

Formalization of Multiway-Join Algorithms

Thibault Dardinier

February 6, 2026

Abstract

Worst-case optimal multiway-join algorithms are recent seminal achievement of the database community. These algorithms compute the natural join of multiple relational databases and improve in the worst case over traditional query plan optimizations of nested binary joins. In 2014, Ngo, Ré, and Rudra [1] gave a unified presentation of different multi-way join algorithms. We formalized and proved correct their "Generic Join" algorithm and extended it to support negative joins.

Contents

1	The algorithm	1
1.1	Generic algorithm	1
1.2	An instantiation	3
2	Correctness	3
2.1	Well-formed queries	3
2.2	Correctness	7
3	Example instantiations and queries	11
3.1	Instantiations	11
3.2	Queries	12

1 The algorithm

```
theory Generic_Join
  imports MFOTL_Monitor.Table
begin
```

```
type_synonym 'a atable = nat set × 'a table
type_synonym 'a query = 'a atable set
type_synonym vertices = nat set
```

1.1 Generic algorithm

```
locale getIJ =
  fixes getIJ :: 'a query ⇒ 'a query ⇒ vertices ⇒ vertices × vertices
  assumes coreProperties:  $\text{card } V \geq 2 \implies \text{getIJ } Q\_pos \ Q\_neg \ V = (I, J) \implies$ 
     $\text{card } I \geq 1 \wedge \text{card } J \geq 1 \wedge V = I \cup J \wedge I \cap J = \{\}$ 
begin
```

```
lemma getIJProperties:
  assumes  $\text{card } V \geq 2$ 
  assumes  $(I, J) = \text{getIJ } Q\_pos \ Q\_neg \ V$ 
  shows  $\text{card } I \geq 1$  and  $\text{card } J \geq 1$  and  $\text{card } I < \text{card } V$  and  $\text{card } J < \text{card } V$ 
```

and $V = I \cup J$ **and** $I \cap J = \{\}$
 <proof>

fun *projectTable* :: *vertices* \Rightarrow 'a *atable* \Rightarrow 'a *atable* **where**
projectTable V (s , t) = ($s \cap V$, *Set.image* (*restrict* V) t)

fun *filterQuery* :: *vertices* \Rightarrow 'a *query* \Rightarrow 'a *query* **where**
filterQuery V Q = *Set.filter* ($\lambda(s, _).$ \neg *Set.is_empty* ($s \cap V$)) Q

fun *filterQueryNeg* :: *vertices* \Rightarrow 'a *query* \Rightarrow 'a *query* **where**
filterQueryNeg V Q = *Set.filter* ($\lambda(A, _).$ $A \subseteq V$) Q

fun *projectQuery* :: *vertices* \Rightarrow 'a *query* \Rightarrow 'a *query* **where**
projectQuery V s = *Set.image* (*projectTable* V) s

fun *isSameIntersection* :: 'a *tuple* \Rightarrow *nat set* \Rightarrow 'a *tuple* \Rightarrow *bool* **where**
isSameIntersection $t1$ s $t2$ = ($\forall i \in s.$ $t1!i = t2!i$)

fun *semiJoin* :: 'a *atable* \Rightarrow (*nat set* \times 'a *tuple*) \Rightarrow 'a *atable* **where**
semiJoin (s , tab) (st , tup) = (s , *Set.filter* (*isSameIntersection* tup ($s \cap st$)) tab)

fun *newQuery* :: *vertices* \Rightarrow 'a *query* \Rightarrow (*nat set* \times 'a *tuple*) \Rightarrow 'a *query* **where**
newQuery V Q (st , t) = *Set.image* ($\lambda tab.$ *projectTable* V (*semiJoin* tab (st , t))) Q

fun *merge_option* :: 'a *option* \times 'a *option* \Rightarrow 'a *option* **where**
merge_option (*None*, *None*) = *None*
 | *merge_option* (*Some* x , *None*) = *Some* x
 | *merge_option* (*None*, *Some* x) = *Some* x
 | *merge_option* (*Some* a , *Some* b) = *Some* a

definition *merge* :: 'a *tuple* \Rightarrow 'a *tuple* \Rightarrow 'a *tuple* **where**
merge $t1$ $t2$ = *map* *merge_option* (*zip* $t1$ $t2$)

