OptiQL: LINQ on Delite

What is LINQ?

- Language Integrated Query (LINQ) is a set of language and framework features for writing structured type-safe queries over local object collections and remote data sources.
- Can query any collection implementing IEnumerable<T>
  - Equivalent to Iterable[T] in Scala

What is OptiQL

- The initial version is LINQ with some modifications implemented on Delite
- Get parallelization from using Delite
- Add Relational Algebra rules to further optimize OptiQL programs

Outline

- LINQ Architecture
  - Implications for Scala, OptiQL and Delite
- LINQ Queries
  - Implementation strategies on Delite
- Benchmarking LINQ/OptiQL

LINQ: Intro

- Basic units are *sequences* and *elements*

```
string[] names = { "Tom", "Dick", "Harry" };
```

- This is a local sequence represented by a local collection of objects in memory
- Query operators are methods that typically accept an *input sequence* and emit a transformed *output sequence*

Query Operator Example

```
string[] names = { "Tom", "Dick", "Harry" };
IEnumerable<string> filteredNames = System.Linq.Enumerable.Where
                                    (names, n => n.Length >= 4);
foreach (string n in filteredNames)
  Console.WriteLine (n);

Dick
Harry
```

- However, operators are implemented as extension methods (similar to infix_ methods)

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
```

- So can write queries as this:

```
IEnumerable<string> filteredNames = names.Where (n => n.Length >= 4);
```

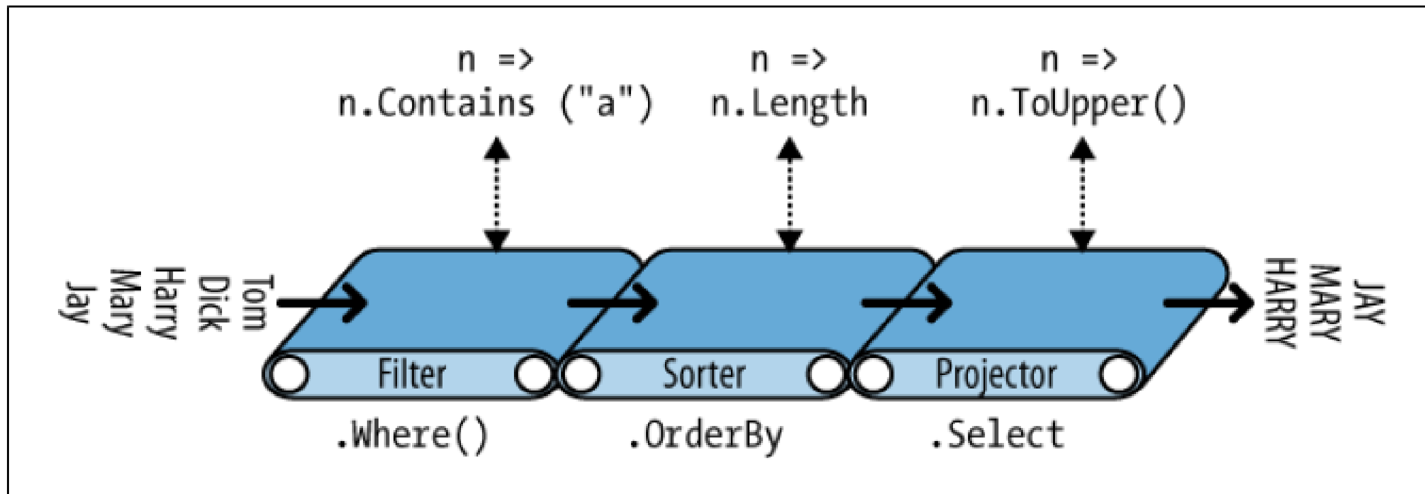
Fluent Syntax: Chaining Query Operators

- Similar to other DSLs we have seen, LINQ uses chaining to allow for more complex queries

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IEnumerable<string> query = names
  .Where   (n => n.Contains ("a"))
  .OrderBy (n => n.Length)
  .Select  (n => n.ToUpper());

foreach (string name in query) Console.WriteLine (name);
```

```
JAY
MARY
HARRY
```

