conclusions)
  - Triple component access resolution
- AQL to SQL
- Conclusions

For reference:
Type-ARQuE translation passes

SPARQL front-end	SPARQL parse to AST
	AST normalize
	Variable binding
	Generate AQL
General preparation	Normalization passes
	Type inference
	Empty type sets to `nulls`
	Nested join flattening
	Comparison optimization
	Function variant selection
Specialization	Property value requirer
	Triple access resolution
	Expression optimization
	Function variant selection
	Typecast injection
AQL to SQL	SQL access collection
	SQL emit

Contents of the Rest of the Talk

- AQL Semantics
  - Basics, joins, expressions
- Type inference
- SPARQL to AQL translation
  - Variable binding
- AQL transformations / lowering
  - Nested join flattening (postponed after conclusions)
  - Triple component access resolution
- AQL to SQL
- Conclusions

For reference:
Type-ARQuE translation passes

SPARQL front-end	SPARQL parse to AST
	AST normalize
	Variable binding
	Generate AQL
General preparation	Normalization passes
	Type inference
	Empty type sets to `nulls`
	Nested join flattening
	Comparison optimization
	Function variant selection
Specialization	Property value requirer
	Triple access resolution
	Expression optimization
	Function variant selection
	Typecast injection
AQL to SQL	SQL access collection
	SQL emit

Contents of the Rest of the Talk

- AQL Semantics
  - Basics, joins, expressions
- Type inference
- SPARQL to AQL translation
  - Variable binding
- AQL transformations / lowering
  - Nested join flattening (postponed after conclusions)
  - Triple component access resolution
- AQL to SQL
- Conclusions

For reference:
Type-ARQuE translation passes

SPARQL front-end	SPARQL parse to AST AST normalize Variable binding Generate AQL
General preparation	Normalization passes Type inference Empty type sets to `nulls` Nested join flattening Comparison optimization Function variant selection
Specialization	Property value requirer Triple access resolution Expression optimization Function variant selection Typecast injection
AQL to SQL	SQL access collection SQL emit

Contents of the Rest of the Talk

- AQL Semantics
  - Basics, joins, expressions
- Type inference
- SPARQL to AQL translation
  - Variable binding
- AQL transformations / lowering
  - Nested join flattening (postponed after conclusions)
  - Triple component access resolution
- AQL to SQL
- Conclusions

For reference:
Type-ARQuE translation passes

SPARQL front-end	SPARQL parse to AST AST normalize Variable binding Generate AQL
General preparation	Normalization passes Type inference Empty type sets to `nulls` Nested join flattening Comparison optimization Function variant selection
Specialization	Property value requirer Triple access resolution Expression optimization Function variant selection Typecast injection
AQL to SQL	SQL access collection SQL emit

Contents of the Rest of the Talk

- AQL Semantics
  - Basics, joins, expressions
- Type inference
- SPARQL to AQL translation
  - Variable binding
- AQL transformations / lowering
  - Nested join flattening (postponed after conclusions)
  - Triple component access resolution
- AQL to SQL
- Conclusions

For reference:
Type-ARQuE translation passes

SPARQL front-end	SPARQL parse to AST AST normalize Variable binding Generate AQL
General preparation	Normalization passes Type inference Empty type sets to `nulls` Nested join flattening Comparison optimization Function variant selection
Specialization	Property value requirer Triple access resolution Expression optimization Function variant selection Typecast injection
AQL to SQL	SQL access collection SQL emit

Contents of the Rest of the Talk

- AQL Semantics
  - Basics, joins, expressions
- Type inference
- SPARQL to AQL translation
  - Variable binding
- AQL transformations / lowering
  - Nested join flattening (postponed after conclusions)
  - Triple component access resolution
- AQL to SQL
- Conclusions

For reference:
Type-ARQuE translation passes

SPARQL front-end	SPARQL parse to AST
	AST normalize
	Variable binding
	Generate AQL
General preparation	Normalization passes
	Type inference
	Empty type sets to `nulls`
	Nested join flattening
	Comparison optimization
	Function variant selection
Specialization	Property value requirer
	Triple access resolution
	Expression optimization
	Function variant selection
	Typecast injection
AQL to SQL	SQL access collection
	SQL emit

Contents of the Rest of the Talk

- AQL Semantics
  - Basics, joins, expressions
- Type inference
- SPARQL to AQL translation
  - Variable binding
- AQL transformations / lowering
  - Nested join flattening (postponed after conclusions)
  - Triple component access resolution
- AQL to SQL
- Conclusions

For reference:
Type-ARQuE translation passes

SPARQL front-end	SPARQL parse to AST AST normalize Variable binding Generate AQL
General preparation	Normalization passes Type inference Empty type sets to `nulls` Nested join flattening Comparison optimization Function variant selection
Specialization	Property value requirer Triple access resolution Expression optimization Function variant selection Typecast injection
AQL to SQL	SQL access collection SQL emit

Contents of the Rest of the Talk

- AQL Semantics
  - Basics, joins, expressions
- Type inference
- SPARQL to AQL translation
  - Variable binding
- AQL transformations / lowering
  - Nested join flattening (postponed after conclusions)
  - Triple component access resolution
- AQL to SQL
- Conclusions

