LOGICBLOX®

TECHNICAL REPORT

Leapfrog Triejoin: A Worst-Case Optimal Join Algorithm

October 1, 2012
Todd L. Veldhuizen

LB1201

A∩B∩C

A

B

C

0   0   0

1   *seek*(2)

2   2

3   *seek*(3)

4   4

seek(6)   5   *seek*(8)

6   6   6

7   7   *seek*(8)

8   8   8   8

next()   9   9

10   *seek*(11)   *seek*(10)

11

+∞
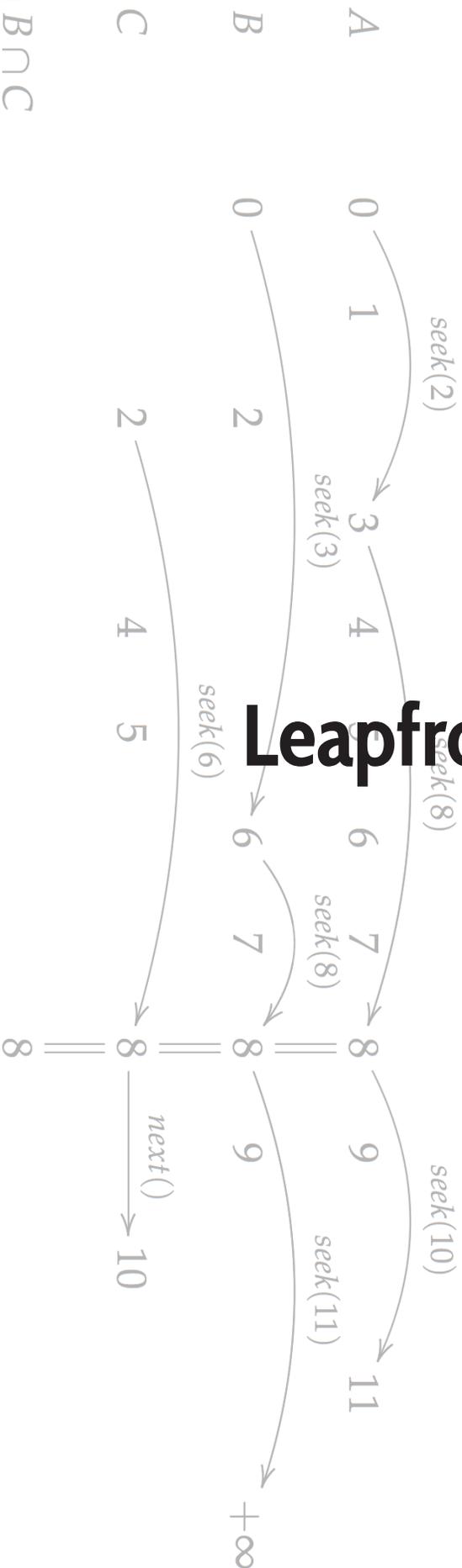
Leapfrog Triejoin: a worst-case optimal join algorithm

Todd L. Veldhuizen[*]

Contents

1 Introduction

We recently had occasion to study and admire the join algorithm of Ngo, Porat, Ré and Rudra [2] (henceforth NPRR). Given bounds on the input relation sizes, the running time of NPRR is bounded by the largest possible result size, which may be determined using the fractional edge cover method [1].

Our commercial Datalog system LogicBlox®employs a novel join algorithm (*leapfrog triejoin*) which performs conspicuously well over diverse benchmarks; we were curious how it would compare. We establish that leapfrog triejoin is also worst-case optimal for fully conjunctive queries, and in fact satisfies a more exacting optimality criterion. Our algorithm may offer a practical competitor to NPRR, being easy to absorb, simple to implement, and having a concise optimality proof.

Acknowledgements

---

[*]LogicBlox Inc., `tveldhui@acm.org`

1.1 Notations and conventions

All logarithms are base 2, and $[n] = \{1, \ldots, n\}$. Complexity analyses assume the RAM machine model.

We assume structures defined over universes which are subsets of $\mathbb{N}$, the natural numbers. In algorithm descriptions we use int as a synonym for $\mathbb{N}$.

For a binary relation $R(a, b)$, we write $R(a, \_)$ for the projection $\pi_a(R)$, i.e., the set $\{a : \exists b . (a, b) \in R\}$. For a parameter a, we write $R_a(b)$ for the *curried* version of R, i.e., the relation $\{b : (a, b) \in R\}$. Similarly for relations of arity > 2, e.g., for $S(a, b, c)$ we write $S_a(b, c)$ and $S_{a,b}(c)$ for currying.