Chaining Query Operators



Lambda Expressions

- Lambda expressions provide flexibility

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
{
  foreach (TSource element in source)
    if (predicate (element))
      yield return element;
}
```

- The operators encode common machinery, while lambda provide specialization
  - Lots of DSLs do this, hence why functional languages are ideal for DSL implementation

Query Expressions

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IEnumerable<string> query =
  from    n in names
  where   n.Contains ("a")     // Filter elements
  orderby n.Length             // Sort elements
  select  n.ToUpper();         // Translate each element (project)

foreach (string name in query) Console.WriteLine (name);
```

```
JAY
MARY
HARRY
```

- Need special compiler support for this
- Can be achieved via a Scala compiler plugin
  - Maybe we can do better in the future

Deferred Execution

- Most query operators execute *not* when constructed, but when enumerated

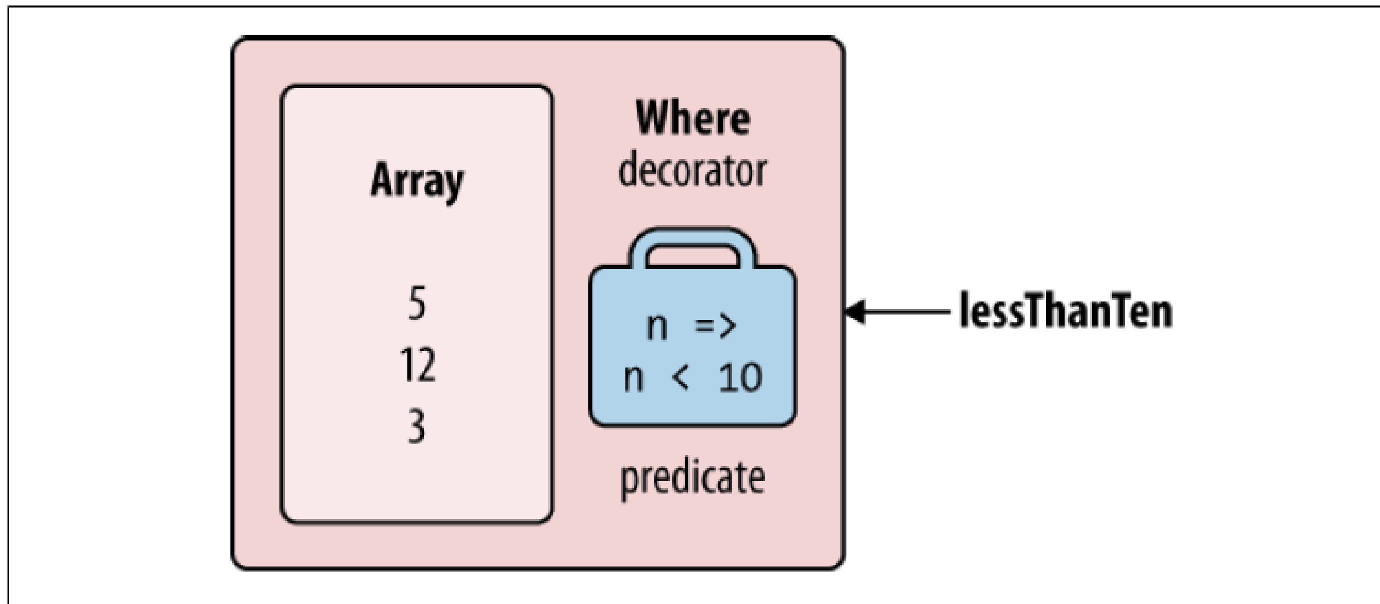
```
var numbers = new List<int>();
numbers.Add (1);

IEnumerable<int> query = numbers.Select (n => n * 10);    // Build query

numbers.Add (2);                    // Sneak in an extra element

foreach (int n in query)
  Console.Write (n + "|");          // 10|20|
```

- Some operators that have no way of deferring (like Count) execute immediately

Implementing Deferred Execution

- Return *decorator* sequence with no backing structure of its own

```
IEnumerable<int> lessThanTen = new int[] { 5, 12, 3 }.Where (n => n < 10);
```