For reference:
Type-ARQuE translation passes

SPARQL front-end	SPARQL parse to AST
	AST normalize
	Variable binding
	Generate AQL
General preparation	Normalization passes
	Type inference
	Empty type sets to `nulls`
	Nested join flattening
	Comparison optimization
	Function variant selection
Specialization	Property value requirer
	Triple access resolution
	Expression optimization
	Function variant selection
	Typecast injection
AQL to SQL	SQL access collection
	SQL emit

AQL Semantics: Basics

```
PREFIX p: <.../>
SELECT ?a ?c
WHERE {
  ?a ?b ?c
  FILTER(?c = 'Anne')
}
```

```
(aql-query ("triple_1_1")
  (select "a"
    (property any "triple_1_1" subject))
  (select "c"
    (property any "triple_1_1" object))
  (criterion
    (comp-eq
      (property any "triple_1_1" object)
      (literal string "Anne"))))
```

- Explicitly named triples
- Triple component references instead of variables

AQL Semantics: Basics

```
PREFIX p: <.../>
SELECT ?a ?c
WHERE {
  ?a p:firstname ?c
  FILTER(?c = 'Anne')
}
```

```
(aql-query ("triple_1_1")
  (select "a"
    (property any "triple_1_1" subject))
  (select "c"
    (property any "triple_1_1" object))
  (criterion (and
    (comp-eq
      (property any "triple_1_1" predicate)
      (literal IRI ".../firstname"))
    (comp-eq
      (property any "triple_1_1" object)
      (literal string "Anne")))))
```

- Unified filters: no difference between `FILTER`s and triple match pattern

AQL Semantics: Joins

```
SELECT ?a ?c ?e
WHERE {
  ?a ?b ?c
  OPTIONAL {
    ?c ?d ?e
  }
}
```

```
(aql-query ("triple_1_1")
  (select "a"
    (property any "triple_1_1" subject))
  (select "c"
    (property any "triple_1_1" object))
  (select "e"
    (property any "triple_2_1" object))
  (join left ("triple_2_1")
    (comp-eq
      (property any "triple_2_1" subject)
      (property any "triple_1_1" object)))
  (criterion))
```

- Optional graph group = left join
- Join condition: “outer” `?c == ?c` “inner”
- Joins can be nested

AQL Semantics: Joins

```
SELECT ?a ?c ?e
WHERE {
  ?a ?b ?c
  OPTIONAL {
    ?c ?d ?e
  }
}
```

```
(aql-query ("triple_1_1")
  (select "a"
    (property any "triple_1_1" subject))
  (select "c"
    (property any "triple_1_1" object))
  (select "e"
    (property any "triple_2_1" object))
  (join left ("triple_2_1")
    (comp-eq
      (property any "triple_2_1" subject)
      (property any "triple_1_1" object)))
  (criterion))
```

- Optional graph group = left join
- Join condition: “outer” `?c == ?c` “inner”
- Joins can be nested

AQL Semantics: Joins

```
SELECT ?a ?c ?e
WHERE {
  ?a ?b ?c
  OPTIONAL {
    ?c ?d ?e
  }
}
```

```
(aql-query ("triple_1_1")
  (select "a"
    (property any "triple_1_1" subject))
  (select "c"
    (property any "triple_1_1" object))
  (select "e"
    (property any "triple_2_1" object))
  (join left ("triple_2_1")
    (comp-eq
      (property any "triple_2_1" subject)
      (property any "triple_1_1" object)))
  (criterion))
```

- Optional graph group = left join
- Join condition: “outer” `?c == ?c` “inner”
- Joins can be nested

AQL Join Evaluation Semantics

- Top-to-bottom, recurse before join condition
  1. Join the data
  2. Recurse into nested joins
  3. Apply join condition
- This is different to SPARQL and SQL joins
  - SPARQL (currently) and SQL joins are bottom-up
  - They are more localized: recurse after applying condition
- The rationale: more triples can be referenced at join conditions
- AQL joins are a superset of SPARQL and SQL joins
  - For both bottom-up or left-to-right variable binding semantics

Join Evaluation Semantics Example

```
(aql-query ("tri_1" "tri_2") (tri_1.subj=="s1")
  (join left ("tri_3") (tri_1.subj==tri_2.subj
                        AND tri_1.subj==tri_3.subj)))
```

Empty table
1 rows, 0 columns

Store contains 2 triples:
(s1,p1,o1) and (s2,p2,o2)

We start from an empty table, which is the identity for Cartesian product

Join Evaluation Semantics Example

```
(aql-query ("tri_1" "tri_2") (tri_1.subj=="s1")
  (join left ("tri_3") (tri_1.subj==tri_2.subj
                        AND tri_1.subj==tri_3.subj)))
```

ROW	tri_1			tri_2		
1	s1	p1	o1	s1	p1	o1
2	s1	p1	o1	s2	p2	o2
3	s2	p2	o2	s1	p1	o1
4	s2	p2	o2	s2	p2	o2

Join the triple store once per declared triple in top-level query using Cartesian product

Join Evaluation Semantics Example

```
(aql-query ("tri_1" "tri_2") (tri_1.subj=="s1")
  (join left ("tri_3") (tri_1.subj==tri_2.subj
                        AND tri_1.subj==tri_3.subj)))
```