function (*sequential*) *genericJoin* :: *vertices* \Rightarrow 'a *query* \Rightarrow 'a *query* \Rightarrow 'a *table* **where**
genericJoin V Q_pos Q_neg =
 (*if* *card* $V \leq 1$ *then*
 ($\bigcap (_, x) \in Q_pos.$ x) $-$ ($\bigcup (_, x) \in Q_neg.$ x)
else
let (I , J) = *getIJ* Q_pos Q_neg V *in*
let Q_I_pos = *projectQuery* I (*filterQuery* I Q_pos) *in*
let Q_I_neg = *filterQueryNeg* I Q_neg *in*
let R_I = *genericJoin* I Q_I_pos Q_I_neg *in*
let Q_J_neg = $Q_neg - Q_I_neg$ *in*
let Q_J_pos = *filterQuery* J Q_pos *in*
let X = $\{(t, \text{genericJoin } J (\text{newQuery } J \text{ } Q_J_pos (I, t)) (\text{newQuery } J \text{ } Q_J_neg (I, t))) \mid t . t \in$
 $R_I\}$ *in*
 ($\bigcup (t, x) \in X.$ $\{merge \text{ } xx \text{ } t \mid xx . xx \in x\}$)

<proof>

termination

<proof>

declare *genericJoin.simps* [*simp del*]

definition *wrapperGenericJoin* :: 'a *query* \Rightarrow 'a *query* \Rightarrow 'a *table* **where**
wrapperGenericJoin Q_pos Q_neg =
 (*if* ($(\exists (A, X) \in Q_pos.$ *Set.is_empty* X) \vee ($(\exists (A, X) \in Q_neg.$ *Set.is_empty* $A \wedge \neg$ *Set.is_empty* X))
then

```

    {}
  else
    let Q = Set.filter (λ(A, _). ¬ Set.is_empty A) Q_pos in
    if Set.is_empty Q then
      (∩(A, X)∈Q_pos. X) - (∪(A, X)∈Q_neg. X)
    else
      let V = (∪(A, X)∈Q. A) in
      let Qn = Set.filter (λ(A, _). A ⊆ V ∧ card A ≥ 1) Q_neg in
      genericJoin V Q Qn
end

```

1.2 An instantiation

definition `score` :: 'a query ⇒ nat ⇒ nat **where**
`score Q i = (let relevant = Set.image (λ(_, x). card x) (Set.filter (λ(sign, _). i ∈ sign) Q) in
 let l = sorted_list_of_set relevant in
 foldl (+) 0 l
)`

definition `arg_max_list` :: ('a ⇒ nat) ⇒ 'a list ⇒ 'a **where**
`arg_max_list f l = (let m = Max (set (map f l)) in arg_min_list (λx. m - f x) l)`

lemma `arg_max_list_element`:
assumes `length l ≥ 1` **shows** `arg_max_list f l ∈ set l`
`<proof>`

definition `max_getIJ` :: 'a query ⇒ 'a query ⇒ vertices ⇒ vertices × vertices **where**
`max_getIJ Q_pos Q_neg V = (
 let l = sorted_list_of_set V in
 if Set.is_empty Q_neg then
 let x = arg_max_list (score Q_pos) l in
 ({x}, V - {x})
 else
 let x = arg_max_list (score Q_neg) l in
 (V - {x}, {x}))`

lemma `max_getIJ_coreProperties`:
assumes `card V ≥ 2`
assumes `(I, J) = max_getIJ Q_pos Q_neg V`
shows `card I ≥ 1 ∧ card J ≥ 1 ∧ V = I ∪ J ∧ I ∩ J = {}`
`<proof>`

global_interpretation `New_max`: `getIJ max_getIJ`
defines `New_max_getIJ_genericJoin = New_max.genericJoin`
and `New_max_getIJ_wrapperGenericJoin = New_max.wrapperGenericJoin`
`<proof>`

end

2 Correctness

2.1 Well-formed queries

theory `Generic_Join_Correctness`
imports `Generic_Join`
begin

definition $wf_set :: nat \Rightarrow vertices \Rightarrow bool$ **where**

$wf_set\ n\ V \longleftrightarrow (\forall x \in V. x < n)$

definition $wf_atable :: nat \Rightarrow 'a\ atable \Rightarrow bool$ **where**

$wf_atable\ n\ X \longleftrightarrow table\ n\ (fst\ X)\ (snd\ X) \wedge finite\ (fst\ X)$

definition $wf_query :: nat \Rightarrow vertices \Rightarrow 'a\ query \Rightarrow 'a\ query \Rightarrow bool$ **where**

