LogicBlox

Smart database for next-generation applications

LOGICBLOX 4 MIGRATION GUIDE

Contents

1 Who Should Upgrade to LogicBlox 4

LogicBlox 4 supports applications built using the ServiceBlox service-oriented architecture, with Javascript-based front-end UIs. Applications that use MochaBlox or Client-side Wrapped Controls should migrate to an independent third-party Javascript framework as part of the migration.

Blade-based applications -- applications that rely on the workbook framework, the Java applet grid, and measure rules -- are not at this point candidates for upgrade.

2 What Has Changed in LogicBlox 4

Compared to previous generations of the LogicBlox platform, LogicBlox 4 delivers faster query evaluation, better concurrent transaction throughput, a simplified programming model and reduced database administration overhead, as well as full ACID-compliance. In order to achieve these advantages, it was necessary to make some non-backward compatible changes. The following sections provide guidance on what needs to be changed in applications, as well as how to better take advantage of new features.

2.1 Language Semantics

2.1.1 Numeric Types

Supported and Recommended Types

- LogicBlox 4 adds support for the integer numeric type `int` which replaces signed and unsigned integers of various widths. Similarly, the binary floating-point numeric type `float` replaces the old `float[32]` and `float[64]` types. Porting code from LogicBlox 3.x to LogicBlox 4 is a straightforward source code transformation that can be aided by a script for rewrites, as well as compiler-generated warnings and errors. More information on the migration script can be found here.
  - The new type `int` for integers replaces the old unsigned and signed integer types with explicit bit widths (`uint[8]`, `uint[16]`, `uint[32]`, `uint[64]`, `int[8]`, `int[16]`, `int[32]`, `int[64]`). Integers are implemented as signed 64-bit numbers.
  - The new type `float` binary floating-point replaces the old float types with explicit bit widths (`float[32]`, `float[64]`). Floats are implemented as 64-bit IEEE 754 binary floating-point numbers and are preferable for scientific applications. The LogicBlox 4 type float has the same size and precision as `float[64]` in LogicBlox 3.x and `double` in most languages that implement both `double` and `float`. Floats can represent a wider range of values and provide more precise multiplications and divisions than decimal fixed-point numbers, but they have more severe round-off errors which can lead to unintuitive behavior.
- LogicBlox 4 adds support for `decimal`. Decimals are implemented with an integer part of up to 13 digits and a fractional part of up to 5 digits. Decimal numbers have better behavior over floats, which leads to improved accuracy, efficiency, and safety over floats. Advantages of using `decimal` include exact addition and subtractions (and thus preferable especially to represent financial data), and better total aggregation performance, compared against floating-point types.

- Floating-point decimal types, `decimal[64]` and `decimal[128]`, are no longer supported.
- In LogicBlox 4, `float` literals are written with an an explicit 'f' suffix, for example: '1.2f', '3E4f', and '5.6E7f'. On the other hand, `decimal` literals can have an optional 'd' suffix, for example, '1.2' or '1.2d'.

> **Note**
> Note that, in particular, a literal '1.2' is interpreted as a decimal on LogicBlox 4, instead of as a float on LogicBlox 3.x. This change is intentional to encourage applications to migrate uses of the `float[32]` and `float[64]` types on 3.x to the `decimal` type on 4, while the literal remains '1.2'.

Floating Point Computations

- If a floating-point primitive produces a `NaN` value, it is treated as a failure; no value is produced. If a floating-point operation produces -0.0f, it is silently converted to 0.0f.

Example 2.1

`+F(0.0f/0.0f).` will not result in anything being inserted to `F`, since 0.0f/0.0f produces NaN.

- In LogicBlox 4 floating point numbers are compared bit-wise for equality. The `system:float32Precision` and `system:float64Precision` variables to control the precision of floating-point comparisons are no longer supported.

2.1.2 Built-ins

- Standard library functions for integers use an `int:` path instead of `uint8:`, `int8:`, etc.; for binary floating-point numbers use a `float:` path instead of `float32:` or `float64:`; and for decimal fixed-point numbers use a `decimal:` path instead of `decimal64:` or `decimal128:`.

  – `int:add[i1,i2]=i3` instead of `int32:add[i1,i2]=i3`, and
  – `float:decimal:convert[f]=d` instead of `float64:decimal64:convert[i]=f`.

  Conversions between integers, between floats, or between decimals become unnecessary. For example:

  – `u=i` instead of `uint32:int16:convert[u]=i` or `uint32:int16:eq[u]=i`
  – `f=g` instead of `float32:float64:convert[f]=g` or `float32:float64:eq[f]=g` `d=e` instead of `decimal64:decimal128:convert[d]=e`.