ROW	tri_1			tri_2			tri_3		
1a	s1	p1	o1	s1	p1	o1	s1	p1	o1
1b	s1	p1	o1	s1	p1	o1	s2	p2	o2
2a	s1	p1	o1	s2	p2	o2	s1	p1	o1
2b	s1	p1	o1	s2	p2	o2	s2	p2	o2
3a	s2	p2	o2	s1	p1	o1	s1	p1	o1
3b	s2	p2	o2	s1	p1	o1	s2	p2	o2
4a	s2	p2	o2	s2	p2	o2	s1	p1	o1
4b	s2	p2	o2	s2	p2	o2	s2	p2	o2

Recurse into joins, still using Cartesian product

Join Evaluation Semantics Example

```
(aql-query ("tri_1" "tri_2") (tri_1.subj=="s1")
  (join left ("tri_3") (tri_1.subj==tri_2.subj
                        AND tri_1.subj==tri_3.subj)))
```

ROW	tri_1			tri_2			tri_3			
1a	s1	p1	o1	s1	p1	o1	s1	p1	o1	MATCH
1b	s1	p1	o1	s1	p1	o1	s2	p2	o2	
2a	s1	p1	o1	s2	p2	o2	s1	p1	o1	
2b	s1	p1	o1	s2	p2	o2	s2	p2	o2	
3a	s2	p2	o2	s1	p1	o1	s1	p1	o1	
3b	s2	p2	o2	s1	p1	o1	s2	p2	o2	
4a	s2	p2	o2	s2	p2	o2	s1	p1	o1	
4b	s2	p2	o2	s2	p2	o2	s2	p2	o2	MATCH

No more nested joins, evaluate condition

Join Evaluation Semantics Example

```
(aql-query ("tri_1" "tri_2") (tri_1.subj=="s1")
  (join left ("tri_3") (tri_1.subj==tri_2.subj
                        AND tri_1.subj==tri_3.subj)))
```

ROW	tri_1			tri_2			tri_3			
1a	s1	p1	o1	s1	p1	o1	s1	p1	o1	MATCH
1b	s1	p1	o1	s1	p1	o1	<null>			
2a	s1	p1	o1	s2	p2	o2	<null>			
2b	s1	p1	o1	s2	p2	o2	<null>			
3a	s2	p2	o2	s1	p1	o1	<null>			
3b	s2	p2	o2	s1	p1	o1	<null>			
4a	s2	p2	o2	s2	p2	o2	<null>			
4b	s2	p2	o2	s2	p2	o2	s2	p2	o2	MATCH

Replace joined values by nulls in non-matching condition rows

Join Evaluation Semantics Example

```
(aql-query ("tri_1" "tri_2") (tri_1.subj=="s1")
  (join left ("tri_3") (tri_1.subj==tri_2.subj
                        AND tri_1.subj==tri_3.subj)))
```

ROW	tri_1			tri_2			tri_3			
1a	s1	p1	o1	s1	p1	o1	s1	p1	o1	MATCH
1b	s1	p1	o1	s1	p1	o1	<null>			
2a	s1	p1	o1	s2	p2	o2	<null>			
2b	s1	p1	o1	s2	p2	o2	<null>			
3a	s2	p2	o2	s1	p1	o1	<null>			
3b	s2	p2	o2	s1	p1	o1	<null>			
4a	s2	p2	o2	s2	p2	o2	<null>			
4b	s2	p2	o2	s2	p2	o2	s2	p2	o2	MATCH

Compactify by removing rows which received nulls. However, as this is LEFT OUTER join, leave at least one instance of original rows. INNER join would remove also these.

Join Evaluation Semantics Example

```
(aql-query ("tri_1" "tri_2") (tri_1.subj=="s1")
  (join left ("tri_3") (tri_1.subj==tri_2.subj
                        AND tri_1.subj==tri_3.subj)))
```

ROW	tri_1			tri_2			tri_3		
1a	s1	p1	o1	s1	p1	o1	s1	p1	o1
2a	s1	p1	o1	s2	p2	o2	<null>		
3a	s2	p2	o2	s1	p1	o1	<null>		
4b	s2	p2	o2	s2	p2	o2	s2	p2	o2

Compactify by removing rows which received nulls. However, as this is LEFT OUTER join, leave at least one instance of original rows. INNER join would remove also these.

Join Evaluation Semantics Example

`(aql-query ("tri_1" "tri_2") (`*tri_1.subj=="s1"*`)`

`(join left ("tri_3") (`*tri_1.subj==tri_2.subj*

AND tri_1.subj==tri_3.subj`)))`

ROW	tri_1			tri_2			tri_3			
1a	s1	p1	o1	s1	p1	o1	s1	p1	o1	MATCH
2a	s1	p1	o1	s2	p2	o2	<null>			MATCH
3a	s2	p2	o2	s1	p1	o1	<null>			
4b	s2	p2	o2	s2	p2	o2	s2	p2	o2	

Continue upwards by evaluating top-level conditions

Join Evaluation Semantics Example

```
(aql-query ("tri_1" "tri_2") (tri_1.subj=="s1")
  (join left ("tri_3") (tri_1.subj==tri_2.subj
                        AND tri_1.subj==tri_3.subj)))
```

ROW	tri_1			tri_2			tri_3			
1a	s1	p1	o1	s1	p1	o1	s1	p1	o1	MATCH
2a	s1	p1	o1	s2	p2	o2	<null>			MATCH
3a	s2	p2	o2	s1	p1	o1	<null>			
4b	s2	p2	o2	s2	p2	o2	s2	p2	o2	