$wf_query\ n\ V\ Q_pos\ Q_neg \longleftrightarrow (\forall X \in (Q_pos \cup Q_neg). wf_atable\ n\ X) \wedge (wf_set\ n\ V) \wedge (card\ Q_pos \geq 1)$

definition $included :: vertices \Rightarrow 'a\ query \Rightarrow bool$ **where**

$included\ V\ Q \longleftrightarrow (\forall (S, X) \in Q. S \subseteq V)$

definition $covering :: vertices \Rightarrow 'a\ query \Rightarrow bool$ **where**

$covering\ V\ Q \longleftrightarrow V \subseteq (\bigcup (S, X) \in Q. S)$

definition $non_empty_query :: 'a\ query \Rightarrow bool$ **where**

$non_empty_query\ Q = (\forall X \in Q. card\ (fst\ X) \geq 1)$

definition $rwf_query :: nat \Rightarrow vertices \Rightarrow 'a\ query \Rightarrow 'a\ query \Rightarrow bool$ **where**

$rwf_query\ n\ V\ Qp\ Qn \longleftrightarrow wf_query\ n\ V\ Qp\ Qn \wedge covering\ V\ Qp \wedge included\ V\ Qp \wedge included\ V\ Qn$
 $\wedge non_empty_query\ Qp \wedge non_empty_query\ Qn$

lemma $wf_tuple_empty: wf_tuple\ n\ \{\} v \longleftrightarrow v = replicate\ n\ None$

<proof>

lemma $table_empty: table\ n\ \{\} X \longleftrightarrow (X = empty_table \vee X = unit_table\ n)$

<proof>

context $getIJ$ **begin**

lemma $isSame_equi_dev:$

assumes $wf_set\ n\ V$

assumes $wf_tuple\ n\ A\ t1$

assumes $wf_tuple\ n\ B\ t2$

assumes $s \subseteq A$

assumes $s \subseteq B$

assumes $A \subseteq V$

assumes $B \subseteq V$

shows $isSameIntersection\ t1\ s\ t2 = (restrict\ s\ t1 = restrict\ s\ t2)$

<proof>

lemma $wf_getIJ:$

assumes $card\ V \geq 2$

assumes $wf_set\ n\ V$

assumes $(I, J) = getIJ\ Q_pos\ Q_neg\ V$

shows $wf_set\ n\ I$ **and** $wf_set\ n\ J$

<proof>

lemma $wf_projectTable:$

assumes $wf_atable\ n\ X$

shows $wf_atable\ n\ (projectTable\ I\ X) \wedge (fst\ (projectTable\ I\ X) = (fst\ X \cap I))$

<proof>

lemma $set_filterQuery:$

assumes $QQ = filterQuery\ I\ Q$

assumes $non_empty_query\ Q$

shows $\forall X \in Q. (card\ (fst\ X \cap I) \geq 1 \longleftrightarrow X \in QQ)$

<proof>

lemma *wf_filterQuery*:

assumes $I \subseteq V$

assumes $\text{card } I \geq 1$

assumes $\text{rwf_query } n \ V \ Qp \ Qn$

assumes $QQp = \text{filterQuery } I \ Qp$

assumes $QQn = \text{filterQueryNeg } I \ Qn$

shows $\text{wf_query } n \ I \ QQp \ QQn \ \text{non_empty_query } QQp \ \text{covering } I \ QQp$

<proof>

lemma *wf_set_subset*:

assumes $I \subseteq V$

assumes $\text{card } I \geq 1$

assumes $\text{wf_set } n \ V$

shows $\text{wf_set } n \ I$

<proof>

lemma *wf_projectQuery*:

assumes $\text{card } I \geq 1$

assumes $\text{wf_query } n \ I \ Q \ Qn$

assumes $\text{non_empty_query } Q$

assumes $\text{covering } I \ Q$

assumes $\forall X \in Q. \text{card } (\text{fst } X \cap I) \geq 1$

assumes $QQ = \text{projectQuery } I \ Q$

assumes $\text{included } I \ Qn$

assumes $\text{non_empty_query } Qn$

shows $\text{rwf_query } n \ I \ QQ \ Qn$

<proof>

lemma *wf_firstRecursiveCall*:

assumes $\text{rwf_query } n \ V \ Qp \ Qn$

assumes $\text{card } V \geq 2$

assumes $(I, J) = \text{getIJ } Qp \ Qn \ V$

assumes $Q_I_pos = \text{projectQuery } I \ (\text{filterQuery } I \ Qp)$

assumes $Q_I_neg = \text{filterQueryNeg } I \ Qn$

shows $\text{rwf_query } n \ I \ Q_I_pos \ Q_I_neg$

<proof>

lemma *wf_atable_subset*:

assumes $\text{table } n \ V \ X$

assumes $Y \subseteq X$

shows $\text{table } n \ V \ Y$

<proof>

lemma *same_set_semiJoin*:

$\text{fst } (\text{semiJoin } x \ \text{other}) = \text{fst } x$

<proof>

lemma *wf_semiJoin*:

assumes $\text{card } J \geq 1$

assumes $\text{wf_query } n \ J \ Q \ Qn$

assumes $\text{non_empty_query } Q$

assumes $\text{covering } J \ Q$

assumes $\forall X \in Q. \text{card } (\text{fst } X \cap J) \geq 1$

assumes $QQ = (\text{Set.image } (\lambda \text{tab}. \text{semiJoin } \text{tab } (\text{st}, \text{t})) \ Q)$

shows $\text{wf_query } n \ J \ QQ \ Qn \ \text{non_empty_query } QQ \ \text{covering } J \ QQ$

<proof>

lemma *newQuery_equiv_def*:
 $newQuery\ V\ Q\ (st, t) = projectQuery\ V\ (Set.image\ (\lambda tab.\ semiJoin\ tab\ (st, t))\ Q)$
 ⟨proof⟩

lemma *included_project*:
 $included\ V\ (projectQuery\ V\ Q)$
 ⟨proof⟩

lemma *non_empty_newQuery*:
assumes $Q1 = filterQuery\ J\ Q0$
assumes $Q2 = newQuery\ J\ Q1\ (I, t)$
assumes $\forall X \in Q0.\ wf_atable\ n\ X$
shows $non_empty_query\ Q2$
 ⟨proof⟩

lemma *wf_newQuery*:
assumes $card\ J \geq 1$
assumes $wf_query\ n\ J\ Q\ Qn0$
assumes $non_empty_query\ Q$
assumes $covering\ J\ Q$
assumes $\forall X \in Q.\ card\ (fst\ X \cap J) \geq 1$
assumes $QQ = newQuery\ J\ Q\ t$
assumes $QQn = newQuery\ J\ Qn\ t$
assumes $non_empty_query\ Qn$
assumes $Qn = filterQuery\ J\ Qn0$
shows $rwf_query\ n\ J\ QQ\ QQn$
 ⟨proof⟩

lemma *subset_Q_neg*:
assumes $rwf_query\ n\ V\ Q\ Qn$
assumes $QQn \subseteq Qn$
shows $rwf_query\ n\ V\ Q\ QQn$
 ⟨proof⟩

lemma *wf_secondRecursiveCalls*:
assumes $card\ V \geq 2$
assumes $rwf_query\ n\ V\ Q\ Qn$
assumes $(I, J) = getIJ\ Q\ Qn\ V$
assumes $Qns \subseteq Qn$
assumes $Q_J_neg = filterQuery\ J\ Qns$
assumes $Q_J_pos = filterQuery\ J\ Q$
shows $rwf_query\ n\ J\ (newQuery\ J\ Q_J_pos\ t)\ (newQuery\ J\ Q_J_neg\ t)$
 ⟨proof⟩

lemma *simple_merge_option*:
 $merge_option\ (a, b) = None \longleftrightarrow (a = None \wedge b = None)$
 ⟨proof⟩

lemma *wf_merge*:
assumes $wf_tuple\ n\ I\ t1$
assumes $wf_tuple\ n\ J\ t2$
assumes $V = I \cup J$
assumes $t = merge\ t1\ t2$
shows $wf_tuple\ n\ V\ t$
 ⟨proof⟩

lemma *wf_inter*:

assumes $rwf_query\ n\ \{i\}\ Q\ Qn$
assumes $(sa, a) \in Q$
assumes $(sb, b) \in Q$
shows $table\ n\ \{i\}\ (a \cap b)$
 $\langle proof \rangle$

lemma $table_subset$:
assumes $table\ n\ V\ T$
assumes $S \subseteq T$
shows $table\ n\ V\ S$
 $\langle proof \rangle$

lemma wf_base_case :
assumes $card\ V = 1$
assumes $rwf_query\ n\ V\ Q\ Qn$
assumes $R = genericJoin\ V\ Q\ Qn$
shows $table\ n\ V\ R$
 $\langle proof \rangle$

