The RBAC challenge in LogiQL: Solutions and Limitations

K. Tuncay Tekle, Stony Brook University
tuncay@cs.stonybrook.edu

June 28, 2018

1 Introduction

LogicBlox [1] is a commercial system unifying the programming model for enterprise software development that combines transactions with analytics by using a flavor of Datalog called LogiQL. LogiQL is a strongly-typed, extended form of Datalog that allows coding of entire enterprise applications, including business logic, workflows, user interfaces, statistical modeling, and optimization tasks.

The programming challenge for the Logic and Practice of Programming workshop is based on Role-Based Access Control (RBAC) [2], and provides various problems whose solutions require the handling of (1) rules, (2) queries, (3) updates, (4) type constraints, (5) other constraints, (6) optimization, (7) planning. In this paper, we show that problems related to the first five can be completely handled, one case of optimization can be handled with some caveats, and another can be written but does not run in the latest implementation; and planning cannot be handled in LogiQL.

As an overview, LogicBlox is a state-based system with a persistent database that can be manipulated, where one can add facts to the database, and rules and constraints to the state, and query the database at any point in time. Rules and queries are readily handled by a logic programming system as expected. Type constraints are also intrinsic to LogiQL as every predicate is typed. Constraints that are not type constraints in this challenge can also be intuitively expressed. All of these are maintained in the presence of updates. Optimization problems can also be handled, although we show that it requires a rewrite to fit in the optimization framework, and has some restrictions. There is no capability for planning in LogiQL, however we propose a syntax similar to optimization that requires an implementation. However, since LogicBlox is a closed-source system, we do not expect that this would be implemented.

2 Implementation

We briefly introduce how LogicBlox works, and how it evaluates LogiQL. In LogicBlox, the system starts with an empty state, where a state contains the fact database, rules and constraints. The state can be manipulated by adding facts, rules or constraints, where more facts can be added automatically by the rules due to the addition of facts, or the manipulation can be rejected due to the added facts not satisfying constraints.

The programming system can be separated into three components: (1) installed blocks, (2) updates, (3) queries. Installed blocks contain entity and predicate declarations, rules, and constraints. Updates are inserts, updates or deletes to entities or predicates. Queries retrieve data from predicates or may introduce a temporary rule to retrieve data. A sketch of how LogicBlox evaluates LogiQL as follows: (1) when a new block is installed, all declarations in the block are installed, all rules in the block are evaluated bottom-up with respect to the current state of the database, and all constraints are checked; if a constraint fails, the installation of the block is rolled back; (2) for an update: all rules in all installed blocks are evaluated bottom-up (incrementally whenever possible), then all type constraints are enforced by deleting facts of a predicate related to an entity if that entity is deleted in an update, then if any other constraint is not satisfied after the evaluation of the rules, the update is rolled back; (3) for a query, the results of the query are retrieved as expected.

Next, we dig in to the components of RBAC, where we talk about only new concepts needed for each component in order. The code for each component can be found in the appendix.

2.1 Core RBAC

Entity declarations and updates. This component first defines some types: Users, roles, and permissions. Types are called entities in LogiQL, and entities can be implemented via *entity* and *refmode* (reference mode) declarations. A user can then be added using update statements. All of the update functions for these entities (`AddUser, DeleteUser, AddRole, DeleteRole, AddPerm, DeletePerm`) can be implemented using the entity declarations and update statements.

Relation declarations and updates. Next, user-role pairs, and permission-role pairs are introduced with the constraint that each set of pairs is a subset of the cross-product of the entities. This is trivially satisfied in LogiQL by the nature of the declaration of the set of pairs, where each argument's type is provided by a constraint. Therefore, e.g., any insert to a predicate verifies that the types of its arguments matches the declaration, or else the update fails. If an entity of a type is removed, all predicate facts related to that particular entity is removed as well. All of the update functions for these relations (`AddUR, DeleteUR, AddPR, DeletePR`) can be implemented using these declarations.

Queries. All three queries in this component are easily implemented via logical queries.

2.2 Hierarchical RBAC

Rules. Reflexive-transitive closure of the role hierarchy can be defined via rules.

Complex constraints. So far we have only seen type constraints. More complex constraints can be similarly implemented via the right arrow notation. For example, the acyclicity of the role hierarchy can be enforced by constraints.

