Worst-case Optimal Join in Clojure

Finn Völkel

26.10.2022

Joins

R(a,c) ⋈ T(a,b) ⋈ S(b,c)

```
SELECT
    *
FROM
    R, T, S
WHERE
    R.a = T.a AND T.b = S.b AND R.c = S.c;
```

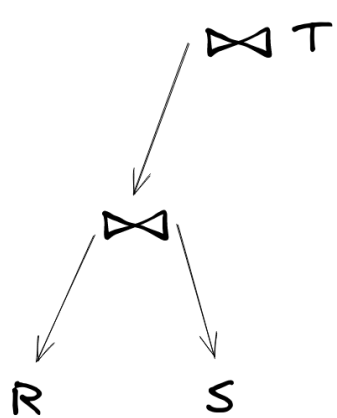
```
'{:find [?a ?b ?c]
  :where [[?a :r ?c]
          [?a :t ?b]
          [?b :s ?c]]}
```

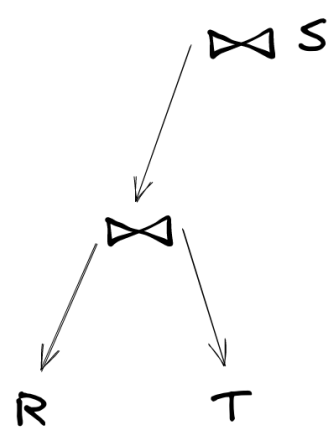
Joins

R(a,c) ⋈ T(a,b) ⋈ S(a,c)

```
SELECT
    *
FROM
    R, T, S
WHERE
    R.a = T.a AND T.b = S.b AND R.c = S.c;
```

```
'{:find [?a ?b ?c]
  :where [[?a :r ?c]
          [?a :t ?b]
          [?b :s ?c]]}
```





R ⋈ S ≈ 1000

R ⋈ S ⋈ T ≈ 100

R ⋈ T ≈ 10

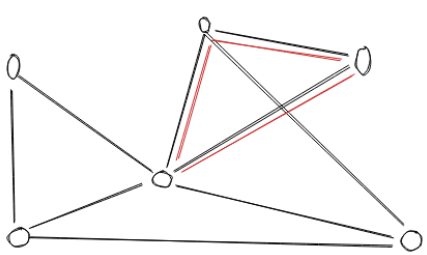
R ⋈ S ⋈ T ≈ 100

Triangle Query

G(a,c) ⋈ G(a,b) ⋈ G(b,c)

```
SELECT
    g1.f AS a, g1.t AS b, g2.t AS c
FROM
    g AS g1, g AS g2, g AS g3
WHERE
    g1.t = g2.f AND g2.t = g3.t AND g1.f = g3.f;
```

```
'{:find [?a ?b ?c]
  :where [[?a :g/to ?c]
          [?a :g/to ?b]
          [?b :g/to ?c]]}
```

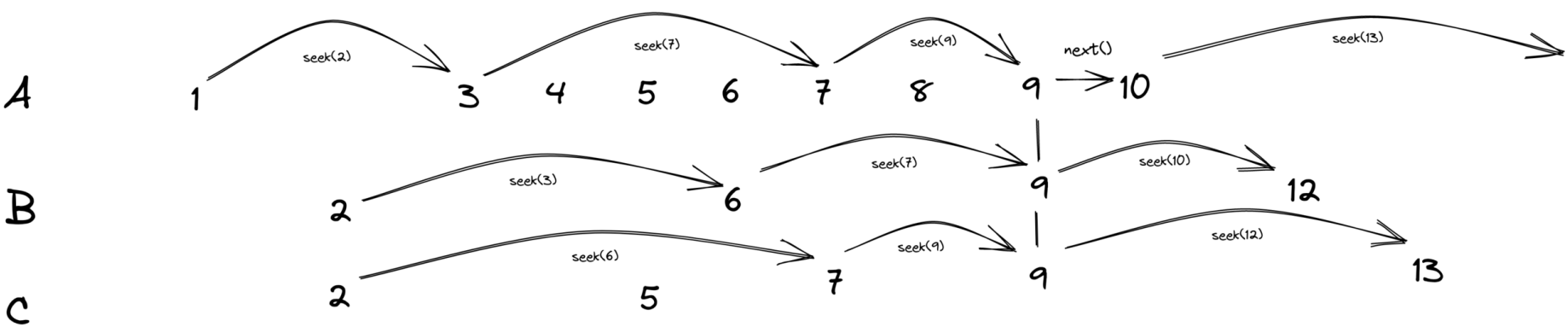


- potentially $O(N^2)$ rows in an intermediate join
- at most $O(N^{3/2})$ triangles in any graph

