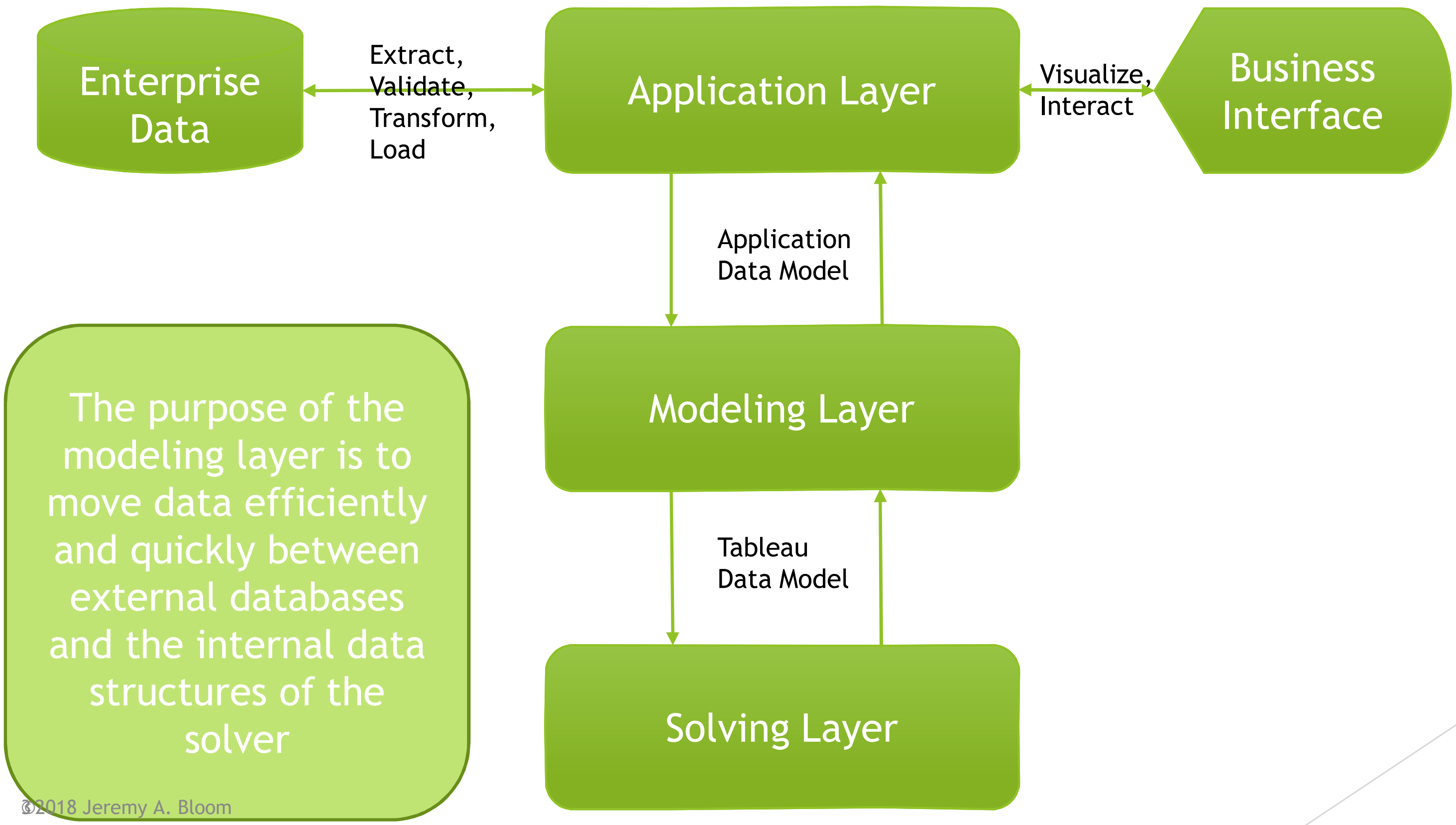
Optimization Modeling and Relational Data

Dr. Jeremy Bloom
jeremyblmca@gmail.com
October 10, 2018

The Story In Brief

- There is a deep relationship between the structure of the data used in optimization modeling and the constructs (variables, constraints) of an optimization model
- That relationship is demonstrated through an example using SQL and IBM OPL
- We propose a design for a new optimization modeling language based on this relationship

Business Solution Architecture



The Standard (Tableau) Form of a Linear Optimization Model

```
min z = sum(j in J} c[j]x[j]
subject to:
sum[j in J] a[i,j]x[j] ≤ b[i] for all i in I
l[j] ≤ x[j] ≤ u[j] for all j in J
```