Compactify by removing non-matching rows

Join Evaluation Semantics Example

```
(aql-query ("tri_1" "tri_2") (tri_1.subj=="s1")
  (join left ("tri_3") (tri_1.subj==tri_2.subj
                        AND tri_1.subj==tri_3.subj)))
```

ROW	tri_1			tri_2			tri_3		
1a	s1	p1	o1	s1	p1	o1	s1	p1	o1
2a	s1	p1	o1	s2	p2	o2	<null>		

Compactify by removing non-matching rows

Join Evaluation Semantics Example

```
(aql-query ("tri_1" "tri_2") (tri_1.subj=="s1")
  (join left ("tri_3") (tri_1.subj==tri_2.subj
                        AND tri_1.subj==tri_3.subj)))
```

ROW	tri_1			tri_2			tri_3		
1a	s1	p1	o1	s1	p1	o1	s1	p1	o1
2a	s1	p1	o1	s2	p2	o2	<null>		

Nothing more to do. This is the evaluated solution set. Each row represents a solution.

AQL Expressions

- Expression classes: literals, triple component expressions (called property expressions), function expressions
- Explicitly typed
- Triple component expressions and function expressions have sets of possible types
  - In SPARQL, variables may be bound to values of different types between solutions, too
- Examples
  - `(property (string integer) "triple_1_1" object)`
  - ```
    (function "builtin:coalesce" (string integer)
       (literal string "ABC")
       (literal integer 55))
    ```
    ```

AQL Expressions

- Expression classes: literals, triple component expressions (called property expressions), function expressions
- Explicitly typed
- Triple component expressions and function expressions have sets of possible types
  - In SPARQL, variables may be bound to values of different types between solutions, too
- Examples
  - `(property (string integer) "triple_1_1" object)`
  - ```
    (function "builtin:coalesce" (string integer)
       (literal string "ABC")
       (literal integer 55))
    ```
    ```

Type Inference

- Based on join condition analysis (*i.e.*, SPARQL filters and triple match patterns)
- Motivational example:

```
SELECT *
WHERE {
  ?s ?p ?o
  FILTER(?o > 3)
}
```

  For each possible solution:
  - `?s` and `?p` must be IRIs
  - `?o` must be numeric
- Performed on AQL queries (explicit expression typing)
- Beneficial for later parts of translation

Type Inference

- Based on join condition analysis (*i.e.*, SPARQL filters and triple match patterns)
- Motivational example:

```
SELECT *
WHERE {
  ?s ?p ?o
  FILTER(?o > 3)
}
```

  For each possible solution:
  - `?s` and `?p` must be IRIs
  - `?o` must be numeric
- Performed on AQL queries (explicit expression typing)
- Beneficial for later parts of translation

Type Inference

- Based on join condition analysis (*i.e.*, SPARQL filters and triple match patterns)
- Motivational example:

```
SELECT *
WHERE {
  ?s ?p ?o
  FILTER(?o > 3)
}
```

  For each possible solution:
  - `?s` and `?p` must be IRIs
  - `?o` must be numeric
- Performed on AQL queries (explicit expression typing)
- Beneficial for later parts of translation

Type Inference

- Based on join condition analysis (*i.e.*, SPARQL filters and triple match patterns)
- Motivational example:

```
SELECT *
WHERE {
  ?s ?p ?o
  FILTER(?o > 3)
}
```

  For each possible solution:
  - `?s` and `?p` must be IRIs
  - `?o` must be numeric
- Performed on AQL queries (explicit expression typing)
- Beneficial for later parts of translation

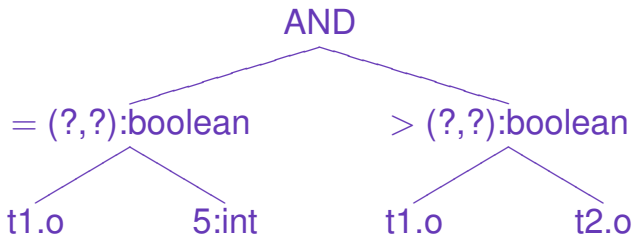
Type Inference Algorithm

- Based on two-level dataflow equations:
  1. Assume initially that every triple component and function expression can be of any type
  2. Find conflicts and constrain the set of possible types (per condition expression)
  3. Propagate triple component type sets between joins
  4. Go back to Step 2 until a fixpoint is found
- As the type sets are always shrinking, a fixpoint is guaranteed to be reached eventually

Type Inference Example for Condition

(t1.o=5) AND (t1.o>t2.o)

STEP 1



STEP 2

STEP 3

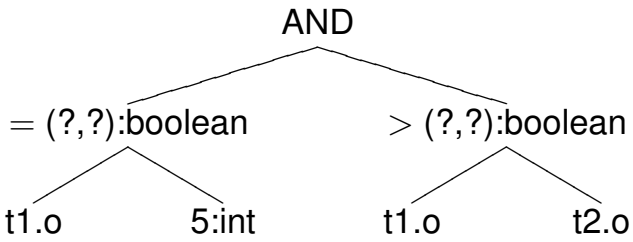
STEP 4

Figure: Illustration of type inference for a simple expression. Type of t1.o has been inferred as and t2.o as type.

