Appendix J: Improving Your Program's Performance

This appendix provides a brief summary of some ways to help improve the performance of your LogiQL programs.

DERIVED-ONLY PREDICATES

By default, predicates computed via rules are treated as materialized views, so their fact populations are stored for use in later transactions. This is typically better because it allows updates to be computed incrementally (e.g., computing a bank account balance from the latest update to the previous balance is much faster than calculating the balance each time by processing all the updates on the account that have occurred since the account was opened).

However, it is sometimes useful to fully re-evaluate a derived predicate, making its result available for other rules within the current transaction, but not installing its result in the database when the transaction is committed. Such *derived-only predicates* are declared using the metapredicate setting `lang:derivationType[`]="Derived"`.

Derived-only predicates are useful for defining a complex computation, which could potentially result in an infinite set of facts, for reuse elsewhere in the transaction in a context that finitely constrains its arguments. Recall the following example discussed in Unit 4.5:

```
Mass(m), hasKgValue(m:kg) -> float(kg).
Energy(e), hasJouleValue(e:j) -> float(j).
cSquared[m] = e -> Mass(m), Energy(e).
```

```
cSquared[m] = e <- m * pow[300000000f, 2f] = e.
lang:derivationType[`cSquared] = "Derived".
isHighEnergySource(m) -> Mass(m).
isHighEnergySource(m) <- Mass(m), cSquared[m] >
    pow[10f, 18f].
isVeryHighEnergySource(m) -> Mass(m).
isVeryHighEnergySource(m) <- Mass(m), cSquared[m] >
    pow[10f, 19f].
```

The `cSquared` rule would normally be treated as unsafe, because there are infinitely many values of `m` and `e` that satisfy it. However, because `cSquared` is a derived-only predicate, its values are computed only in the context of the other rules that use them, and those rule bodies restrict `m` to the finite set of values asserted for the domain predicate `Mass(m)`. The following performance tip summarizes appropriate usage.

PT1: *Use the default setting (derived-and-stored) for computed predicates whenever incremental computation is desired (the typical situation). If you wish to reuse a computation with potentially infinite results within other rules in the same transaction that finitely constrain it, then declare the predicate derived-only.*

Unguarded Delta Rules

Recall how in Exercise 5B of Chapter 3 grandparent facts were added using an intentional database (IDB) rule. Here `isGrandparentOf` is declared instead as an extensional database (EDB) predicate to allow direct assertions about grandparenthood when we do not know at least one of the parenthood facts whose combination would imply it. The delta rules automatically add grandparent facts when relevant parenthood facts are added:

```
Person(p), hasPersonName(p:pn) -> string(pn).
isParentOf(p1, p2) -> Person(p1), Person(p2).
isGrandparentOf(p1, p2) -> Person(p1), Person(p2).
+isGrandparentOf(p1, p2) <- +isParentOf(p1, p3),
    isParentOf(p3, p2).
+isGrandparentOf(p1, p2) <- isParentOf(p1, p3),
    +isParentOf(p3, p2).
```

Executed delta rules are evaluated before the installed delta rules, so the relevant parenthood facts are already there when the installed delta rules are run. For the same reason, the single installed delta rule below expresses the same result as the combination of rules above:

```
+isGrandparentOf(p1, p2) <- isParentOf(p1, p3),
    isParentOf(p3, p2). // Error!
```

This rule would derive all applicable grandparenthood facts on every transaction, in contrast to the guarded delta rules in the first version that derive the relevant grandparenthood facts only when a relevant parenthood fact is added. However, this delta rule generates an error message because its rule body does not include a delta or pulse predicate that provides a condition, or *guard*, that must be satisfied before the rule is executed. Such a delta rule is said to be *unguarded*. If they were allowed, unguarded delta rules would be evaluated on every transaction. Apart from the extra expense incurred by such redundant computation, unguarded delta rules might lead to predicate locking problems that can potentially disable programs that are intended to be concurrent.

Problems can also occur if a delta rule to be executed at a stage other than `initial` references a negated atom from an earlier stage.