Worst-case optimal join

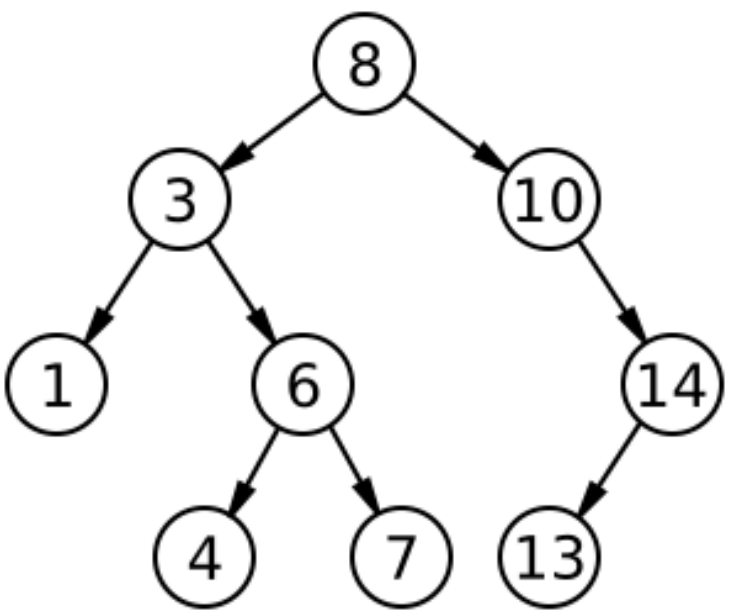
A(x) ⋈ B(x) ⋈ C(x)

```
'{:find [?x]
  :where [[?x :person/hometown "Berlin"]
          [?x :person/firstname "Bob"]
          [?x :person/eats "Spaghetti"]]}
```



Unary Iterator

```
(defprotocol LeapIterator
  (key [this])
  (next [this])
  (seek [this k])
  (at-end? [this]))
```



Unary Iterator

```
(defprotocol LeapIterator
  (key [this])
  (next [this])
  (seek [this k])
  (at-end? [this]))
```

- visiting m of N values should have amortized cose $O(1 + \log(N/m))$
- any `sorted-map` implementation works
- `ISeq` needs to be extended with

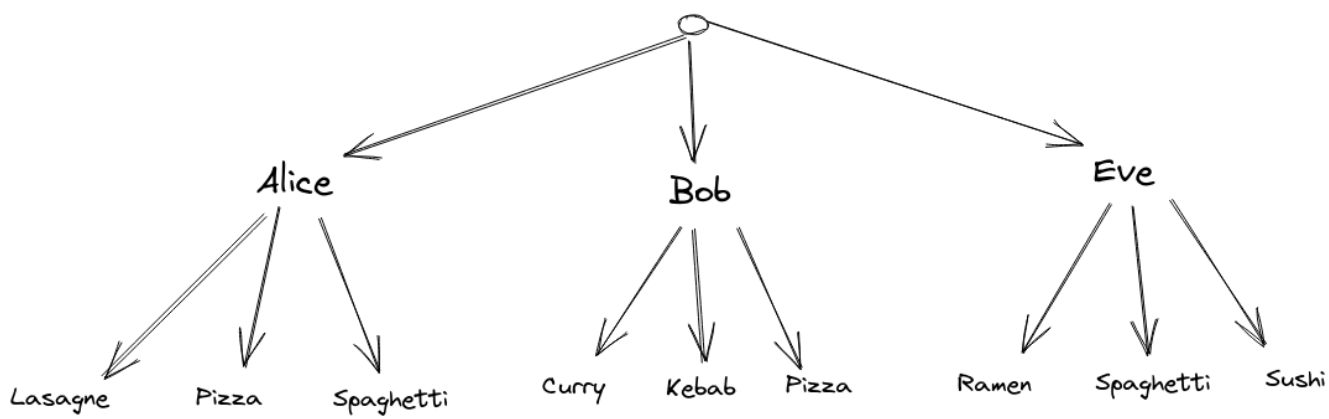
```
(defprotocol Seek
  (seek [this k]))
```

Worst-case optimal join

A(x) ⋈ B(x) ⋈ C(x,y)

```
'{:find [?x ?y]
  :where [[?x :person/hometown "Berlin"]
          [?x :person/firstname "Bob"]
          [?x :person/eats ?y]]}
```

