Relational algebra with discriminative joins and lazy products

Fritz Henglein

Department of Computer Science
University of Copenhagen
Email: henglein@diku.dk

IFIP TC 2 Working Group 2.8 meeting, Frauenchiemsee, 2009-06-08

The problem

A query using list comprehensions:

```
[(dep, acct) | dep <- depositors,
               acct <- accounts,
               depNum dep == acctNum account]
```

Using relational algebra operators:

```
select (\(dep, acct) ->
             depNum dep == acctNum account))
        (prod depositors accounts)
```

+ Compositional, simple (generate and test)
- $\Theta(n^2)$ time and space complexity (not scalable)

Solution 1: Optimize by rewriting

Rewrite and use a sort-merge join (Wadler, Trinder 1989) or hash join; e.g.

```
jmerge (sort s1) (sort s2)
```

- + $O(n \log n + o)$ time complexity
- - Programmer needs to rewrite statically
- - Join algorithm explicit and fixed
- - Requires ordering relation for sorting

Solution 2: Use join

- Introduce `(equi) join` operator and make programmer use it.
- Use hash or sort-merge join algorithm in implementation of join

\+ $O(n \log n + o)$ time complexity

\+ Join algorithm encapsulated, can be changed (even dynamically)

\- Requires using *join* and clever static optimization, e.g. combining two consecutive joins.

Solution 3: Write it naively

- Write query using `select`, `project`, `prod`, no need to use explicit `join`
- Use lazy (symbolic) products to represent Cartesian products
- Employ generic discrimination for asymptotically worst-case optimal joining

+ $O(n + o)$ time complexity
+ Naive query, with symbolic representations of formulas
+ Dynamic optimization, subsumes classical static algebraic optimizations
+ Works generically for equivalences, not just equalities
+ Works for reference types with observable equality only, no need for observable sort order or hash function

Sets, naively

```
data Set a = Set [a]
```

- A *set* is represented by *any* list that contains the right elements
- Same set represented by:
  - $[4, 8, 9, 1]$
  - $[1, 9, 8, 4, 4, 9]$
- Allow any element type, not just tuples of primitive type as in Relational Algebra

Projections, naively

```
data Proj a b = Proj (a -> b)
```

- A *projection* is any function.
- Allow any function, not just proper projections of records to fields.

Predicates, naively

```
data Pred a = Pred (a -> Bool)
```

- A *predicate* is any function to `Bool`.
- Allow any predicate, not just relational operators $=, \neq, \leq, \geq$ applied to fields of records.

Relational operators

```
select (Pred c) (Set xs) =
    Set (filter c xs)

project (Proj f) (Set xs) =
    Set (map f xs)

prod (Set xs) (Set ys) =
    Set [(x, y) | x <- xs, y <- ys]
```

Other operators: `union`, `intersect` similarly

Definable operators

Join operator:

```
join c s1 s2 =
     select c (prod s1 s2)
```

SQL-style SELECT FROM WHERE:

```
selectFromWhere p s c =
     project p (select c s)
```

Problem:

- Intermediate data may require asymptotically more storage space than input and output:
  - `prod` produces large output
  - `select` shrinks it again

Partitioning discriminator

Definition

```
D ::  forall v.  [(k, v)] -> [[v]]
```

is a *(partitioning) discriminator for equivalence e on* k if

- D partitions the value components of key-value pairs into the *e*-equivalence classes of their keys.
- D is parametric wrt. *e*: Replacing a key in the input with any *e*-equivalent key yields the same result.

Example:

- $(x, y) \in \mathit{evenOdd}$ iff both x, y even or both odd.
- Possible result:
  $\mathtt{D}[(5, 100), (4, 200), (9, 300)] = [[100, 300], [200]]$
- By parametricity then also:
  $\mathtt{D}[(\mathbf{3}, 100), (\mathbf{8}, 200), (\mathbf{1}, 300)] = [[100, 300], [200]]$

Discrimination-based equijoin: Algorithm

- Values: Tag records of input sets to identify where they come from
- Keys: Apply specified projections to records
- Concatenate list of key/value pairs
- Discriminate
- Form formal products (formal product: list of records from first input and list of records from second input, all with equivalent keys)
- Multiply out: Each record in a formal product from first input paired with each record from the second input.