- The compiler no longer introduces implicit conversions between numerics. Source programs must include explicit conversions as appropriate.

On LogicBlox 3.x, numerics were implicitly converted from integer to float to decimal of various widths, based on heuristics guided by the types. For example, the comparison `1 < 3.14` of an integer `1` and float `3.14` would be interpreted as `float32:lt_2(int32:float32:eq[1], 3.14)`, where `1` is converted from `int[32]` to `float[32]`, and the comparison `float32:lt_2` is performed on `float[32]` arguments.
Since the conversions between integer, float, and decimal types are lossy (a value in one type may not be representable in the others), inserting implicit conversions led to subtle behaviors.

For example, the `1 < 3.14` comparison can be ported to LogicBlox 4 in several ways. The recommended and most straightforward translation is `1.0 < 3.14` where both literals are now decimals; this is equivalent to `1.0d < 3.14d` and `decimal:lt_2(1.0, 3.14)`. Alternatively, the literal `1` can be explicitly converted from integer to decimal: `int:decimal:convert[i]=d`.
If floats are still preferred, the `1 < 3.14` comparison can be ported as `1.0f < 3.14f` by making the integer `1` a float `1.0f` with the explicit `f` suffix, or as `int:float:convert[1] < 3.14f` with an explicit `int:float:convert` conversion from `int` to `float`.

- Before LogicBlox 4, converting from `float` to `string` would often result in loss of precision. Conversions between `float` and `string` round-trip correctly in LogicBlox 4.

Example 2.2

– In LogicBlox 3.x both 0.1 and 0.099999994 were converted to the string "0.1".
– In LogicBlox 4, `string:float:convert[float:string:convert[f]]=f`.

- `float:range` returns correct results, which did not always work in prior releases.

Example 2.3

In LogicBlox 3.x `float32:range(1.0,1.00001,1E-10,x)` went into an infinite loop, and `float32:range(-1000000,0,0.1,x)` produced 9296504 values.

- In LogicBlox 4, floating-point add `float:add[x,y]=z` and subtract `float:subtract[x,y]=z` operations are never rewritten unsafely. In LogicBlox 3.x, `float32:add[x,y]=z` could be incorrectly 'solved' to `float32:subtract[z,y]=x` when needed.

Example 2.4

There are many `x` such that `float:add[x,1.0]=1.0`; it is wrong to 'solve' this to `x=0.0`.

2.1.3 Negation

- The interaction of negation and equality/inequality has been changed slightly. The new behavior can be summarized as: "equality is symmetric, and ! means not".

Example 2.5

– In LogicBlox 3.x `(A[x] !=B[x])` meant something different than `(B[x] !=A[x])`, due to the way function applications (e.g. `A[x]`) were "lifted" outside of negation. In LogicBlox 4 they both have the same meaning as `(A[x]=u,B[x]=v,!(u=v))`.
– Another anomaly that could arise due to the "lifting" of function applications outside of negation was that, if `A[x]` existed and `B[x]` was missing, neither `(A[x]=B[x])` nor `!(A[x]=B[x])` would be true. Alternatively, in some situations, both `(A[x] < B[x])` and `!(A[x] < B[x])` could be simultaneously true. In LogicBlox 4, if `Foo` is a formula, then exactly one of `Foo` and `!Foo` is true.

The table below illustrates the difference in the treatment of negation and equality between LogicBlox 3.x and 4.

- It is now possible to use primitives in negation.

Scenario	LogicBlox 3.x facts	LogicBlox 4 facts
A[x] = α, B[x] = α	A[x]=B[x] !(A[x] != B[x])	A[x]=B[x] !(A[x] != B[x])
A[x] = α, B[x] = β, $\alpha \neq \beta$	!(A[x]=B[x]) A[x] != B[x]	!(A[x]=B[x]) A[x] != B[x]
A[x] = α, B[x] = missing	-	!(A[x]=B[x]) !(A[x]!=B[x])
A[x] = missing, B[x] = α	!(A[x] = B[x])	!(A[x]=B[x]) !(A[x]!=B[x])
A[x] = missing, B[x] = missing	-	!(A[x]=B[x]) !(A[x]!=B[x])

Table 1: Treatment of negation

Example 2.6

```
f(x) <- h(x), !(x/2=0).
```

2.1.4 Entity Creation and Retraction

- In LogicBlox 4 and beyond, programmers should no longer rely on an entity type's internal identifiers to be sequential, reflecting the order the entity values were created. These properties are accidental behavior of the LogicBlox 3.x runtime, and will not be observed by LogicBlox 4. This change was necessary to support parallelism.
- Entity values can only be created via constructors or refmodes. Any entity insertion without a constructor or a refmode will yield the compile-time error NONIDEMPOTENT_INSERTION.