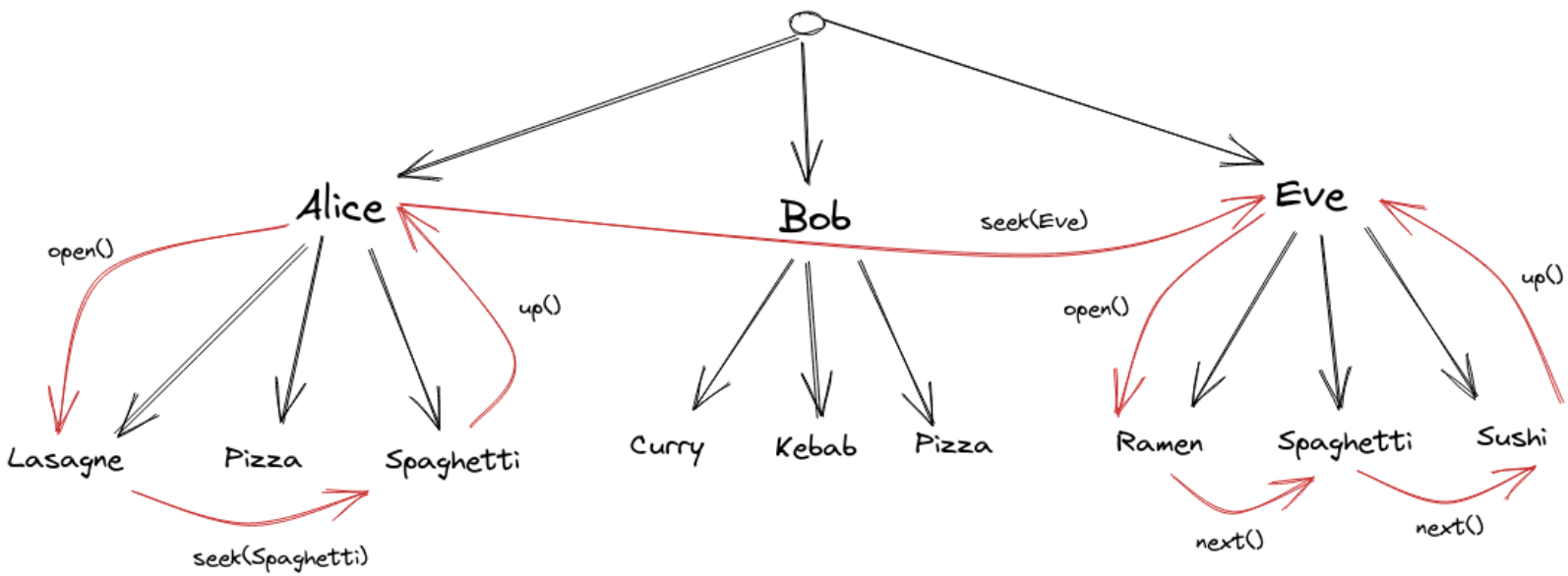
Alice	Lasagne
Alice	Pizza
Alice	Spaghetti
Bob	Curry
Bob	Kebab
Bob	Pizza
Eve	Ramen
Eve	Spaghetti
Eve	Sushi



Multi-arity iterator

```clojure
(defprotocol LeapIterator
  (key [this])
  (next [this])
  (seek [this k])
  (at-end? [this]))
```

```clojure
(defprotocol LeapLevels
  (open [this])
  (up [this])
  (level [this])
  (depth [this]))
```



A simple datalog engine

```
[:db/add "Bob" :person/eats "Spaghetti"]
```

```
(defrecord MemoryGraphIndexed [eav eva ave aev vea vae doc-store opts]
  graph/Graph
  (transact [this tx-data ts] (transact* this tx-data ts))
  ...

  graph/GraphIndex
  (get-iterator [this tuple] (get-iterator* this tuple))
  ...
  )
```

A simple datalog engine

```
[:db/add "Bob" :person/eats "Spaghetti"]
```

```
(defn index-triple-add [{opts :opts :as graph} [e a v :as triple]]
  (let [update-in (avl-util/create-update-in avl/sorted-map)
        [he ha hv] (mapv hash triple)]
    (-> graph
        (update-in [:eav he ha] (fnil conj (avl/sorted-set)) hv)
        (update-in [:eva he hv] (fnil conj (avl/sorted-set)) ha)
        (update-in [:ave ha hv] (fnil conj (avl/sorted-set)) he)
        (update-in [:aev ha he] (fnil conj (avl/sorted-set)) hv)
        (update-in [:vea hv he] (fnil conj (avl/sorted-set)) ha)
        (update-in [:vae hv ha] (fnil conj (avl/sorted-set)) he)
        (update :doc-store into [[he e] [ha a] [hv v]]))))
```

