Using Optimization Services in Datalog

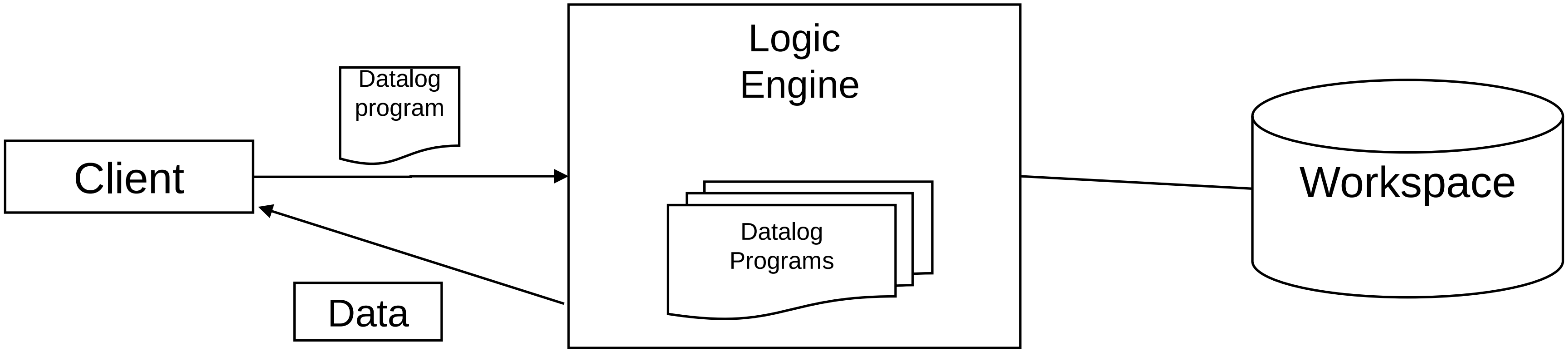
Molham Aref, *Emir Pasalic,* Beata Sarna-Starosta, David Zook
January 12, 2009

LogicBlox Inc.

- Startup company based in Atlanta
  - http://www.logicblox.com
  - ~60 employees + academic collaborators
- Objective
  - Declarative database platform for automated decision support
    - Simulation, optimization, data mining, advanced querying
  - Software as a service
- Applications:
  - Retail supply-chain management (Predictix),
  - On-demand business intelligence (Verabridge),
  - Program analysis (Semmle)

LogicBlox (LB) Technology

- Workspace
  - Provides efficient persistent data storage
  - Organizes data into predicates (like tables)
  - Stores programs, schemas and system parameters in a meta-model
- Logic Engine
  - Manipulates data based on program rules and constraints
  - Manages program execution
  - Handles concurrency, locking and transactions
- Clients
  - Communicate with the engine through Datalog programs
    - Write and install Datalog programs with the engine
    - Receive results of Datalog programs as data



Introduction to Datalog

- A **program** is a set of logical statements about a database
  - If stored data is changed, the Logic Engine recalculates the data to satisfy the program
  - The logic engine executes each program in a transaction
- A **value** is an atomic piece of data
  - Primitive types: floats, strings, integers
  - Entities: abstract user-defined atoms (like a C enum)
- A **predicate** is the only complex data-structure
  - Relations between values, like tables in SQL
    - e.g., parent, ancestor, name, etc.
  - Typed
    - Unary predicates (e.g., person) define types.

```
person(x) -> .
person:firstName(x,s) -> person(x), string(s).
person:lastName(x,s) -> person(x), string(s).
```

    - `person` is a collection of abstract values (entities)
    - `person:firstName` and `person:lastName` are binary predicates relating each person entity to strings

Datalog predicates

- Extensional predicates
  - Store externally provided data
  - Values can be added to extensional predicates declaratively

```
+person(x), +person:firstName(x,"David"), +person:lastName(x,"Z.").
+person(x), +person:firstName(x,"Beata"), +person:lastName(x,"S.").
+person(x), +person:firstName(x,"Emir"), +person:lastName(x,"P.").
```

    - The program asserts three facts about the database
    - The Logic Engine satisfies this program by creating new entities, and inserting the appropriate tuples into the predicate storage
- Intensional predicates
  - Derived from extensional through rules

```
person:name(x,n) <- person:firstName(x,first), person:lastName(x,last), n = first + last.
```

- Built-in support for functional predicates
  - `person:firstName[x]=s` means: for all `x,s1,s2`, if `person:firstName(x,s1)` and `person:firstName(x,s2)` then `s1=s2`.
  - The Logic Engine uses this fact for efficient execution and static checking