2 Leapfrog Triejoin

Leapfrog triejoin is a a join algorithm for $\exists_1$ queries, that is, queries definable by first-order formulae without universal quantifiers (and, needless to say, excluding negated existential quantifiers.) We first describe the leapfrog join for unary relations (Section 2.1). This extends without fuss to the triejoin algorithm for fully conjunctive queries (Section 2.4). With minor embellishments, leapfrog triejoin can tackle $\exists_1$ queries; we mention these extensions in passing, but the focus (and particularly, the complexity analysis) is for fully conjunctive queries.

2.1 Leapfrog join for unary predicates

Leapfrog join is a variant of sort-merge join which simultaneously joins unary relations $A_1(x), \ldots, A_k(x)$ and has running time proportional to the size of the smallest relation. It is of no particular novelty, but serves as the basic building block for Leapfrog Triejoin. Its performance bound underpins the complexity analyses which follow.

For the purposes of leapfrog join, the relations $A_i \subseteq \mathbb{N}$ are presented in sorted order by linear iterators, one for each relation, with this interface:

int key()	Returns the key at the current iterator position
next()	Proceeds to the next key
seek(int seekKey)	Position the iterator at a least upper bound for seekKey, i.e. the least key $\geq$ seekKey, or move to end if no such key exists. The sought key must be $\geq$ the key at the current position.
bool atEnd()	Returns true if iterator is at the end.

All iterator methods for a relation A are required to take $O(\log N)$ time, where $N = |A|$ is the cardinality. Moreover, if m keys are visited in ascending order, the amortized complexity is required to be $O(1 + \log(N/m))$, as usual for balanced tree data structures.[1]

---

[1] For example, if every key is visited in order then $m = N$ and the amortized complexity is $O(1)$. Rather than returning to the tree root for each seek() request, the iterator ascends just far enough to find an upper bound for the key sought.

Leapfrog join is itself implemented as an instance of the linear iterator interface; it provides an iterator for the intersection $A_1 \cap \cdots \cap A_k$. The algorithm uses an array $\mathsf{Iter}[0 \ldots k-1]$ of pointers to iterators, one for each relation. In operation, the join tracks the smallest and largest keys at which iterators are positioned, and repeatedly moves an iterator at the smallest key to a least upper bound for the largest key, 'leapfrogging' the iterators until they are all positioned at the same key. Detailed descriptions of the algorithm follow; some readers may choose to skip to the complexity analysis (Section 2.2).

When the leapfrog join iterator is constructed, the following method is used to initialize and find the first result:

Function leapfrog-init

> **if** *any iterator has* atEnd() *true* **then**
> > atEnd := true ;
>
> **else**
> > atEnd := false ;
> > sort the array $\mathsf{Iter}[0..k-1]$ by keys at which the iterators are positioned ;
> > p := 0 ;
> > leapfrog-search()

The main workhorse is the leapfrog-search algorithm, which finds the next key in the intersection $A_1 \cap \cdots \cap A_k$:

Function leapfrog-search

> $x' := \mathsf{Iter}\big[(p-1) \mod k\big].\mathrm{key}()$;   // Max key of any iter
> **while** *true* **do**
> > $x := \mathsf{Iter}[p].\mathrm{key}()$;   // Least key of any iter
> > **if** $x = x'$ **then**
> > > key := x ;   // All iters at same key
> > > return;
> >
> > **else**
> > > $\mathsf{Iter}[p].\mathrm{seek}(x')$;
> > > **if** $\mathsf{Iter}[p].\mathrm{atEnd}()$ **then**
> > > > atEnd := true ;
> > > > return;
> > >
> > > **else**
> > > > $x' := \mathsf{Iter}[p].\mathrm{key}()$;
> > > > p := p+1 mod k;

Immediately after leapfrog-init(), the leapfrog join iterator is positioned at the first result, if
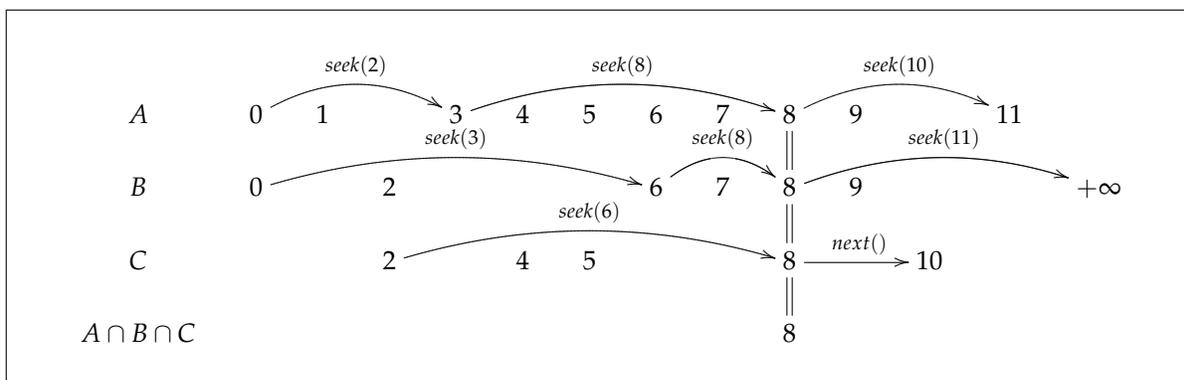
Fig. 1: Example of a leapfrog join of three relations A, B, C, with $A = \{0, 1, 3, 4, 5, 6, 7, 8, 9, 11\}$ and B, C as shown in the second and third rows. Initially the iterators for A,B,C are positioned (respectively) at 0, 0, and 2. The iterator for A performs a seek(2) which lands it at 3; the iterator for B then performs a seek(3) which lands at 6; the iterator for C does seek(6) which lands at 8, etc.