PT2: *Avoid unguarded delta rules (i.e., delta rules in which the rule body contains no delta predicate or pulse predicate condition that is required for the rule to be evaluated).*

FOLDING DISJUNCTIONS

Consider the following program, which includes a derivation rule to select a short list of applicants to be interviewed for a technical position:

```
Applicant(a), hasApplicantNr(a:n) -> int(n).
Language(la), hasLanguageName(la:n) -> string(n).
hasIndustryExperience(a) -> Applicant(a).
isFluentIn(a, la) -> Applicant(a), Language(la).
nrPublicationsOf[a] = n -> Applicant(a), int(n).
Discipline(d), hasDisciplineName(d:n) -> string(n).
hasPhDIn(a, d) -> Applicant(a), Discipline(d).
isShortListed(a) -> Applicant(a).
isShortListed(a) <-
    hasIndustryExperience(a), isFluentIn(a, "English"),
    nrPublicationsOf[a] >= 20,
    (hasPhDIn(a, "Computer Science") ;
    hasPhDIn(a, "Logic")).
```

The derivation rule includes a conjunction of conditions to be satisfied (industrial experience, fluency in English, and at least 20 publications), as well as a disjunction of conditions to be satisfied (must have a Ph.D. in either computer science or logic).

Although this program works, internally the LogiQL engine transforms the derivation rule into the following, disjunction-free rules, in order to execute it:

```
isShortListed(a) <-
    hasIndustryExperience(a), isFluentIn(a, "English"),
    nrPublicationsOf[a] >= 20,
    hasPhDIn(a, "Computer Science").

isShortListed(a) <-
    hasIndustryExperience(a), isFluentIn(a, "English"),
    nrPublicationsOf[a] >= 20,
    hasPhDIn(a, "Logic").
```

Each of the transformed rules includes the original conjunction of three conditions, as well as one of the original disjuncts. Hence, the conjunction in the original rule has to be processed twice, once for each original disjunct. If the original conjunction is complex or the number of disjuncts is high, this kind of duplication of effort can significantly impact performance. In such cases, it is better to specify one rule to compute just the disjunction (known as "*folding* the disjunction"), and then use that computed predicate instead of the disjunction in a second rule. For the current example, this leads to the following reformulation:

```
hasRelevantPhD(a) <-
    hasPhDIn(a, "Computer Science") ;
    hasPhDIn(a, "Logic").
isShortListed(a) <-
    hasIndustryExperience(a), isFluentIn(a, "English"),
    nrPublicationsOf[a] >= 20, hasRelevantPhD(a).
```

With this approach, the conjunction is computed once only, leading to better performance. For some more complicated disjunctions, folding cannot be done (e.g., if a disjunct negates over a variable that is positively bound only outside the disjunction). However, where it can be done, folding disjunctions often improves performance.

PT3: *If the rule body includes a complex conjunction as well as a disjunction, consider folding the disjunction to a separate rule.*

DISJOINT PREDICATE RULES

Unit 2.6 used the following example code to compute the number of children that a `Person` has:

```
nrChildrenOf[p] = 0 <- Person(p), !isParentOf(p, _).
nrChildrenOf[p] = positiveNrChildrenOf[p] <-
    isParentOf(p, _).
// The number of children of p = 0
// if p is a person who is not a parent of someone ,
// else it's the positive number of children of p .
```

Notice that there are two rules for `nrChildrenOf`. Because `nrChildrenOf` is a functional predicate, there is a danger that the two rules will produce different values for a given argument `Person`. To make sure this does not happen, the execution engine must specifically check for violations. For these two rules, however, we know that a given argument `Person` can only satisfy one of the two rules. Hence, in principle, the check need never be made.

Fortunately, LogiQL has a way for you to let the execution engine know that at most one of these possibilities could ever hold. To do this, you should make use of the `lang:isDisjoint` metapredicate:

```
lang:isDisjoint[`nrChildrenOf] = true.
// The two rules for nrChildrenOf return mutually
// exclusive results.
```

Notice that the predicate `nrChildrenOf` serves as argument of the metapredicate `lang:isDisjoint`. Although this metapredicate is not needed to get the right result, it can improve performance by notifying the compiler that the result sets returned individually by the two rules for `nrChildrenOf` are mutually exclusive. Since each of the two individual result sets is known to be functional, disjointedness between them implies that the overall result set is also functional. Hence, the LogiQL execution engine does not need to perform a separate check to ensure that this functional dependency is satisfied.

PT4: *If multiple rule bodies defining the same functional predicate can never be applied to the same argument, use the* `lang:isDisjoint` *metapredicate to avoid making unnecessary functional dependency violation checks.*