2.3 Static Separation of Duty (SSD)

SSD can be implemented using the concepts introduced above, and the hieararchical version only needs to change the predicate to use the Hierarchical RBAC version rather than the Core.

2.4 Administrative RBAC

There are two types of new challenges in this component: optimization and planning. There is no planning functionality in LogiQL, and optimization has restrictions. We show how to solve the `MinRoleAssignments` and `MinRoleAssignmentsWithHierarchy` optimization problems with caveats, and suggest a syntax for planning.

Optimization. In LogiQL, optimization is performed by translating the rules and variables into a mathematical program. Therefore, the variables and the predicate to solve for need to be defined, and there are various restrictions on what the rules and constraints can look like. Then, the mathematical problem is solved via an invocation of a solver using the facts of the current state. Note that this is significantly different than how the system normally operates.

For the hierarchical version of the problem, the definition of rules needs to be recursive. However, the preprocessor fails at translating the rules written this way, although a mapping seems obvious for this example.

Planning. Planning problems in Administrative RBAC cannot be specified in LogiQL, but the lack of an implementation (and the possibility of it being implemented being close to zero) notwithstanding, it is not difficult to imagine that directives such as the ones used in optimization could be used to construct action variables based on each action that can be taken, and a solver variable for the objective.

References

[1] Aref, M., ten Cate, B., Green, T. J., Kimelfeld, B., Olteanu, D., Pasalic, E., Veldhuizen, T. L., and Washburn, G. Design and implementation of the logicblox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015* (2015), T. K. Sellis, S. B. Davidson, and Z. G. Ives, Eds., ACM, pp. 1371–1382.

[2] Liu, Y. A. Role-based access control as a programming challenge. In *Logic and Practice of Programming Workshop* (2018).

A Core RBAC

- Definition of Users, Roles, Permissions (types):

```
User(u), UserName(u:n) -> string(n).
Role(r), RoleName(r:n) -> string(n).
Permission(p), PermissionName(p:n) -> string(n).
```

- Addition/removal of a user (update):

```
+UserName[_] = "tuncay".
-UserName[_] = "tuncay".
```

- Definition of user-role, permission-role pairs (relations):

```
UR(u,r) -> User(u), Role(r).
PR(p,r) -> Permission(p), Role(r).
```

- Addition/deletion of user-role, permission-role pairs (update):

```
+UR(u,r) <- UserName[u] = "tuncay", RoleName[r] = "admin".
-PR(p,r) <- PermissionName[u] = "write", RoleName[r] = "admin".
```

- `AssignedRoles`, `UserPermissions`, `CheckAccess` queries:

```
_(r) <- UserName[u] = "tuncay", UR(u,r).
_(p) <- UserName[u] = "tuncay", UR(u,r), PR(p,r).
_(true) <- UserName[u] = "tuncay", PermissionName[p] = "read", UR(u,r), PR(p,r).
```

B Hierarchical RBAC

- Role hierarchy definition:

```
RH(r1,r2) -> Role(r1), Role(r2).
```

- Definition of the reflexive-transitive closure of the role hierarchy, and `AuthorizedRoles` (rules):

```
Trans(r1,r1) <- Role(r1).
Trans(r1,r2) <- RH(r1,r2).
Trans(r1,r2) <- RH(r1,r3), Trans(r3,r2).

AuthorizedRoles(u,r1) <- UR(u,r), Trans(r,r1).
```

- Enforcement of the acyclicity of the role hierarchy (constraint):

```
Trans(p,p2), Trans(p2,p) -> p = p2.
```

C Static Separation of Duty

- Declarations of types and relations:

```
RoleSet(rs), RoleSetName(rs:n) -> string(n).
RoleSetRoles(rs,r) -> RoleSet(rs), Role(r).
SSDItem(s), SSDItemName(s:n) ->string(n).
SSDItemRoleSet[s] = rs -> SSDItem(s), RoleSet(rs).
SSDItemCardinality[s] = c -> SSDItem(s), int(c).
SSDItem(s) -> SSDItemRoleSet[s] = _.
SSDItem(s) -> SSDItemCardinality[s] = _.
```

- Deletion of an `SSDItem`:

```
-SSDItem[s], -RoleSet(rs) <- SSDItemName@prev[s] = "s1", SSDItemRoleSet@prev[s] = rs.
```

  Note here the use of `@prev` on the right hand side of the rule. For an update of a predicate that depends on a fact of that predicate before the update starts, this annotation is utilized to avoid recursion. So this rule says if there is an SSD item with the name `s1` before the update is started, then delete that SSD item.