lemma $filter_Q_J_neg_same$:
assumes $card\ V \geq 2$
assumes $(I, J) = getIJ\ Q\ Qn\ V$
assumes $Q_I_neg = filterQueryNeg\ I\ Qn$
assumes $rwf_query\ n\ V\ Q\ Qn$
shows $filterQuery\ J\ (Qn - Q_I_neg) = Qn - Q_I_neg$ (**is** $?A = ?B$)
 $\langle proof \rangle$

lemma $vars_genericJoin$:
assumes $card\ V \geq 2$
assumes $(I, J) = getIJ\ Q\ Qn\ V$
assumes $Q_I_pos = projectQuery\ I\ (filterQuery\ I\ Q)$
assumes $Q_I_neg = filterQueryNeg\ I\ Qn$
assumes $R_I = genericJoin\ I\ Q_I_pos\ Q_I_neg$
assumes $Q_J_neg = filterQuery\ J\ (Qn - Q_I_neg)$
assumes $Q_J_pos = filterQuery\ J\ Q$
assumes $X = \{(t, genericJoin\ J\ (newQuery\ J\ Q_J_pos\ (I, t))\ (newQuery\ J\ Q_J_neg\ (I, t))) \mid t . t \in R_I\}$
assumes $R = (\bigcup (t, x) \in X. \{merge\ xx\ t \mid xx . xx \in x\})$
assumes $rwf_query\ n\ V\ Q\ Qn$
shows $R = genericJoin\ V\ Q\ Qn$
 $\langle proof \rangle$

lemma $base_genericJoin$:
assumes $card\ V \leq 1$
shows $genericJoin\ V\ Q\ Qn = (\bigcap (_, x) \in Q. x) - (\bigcup (_, x) \in Qn. x)$
 $\langle proof \rangle$

lemma $wf_genericJoin$:
 $\llbracket rwf_query\ n\ V\ Q\ Qn; card\ V \geq 1 \rrbracket \implies table\ n\ V\ (genericJoin\ V\ Q\ Qn)$
 $\langle proof \rangle$

2.2 Correctness

lemma $base_correctness$:
assumes $card\ V = 1$
assumes $rwf_query\ n\ V\ Q\ Qn$
assumes $R = genericJoin\ V\ Q\ Qn$
shows $z \in genericJoin\ V\ Q\ Qn \iff wf_tuple\ n\ V\ z \wedge (\forall (A, X) \in Q. restrict\ A\ z \in X) \wedge (\forall (A, X) \in Qn.$

restrict A z ∉ X
⟨*proof*⟩

lemma *simple_list_index_equality*:
 assumes *length a = n*
 assumes *length b = n*
 assumes $\forall i < n. a!i = b!i$
 shows *a = b*
 ⟨*proof*⟩

lemma *simple_restrict_none*:
 assumes *i < length X*
 assumes *i ∉ A*
 shows *(restrict A X)!i = None*
 ⟨*proof*⟩

lemma *simple_restrict_some*:
 assumes *i < length X*
 assumes *i ∈ A*
 shows *(restrict A X)!i = X!i*
 ⟨*proof*⟩

lemma *merge_restrict*:
 assumes *A ∩ J = {}*
 assumes *True*
 assumes *length xx = n*
 assumes *length t = n*
 assumes *restrict J xx = xx*
 shows *restrict A (merge xx t) = restrict A t*
 ⟨*proof*⟩

lemma *restrict_idle_include*:
 assumes *wf_tuple n A v*
 assumes *A ⊆ I*
 shows *restrict I v = v*
 ⟨*proof*⟩

lemma *merge_index*:
 assumes *I ∩ J = {}*
 assumes *wf_tuple n I tI*
 assumes *wf_tuple n J tJ*
 assumes *t = merge tI tJ*
 assumes *i < n*
 shows $(i \in I \wedge t!i = tI!i) \vee (i \in J \wedge t!i = tJ!i) \vee (i \notin I \wedge i \notin J \wedge t!i = \text{None})$
 ⟨*proof*⟩

lemma *restrict_index_in*:
 assumes *i < length X*
 assumes *i ∈ I*
 shows *(restrict I X)!i = X!i*
 ⟨*proof*⟩

lemma *restrict_index_out*:
 assumes *i < length X*
 assumes *i ∉ I*
 shows *(restrict I X)!i = None*
 ⟨*proof*⟩