Datalog constraints

- A **constraint** is a logical assertion that is always satisfied by a database
  - Any program that violates the assertion is aborted
    - Works on both intensional and extensional predicates
  - Typing constraints

```
person:firstName[x] = s -> person(x), string(s).
person:firstName[x] = 43. // REJECTED!
```

  - Runtime constraints

```
parent(p,c) -> person(p), person(c).
!(parent(x,x)).
```

    - Declaring a new predicate `parent` that relates a parent to a child
    - Asserting a constraint that nobody can be their own parent
    - Can be (syntactically) positive or negative

```
!(person(x), person:Age[x] < 0).
person(x) -> person:Age[x] >= 0.
```

LOGICBLOX®

Derivation rules

- A **derivation rule** is a logical specification of how predicates are computed from other predicates

```
person:name[x] = n -> person(x), string(n).
person:name[x] = n <-
   n = person:firstName[x] + " " + person:lastName[x].
```

- The logic engine finds a set of tuples such that the head (person:name) of the rule is true whenever the body (n = person:firstName[x] + " " + person:lastName[x]) is true
  - Bottom-up evaluation: all possible facts will be derived
  - Incremental evaluation: if any predicates in the body change, only the smallest amount of computation will be performed to update person:name

Rules II

- Support for recursion

```
parent(x,y) -> person(x), person(y).
ancestor(x,y) -> person(x), person(y).
ancestor(x,y) <- parent(x,y).
ancestor(x,y) <- ancestor(x,z), ancestor(z,y).
```

- Support for aggregation
  - Aggregation expressed by special rule syntax

```
person:salary[p,m] = n -> person(p), month(m), float[32](n).
person:salary:toDate[p,m] +=
        (person:salary[p,prevM] where prevM < m).
```

  - Aggregation is built-in
    - Any particular aggregation is expressible in pure Datalog
    - But the general aggregation operator (like sum, count etc.) is not
    - Thus += is a special kind of rule that
      - iterates through all tuples on the right hand side that produce a value
      - sums up those values and stores them in the left-hand-side predicate

LOGICBLOX®

Blocks

- A **block** is a collection of predicates, rules and constraints
  - Clients communicate with the logic engine by sending blocks to it
  - Logic Engine installs and executes blocks
    - Installed blocks: database lifetime
    - Executed blocks: single-transaction lifetime
  - Blocks form modules in a workspace
    - Control visibility of predicates
    - Can be added and removed as a unit

Properties of (LB) Datalog

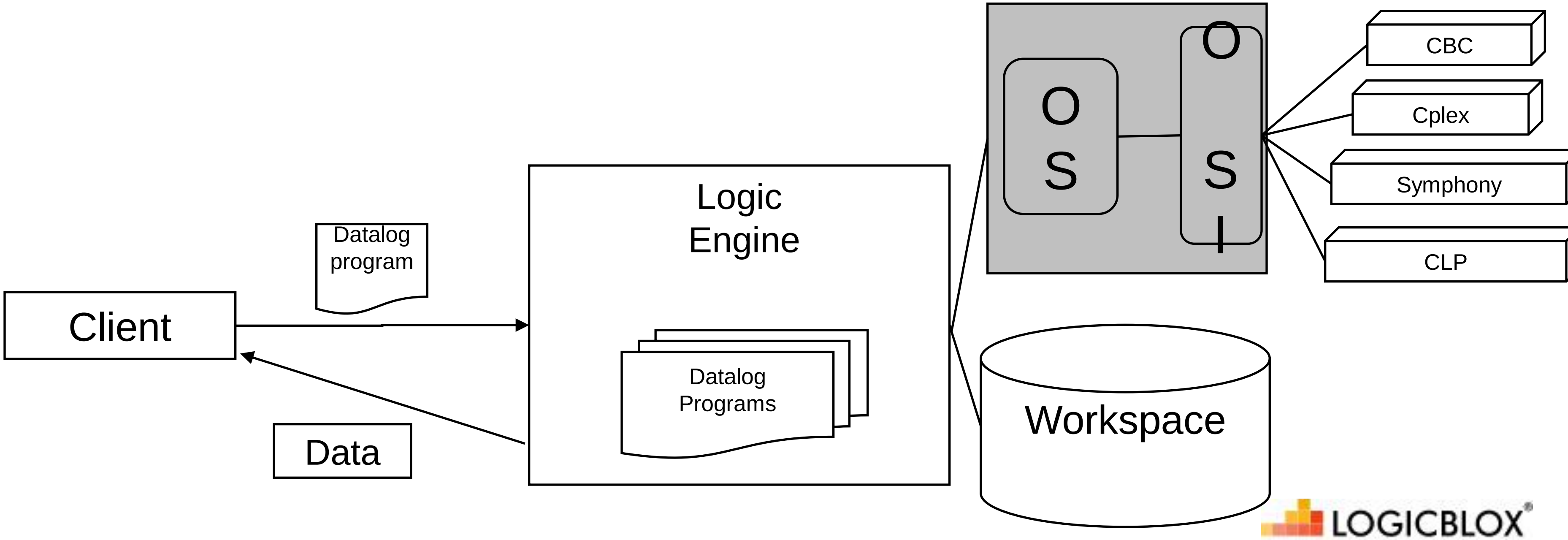
- Guaranteed termination
  - Datalog programs capture exactly the PTIME complexity class
- Purely declarative
  - A subset of first order logic
    - E.g., unlike prolog conjunction is commutative
    - No fixed evaluation strategy implicit in the program
  - Logic engine determines data structures, persistence, execution strategy, memory management
- Efficient execution
  - Persistence
  - Parallelization
  - Query optimization techniques
  - Incremental execution