any; subsequent results are had by calling leapfrog-next().

Function leapfrog-next

Iter$[p]$.next();
if Iter$[p]$.atEnd() **then**
   | atEnd := true;
else
   | p := p+1 mod k;
   | leapfrog-search();

Figure 1 illustrates a join of three relations.

To complete the linear iterator interface, we define a leapfrog-seek() function which finds the first element of $R_1 \cap \cdots \cap R_k$ which is $\geq$ seekKey:

Function leapfrog-seek(int seekKey)

Iter$[p]$.seek($seekKey$);
if Iter$[p]$.atEnd() **then**
   | atEnd := true;
else
   | p := p+1 mod k;
   | leapfrog-search();

The leapfrog join is able to do substantially better than pairwise joins in some scenarios. For instance, suppose we have relations A, B, C where $A = \{0, \cdots, 2n-1\}$, $B = \{n, \cdots, 3n-1\}$, and $C = \{0, \cdots, n-1, 2n, \ldots, 3n-1\}$. Any pairwise join will produce n results, but the intersection $A \cap B \cap C$ is empty; the leapfrog join determines this in $O(1)$ steps.[2]

---

[2] Note, however, that this bound does not hold if we apply a random permutation to the universe.
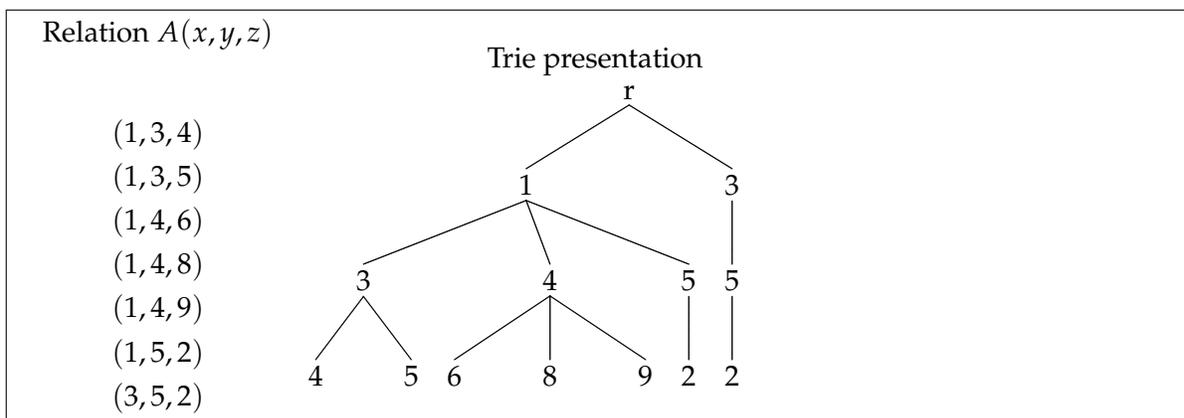
Relation $A(x,y,z)$

Trie presentation

Fig. 2: Example: Trie presentation of a relation $A(x,y,z)$. When open() is invoked at some node n, the linear iterator methods next(), seek() and atEnd() present the children of n. In the above example, invoking open() thrice on an iterator positioned at r would move to the leaf node $[1,3,4]$; next() would then move to leaf node $[1,3,5]$; another next() would result in the iterator being atEnd(). The sequence up(), next(), open() would then move the iterator to leaf node $[1,4,6]$.

2.2 Complexity of leapfrog join

In the complexity analyses which follow, we omit constant factors which depend only on the join structure; e.g. for a join of k unary relations, factors depending only on k are omitted.

Let $N_{min} = \min\{|A_1|,\ldots,|A_k|\}$ be the cardinality of the smallest relation in the join, and $N_{max} = \max\{|A_1|,\ldots,|A_k|\}$ the largest.