Implementing Deferred Execution

- Very easy to do in C#

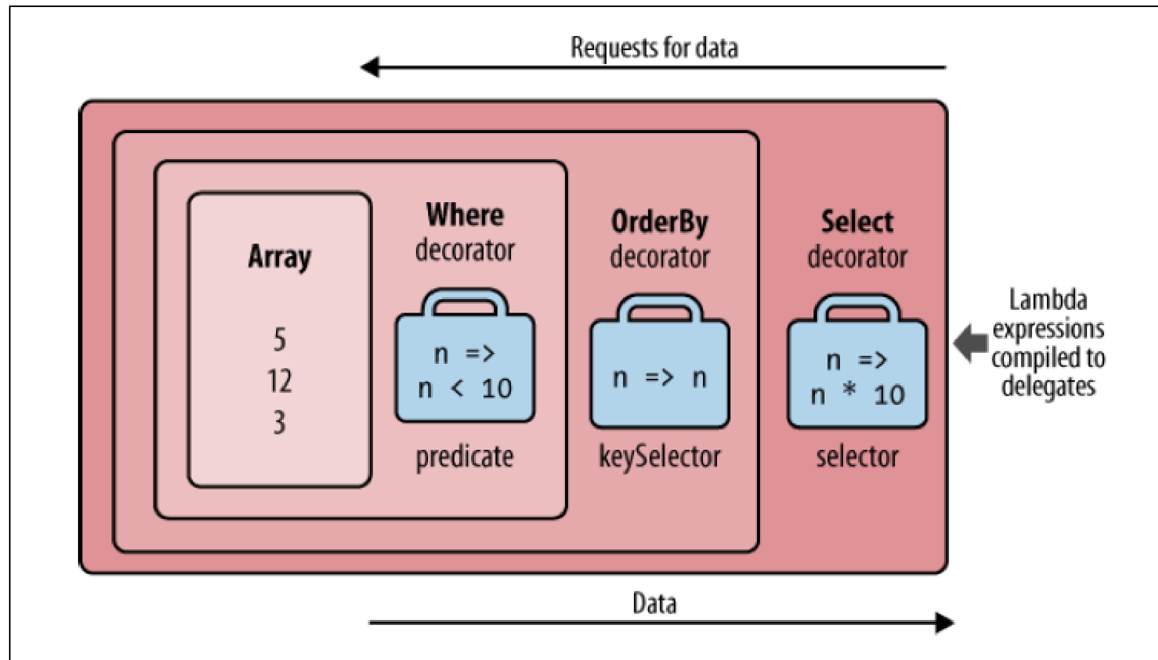
```
public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
{
  foreach (TSource element in source)
    yield return selector (element);
}
```

- The yield automatically constructs a decorator with source as the backing structure
  - yield in Scala is different and won't cause deferral