Example 2.7

If unique entity-id needs to be generated without a constructor, it is possible to use the `transaction:id[]=id -> int(id)` predicate, which contains a unique id for each transaction; this can be used as a key value for a constructor, e.g.:

```
+A(x), +A:ctor[tid]=x <- transaction:id[]=tid.
```

To create, e.g., 10 new entities in a transaction, use a constructor with two integer keys and a rule like:

```
+A(x), +A:ctor[tid,n]=x <- transaction:id[]=tid, int:range(1,10,1,n).
```

- An auto-numbered refmode must have `int` as the value type; the refmode predicate itself is IDB regardless of the derivation type of the entity. Consequently, the values in the auto-numbered refmode at stage initial does not reflect entities added at stage initial. Note that since entity creation in 4 is always via constructors (see preceding bullet above), a separate constructor is required to create the entities themselves.
- Auto-retraction rules for entities and constructors have been improved in 4. Retracting a record from a constructor will destroy the entity-id; all occurrences of the entity-id will be removed from the workspace.

Example 2.8

If `hat` and `person` are entities, and `hat:by_person[p]=h` is a constructor, and `waterproof(h)` is a relation, then retracting `-person(p).` will automatically retract `hat:by_person[p]=h` and `waterproof(h)`.

- Type safety for entities has been improved.

Example 2.9

– In LogicBlox 4 if `person` is an entity and `adult(p) -> person(p).`, it is impossible to commit in a state where some `p` is in `adult` but not in `person`.

– In LogicBlox 3.x there were several ways to commit workspaces with violations of the declared types like this, e.g.: `+adult(p) <--person(p).`

- The compiler implements a ghost entity check in LogicBlox 4. A ghost entity is a entity-typed value that occurs in a predicate, but not in the predicate enumerating values of that entity type.

Example 2.10

In the following [1] is a ghost entity:

```
entity Band                 nameOf
----------------            --------------------------------
[0]                         [0], The National
[2]                         [1], The Mountain Goats
                            [2], Belle and Sebastian
```

The compiler looks for rules that may allow for the creation of ghost entities and issues an error for such rules. For details see the Reference Manual

2.1.5 Predicates

- LogicBlox 4 no longer requires the declaration of certain predicate properties regarding data storage and locking. The following predicate properties are no longer necessary, and will be removed in subsequent 4.x releases:
  - entity capacities
  - storage models
  - locking policies
  - scalable types
  - supplemental indexes
- Data import/export via delimited file predicates is supported, but with several differences compared to LogicBlox 3.x:

Note
For most users, the recommended CSV import/export facility is now Tabular Data Exchange services.

  - The declaration `lang:physical:storageModel[`p]="DelimitedFile"` is no longer required. Only `lang:physicalfilePath[`p]="filename.csv"` is needed.
  - The options `lang:physical:lineNumbers` and `lang:physical:hasColumnNames` are not yet supported.
  - Import file predicates now require an extra attribute: `_file(x,y,z)` is now `_file(offset;x,y,z)`.
  - Export file predicates now require a pragma: `lang:physical:fileMode[`_file] ="export"`.

- Predicates can be presented in sorted order via the `seq<<>>` (array syntax) or `list<<>>` (list syntax) predicate-to-predicate mappings. In a future version there will be a more elegant syntax for these, but for the time being the syntax of the examples below can be used.

Example 2.11

```
b(x,y) -> string(x), string(y).

// Array style: tuples (x,y) in sorted order
b:seq1(n;x,y) -> int(n), string(x), string(y).
b:seq1(n;x,y) <- seq<<[n]=(x,y)>> b(x,y).

// Array style: for each x, present yâ€™s in sorted order
b:seq2[x,n]=y -> int(n), string(x), string(y).
b:seq2[x,n]=y <- seq<<[x,n]=y>> b(x,y).

// List style: tuples (x,y) in sorted order
b:first1(x,y) -> int(x), int(y).
b:next1(x,y,u,v) -> int(x), int(y), int(u), int(v).
b:first1(x,y), b:next1(x,y,u,v) <- list<< >> b(x,y).

// List style: for each x, present yâ€™s in sorted order
b:first2(x,y) -> int(x), int(y).
b:next2(x,y,z) -> int(x), int(y), int(z).
b:first2(x,y), b:next2(x,y,z) <- list<< group-by(x) >> b(x,y).
```

- Ordered entities (via `lang:ordered`) is not currently directly supported in LogicBlox 4, but can be emulated using `list<<>>` or `seq<<>>`.