Proposition 2.1. *The running time of leapfrog join is $O\left(N_{min}\log\left(N_{max}/N_{min}\right)\right)$.*

Proof. The leapfrog algorithm advances the iterators in a fixed order: each iterator is advanced every k steps of the algorithm. An iterator for a relation with cardinality N can be advanced at most N times before reaching the end; therefore the number of steps is at most $k \cdot N_{min}$. An iterator which visits m of N values in order is stipulated to have amortized cost $O(1 + \log(N/m))$; the iterator for a largest relation will have $N = N_{max}$ and $m = N_{min}$, for total cost $N_{min} \cdot O(1 + \log(N_{max}/N_{min}))$.   □

2.3 Trie iterators

We extend the linear iterator interface to handle relations of arity > 1. Relations such as $A(x,y,z)$ are presented as *tries* with each tuple $(x,y,z) \in A$ corresponding to a unique path through the trie from the root to a leaf (Figure 2).

Upon initialization, trie iterators are positioned at the root r. The linear iterator API is augmented with two methods for trie-navigation:

void open();	Proceed to the first key at the next depth
void up();	Return to the parent key at the previous depth

A trie iterator for a materialized relation is required to have $O(\log N)$ time for the open() and up() methods.

With a bit of bookkeeping and no particular cleverness, a linear iterator for a sequence of tuples (x, y, z) can be presented as a TrieIterator and vice versa, with each operation taking $O(\log N)$ time.[3]

2.4   Leapfrog Triejoin

We describe here the *Leapfrog Triejoin* algorithm for conjunctive joins.

The triejoin algorithm requires the optimizer to choose a *variable ordering*, i.e., some permutation of the variables appearing in the join. For example, in the join $R(a, b), S(b, c), T(a, c)$ we might choose the variable ordering $[a, b, c]$. Choosing a good variable ordering can be crucial for performance;

Triejoin requires a restricted form of conjunctive joins, attained via some simple rewrites:

1. Each variable can appear at most once in each argument list. For example, $R(x, x)$ would be rewritten to $R(x, y), x = y$ to satisfy this requirement.[4]

2. Each argument list must be a subsequence of the variable-ordering. For example, if the chosen variable ordering were $[a, b, c]$ and the join contained a term $U(c, a)$, we would rewrite this to $U'(a, c)$ and define an index $U'(a, c) \equiv U(c, a)$. (In practice we install indices automatically when required by such rewrites, and maintain them for use in future queries.)

3. Each relation symbol appears at most once in the query. For a query such as $E(x, y)$, $E(y, z)$ we simply introduce a copy $E' \equiv E$ and rewrite to $E(x, y), E'(y, z)$. This avoids awkwardness in the complexity analysis, and is not required for implementation purposes.

4. Constants may not appear in argument lists. A subformula such as $A(x, 2)$ is rewritten to $A(x, y), C_2(y)$, where $C_2 = \{2\}$.[5]

Leapfrog triejoin employs one leapfrog join for each variable. Consider the example $R(a, b)$, $S(b, c), T(a, c)$ with the variable ordering $[a, b, c]$. The leapfrog joins employed for the variables a, b, c are:[6]

Variable	Leapfrog join	Remarks
a	$R(a, \_), T(a, \_)$	Finds a present in R, T projections
b	$R_a(b), S(b, \_)$	For specific a, finds b values
c	$S_b(c), T_a(c)$	For specific a, b, finds c values

---

[3] For example, to perform a next() operation when positioned at the node $x = 1$ of Figure 2, seek the least upper bound of $(1, +\infty, +\infty)$ in the linear presentation of tuples; this will reach the record $(3, 5, 2)$.

[4] The $x = y$ term may be presented as a nonmaterialized view of a predicate $Id(x, y) \Leftrightarrow (x = y)$.

[5] In practice C_2 can be presented as a nonmaterialized view.

[6] Recall that $R(a, \_)$ is the projection $\{a : \exists b . (a, b) \in R\}$, and $R_a(b)$ is the 'curried' form $\{b : (a, b) \in R\}$.
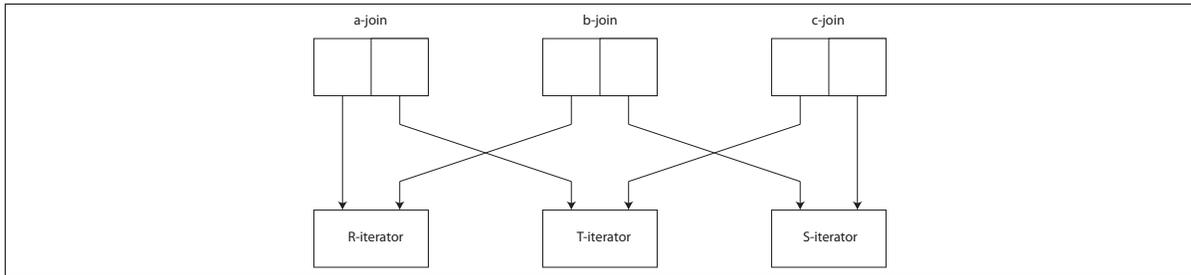
Fig. 3: Example for $R(a,b)$, $S(b,c)$, $T(a,c)$ with variable ordering [a,b,c] of the relationship between the iterator arrays of the leapfrog joins for each variable, and the trie iterators for R, S, and T.