- Adding/deleting role sets, updating cardinality:

```
+RoleSetRoles(rs,r) <- SSDItemName[s] = "s1", SSDItemRoleSet[s] = rs.
-RoleSetRoles(rs,r) <- SSDItemName[s] = "s1", SSDItemRoleSet[s] = rs, RoleName[r] = "r1".
^SSDItemCardinality[s] = c <- SSDItemName[s] = "s1".
```

  The ^ symbol indicates an update of an existing functional value. We need to define the number of items in a role set to support the constraints as follows.

- Constraints for the role count and cardinality relations:

```
RoleCount[s] = rc -> SSDItem(s), int(rc).
RoleCount[s] = rc <- agg<<rc = count()>>
                     SSDItemRoleSet[s] = rs, RoleSetRoles(rs,_).

SSDItemCardinality[s] = c -> c > 0, c < RoleCount[s].

UserSSDRoleCount[u,s] = rc -> User(u), SSDItem(s), int(rc).
UserSSDRoleCount[u,s] = rc <- agg<<rc = count()>>
                              UR(u,r), SSDItemRoleSet[s] = rs, RoleSetRoles(rs,r).
UserSSDRoleCount[_,s] < SSDItemCardinality[s].
```

- Queries:

```
_(name) <- SSDItemName[_] = name.
_(r) <- SSDItemName[s] = "s1", SSDItemRoleSet[s] = rs, RoleSetRoles(rs,r).
_(c) <- SSDItemName[s] = "s1", SSDItemCardinality[s] = c.
```

- For Hierarchical RBAC, the only necessity is to change the rule defining `UserSSDRoleCount` to use `AuthorizedRoles` instead of `UR`.

D Administrative RBAC

For the first problem, MinRoleAssignments, the total size of new UR' and PR' need to be minimized; however ROLES' can also contain new roles. There is no way to invent new values during optimization in LogiQL. Therefore, we restrict the question so that ROLES'=ROLES. Then, we first define two predicates which define whether a role subsumes another and the original permissions.

```
Subsume[r1,r2] = b -> Role(r1), Role(r2), int(b).
Subsume[r1,r2] = 1 <- Trans(r1,r2).
OrigPerms[u,p] = b -> User(u), Permission(p), int(b).
OrigPerms[u,p] = 1 <- AuthorizedRoles(u,r), PR(r,p).
```

Then, we define the new variables and constraints for the optimization problem as follows, where NewPerms are the user-permissions based on the NewUR and NewPR sets which are self-explanatory:

```
NewUR[u,r] = b -> User(u), Role(r), int(b), b >= 0, b <= 1
NewPR[p,r] = b -> Permission(p),  Role(r), int(b), b >= 0, b <= 1.
lang:solver:variable(‘NewUR).
lang:solver:variable(‘NewPR).

NewPerms[u,p] = b -> User(u), Permission(p), int(b), b >= 0, b <= 1.
NewPerms[u,p] = b1, OrigPerms[u,p] = b2 -> b1 = b2.
NewPerms[u,p] = b1, !OrigPerms[u,p] = _ -> b1 = 0.
NewPerms[u,p] = v, NewUR[u,r1] = v1, Subsume[r1,r2] = v2, NewPR[p,r2] = v3 -> v >= v1 * v2 * v3.
```

The directive lang:solver:variable defines the variables of the optimization problem. Finally, we define the optimization function as follows:

```
NewURSize[] = nus <- agg<<nus=total(v)>>
                     v = NewUR[_,_].
NewPRSize[] = nps <- agg<<nps=total(v)>>
                     v = NewPR[_,_].
TotalSize[] = tc -> int(tc).
TotalSize[] = NewURSize[] + NewPRSize[].
lang:solver:minimal(‘TotalSize).
```

The directive lang:solver:minimal defines the predicate to solve for (by way of minimization).