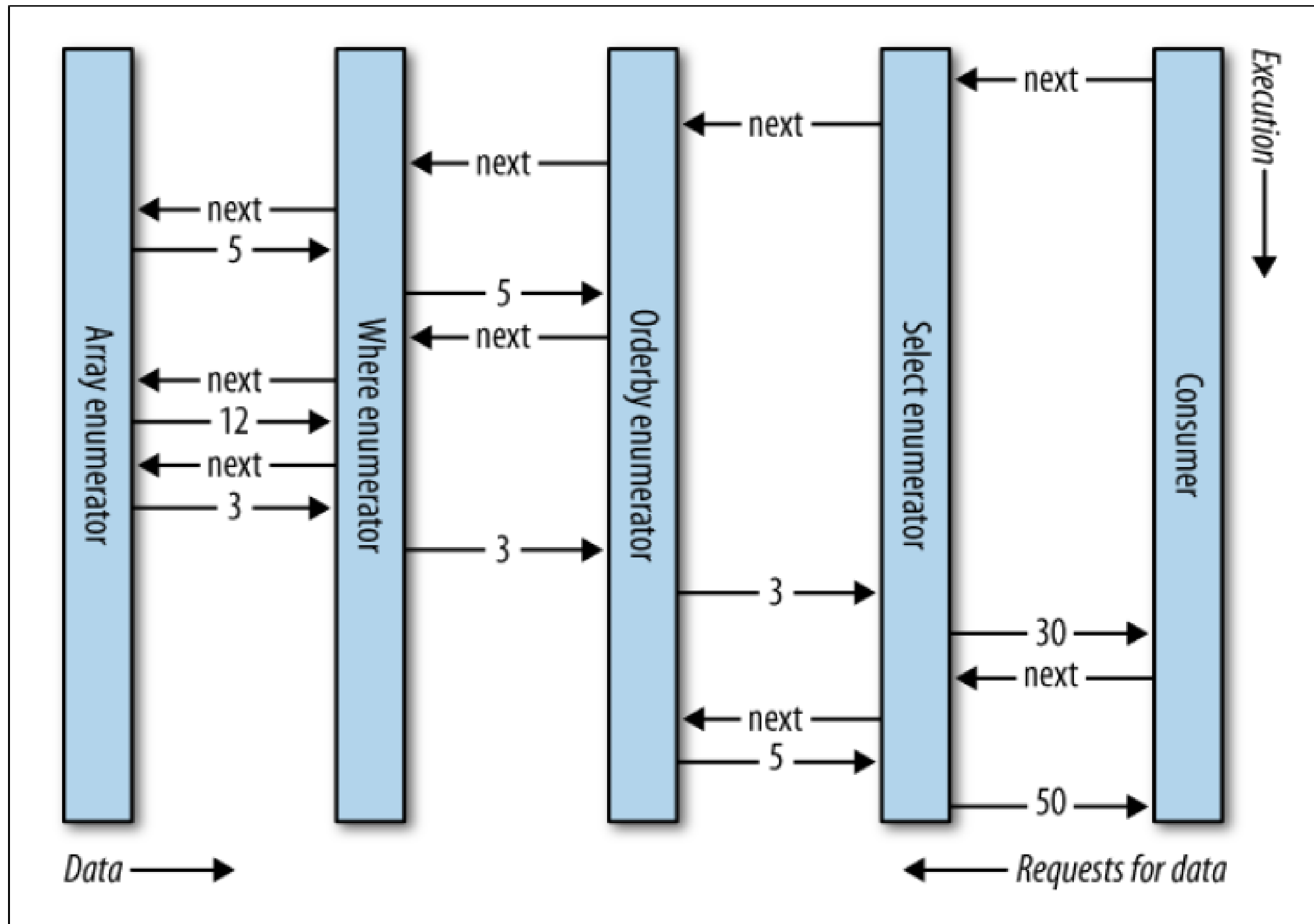
Chaining Decorators

- C# yields will cause automatic chaining

```
IEnumerable<int> query = new int[] { 5, 12, 3 }.Where   (n => n < 10)
                                               .OrderBy (n => n)
                                               .Select  (n => n * 10);
```