The topmost leapfrog join iterates values for a which are in both the projections $R(a,\_)$ and $T(a,\_)$. When this leapfrog join emits a binding for a, we can proceed to the next level join and seek bindings for b from $R_a(b)$, $S(b,\_)$. For each such b, we can proceed to the next level and seek a binding for c in $S_b(c)$, $T_a(c)$. When a leapfrog join exhausts its bindings, we can retreat to the previous level and seek another binding for the previous variable. Conceptually, we can regard triejoin as a backtracking search through a 'binding trie.'

At initialization, the triejoin is provided with a trie iterator for each relation in the join. Since relations are presented by trie iterators, nonmaterialized views may be used in place of relations. (Example: we could construct a triejoin for $A(x)$, $D(x,y)$ where $D(x,y)$ presents a nonmaterialized view of $B(x,y) \vee C(x,y)$.)

It profits us to define triejoin as an implementation of the trie iterator interface. That is, triejoin presents a nonmaterialized view of the join result, so result tuples can be retrieved by exploring the trie using the open(), next(), etc. methods. We can then define a variant of triejoin for disjunctive joins, which also implements the trie iterator interface; this lets us build trie iterators for queries containing arbitrary nestings of disjunction and conjunction. Adding trie iterators for complements and projections completes the toolbox needed to tackle $\exists_1$ queries.

2.5   Triejoin implementation

The triejoin initialization constructs an array of leapfrog join instances, one for each variable. The leapfrog join for a variable x is given an array of pointers to trie-iterators, one for each atom in whose argument list the variable appears. The leapfrog joins use the linear-iterator portion of the trie iterator interfaces; the up-down trie navigation methods are used only by the triejoin algorithm. The triejoin uses a variable *depth* to track the current variable for which a binding is being sought; initially $depth = -1$ to indicate the triejoin is positioned at the root of the binding trie (i.e., before the first variable.) Depths $0, 1, \ldots$ refer to the first, second, etc. variables of the variable-ordering.

The linear iterator portions of the trie-iterator interface (namely key(), atEnd(), next(), and seek()) are delegated to the leapfrog join for the current variable. (At depth -1, i.e., the root, only the operation open() is permitted, which moves to the first variable.) It remains to

define the open() and up() methods, which are trivial:

Function triejoin-open

$depth := depth + 1$;   // Advance to next variable
for *each iter in leapfrog join at current depth* **do**
   | iter.open() ;
end
call leapfrog-init() for leapfrog join at current depth

Function triejoin-up

for *each iter in leapfrog join at current depth* **do**
   | iter.up() ;
end
$depth := depth$ - 1 ;   // Backtrack to previous variable

This completes the trie iterator interface. To enumerate the satisfying assignments of the join, we employ an adaptor which turns a trie-iterator into a linear iterator of tuples, a simple exercise we omit here.

3   Complexity of Leapfrog Triejoin

We consider here the complexity of triejoin for fully conjunctive joins of materialized relations.

3.1   The proof strategy

To introduce the proof strategy, consider the example join:

$$Q(a, b, c) \equiv R(a, b), S(b, c), T(a, c)$$

with variable-ordering $[a, b, c]$. Suppose that $|R| \leq n$, $|S| \leq n$, and $|T| \leq n$. The fractional cover bound yields $|Q| \leq n^{3/2}$, a worst case realized by setting $R = S = T = [n^{1/2}] \times [n^{1/2}]$.

We wish to show that the triejoin runs in $O(n^{3/2} \log n)$ time for this example. Recall that a leapfrog join of two unary relations U, V requires at most $\inf\{|U|, |V|\}$ iterator operations. It is readily seen that the cost at the first two trie levels $[a, b]$ cannot exceed $|R|$ linear iterator operations: at the first trie level the leapfrog join is limited by $|R(a, \_)| \leq |R|$, and at the second trie level the number of iterator operations is controlled by:

$$\sum_{a \in R_a(\_), T(a, \_)} \inf\{|R_a(b)|, |S(b, \_)|\} \leq \sum_{a \in R_a(\_), T(a, \_)} |R_a(b)|$$

$$\leq |R|$$

Therefore the number of linear iterator operations at the first two trie levels is $O(n)$. At the

third trie level, the number of linear iterator operations is controlled by:

$$\sum_{(a,b)\in R(a,b),S(b,\_),T(a,\_)} \inf\{|S_b(c)|, |T_a(c)|\} \tag{1}$$

We now wish to prove that the quantity (1) is $\leq n^{3/2}$. We do this by renumbering the c values of the relations such that the join produces a number of results equal to (1).