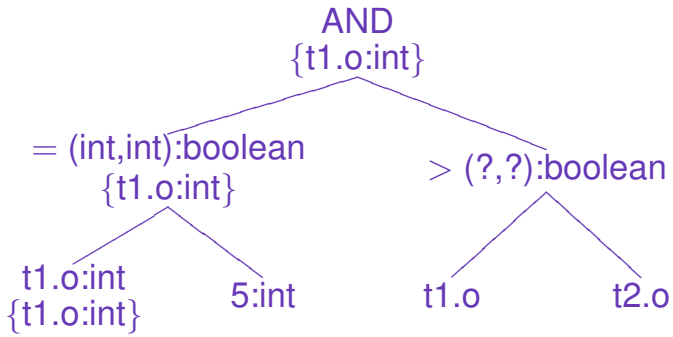
Type Inference Example for Condition

(t1.o=5) AND (t1.o>t2.o)

STEP 1



STEP 2



STEP 3

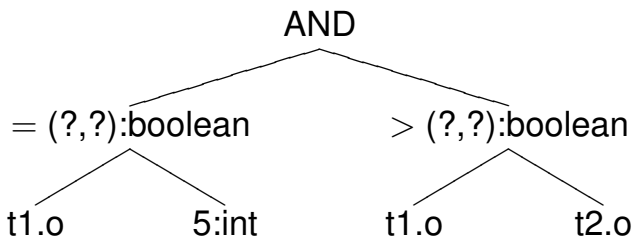
STEP 4

Figure: Illustration of type inference for a simple expression. Type of t1.o has been inferred as and t2.o as type.

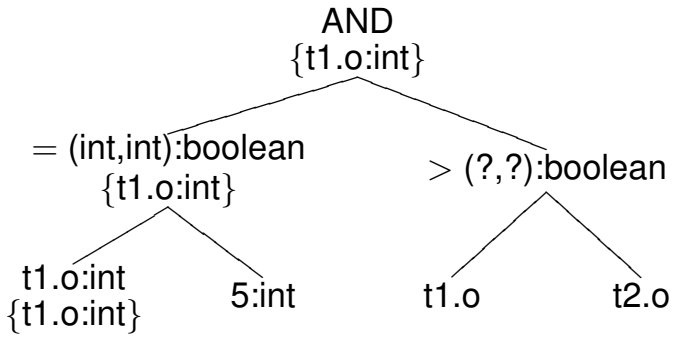
Type Inference Example for Condition

(t1.o=5) AND (t1.o>t2.o)

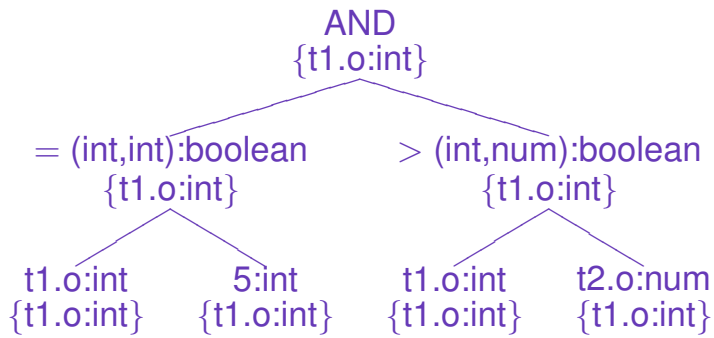
STEP 1



STEP 2



STEP 3



STEP 4

Figure: Illustration of type inference for a simple expression. Type of t1.o has been inferred as and t2.o as type.

Type Inference Example for Condition

(t1.o=5) AND (t1.o>t2.o)