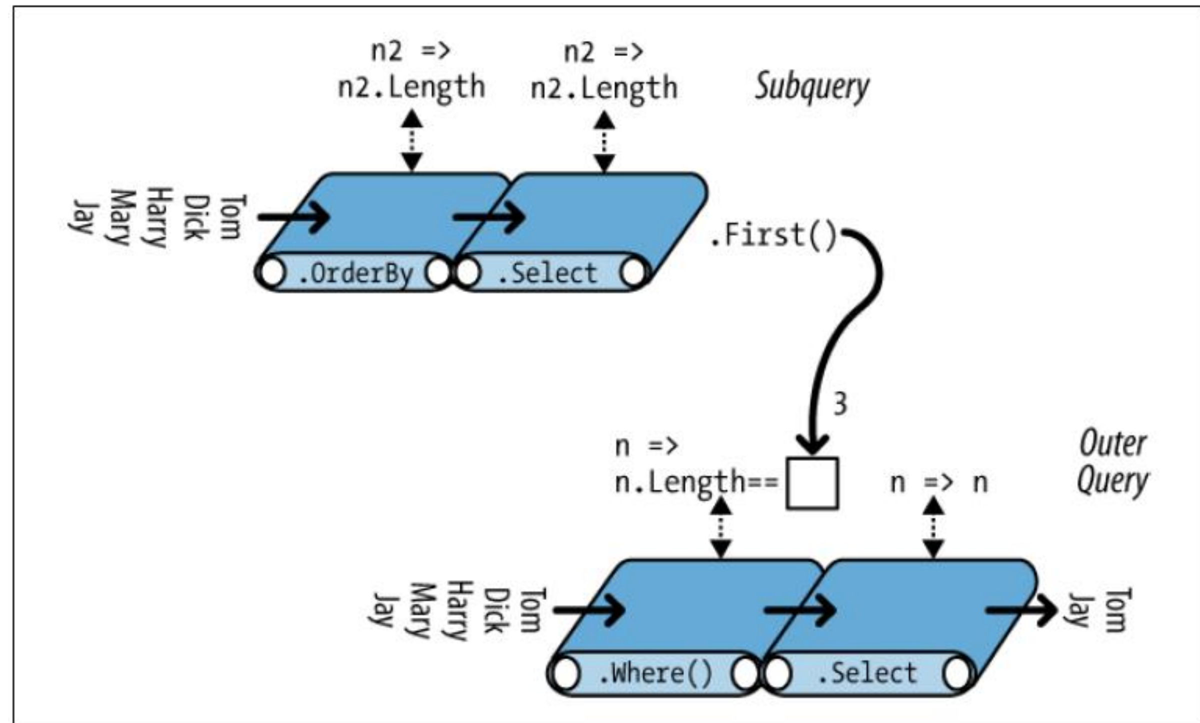
Chaining Decorators



Subqueries

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IEnumerable<string> outerQuery = names
  .Where (n => n.Length == names.OrderBy (n2 => n2.Length)
                                 .Select  (n2 => n2.Length).First());
```

```
Tom, Jay
```



Implications for OptiQL and Delite

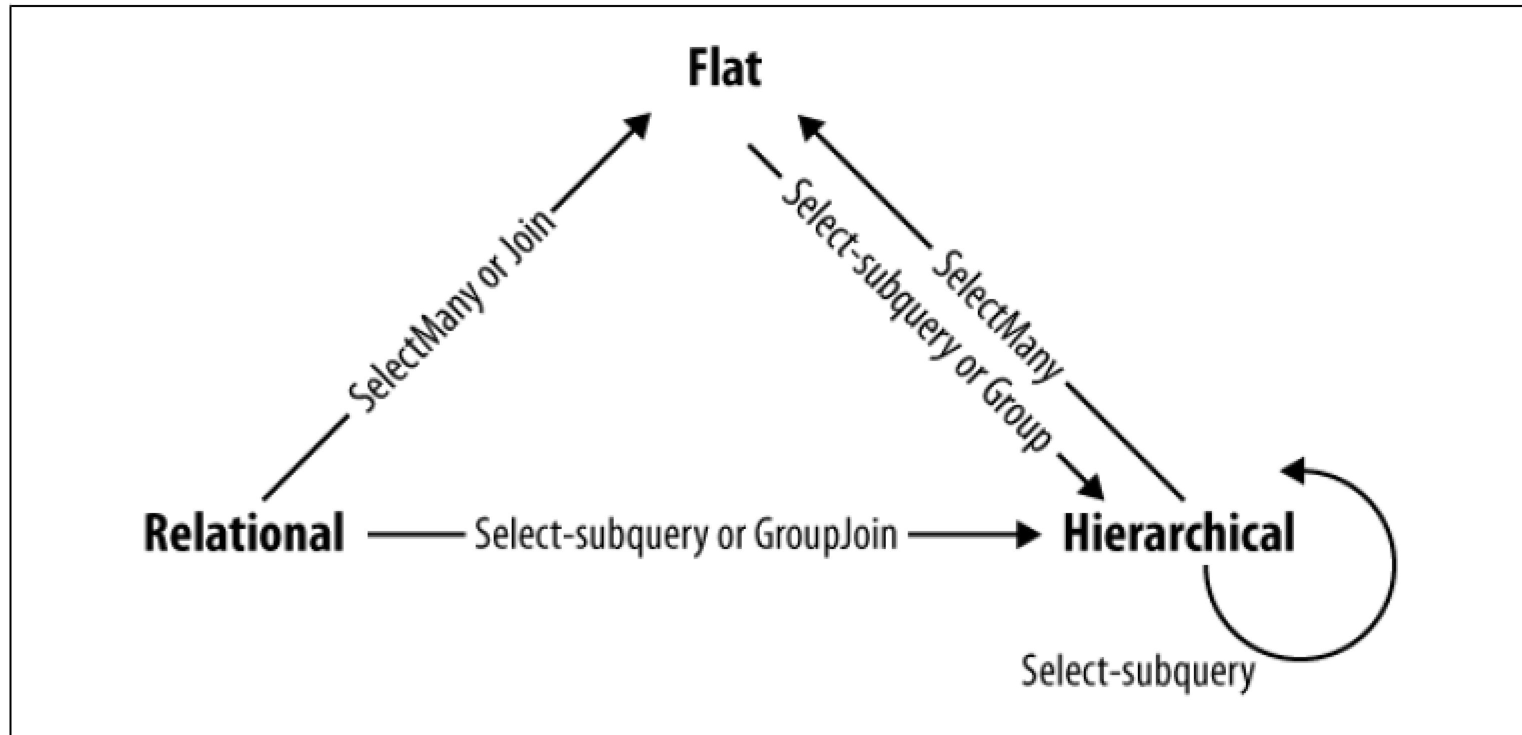
- Subquery performance can be improved dramatically
- Each Query Operator should be implemented as a Delite Op
- While Scala doesn't support deferral, we can achieve same or better result with fusing Query Ops

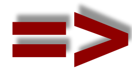
LINQ OPERATORS

LINQ Operator Overview

- Standard LINQ query operators fall into three categories:
  - Sequence in, sequence out
  - Sequence in, single element or scala out
  - Nothing in, sequence out

Sequence => Sequence



Sequence  Sequence

- Filtering
  - Where, Take, TakeWhile, Skip, SkipWhile, Distinct
- Projecting
  - Select, SelectMany
- Joining
  - Join, GroupJoin
- Ordering
  - OrderBy, ThenBy, Reverse
- Grouping
  - GroupBy
- Set operators
  - Concat, Union, Intersect, Except
- Zip operator

Sequence => Element or Scalar

- Element operators
  - First, Last, Single, ElementAt, DefaultIfEmpty,…
- Aggregation methods
  - Aggregate, Average, Count, Sum, Max,Min
- Quantifiers
  - All, Any, Contains, SequenceEqual

Void => Sequence

- Manufactures a simple sequence
  - Empty, Range, Repeat

Operator Example: Where

- returns elements of the sequence that satisfy the predicate

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<string> query = names.Where (name => name.EndsWith ("y"));

// Result: { "Harry", "Mary", "Jay" }
```