A simple datalog engine

Triple

```
[:db/add "Bob" :person/eats "Spaghetti"]
```

"hashed" triple

```
[:db/add 1 2 3]
```

eav index

```
{1 {2 #{3}}}
```

aev index

```
{2 {1 #{3}}}
```

doc-store

```
{1 "Bob" 2 :person/eats 3 "Spaghetti"}
```

A simple datalog engine

```
{:triple '[?person :person/eats ?food]
 :triple-order [:a :e :v]}
```

```
(defn simplify [binding] (map #(if (util/variable? %) '? :v) binding))

(defmulti get-index (fn [graph {:keys [triple] :as _tuple}] (simplify triple)))

(defmethod get-index '[? ? ?]
  [graph {[t1 t2 t3] :triple-order :as _tuple}]
  (get graph (keyword (str (name t1) (name t2) (name t3)))))

(defmethod get-index '[? :v ?]
  [graph {[t1 t2 t3] :triple-order [_ v2 _] :triple :as _tuple}]
  (get-in graph [(keyword (str (name t2) (name t1) (name t3))) v2]))
...
```

A simple datalog engine

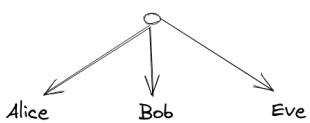
```
(defrecord LeapIteratorAVL [index stack depth max-depth]
  LeapIterator
  (key [this] (first-key index depth max-depth))

  (next [this] (->LeapIteratorAVL (clojure.core/next index) stack depth max-depth))

  (seek [this k]
    (when (seq index)
      (->LeapIteratorAVL (avl/seek index k) stack depth max-depth)))

  (at-end? [this] (empty? index))

  LeapLevels
  (open [this]
    (assert (< (inc depth) max-depth))
    (->LeapIteratorAVL (-> index first second seq) (conj stack index) (inc depth) max-depth))

  (up [this]
    (->LeapIteratorAVL (peek stack) (pop stack) (dec depth) max-depth))

  (level [this] depth)

  (depth [this] max-depth))
```

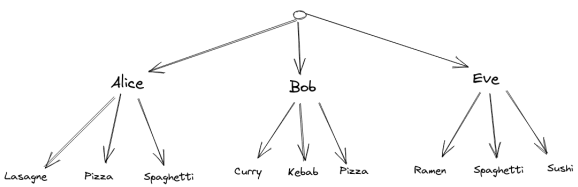
```
(defn ->leap-iterator-avl [index max-depth]
  {:pre [(avl-index? index)]}
  (->LeapIteratorAVL (seq index) [] 0 max-depth))

(->leap-iterator-avl (get-index graph tuple) nb-vars)
```

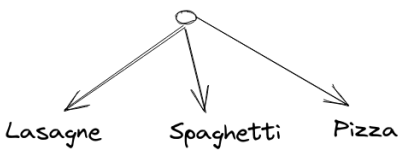
A simple datalog engine

```
'{:find [?person ?food]
  :where [[?person :person/hometown "Berlin"]
          [?person :person/eats ?food]
          [?food :food/healthy true]]}
```

- decide on a variable order (?person ?food)
- get iterators







(simplified) complexity proof

```
for each a in intersect(R, T):
  for each b in intersect(R[a], S):
    for each c in intersect(S[b], T[a]):
      emit((a, b, c))
```

$O(N)$

$O(N)$

$O(N^{3/2})$

- by the proof we emit at most $O(N^{3/2})$ results
- every intersect is roughly a unary iterator

Final remarks

- no silver bullet
- variable join order matters
- is it really worth it?
- disk access the heavy part

References

- Skew strikes back https://arxiv.org/abs/1310.3314
- Leapfrog Triejoin https://arxiv.org/abs/1210.0481
- Radix Triejoin https://arxiv.org/abs/1912.12747
- WCOJ review https://arxiv.org/abs/1803.09930
- https://justinjaffray.com/a-gentle-ish-introduction-to-worst-case-optimal-joins/
- http://www.frankmcsherry.org/dataflow/relational/join/2015/04/11/genericjoin.html
- https://github.com/FiV0/hooray