Issues:

- The data are usually very sparse
- The index sets are often more general than integers

An Example – Warehouse Location

- A consumer packaged goods supplier needs to decide where to locate its warehouses to serve a set of retail stores at different locations.
- At the same time, it also needs to determine how much capacity each warehouse should have.
- The cost of opening a warehouse has a fixed component, related to the acquisition of land and designing the facility, and a variable component proportional to the capacity of the warehouse.
- The cost to ship the goods from a warehouse to a store depends on the distance between them.
- The objective is to minimize the cost of opening the warehouses and shipping the goods.
- Such an optimization application would typically be used as part of an annual planning process in which the company's management would decide on sales targets and the capital investments needed to support them.
- See Optimization+Modeling+and+Relational+Data+pub on https://github.com/JeremyBloom/Optimization---Sample-Notebooks

Application Data Model – Inputs

```
tuple Warehouse {
      key string location;
      float fixedCost;
      float capacityCost;
}
tuple Store {
      key string storeId;
}
tuple Route {
      key string location;
      key string store;
      float shippingCost;
}
tuple Demand {
      key string store;
      float amount;
}
```

```
//Input Data
{Warehouse} warehouses= ...;
{Store} stores= ...;
{Route} routes= ...;
{Demand} demands= ...;
//demand at the store at the end of route r
float demand[routes]=
      [r: d.amount | r in routes,  d in demands: r.store==d.store];
```

Optimization Model

```
dvar boolean open[warehouses];

dvar float+ capacity[warehouses];

dvar float+ ship[routes] in 0.0..1.0;


dexpr float capitalCost=

    sum(w in warehouses)
    (w.fixedCost*open[w] +
    w.capacityCost*capacity[w]);

dexpr float operatingCost=

    sum(r in routes)
    r.shippingCost*demand[r]*ship[r];

dexpr float totalCost=

    capitalCost + operatingCost


constraint ctCapacity[warehouses];

constraint ctDemand[stores];

constraint ctSupply[routes];
```

```
minimize totalCost;

subject to {


    forall(w in warehouses)
//  Cannot ship more out of a warehouse than its capacity
     ctCapacity[w]: capacity[w] >=
         sum(r in routes: r.location==w.location) demand[r]*ship[r];


    forall(s in stores)
//  Must ship at least 100% of each store's demand
     ctDemand[s]: sum(r in routes: r.store==s.storeId) ship[r] >= 1.0;


    forall(r in routes, w in warehouses: w.location==r.location)
//  Can only ship along a supply route if its warehouse is open
     ctSupply[r]: ship[r] <= open[w];

}
```

Application Data Model – Outputs

```
tuple Objective {
    key string problem;
    key string dExpr;
    float value;
}

tuple Shipment {
    key string location;
    key string store;
    float amount;
}

tuple OpenWarehouse {
    key string location;
    int open;
    float capacity;
}
```

```
{Objective} objectives= {
    <"Warehousing", "capitalCost", capitalCost>,
    <"Warehousing", "operatingCost", operatingCost>,
    <"Warehousing", "totalCost", totalCost>
};

{Shipment} shipments= {
    <r.location, r.store, ship[r]*d.amount> |
    r in routes, d in demands: r.store==d.store && ship[r]>0.0};

{OpenWarehouse} openWarehouses= {
    <w.location, open[w], capacity[w]> |
    w in warehouses};
```

Transforming an Optimization Problem to Tableau Form

Use Relational Database Operators (SQL) to Reshape the Instance Data

- Map Decision Variables and Constraints to Columns and Rows
- Reshape the Instance Data into the Tableau
  - Encode the Decision Variables
  - Encode the Constraints and Decision Expressions
  - Encode the Matrix Entries

	columns_open	columns_capacity	columns_ship
rows_dexpr	entries_dexpr_open	entries_dexpr_capacity	entries_dexpr_ship
rows_ctCapacity		entries_ctCapacity_capacity	entries_ctCapacity_ship
rows_ctDemand			entries_ctDemand_ship
rows_ctSupply	entries_ctSupply_open		entries_ctSupply_ship

Why SQL?