- Implemented as follows:

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func <TSource, bool> predicate)
{
  foreach (TSource element in source)
    if (predicate (element))
      yield return element;
}
```

Where: Baseline Scala (no deferral)

```
class Queryable[TSource](source: Iterable[TSource]) {
  import OptiQL._


  def Where(predicate: TSource => Boolean) = {
    if(predicate == null) throw new IllegalArgumentException("Predicate is Null")
    source.filter(predicate)
  }

}
```

Where: Delite Op Version

```
trait QueryableOpsExp extends QueryableOps with EffectExp {
  this: QueryableOps with OptiQLExp =>


  case class QueryableWhere[TSource:Manifest](s: Exp[DataTable[TSource]],
            predicate: Exp[TSource] => Exp[Boolean]) extends DeliteOpLoop[DataTable[TSource]] {
    val size = s.size
    val v = fresh[Int]
    val body : Def[DataTable[TSource]] = new DeliteCollectElem[TSource, DataTable[TSource]](
      alloc = reifyEffects(DataTable[TSource]()),
      func = reifyEffects(s(v)),
      cond = reifyEffects(predicate(s(v)))::Nil
    )
  }


  def queryable_where[TSource:Manifest](s: Exp[DataTable[TSource]],
       predicate: Exp[TSource] => Exp[Boolean]) = QueryableWhere(s,predicate)
}
```

Operator Example: Select

- This is basically a map

```
var query =
  from f in FontFamily.Families
  select new { f.Name, LineSpacing = f.GetLineSpacing (FontStyle.Bold) };
```

- Implementation is also pretty simple:

```
public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
{
  foreach (TSource element in source)
    yield return selector (element);
}
```

- In Scala, just use a map (but no deferral)

Select: Using Delite Ops

- Very simple to implement using Delite Ops

```
case class Select[A:Manifest,B:Manifest](in: Exp[DataTable[A]]
    , selector: Exp[A] => Exp[B]) extends DeliteOpMap[A,B,Vector] {
  val alloc = reifyEffects(DataTable[B]())
  val v = fresh[A]
  val func = reifyEffects(selector(v))
}
```

Operator Example: Join

- Join (inner), combines to sequences and creates a sequence that contains all the elements from each sequence that agree on join conditions merged in some fashion

```
val q4 = calls.Join(contacts)(_.Number, _.Phone, (call, contact) => new {
    val Name = contact.FirstName + " " + contact.LastName
    val Number = call.Number
    val Duration = call.Duration
})
```

Needs its own Delite Op

- Join needs its own Op, too different of a pattern to be implemented by an existing OP
- There are also multiple possible "physical" implementation of a Join, so Join is a good candidate for an Op

BENCHMARKING OPTIQL

TPCH Intro

- Transaction Processing Performance Council (TPC) is non-profit organization with the mission of disseminating objective, verifiable TPC performance data to the industry
- TPC-C: an on-line transaction processing benchmark
  - Not a good candidate, about making transactions
- TPC-H: An ad-hoc, decision support benchmark
  - Good candidate, about making queries of data
  - Challenge: Smallest dataset is very taxing

OptiQL: TPCH Example