For example, suppose we had these trie presentations of R, S, T:
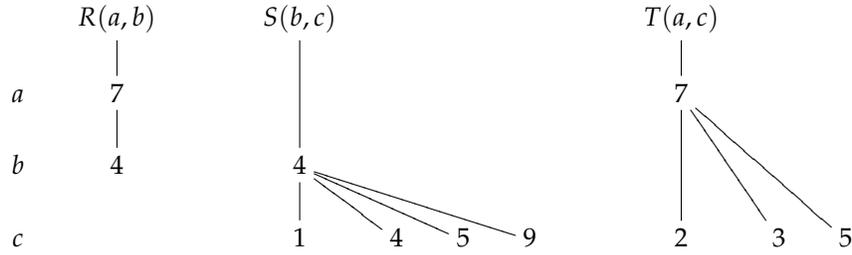
This would produce only the result tuple $(7, 4, 5)$. To get a result size equal to (1) we can renumber the c values to produce one result for every leaf of T (the smaller relation):
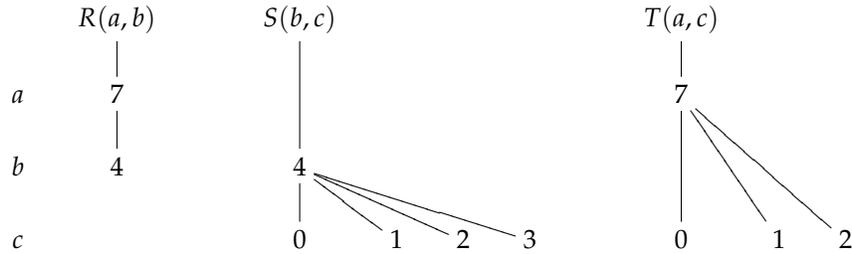
This results in exactly three results $(7, 4, 0)$, $(7, 4, 1)$, and $(7, 4, 2)$, equalling (1).

In general, the renumbering produces modified relations S', T' which each have cardinality $\leq n$. Since $n^{3/2}$ is an upper bound on the result size, it follows that (1) is at most $n^{3/2}$.

The renumbering is accomplished as follows:

- Construct $S'(b, c)$ by renumbering the c values of each S_b-subtree to be $0, 1, \ldots$, i.e.:

$$S'(b, \_) = S(b, \_) \qquad \text{Keep b values the same}$$
$$S'_b = \{0, 1, \ldots, |S_b| - 1\} \qquad \text{For each b, renumber the c values}$$

- Similarly, renumber the c values of each T_a subtree:

$$T'(a, \_) = T(a, \_) \qquad \text{Keep a values the same}$$
$$T'_a = \{0, 1, \ldots, |T_a| - 1\} \qquad \text{Renumber the c values}$$

When we compute the leapfrog join of $S_b' = \{0, 1, \ldots, |S_b| - 1\}$ with $T_a' = \{0, 1, \ldots, |T_a| - 1\}$, we get exactly $\inf\{|S_b|, |T_a|\}$ results. This holds for every join at the third trie level; therefore the query result size is exactly the quantity (1). Since the fractional cover bound gives an upper bound of $n^{3/2}$ on the query result size, we have:

$$\sum_{(a,b) \in R(a,b), S(b,\_), T(a,\_)} \inf\{|S_b(c)|, |T_a(c)|\} \le n^{3/2}$$

Hence the running time of leapfrog triejoin for the example is $O(n^{3/2} \log n)$.[7]

3.2   The renumbering transform

We generalize the renumbering transformation introduced in the previous section. For a relation $R(x, y, z)$, a renumbering at variable v is obtained by traversing the trie representation of R, and:

- If the variable v appears in the argument list at depth d, then renumber the children of nodes at depth $d - 1$ to be $0, 1, \ldots$; otherwise, do nothing.

- Replace all values for variables appearing after v in the key-ordering with 0.

- Eliminating any duplicate tuples.

The resulting relation R' is called a renumbering of R. Figure 4 illustrates renumberings of a relation $R(x, y, z)$ at various depths.

3.3   Triejoin costs

Let $R^1, \ldots, R^m$ be the relations in the join, and $V = [v_1, \ldots, v_k]$ be the chosen variable ordering. Each atom (relation) in the join takes as arguments some subset of the variables V, in order. For a relation $R(v_1, v_2, v_3, v_4)$, we use this notation for currying:

$$R_{v_1, v_2}(v_3, v_4) = \{(v_3, v_4) \; : \; (v_1, v_2, v_3, v_4) \in R\}$$

We write $R_{<i}(v_i, \ldots)$ for the curried version of all variables strictly before v_i in the ordering; e.g. $R_{<4}(v_4) = R_{v_1, v_2, v_3}(v_4)$.