- Widely known and used by developers
- Relatively easy to learn
- Many platforms support it
- ANSI standard (with product-specific variances)
- Portable
- Query optimization

SQL with Apache Spark™

- Apache Spark™ is a unified analytics engine for large-scale data processing
- Open Source, written in Scala
  - more than 1200 developers from over 300 companies
- Supports Scala, Java, Python, R, and SQL
- Built on distributed datasets with parallel processing (like Hadoop)
  - Driver
  - Multiple Workers
- Spark operations fall into two classes
  - Transformations (e.g. map, filter) create a new dataset; they are lazy, get executed in parallel when action is called, can be rearranged to optimize execution
  - Actions (e.g. reduce, count, collect) return a non-dataset result (e.g. a scalar) to a task driver; trigger the optimized transformation chain
- Spark can execute SQL directly or by chained method calls (like Pandas)
- Spark datasets are not peristent. If you want to persist your data, you need to interface with a persistent data store
- My approach would also work with alternative databases that support SQL

Terminology

SQL	Spark	OPL
Schema	StructType	Tuple*
Record	Row	Tuple*
Table	Dataset<Row>	Tupleset
SELECT Query	select(…)	filter or slice

*tuple is used both as a schema and a record

Mapping Decision Variables and Constraints to Columns and Rows

indices_capacity = SELECT location, CONCAT (‘capacity_’, location) AS column
FROM warehouses

indices_open = …

indices_ship = SELECT location, store, CONCAT (‘ship_’, location, ‘_’, store) AS column
FROM routes

indices_ctDemand = SELECT storeId, CONCAT (‘ctDemand_’, storeId) AS row
FROM stores

indices_ctCapacity = …

indices_ctSupply = …

Encoding the Decision Variables

```
columns_open =   SELECT indices_open.column AS column,
                        'open' AS variable,
                        warehouses.fixedCost AS c
                    FROM warehouses, indices_open
                    WHERE warehouses .location = indices_open.location
columns_capacity = …
columns_ship =    …

columns_boolean= SELECT * FROM columns_open ORDER BY column
columns_float=     SELECT * FROM columns_capacity
                    UNION ALL SELECT * FROM columns_ship
                    ORDER BY column
```

Encoding the Constraints

```
rows_ctCapacity =         SELECT  indices_ctCapacity.row AS row,
                                  'ctCapacity' AS constraint,
                                  CAST('0.0' AS double) AS b
                              FROM warehouses, indices_ctCapacity
                              WHERE warehouses.location = indices_ctCapacity.location
rows_ctDemand =      …
rows_ctSupply =      …

rows_all =    SELECT * FROM rows_ctCapacity
                UNION ALL SELECT * FROM rows_ctDemand
                UNION ALL SELECT * FROM rows_ctSupply
                ORDER BY row
```

Encoding the Matrix Entries

```
coefs_ctCapacity_ship=
    SELECT  indices_ctCapacity.row AS row,
            indices_ship.column AS column,
            -demands.amount AS a
      FROM warehouses,  routes, indices_ctCapacity, indices_ship, demands
      WHERE routes .location = warehouses .location
            AND indices_ctCapacity.location = warehouses .location
            AND routes .location = indices_ship.location
            AND routes.store = indices_ship.store
            AND demands.store = routes.store
…
```

```
coefs_boolean= SELECT *
               FROM coefs_ctSupply_open
               ORDER BY row, column
```

```
coefs_float=  SELECT * FROM coefs_ctCapacity_capacity
   UNION ALL SELECT * FROM coefs_ctCapacity_ship
   UNION ALL SELECT * FROM coefs_ctDemand_ship
   UNION ALL SELECT * FROM coefs_ctSupply_ship
   ORDER BY row, column
```

MODSL: Mathematical Optimization Domain Specific Language

```
Data warehouses: set of <
 *location: String,
  fixedCost: Double,
  capacityCost: Double
> <- ;
Data stores: set of <*storeId: String> <- ;
Data routes: set of <
 *location: String,
 *store: String,
  shippingCost: Double
> <- ;
Data demands: set of <
 *store: String,
  amount: Double
> <- ;
```

```
Variable open: array[warehouses] of Binary;
Variable capacity: array[warehouses] of Double in interval 0.0 to infinity;
Variable ship: array[routes] of Double in interval 0.0 to 1.0;

Objective capitalCost: Double := sum(for w in warehouses)
(w.fixedCost*open[w] + w.capacityCost*capacity[w]);
Objective operatingCost: Double := sum(for r in routes)
r.shippingCost*demands[r.store]*ship[r];
Objective minimize totalCost: Double := capitalCost + operatingCost;

Constraint ctCapacity[for w in warehouses] :=
      capacity[w] >= sum(r in routes where r matches w)
      demand[r.store]*ship[r];
Constraint ctDemand[for s in stores] :=
      sum(r in routes where r.store==s.storeId) ship[r] >= 1.0;
Constraint ctSupply[for r in routes] :=
      ship[r] <= open[r.location];
```

MODSL Parsing Service Architecture