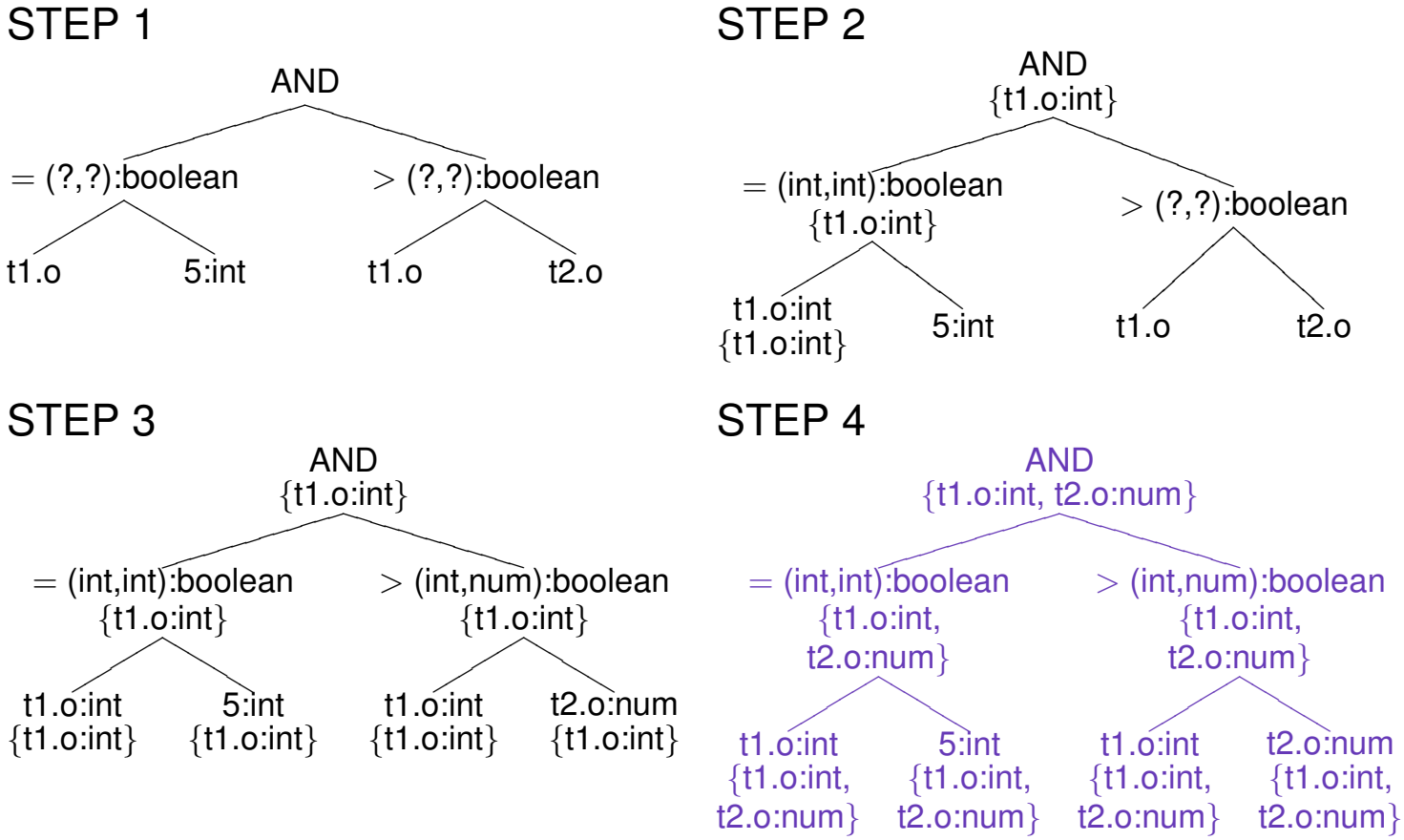


Figure: Illustration of type inference for a simple expression. Type of t1.o has been inferred as integral and t2.o as general numeric type.

Translating SPARQL to AQL

Basically straightforward:

- Graph group structure is preserved in AQL join expression structure
- Triple match patterns are named and the names are inserted into AQL joins, as triple names
- If match pattern has constraints, add respective conditions for AQL joins (or the top-level query)
- Add `FILTER`s as additional constraints to AQL joins (or the top-level query)
- However, variable dereferencing needs additional consideration

SPARQL Variables to AQL Expressions

- The idea: When dereferencing a variable, determine where it can be bound before the dereference point (per solution)
- If multiple bound options, use `coalesce`
- If variable is used in a triple match pattern with possible previous bind, add condition

Variables Example

```
SELECT ?x
WHERE {
  ?a ?b ?x
}
```

```
(aql-query ("triple_1_1")
  (select "x" (property any "triple_1_1" object))
  (criterion))
```

Variables Example

```
SELECT ?x
WHERE {
  OPTIONAL { ?c ?d ?x }
  ?a ?b ?x
}
```

```
(aql-query ("triple_1_1")
  (select "x" (property any "triple_1_1" object))
  (join left ("triple_2_1"))
  (criterion
    (or
      (function"builtin:is-null" any
        (property any "triple_2_1" object))
      (comp-eq
        (property any "triple_1_1" object)
        (property any "triple_2_1" object)))))
```

Variables Example

```
SELECT ?x
WHERE {
  OPTIONAL { ?c ?d ?x }   OPTIONAL { ?e ?f ?x }
  ?a ?b ?z
}
```

```
(aql-query ("triple_1_1")
  (select "x" (function"builtin:coalesce" any
      (property any "triple_2_1" object)
      (property any "triple_3_1" object)))
  (join left ("triple_2_1"))
  (join left ("triple_3_1")
    (or
      (function"builtin:is-null" any
        (property any "triple_2_1" object))
      (comp-eq
        (property any "triple_3_1" object)
        (property any "triple_2_1" object))))
  (criterion))
```

Adapting to SQL Schema: Triple Component Access Resolution

- Replace schema-agnostic `property` expressions with schema-specific low-level expressions
- For faceted schemas, explicit type information of `property` expressions is used to determine which value tables are required
- Example:

```
(property (double integer) "triple_1_1" object)
      ⇒
(function"builtin:coalesce" (double integer)
  (custom (double)
    SQLAccessExpr INDEX VC_Doubles.{ix:  id, value:  double_value,
       triple-table-column:  obj(object)} USING JOIN triple_1_1)
  (custom (integer)
    SQLAccessExpr INDEX VC_Integers.{ix:  id, value:  int_value,
       triple-table-column:  obj(object)} USING JOIN triple_1_1))
```

Adapting to SQL Schema: Triple Component Access Resolution

- Replace schema-agnostic `property` expressions with schema-specific low-level expressions
- For faceted schemas, explicit type information of `property` expressions is used to determine which value tables are required
- Example:

```
(property (double integer) "triple_1_1" object)
      ⇒
(function"builtin:coalesce" (double integer)
  (custom (double)
    SQLAccessExpr INDEX VC_Doubles.{ix:  id, value:  double_value,
       triple-table-column:  obj(object)} USING JOIN triple_1_1)
  (custom (integer)
    SQLAccessExpr INDEX VC_Integers.{ix:  id, value:  int_value,
       triple-table-column:  obj(object)} USING JOIN triple_1_1))
```