lemma *merge_length*:
assumes *length a = n*
assumes *length b = n*
shows *length (merge a b) = n*
<proof>

lemma *real_restrict_merge*:
assumes $I \cap J = \{\}$
assumes *wf_tuple n I tI*
assumes *wf_tuple n J tJ*
shows $\text{restrict } I \text{ (merge } tI \text{ } tJ) = \text{restrict } I \text{ } tI \wedge \text{restrict } J \text{ (merge } tI \text{ } tJ) = \text{restrict } J \text{ } tJ$
<proof>

lemma *simple_set_image_id*:
assumes $\forall x \in X. f x = x$
shows $\text{Set.image } f X = X$
<proof>

lemma *nested_include_restrict*:
assumes $\text{restrict } I z = t$
assumes $A \subseteq I$
shows $\text{restrict } A z = \text{restrict } A t$
<proof>

lemma *restrict_nested*:
 $\text{restrict } A \text{ (restrict } B x) = \text{restrict } (A \cap B) x$ (**is** *?lhs = ?rhs*)
<proof>

lemma *newQuery_equi_dev*:
 $\text{newQuery } V Q (I, t) = \text{Set.image } (\text{projectTable } V) (\text{Set.image } (\lambda \text{tab. semiJoin } \text{tab } (I, t)) Q)$
<proof>

lemma *projectTable_idle*:
assumes *table n A X*
assumes $A \subseteq I$
shows $\text{projectTable } I (A, X) = (A, X)$
<proof>

lemma *restrict_partition_merge*:
assumes $I \cup J = V$
assumes *wf_tuple n V z*
assumes $xx = \text{restrict } J z$
assumes $t = \text{restrict } I z$
assumes $\text{Set.is_empty } (I \cap J)$
shows $z = \text{merge } xx t$
<proof>

lemma *restrict_merge*:
assumes $zI = \text{restrict } I z$
assumes $zJ = \text{restrict } J z$
assumes $\text{restrict } (A \cap I) zI \in \text{Set.image } (\text{restrict } I) X$
assumes $\text{restrict } (A \cap J) zJ \in \text{Set.image } (\text{restrict } J) (\text{Set.filter } (\text{isSameIntersection } zI (A \cap I)) X)$
assumes $z = \text{merge } zJ zI$
assumes *table n A X*
assumes $A \subseteq I \cup J$
assumes $\text{card } (A \cap I) \geq 1$
assumes *wf_set n (I ∪ J)*
assumes *wf_tuple n (I ∪ J) z*

shows $\text{restrict } A \ z \in X$
 ⟨*proof*⟩

lemma *partial_correctness*:

assumes $V = I \cup J$
assumes $\text{Set.is_empty } (I \cap J)$
assumes $\text{card } I \geq 1$
assumes $\text{card } J \geq 1$
assumes $Q_I_pos = \text{projectQuery } I \ (\text{filterQuery } I \ Q)$
assumes $Q_J_pos = \text{filterQuery } J \ Q$
assumes $Q_I_neg = \text{filterQueryNeg } I \ Qn$
assumes $Q_J_neg = \text{filterQuery } J \ (Qn - Q_I_neg)$
assumes $NQ_pos = \text{newQuery } J \ Q_J_pos \ (I, t)$
assumes $NQ_neg = \text{newQuery } J \ Q_J_neg \ (I, t)$
assumes $R_NQ = \text{genericJoin } J \ NQ_pos \ NQ_neg$
assumes $\forall x. (x \in R_I \longleftrightarrow \text{wf_tuple } n \ I \ x \wedge (\forall (A, X) \in Q_I_pos. \text{restrict } A \ x \in X) \wedge (\forall (A, X) \in Q_I_neg. \text{restrict } A \ x \notin X))$
assumes $\forall y. (y \in R_NQ \longleftrightarrow \text{wf_tuple } n \ J \ y \wedge (\forall (A, X) \in NQ_pos. \text{restrict } A \ y \in X) \wedge (\forall (A, X) \in NQ_neg. \text{restrict } A \ y \notin X))$
assumes $z = \text{merge } xx \ t$
assumes $t \in R_I$
assumes $xx \in R_NQ$
assumes $\text{rwf_query } n \ V \ Q \ Qn$
shows $\text{wf_tuple } n \ V \ z \wedge (\forall (A, X) \in Q. \text{restrict } A \ z \in X) \wedge (\forall (A, X) \in Qn. \text{restrict } A \ z \notin X)$
 ⟨*proof*⟩