Discrimination-based equijoin: Code

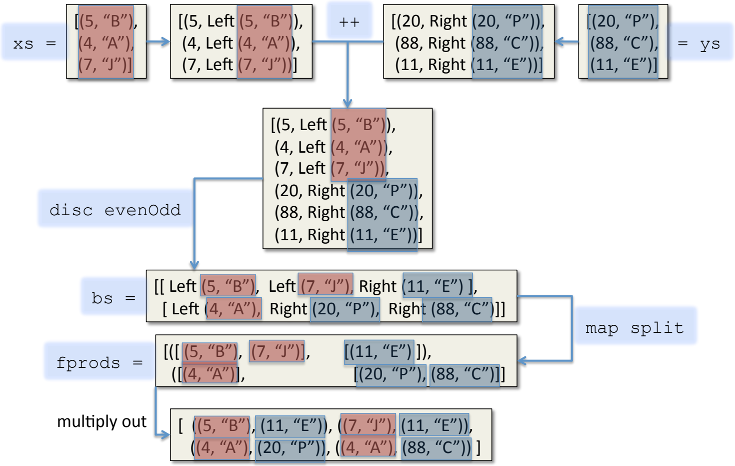
```
join (Set xs, Set ys) (Proj f1) e (Proj f2)=
  Set [(x, y) | (xs, ys) <- fprods,
                x <- xs, y <- ys ]
  where bs = disc e
             ([(f1 x, Left x) | x <- xs] ++
              [(f2 y, Right y) | y <- ys])
        fprods = map split bs
```

Auxiliary function

```
split :: [Either a b] -> ([a], [b])
```

splits a group of tagged values into their left, respective right values.

Discrimination-based equijoin: Example



Complexity

Assume:

- Worst-case time complexity of projection application: $O(1)$.
- s_1, s_2 are the respective lengths of the two inputs.
- o is the length of the output.

Observe:

- Discrimination-based join runs in worst-case time $O(s_1 + s_2 + o)$.
- Each step runs in time $O(s_1 + s_2)$ except for the last: multiplying out the results.

Idea: Be lazy! (Why multiply out if it's a lot of work?)

Lazy sets

Constructors for sets:

```
data Set :: * -> *  where
     Set        :: [a] -> Set a
     U          :: Set a -> Set a -> Set a
     X          :: Set a -> Set b -> Set (a, b)
```

- `Set xs`: Set represented by list `xs`
- ``s1 `U` s2``: Union of sets `s1, s2`
- ``s1 `X` s2``: Cartesian product of `s1, s2`

Lazy projections

```
data Proj :: * -> * -> * where
    Proj     :: (a -> b) -> Proj a b
    Par      :: Proj a b -> Proj c d ->
                Proj (a, c) (b, d)
```

- `Proj f`: Projection given by function *f*
- `Par p q`: Parallel composition of `p, q`

Why parallel compositions?
Permit symbolic execution at run-time.

Lazy predicates

```
data Pred :: * -> * where
   Pred :: (a -> Bool) -> Pred a
   TT   :: Pred a
   FF   :: Pred a
   PAnd :: Pred a -> Pred b -> Pred (a, b)
   In   :: (Proj a k, Proj b k) -> Equiv k
           -> Pred (a, b)
```

- `Pred f`: Predicate given by characteristic function
- `TT, FF`: Constant true, false
- `PAnd`: Parallel conjunction
- `In`: Join condition constructor.

Relational algebra operators

```
select   :: Pred a -> Set a -> Set a
project  :: Proj a b -> Set a -> Set b
prod     :: Set a -> Set b -> Set (a, b)
```

Example:

```
select ((depNum, acctNum) `In` eqNat16)
       (prod depositors accounts)
```

Like original naive definition, but:

- runs in time $O(n)$ (size of the input);
- *listing* result takes time $O(o)$ (size of the output).