Write $Q_i(v_1, v_2, \ldots, v_i)$ for the join 'up to' variable i; this is obtained by replacing variables $v_{i+1}, \ldots, v_k$ with the projection symbol $\_$ in the query, and omitting any atoms which contain only projection symbols.[8] For example, with $Q = R(a, b), S(b, c), T(a, c)$, and key order

---

[7] It turns out for the specific worst case where $R = S = T = [n^{1/2}] \times [n^{1/2}]$ the running time is $O(n^{3/2})$, i.e., we can drop the log factor because of the amortized complexity of the iterators. It is plausible that the log factor could be dropped for arbitrary R, S, T. However, there are some queries (i.e. not the R-S-T example) where the log factor appears unavoidable.

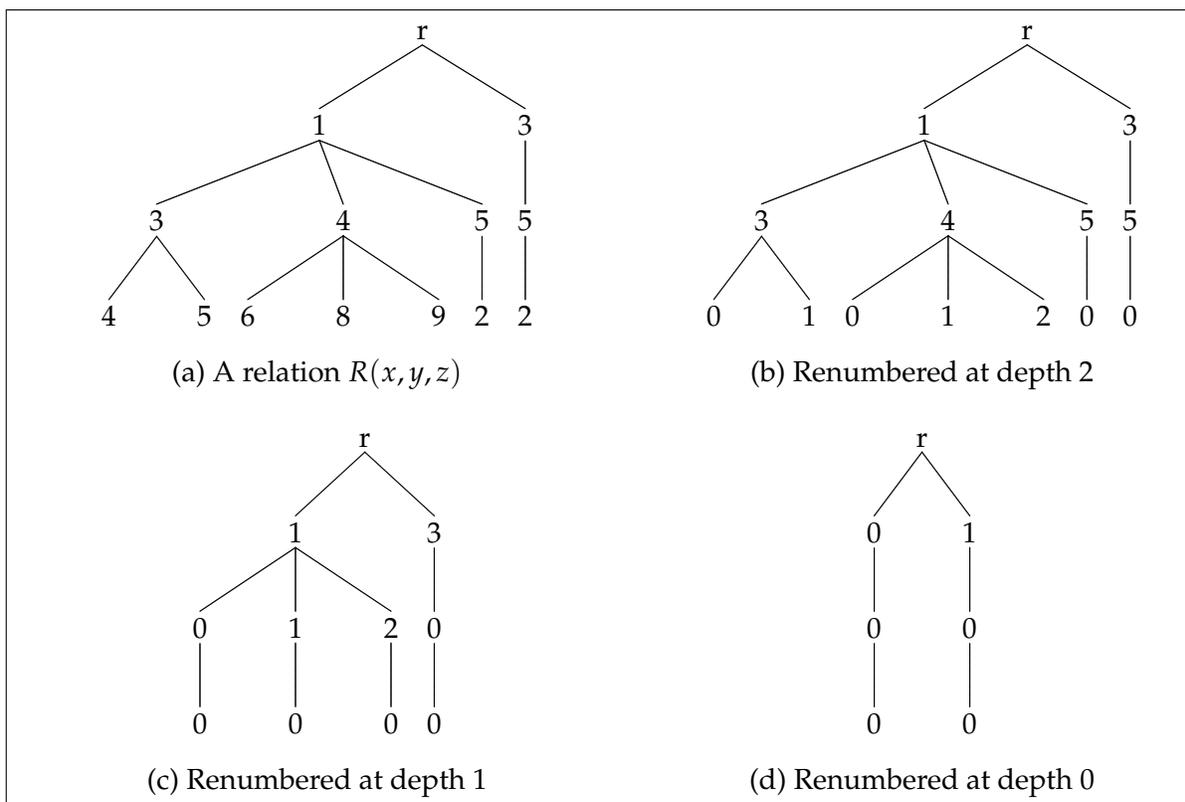[8] Note that Q_i is generally a strict superset of the projection of the query result $Q(v_1, v_2, \ldots, v_i, \_, \_, \ldots, \_)$.

(a) A relation $R(x, y, z)$

(b) Renumbered at depth 2

(c) Renumbered at depth 1

(d) Renumbered at depth 0

Fig. 4: Example of the renumbering transform applied to a relation $R(x, y, z)$.

$[a, b, c]$, we would have:

$$Q_1 = R(a, \_), T(a, \_)$$
$$Q_2 = R(a, b), S(b, \_), T(a, \_)$$
$$Q_3 = R(a, b), S(b, c), T(a, c)$$

Let $R^1_{v_{<i}}(v_i), \dots, R^l_{v_{<i}}(v_i)$ be the relations in the leapfrog join at level i. Let C_i be the sum-inf of the leapfrog triejoin at tree depth i:

$$C_i = \sum_{(v_1, \dots, v_{i-1}) \in Q_{i-1}} \inf\{|R^1_{v_{<i}}(v_i, \_, \dots, \_)|, \dots, |R^l_{v_{<i}}(v_i, \_, \dots, \_)|\}$$

To compute the join result, one uses the trie iterator presented by leapfrog triejoin to completely traverse the trie. The time cost of this is immediate from the leapfrog complexity bound, and the $O(\log N)$ performance requirement for open() and up():

Proposition 3.1. *The running time of Leapfrog Triejoin is* $O((\sum_{i \in [k]} C_i) \log N_{max})$.