lemma *simple_set_inter*:

assumes $I \subseteq (\bigcup X \in A. f \ X)$
shows $I \subseteq (\bigcup X \in A. (f \ X) \cap I)$
 ⟨*proof*⟩

lemma *union_restrict*:

assumes $\text{restrict } I \ z1 = \text{restrict } I \ z2$
assumes $\text{restrict } J \ z1 = \text{restrict } J \ z2$
shows $\text{restrict } (I \cup J) \ z1 = \text{restrict } (I \cup J) \ z2$
 ⟨*proof*⟩

lemma *partial_correctness_direct*:

assumes $V = I \cup J$
assumes $\text{Set.is_empty } (I \cap J)$
assumes $\text{card } I \geq 1$
assumes $\text{card } J \geq 1$
assumes $Q_I_pos = \text{projectQuery } I \ (\text{filterQuery } I \ Q)$
assumes $Q_J_pos = \text{filterQuery } J \ Q$
assumes $Q_I_neg = \text{filterQueryNeg } I \ Qn$
assumes $Q_J_neg = \text{filterQuery } J \ (Qn - Q_I_neg)$
assumes $R_I = \text{genericJoin } I \ Q_I_pos \ Q_I_neg$
assumes $X = \{(t, \text{genericJoin } J \ (\text{newQuery } J \ Q_J_pos \ (I, t)) \ (\text{newQuery } J \ Q_J_neg \ (I, t))) \mid t . t \in R_I\}$
assumes $R = (\bigcup (t, x) \in X. \{\text{merge } xx \ t \mid xx . xx \in x\})$
assumes $R_NQ = \text{genericJoin } J \ NQ_pos \ NQ_neg$
assumes $\forall x. (x \in R_I \longleftrightarrow \text{wf_tuple } n \ I \ x \wedge (\forall (A, X) \in Q_I_pos. \text{restrict } A \ x \in X) \wedge (\forall (A, X) \in Q_I_neg. \text{restrict } A \ x \notin X))$
assumes $\forall t \in R_I. (\forall y. (y \in \text{genericJoin } J \ (\text{newQuery } J \ Q_J_pos \ (I, t)) \ (\text{newQuery } J \ Q_J_neg \ (I, t))) \longleftrightarrow \text{wf_tuple } n \ J \ y \wedge (\forall (A, X) \in (\text{newQuery } J \ Q_J_pos \ (I, t)). \text{restrict } A \ y \in X) \wedge (\forall (A, X) \in (\text{newQuery } J \ Q_J_neg \ (I, t)). \text{restrict } A \ y \notin X))$
assumes $\text{wf_tuple } n \ V \ z \wedge (\forall (A, X) \in Q. \text{restrict } A \ z \in X) \wedge (\forall (A, X) \in Qn. \text{restrict } A \ z \notin X)$

```

assumes rwf_query n V Q Qn
shows  $z \in R$ 
<proof>

```

```

lemma obvious_forall:
assumes  $\forall x \in X. P x$ 
assumes  $x \in X$ 
shows  $P x$ 
<proof>

```

```

lemma correctness:
 $\llbracket \text{rwf\_query } n V Q Qn; \text{card } V \geq 1 \rrbracket \implies (z \in \text{genericJoin } V Q Qn \iff \text{wf\_tuple } n V z \wedge$ 
 $(\forall (A, X) \in Q. \text{restrict } A z \in X) \wedge (\forall (A, X) \in Qn. \text{restrict } A z \notin X))$ 
<proof>

```

```

lemma wf_set_finite:
assumes wf_set n A
shows finite A
<proof>

```

```

lemma vars_wrapperGenericJoin:
fixes  $Q :: 'a \text{ query}$  and  $Q\_pos :: 'a \text{ query}$  and  $Q\_neg :: 'a \text{ query}$ 
and  $V :: \text{nat set}$  and  $Qn :: 'a \text{ query}$ 
assumes  $Q = \text{Set.filter } (\lambda(A, \_). \neg \text{Set.is\_empty } A) Q\_pos$ 
and  $V = (\bigcup (A, X) \in Q. A)$ 
and  $Qn = \text{Set.filter } (\lambda(A, \_). A \subseteq V \wedge \text{card } A \geq 1) Q\_neg$ 
and  $\neg \text{Set.is\_empty } Q$ 
and  $\neg((\exists (A, X) \in Q\_pos. \text{Set.is\_empty } X) \vee (\exists (A, X) \in Q\_neg. \text{Set.is\_empty } A \wedge \neg \text{Set.is\_empty } X))$ 
shows  $\text{wrapperGenericJoin } Q\_pos Q\_neg = \text{genericJoin } V Q Qn$ 
<proof>