Optimization

- Cannot solve problems harder than PTIME in Datalog
  - E.g., MIP solvers (they are NP problems)
  - There are many applications for linear/mixed-integer programming
    - Scheduling (shift assignment, flight assignment etc)
    - Retail replenishment planning
- Problem: How to give LogicBlox users the power of escaping the PTIME complexity bounds while maintaining good properties of Datalog?
- Solution: Interface LB Datalog with specialized solvers
- Challenges:
  - Interface with solvers in a declarative/pure way
  - Leverage specialized knowledge embodied in existing implementations
  - Give users the flexibility in interacting with the solvers without compromising the purity of the language
  - Integrate optimization seamlessly with the LB logic engine

Integrating Optimization with LB Datalog

- Use existing Datalog syntax
  - Represent variables and parameters as predicates
  - Represent constraints as runtime constraints
  - Invoke solvers using (slightly extended) rule syntax
- Users still write only high-level specifications
  - Logic engine delegates the solver to calculate values satisfying variable predicates
  - The engine verifies the solutions “for free” by executing runtime constraints



LOGICBLOX

Optimization Services (OS)

- A project developed under COIN/OR
  - "a set of standards for representing optimization instances, results, solver options, and communication between clients and solvers in a distributed environment using Web Services."
- Key technology in integrating solvers with LB
- C/C++ and XML Implementation
  - Integrates well with our code base (in C++)
  - Cross-platform

COIN/OR Optimization Services

- OSiL
  - Provides a standard API to construct a single problem instance
- OSoL
  - Provides a standard API to specify solver options
- OSrL
  - Provides a standard API to retrieve results of a solver running on an instance
    - Returns values for instance variables
    - Other information (not used in LB)
- OSI
  - Provides a standard API to interact with multiple solvers
  - Pluggable drivers for executing instances on different solvers
    - COIN-OR LP solver (OsiClp) and COIN-OR Branch and Cut solver (CoinBcp); CPLEX (OsiCpx); DyLP (OsiDylp); FortMP (OsiFmp); GLPK, the GNU Linear Programming Kit (OsiGlpk); Mosek (OsiMsk); OSL, the IBM Optimization Subroutine Library (OsiOsl); SYMPHONY (OsiSym); The Volume Algorithm (OsiVol); XPRESS-MP (OsiXpr).

Optimization in Datalog

- Compiler
  - Rewrites a Datalog block containing definitions of variables, parameters, objective functions and constraints into a low-level mathematical specification of the optimization problem
- Execution engine extension
  - Executes the optimization problem specification by supplementing standard Datalog rule execution semantics
  - Invisible to the user
    - Seamlessly integrates with non-optimization based rules
    - Logically preserves semantics of Datalog

Optimization Compiler

- Find (based on special syntax) the set of variable predicates
- Identify the runtime constraints that involve the variable predicates
  - check constraints for basic feasibility
    - linear arithmetic
    - correct use of indexes on variables and parameters
- Identify predicates that represent index sets and parameters
- Build a low-level runtime specification in an intermediate mathematical notation
  - similar to AMPL or other modeling languages
  - the objective function, constraints, bounds, types of variables, set, parameter, and variable predicates, and direction of optimization
- Pass the low-level runtime specification to the logic engine as part of the definition of the variable predicates