```
select
        l_returnflag,
        l_linestatus,
        sum(l_quantity) as sum_qty,
        sum(l_extendedprice) as sum_base_price,
        sum(l_extendedprice*(1-l_discount)) as sum_disc_price,
        sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge,
        avg(l_quantity) as avg_qty,
        avg(l_extendedprice) as avg_price,
        avg(l_discount) as avg_disc,
        count(*) as count_order
from
        lineitem
where
        l_shipdate <= date '1998-12-01' - interval '[DELTA]' day (3)
group by
        l_returnflag,
        l_linestatus
order by
        l_returnflag,
        l_linestatus;
```

OptiQL: TPCH Example

```
val q1 = lineItems Where(_.shipDate <= Date("1998-12-01") + Interval(90).days) GroupBy(l => (l.returnFlag,l.line
  val returnFlag = g.key._1
  val lineStatus = g.key._2
  val sumQty = g.Sum(_.quantity)
  val sumBasePrice = g.Sum(_.extendedPrice)
  val sumDiscountedPrice = g.Sum(l => l.extendedPrice * (1-l.discount))
  val sumCharge = g.Sum(l=> l.extendedPrice * (1-l.discount) * (1+l.tax))
  val avgQty = g.Average(_.quantity)
  val avgPrice = g.Average(_.extendedPrice)
  val avgDiscount = g.Average(_.discount)
  val countOrder = g.Count
}) OrderBy(_.lineStatus) ThenBy(_.returnFlag)
```

returnFlag	lineStatus	sumQty	sumBasePrice	sumDiscountedPrice	sumCharge	avgQty	avgPrice	avgDiscount	countOrder
A	F	622.0	917881.9	866039.6	905126.0	28.272728	41721.902	0.055909093	22
R	F	409.0	553119.1	521259.12	538893.6	25.5625	34569.945	0.050624993	16
N	O	1564.0	2390615.0	2260066.5	2363488.0	26.508474	40518.9	0.05644065	59

OptiQL: TPCH Example

```
select
        l_orderkey,
        sum(l_extendedprice*(1-l_discount)) as revenue,
        o_orderdate,
        o_shippriority
from
        customer,
        orders,
        lineitem
where
        c_mktsegment = '[SEGMENT]'
        and c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and o_orderdate < date '[DATE]'
        and l_shipdate > date '[DATE]'
group by
        l_orderkey,
        o_orderdate,
        o_shippriority
order by
        revenue desc,
        o_orderdate;
```

OptiQL: TPCH Example

```
val q3 = customers.Where(_.marketSegment == "BUILDING").
  Join(orders)(_.key, _.customerKey, (customer, order)=> new {
  val orderKey = order.key
  val orderDate = order.date
  val orderShipPriority = order.shipPriority
}).Join(lineItems)(_.orderKey, _.orderKey, (co, li)=> new {
  val orderKey = co.orderKey
  val orderDate = co.orderDate
  val orderShipPriority = co.orderShipPriority
  val orderShipDate = li.shipDate
  val extendedPrice = li.extendedPrice
  val discount = li.discount
}).Where(col => col.orderDate < Date("1995-03-15") && col.orderShipDate < Date("1995-03-15")
).GroupBy(col => (col.orderKey,col.orderDate,col.orderShipPriority)) Select(g => new {
  val orderKey = g.key._1
  val revenue = g.Sum(e => e.extendedPrice * (1 - e.discount))
  val orderDate = g.key._2
  val shipPriority = g.key._2
})
```

OptiQL: Challenges

- Requires efficient *Filter* and *Join* (database) operations
  - Need to add DeliteOpScan and DeliteOpJoin

- Anonymous classes and user-defined structural types

```
val result = lineItems Where(_.shipDate <= Date("1998-12-01") +
   Interval(90).days.Select (g => new {
     val returnFlag = g.key._1
     val lineStatus = g.key._2
})
```

Must be able to preserve type safety in lifted representation!
e.g. result.returnFlag should work

Implications from this benchmark

- Need to optimize sub-queries and aggregates (fusing will be key)
- Need to modify LINQ join to accept more than one collection