Observe:
No separate join! Defined *naively*:

```
join c s1 s2 = select c (prod s1 s2)
```

Select: Nonjoins

```
select TT s          = s
select FF s          = Set []
select p (Set xs)    = Set (filter (sat p) xs)
select p (s1 `U` s2) =
    select p s1 `U` select p s2
select (Pred f) s@(s1 `X` s2) =
    Set (filter f (toList s))
select (p `PAnd` q) (s1 `X` s2) =
    select p s1 `X` select q s2
select ((p, q) `In` e) s@(s1 `X` s2) = ...
```

What do lazy (symbolic) representations buy?

- `TT, FF`: Argument set not traversed (good!)
- `p` with `` `U` ``: Lazy selection (good!)
- `Pred f` with `` `X` ``: Multiplying out (ouch!)
- ``p `PAnd` q`` with `` `X` ``: Lazy product (good!)

Select: Join

```
select ((f1, f2) `In` e) (s1 `X` s2) =
  foldr (\b s -> let (xs, ys) = split b
     in (Set xs `X` Set ys) `U` s) empty bs
  where bs = disc e
   ([(ext f1 r, Left r) | r <- toList s1] ++
    [(ext f2 r, Right r) | r <- toList s2])
```

- Recognize dynamically when `select` has an (equi)join condition applied to a lazy product.
- Invoke discrimination-based join algorithm
- Avoid multiplying out result in final step

Theorem

Join executes in time $O(s_1 + s_2)$ for $O(1)$-time projections where s_1, s_2 are the sizes (as lists) of `s1, s2`*, respectively.*

Observe: No *o* in that formula! Not $s_1 \times s_2$, but $s_1 + s_2$!

Project

```
project f (Set xs) = Set (map (ext f) xs)
project f (s1 `U` s2) =
   project f s1 `U` project f s2
project (Proj f) s@(s1 `X` s2) =
   Set (map f (toList s))
project (Par f1 f2) (s1 `X` s2) =
   project f1 s1 `X` project f2 s2
```

At run time:

- `Set`: Iterate (okay, not much else to do)
- `` `U` ``: Lazy union (good!)
- `Proj f` with `` `X` ``: Multiply out (ouch!)
- `Par f1 f2` with `` `X` ``: Lazy product (good!)

Prod

```
prod s1 s2 = s1 `X` s2
```

- Constant time!

Relation to query optimization

Implementation performs classical algebraic query optimizations, including

- filter promotion (performing selections early)
- join introduction (replacing product followed by selection by join)
- join composition (combining join conditions to avoid intermediate multiplying out)

Observe:

- Done at run-time
- No static preprocessing
- Data-dependent optimization possible.
- Deforestatation of intermediate materialized data structures not necessary due to lazy evaluation.

The problem
Relational algebra, naively
Relational algebra, cleverly

Applicability

- Assumption: RAM-model, all memory accesses cost the same
- Out-of-the-box applicability: In-memory bulk data.
- Just as you would not dream of applying sorting or hashing out-of-the-box to disk data, do not apply discrimination to disk data out of the box.
- As for sorting and hashing, does not rule out usability of generic discrimination as a *technique* to be combined with I/O efficiency techniques; e.g. block-by-block discrimination.

Related work

Database theory:

- Discrimination as an alternative/complement to sorting and hashing: Not previously explored.
- Lazy products, unions: Where? (Couldn't find in literature)
- Dynamic algebraic query optimization: Where? (Couldn't find in literature)

Functional Programming:

- Buneman et al., HaskellDB, LINQ, Links: Type-safe interfaces to SQL database systems
- Query optimization for in-memory non-SQL data: HaskellDB (?), LINQ (?)
- Kleisli: Distributed database system with functional query language based on Nested Relational Calculus
- Trinder, Wadler (1990), *Improving list comprehension database queries*: Classical query optimizations on list comprehensions

Contributions

- Partitioning discrimination: New generic technique for "bringing data together"
  - complements hashing and sorting techniques
  - makes only equivalence observable (no order, no hash function)
- Lazy products (and derived lazy data structures): New (?) data structure for compact representation of cross-products
- Generic relational algebra
  - User-definable equivalences, not just equalities
  - User-defined data types, including reference types (pointers)