LOGICBLOX®

Optimization Execution

The logic engine uses the low-level specification to compute the values of the variable predicates:

- Evaluate all index sets
- Check that parameter predicates have values at required parameters
- Create a OSiL instance data-structure
  - for each variable predicate, at each index, create a unique symbolic instance variable
  - use a small interpreter for the low-level representation
    - symbolically evaluate the objective function, binding instance variables
    - symbolically evaluate the constraints to obtain a constraint matrix
- Invoke OSI library to call the solver on the instance
  - returns a binding of each instance variable to a value (OSrL)
- Map the values of instance variables back to the variable predicate
- Continue with LB execution
  - Eventually runtime constraints are executed with real values of variables to *check* the solution

Discussion

- Both index sets and parameters can be computed by rules
  - e.g., dependent and arbitrary patterns of indexing of variables and sets
- Incremental evaluation forces re-computation of the optimization solution when parameter data changes
- Additional options can be passed to OSoL through the meta-model
  - predicates in the database that store the information about executing the optimizer, e.g., which optimizer to use
- Optimization Services library is essential
  - we avoid re-implementing our infrastructure for different solvers
  - clear, high-level API for solver/optimization users (not necessarily experts)
  - good XML support for debugging of instances

Future Work

- Increase expressiveness
  - Disjunctive constraints
    - Compile into conjunctive constraints using known techniques
  - Non-linear constraints
    - Already supported by OS. We just need to extend our compiler slightly to construct the non-linear OS instances.
  - What-if evaluation
    - If user interactively changes data in a variable predicate, turn it into extra constraints and re-solve
- Use optimization techniques to extend expressive power of Datalog
  - Compile disjunctive Datalog into integer optimization problems
- Increase efficiency
  - Automatically detect fastest possible solver based on types and constraints (e.g., don't use an integer solver if all variables are reals)
  - Warm-start (e.g., can we make solvers at least partly incremental, like LB Datalog rules)
  - Automatically break up problems based on data dependencies and solve in parallel

THE END

Example: Diet

- Given
  - Index sets
    - FOOD : set of foods
    - NUTR : set of nutrients
  - Parameters
    - amt[n,f] = v: the amount v of a nutrient n in food f
    - nutrLow[n]=v: the minimum daily amount of nutrient n
    - cost[f]=v: the cost of food f
- Objective
  - Variable(s): buy[f] = v : amount of food to buy, for each f in FOOD
  - Minimize total cost of food while satisfying the constraint that the daily minimum amount for each nutrient is met

Diet problem in LB Datalog

- Index sets are implemented as *entities*
  - Abstract values denoting distinct items

```
NUTR(x), NUTR:name(x:n) -> string(n).
FOOD(x), FOOD:name(x:n) -> string(n).
```

- Parameters are represented as functional predicates with an index set domain, and a numeric range

```
amt[n, f] = a -> NUTR(n), FOOD(f), float[64](a), a >= 0.
nutrLow[n] = nL -> NUTR(n), float[64](nL), nL >= 0.
        cost[f] = c -> FOOD(f), float[64](c), c >= 0.
```

- Variables are represented as functional predicates with an index set domain, and a numeric range
  - Note the runtime constraint that `buy[f] >= 0` as a part of type declaration

```
Buy[f] = b -> FOOD(f), float[64](b), b>=0.
```

Diet problem in LB Datalog

- Objective function is represented as a predicate that computes its value from the index sets and variables

```
totalCost[] = x -> float[32](x).
totalCost[] += cost[f] * Buy[f] where FOOD(f).
```

  - `totalCost` is a functional predicate containing only one value
  - It is an aggregation (sum) of `cost[f] * buy[f]` for each food `f`
- Constraints are represented as LB Datalog runtime constraints involving the variable predicate

```
totalNutrAmt[n] += amt[n,f] * Buy[f] where FOOD(f).
!(NUTR(n), totalNutrAmt[n] < nutrLow[n]).
```

  - The constraint states that the total amount of each nutrient is never less than the required daily minimum
  - Auxiliary predicate `totalNutrAmt[n]` computes the total amount of nutrient `n` in all purchased foods

Diet problem in LB Datalog

- The Datalog rule that computes `buy[f]` puts all those elements together
  - Built-in *higher-order* predicate `solve:minimize` is the interface to the solver
    - Takes the value of the objective function
    - Based on runtime constraints in the block, runs the solver to compute the value of the variable predicates

```
Buy[f] = result <- result = (solve:minimize[cost][f] where totalCost[]=cost).
```