Adapting to SQL Schema: Triple Component Access Resolution

- Replace schema-agnostic `property` expressions with schema-specific low-level expressions
- For faceted schemas, explicit type information of `property` expressions is used to determine which value tables are required
- Example:

```
(property (double integer) "triple_1_1" object)
      ⇒
(function"builtin:coalesce" (double integer)
  (custom (double)
    SQLAccessExpr INDEX VC_Doubles.{ix:  id, value:  double_value,
       triple-table-column:  obj(object)} USING JOIN triple_1_1)
  (custom (integer)
    SQLAccessExpr INDEX VC_Integers.{ix:  id, value:  int_value,
       triple-table-column:  obj(object)} USING JOIN triple_1_1))
```

Adapting to SQL Schema: Triple Component Access Resolution

- Replace schema-agnostic `property` expressions with schema-specific low-level expressions
- For faceted schemas, explicit type information of `property` expressions is used to determine which value tables are required
- Example:

```
(property (double integer) "triple_1_1" object)
      ⇒
(function"builtin:coalesce" (double integer)
  (custom (double)
    SQLAccessExpr INDEX VC_Doubles.{ix:  id, value:  double_value,
       triple-table-column:  obj(object)} USING JOIN triple_1_1)
  (custom (integer)
    SQLAccessExpr INDEX VC_Integers.{ix:  id, value:  int_value,
       triple-table-column:  obj(object)} USING JOIN triple_1_1))
```

Translating Lowered AQL to SQL

- This is very straightforward
  - Join structure is naturally preserved (multiple triples in AQL joins are joined using CROSS JOIN)
  - AQL literal and function expressions are translated into SQL expressions, usually simple one-to-one translations
  - Triple component expressions (AQL `property` expressions) are already translated into low-level SQL expressions
- However, not all AQL queries can be translated into legal SQL queries
  - Possible, when we use left-to-right variable binding semantics in SPARQL
  - Partial remedy: nested join flattening. Exemplified in our paper

Conclusions

- We presented a design for SPARQL-to-SQL translation using a purpose-built intermediate language
- Intermediate language can provide additional flexibility in translation. In our case:
  - Clean separation of the front-end (SPARQL) and the back-end (SQL)
  - Variable dereferencing—does not result in sub-`SELECT`s
  - Explicit expression typing and type inference based on join condition analysis
  - Untranslatable query detection and remedy by query translation (left-to-right variable binding semantics only)
    - Join flattening is not generally doable in SPARQL (without further transformations) but easily done in AQL
  - Back-end schema flexibility
- See further examples and translator source at `http://esg.cs.hut.fi/software/type-arque/`

Questions, comments?

More examples after this slide

Nested Join Flattening

- Nested optional graph patterns with two-levels up references in `FILTER`s cannot be translated directly into SQL (issue only with left-to-right variable binding)
- Consider:

```
SELECT ?i
WHERE {
  ?a ?b ?c
  OPTIONAL { ?d ?e ?f
    OPTIONAL { ?g ?h ?i FILTER(?c='123') }
  }
}
```

  - `FILTER(?c ...)` refers to a variable that is bound two-levels up (using left-to-right semantics)
- However, in AQL, we can often transform these queries to equivalent but translatable queries
- This is done by moving the conflicting left joins upwards and adding additional join conditions

Example: Nested Join Flattening

```
SELECT ?i
WHERE { ?a ?b ?c OPTIONAL { ?d ?e ?f OPTIONAL { ?g ?h ?i
                                        FILTER(?c='123') }}}
```

```
(aql-query ("tri_1")
  (select "i" (property any "tri_3" object))
  (join left ("tri_2")
    (literal boolean true)
    (join left ("tri_3")
      (function"builtin:comp-eq" (boolean)
        (property (string) "tri_1" object)
        (literal string "123"))))
  (criterion))
```

```
SELECT tri_3.obj_value AS c0
FROM InlinedTriples AS tri_1
  LEFT JOIN (InlinedTriples AS tri_2
      LEFT JOIN InlinedTriples AS tri_3 ON tri_1.obj_value='123')
    ON TRUE
```

Example: Nested Join Flattening

```
SELECT ?i
WHERE { ?a ?b ?c OPTIONAL { ?d ?e ?f OPTIONAL { ?g ?h ?i
                                        FILTER(?c='123') }}}
```

```
(aql-query ("tri_1")
  (select "i" (property any "tri_3" object))
  (join left ("tri_2")
    (literal boolean true))
  (join left ("tri_3")
    (and
      (function"builtin:comp-eq" (boolean)
        (property (string) "tri_1" object)
        (literal string "123"))
      (function"builtin:is-not-null" any
        (property (reference) "tri_2" subject))))
  (criterion))
```

```
SELECT tri_3.obj_value AS c0
FROM InlinedTriples AS tri_1
  LEFT JOIN InlinedTriples AS tri_2 ON TRUE
  LEFT JOIN InlinedTriples AS tri_3 ON
    tri_1.obj_value='123' AND tri_2.subj_value IS NOT NULL
```

Example: Type Inference and Value Joins SPARQL vs AQL

```
SELECT ?c
WHERE {
  ?a ?b ?c
}
```

```
(aql-query ("triple_1_1")
  (select "c"
    (property (string IRI double integer boolean datetime)
      "triple_1_1" object))
  (criterion (literal boolean true))
```