3.4 Families of problem instances

We write $\text{Str}[\sigma]$ for finite structures with signature (vocabulary) σ. A *family of problem instances* is a family $(\mathbf{K}_n)_{n \in \mathbb{N}}$ indexed by a parameter $n \in \mathbb{N}$, where each $\mathbf{K}_n \subseteq \text{Str}[\sigma]$ is a class of finite relational structures, and $(i \leq j) \implies (\mathbf{K}_i \subseteq \mathbf{K}_j)$. (Example: *graphs with at most n edges* is a family of problem instances.)

More generally, we can choose a tuple of parameters $\bar{n} \in \mathbb{N}^k$, with the usual partial ordering on tuples, so that $n_1 \leq n'_1, \dots, n_k \leq n'_k$ implies $\mathbf{K}_{[n_1, \dots, n_k]} \subseteq \mathbf{K}_{[n'_1, \dots, n'_k]}$. (Example: let σ contain the binary relation symbols R, S, T, and define $\mathbf{K}_{r,s,t}$ to be structures with $|R| \leq r$, $|S| \leq s$, and $|T| \leq t$.)

The signature σ is stipulated to include a query relation symbol Q. The query is defined by some formula $\psi(\bar{x})$, with every structure $\mathcal{A} \in \mathbf{K}_n$ satisfying $\mathcal{A} \models (Q(\bar{x}) \leftrightarrow \psi(\bar{x}))$. (For now, we restrict the query relation to be definable by a fully conjunctive join.) For simplicity, we take $\bar{x}$ to be the variable ordering for the triejoin.

Given structures $\mathcal{A}, \mathcal{A}'$, we say $\mathcal{A}'$ is a renumbering of $\mathcal{A}$ if it is obtained by choosing some relation of $\mathcal{A}$ and renumbering it at some depth, as per Section 3.2.

A family of problem instances is *closed under renumbering* when for every $\mathcal{A} \in \mathbf{K}_n$, if $\mathcal{A}'$ is a renumbering of $\mathcal{A}$, then $\mathcal{A}' \in \mathbf{K}_n$ also.

Theorem 3.2. *Let*

1. $(\mathbf{K}_n)_{n \in \mathbb{N}}$ *be a family of problem instances closed under renumbering,*

2. $q(n) = \sup_{\mathcal{A} \in \mathbf{K}_n} |Q^{\mathcal{A}}|$ *be the largest query result size for any structure in* $\mathbf{K}_n$, *and*

3. $M(n)$ *be the cardinality of the largest relation (excluding Q) in any structure of* $\mathbf{K}_n$.

Then, Leapfrog Triejoin computes Q in $O(q(n) \log M(n))$ time over $(\mathbf{K}_n)_{n \in \mathbb{N}}$.

Proof. (By contradiction). Suppose the running time is $\omega(q(n) \log M(n))$. From Prop. 3.1, the running time of leapfrog triejoin is $O((C_0 + \cdots + C_{k-1}) \log M(n))$, where C_i is the sum-inf for the leapfrog join of variable v_i. For this to be $\omega(q(n) \log M(n))$, some variable v_i must have $C_i \in \omega(q(n))$ for infinitely many instances $\mathcal{A}$. For each such $\mathcal{A}$, renumber all relations for variable v_i. Revise $Q(\bar{v})$ appropriately. This results in structures $\mathcal{A}'$ with $|Q^{\mathcal{A}'}| = C_i$. Since the family is closed under renumbering, $\mathcal{A}' \in \mathbf{K}_n$; but $|Q^{\mathcal{A}'}| \in \omega(q(n))$, contradicting the definition of $q(n)$. $\qquad\square$

The worst-case optimality in the sense of NPRR [2] is a trivial corollary:

Corollary 3.3. *The running time of Leapfrog Triejoin is bounded by the the fractional edge cover bound, up to a log factor.*

Example: for the R, S, T example we could define the family of instances to be $|R| \leq n, |S| \leq n, |T| \leq n$; since renumbering does not increase the sizes of the relations, the family is closed under renumbering. The fractional edge cover provides the bound $q(n)$ of Theorem 3.2.

References

[1] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *FOCS*, pages 739–748. IEEE Computer Society, 2008.

[2] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms: [extended abstract]. In *Proceedings of the 31st symposium on Principles of Database Systems*, PODS '12, pages 37–48, New York, NY, USA, 2012. ACM.