```

```

lemma wrapper_correctness:
assumes  $\text{card } Q\_pos \geq 1$ 
assumes  $\forall (A, X) \in (Q\_pos \cup Q\_neg). \text{table } n A X \wedge \text{wf\_set } n A$ 
shows  $z \in \text{wrapperGenericJoin } Q\_pos Q\_neg \iff \text{wf\_tuple } n (\bigcup (A, X) \in Q\_pos. A) z \wedge (\forall (A, X) \in Q\_pos. \text{restrict } A z \in X) \wedge (\forall (A, X) \in Q\_neg. \text{restrict } A z \notin X)$ 
<proof>

```

end

end

3 Example instantiations and queries

```

theory Examples_Join
imports Generic_Join
begin

```

3.1 Instantiations

```

global_interpretation Max_getIJ: getIJ  $\lambda\_ \_ V. (V - \{\text{Max } V\}, \{\text{Max } V\})$ 
defines  $\text{Max\_getIJ\_genericJoin} = \text{Max\_getIJ.genericJoin}$ 
and  $\text{Max\_getIJ\_wrapperGenericJoin} = \text{Max\_getIJ.wrapperGenericJoin}$ 
<proof>

```

```

global_interpretation Min_getIJ: getIJ  $\lambda\_ \_ V. (\{\text{Min } V\}, V - \{\text{Min } V\})$ 

```

```

defines Min_getIJ_genericJoin = Min_getIJ.genericJoin
and Min_getIJ_wrapperGenericJoin = Min_getIJ.wrapperGenericJoin
(proof)

```

3.2 Queries

```

value Max_getIJ.genericJoin {0, 1} {{(0, 1), {[Some (0 :: nat), Some 0], [Some 1, Some 1]}}, ({0, 1}, {[Some 0, Some 0], [Some 0, Some 1]})} {} :: nat table
value Min_getIJ.genericJoin {0, 1} {{(0, 1), {[Some (0 :: nat), Some 0], [Some 1, Some 1]}}, ({0, 1}, {[Some 0, Some 0], [Some 0, Some 1]})} {} :: nat table

```

```

fun protoTableTriangle :: nat ⇒ nat table where
  protoTableTriangle 0 = {[Some 0, Some 0]}
| protoTableTriangle (Suc n) = (protoTableTriangle n) ∪ {[Some (Suc n), Some 0], [Some 0, Some (Suc n)]}

```

```

fun auxInsertNoneTriangle :: nat tuple ⇒ nat ⇒ nat tuple where
  auxInsertNoneTriangle l 0 = None # l
| auxInsertNoneTriangle (x # q) (Suc n) = x # (auxInsertNoneTriangle q n)
| auxInsertNoneTriangle [] (Suc v) = undefined

```

```

fun insertNoneTriangle :: nat table ⇒ nat ⇒ nat table where
  insertNoneTriangle t n = {auxInsertNoneTriangle x n | x . x ∈ t}

```

```

value set [0 ..< 5]

```

```

fun getTableTriangle :: nat ⇒ nat ⇒ nat atable where
  getTableTriangle n i = ({0, 1, 2} - {i}, insertNoneTriangle (protoTableTriangle n) i)

```

```

fun getQueryTriangle :: nat ⇒ nat query where
  getQueryTriangle n = {getTableTriangle n 0, getTableTriangle n 1, getTableTriangle n 2}

```

```

definition verticesTriangle :: vertices where verticesTriangle = {0, 1, 2}

```

```

value getQueryTriangle 2

```

```

value Max_getIJ.genericJoin verticesTriangle (getQueryTriangle 2) {{(0, 2), {[Some 0, None, Some 0]}}

```

```

value let n = 2 in let ((_, A), (_, B), (_, C)) = (getTableTriangle n 0, getTableTriangle n 1, getTableTriangle n 2) in

```

```

let AB = join A True B in join AB True C

```

```

value Min_getIJ.wrapperGenericJoin (getQueryTriangle 2) {}

```

```

value Max_getIJ.wrapperGenericJoin (getQueryTriangle 2) {}

```

```

value New_max.wrapperGenericJoin (getQueryTriangle 2) {}

```

```

end

```

References

- [1] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: New developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4):5–16, Feb. 2014.