Example: Type Inference and Value Joins SPARQL vs AQL

```
SELECT ?c
WHERE {
  ?a ?b ?c
  FILTER(?c < 30)
}
```

```
(aql-query ("triple_1_1")
  (select "c"
    (property (double integer)
      "triple_1_1" object))
  (criterion
       (function"builtin:comp-lt" (boolean)
          (property (double integer) "triple_1_1" object)
          (literal integer 30))))
```

Example: Type Inference and Value Joins SPARQL vs AQL

```
SELECT ?c
WHERE {
  ?a ?b ?c
  FILTER(?c < 30 || ?c < 'B')
}
```

```
(aql-query ("triple_1_1")
  (select "c"
    (property (string double integer)
      "triple_1_1" object))
  (criterion (function"builtin:or" (boolean)
       (function"builtin:comp-lt" (boolean)
           (property (double integer) "triple_1_1" object)
           (literal integer 30))
       (function"builtin:comp-lt" (boolean)
           (property (string) "triple_1_1" object)
           (literal string "B")))))
```

Example: Type Inference and Value Joins
SQL Schema

Table	Description
VC_Triples	Triple table. Columns: subj, pred, obj. Values are id references to value tables.
VC_Strings	String value table. Columns: id, str_value. Contains small strings.
VC_BigStrings	String value table. Columns: id, text_value. Contains big strings.
VC_Integers	Integer value table. Columns: id, int_value.
VC_Doubles	Double value table. Columns: id, double_value.
VC_Booleans	Boolean values. Columns: id, boolean_value.
VC_Datetimes	Datetime values. Columns: id, datetime_value.

Example: Type Inference and Value Joins SPARQL vs SQL

```
SELECT ?c
WHERE {
  ?a ?b ?c
}
```

```
SELECT COALESCE(tri_obj_strs.str_value,tri_obj_bstrs.text_value,
    tri_obj_VC_IRIs.iri_value,
    CAST(tri_obj_dbls.double_value AS TEXT),
    CAST(tri_obj_ints.int_value AS TEXT),
    aqltosql_boolean_to_text(tri_obj_bools.boolean_value),
    aqltosql_timestamp_to_text(tri_obj_dts.datetime_value)) AS c0
FROM VC_Triples AS tri
  LEFT JOIN VC_Strings AS tri_obj_strs ON tri_obj_strs.id=tri.obj
  LEFT JOIN VC_BigStrings AS tri_obj_bstrs ON tri_obj_bstrs.id=tri.obj
  LEFT JOIN VC_IRIs AS tri_obj_VC_IRIs ON tri_obj_VC_IRIs.id=tri.obj
  LEFT JOIN VC_Doubles AS tri_obj_dbls ON tri_obj_dbls.id=tri.obj
  LEFT JOIN VC_Integers AS tri_obj_ints ON tri_obj_ints.id=tri.obj
  LEFT JOIN VC_Booleans AS tri_obj_bools ON tri_obj_bools.id=tri.obj
  LEFT JOIN VC_Datetimes AS tri_obj_dts ON tri_obj_dts.id=tri.obj
WHERE TRUE
```

Example: Type Inference and Value Joins SPARQL vs SQL

```
SELECT ?c
WHERE {
  ?a ?b ?c
  FILTER(?c < 30)
}
```

```
SELECT COALESCE(tri_obj_dbls.double_value,tri_obj_ints.int_value) AS c0
FROM VC_Triples AS tri
  LEFT JOIN VC_Doubles AS tri_obj_dbls ON tri_obj_dbls.id=tri.obj
  LEFT JOIN VC_Integers AS tri_obj_ints ON tri_obj_ints.id=tri.obj
WHERE COALESCE(tri_obj_dbls.double_value,tri_obj_ints.int_value)<30
  AND (tri_obj_dbls.double_value IS NOT NULL
    OR tri_obj_ints.int_value IS NOT NULL)
```

Example: Type Inference and Value Joins SPARQL vs SQL

```
SELECT ?c
WHERE {
  ?a ?b ?c
  FILTER(?c < 30 || ?c < 'B')
}
```

```
SELECT COALESCE(tri_obj_strs.str_value,tri_obj_bstrs.text_value,
    CAST(tri_obj_dbls.double_value AS TEXT),
    CAST(tri_obj_ints.int_value AS TEXT)) AS c0
FROM VC_Triples AS tri
  LEFT JOIN VC_Strings AS tri_obj_strs ON tri_obj_strs.id=tri.obj
  LEFT JOIN VC_BigStrings AS tri_obj_bstrs ON tri_obj_bstrs.id=tri.obj
  LEFT JOIN VC_Doubles AS tri_obj_dbls ON tri_obj_dbls.id=tri.obj
  LEFT JOIN VC_Integers AS tri_obj_ints ON tri_obj_ints.id=tri.obj
WHERE (COALESCE(tri_obj_dbls.double_value,tri_obj_ints.int_value)<30
    OR COALESCE(tri_obj_strs.str_value,tri_obj_bstrs.text_value)<'B')
  AND (tri_obj_strs.str_value IS NOT NULL
    OR tri_obj_bstrs.text_value IS NOT NULL
    OR tri_obj_dbls.double_value IS NOT NULL
    OR tri_obj_ints.int_value IS NOT NULL)
```