Example 2.12

Sorting by raw entity-id is disallowed, but an order can still be computed via entity constructor arguments:

```
a(_) -> .
b[x]=y -> string(x), a(y).
lang:constructor(`b).

b_1(x) <- b[x]=_.

b_1:first(x) -> string(x).
b_1:next(x,y) -> string(x), string(y).
b_1:first(x), b_1:next(x,y) <- list<<>> b_1(x).
```

```
a:first(x) <- b_1:first(y), b[y]=x.
a:next(x,y) <- b_1:next(u,v), b[u]=x, b[v]=y.
```

2.1.6 Transactions

A detailed overview of the overall transaction handling in LogicBlox 4 can be found in the Reference Manual

- Transactions are never aborted because of locking conflicts. Transactions will either succeed, or fail because they caused an integrity constraint violation.
- In previous releases if a transaction had multiple `exec` blocks, they were executed in sequence, and you could examine intermediate results after each block. In LogicBlox 4 stage `initial` is not evaluated until the **commit** command; the **exec** command merely adds the block to the set of stage-initial blocks.

Example 2.13

Consider the following example:

```
transaction
exec "+bar(0)."
print bar
exec "+bar(1)."
print bar
commit  // +bar(0), +bar(1) executed here
```

– In LogicBlox 3.x this would print `bar={0}` and then `bar={0,1}`.
– In LogicBlox 4 the `+bar(0).` and `+bar(1).` blocks are not executed until the **commit** command, so both print statements will show `bar` to be empty.

- Historically, the workspace could be committed successfully in a state where re-evaluating IDB rules (or running an empty transaction) would cause a constraint failure or changes to the workspace. In LogicBlox 4 this is impossible; if a commit succeeds, the workspace is guaranteed to be at a fixpoint with no constraint failures.
- Effects of the **lb import-protobuf** command will be applied to the workspace at stage `initial`. That means, using **print** immediately after the import will not show the imported data since it will be added to the workspace only when commit is called.

2.1.7 Semantics of Deltas

In previous releases the use of delta predicates did not always have a well-defined meaning. This has been clarified in LogicBlox 4. A detailed overview of the handling of updates in LogicBlox 4 can be found in the Reference Manual

2.1.7.1 Deltas of EDB predicates

In LogicBlox 4 deltas to EDB predicates generally capture requested changes.

Example 2.14

- `+foo@initial(x)` means facts derived into `+foo` in stage `initial`
- `+foo@final(x)` means facts derived into `+foo` in stage `final`.
- For a rule in stage `initial` `+foo(x)` means `+foo@initial(x).`
- For a rule in stage `final`, `+foo(x)` means `(+foo@initial(x);+foo@final(x)).`

2.1.7.2 Deltas to IDB predicates

- In LogicBlox 3.x, if `F[x]=y` was an IDB predicate, then `+F[x]=y` could contain records that did not reflect changes made to `F`.
- In LogicBlox 4, deltas for IDB predicates capture actual changes. `+F[x]=y` and `-F[x]=y` contain only facts inserted (resp. removed) to/from `F`. Deltas for IDB predicates can only be accessed in stage `final`.

2.1.7.3 Deltas to entity predicates and autonumbered refmodes

Deltas to entity predicates and autonumbered refmodes always capture actual changes, even if the predicate is EDB.

2.1.7.4 Retractions

It is no longer an error to request the retraction of a fact that is not there.

2.1.8 Aggregations

- In LogicBlox 4, `count` aggregations must have type `int`. In previous releases `count` aggregations could be defined with an arbitrary type, e.g., `int[32]`, which was a potential source of bugs.
- In LogicBlox 3.x floating-point `total()` aggregations could contain significant errors after maintenance. This has been resolved in LogicBlox 4.

Example 2.15

`A:total[]=tA[x]=yA[0]=1000000A[1]=0.01A[0]`

– In LogicBlox 3.x `print A:total` would show `A:total[]=0.0`.

– In LogicBlox 4 `print A:total` would show `A:total[]=0.01` as expected.

- The `top` and `bottom` aggregates are not currently supported. Instead, use `list` or `seq` as described earlier in the notes.

2.1.9 Recursion

Arbitrary use of recursion (including through negation or aggregation) is now allowed by the runtime. Warnings are reported for certain kinds of recursion which are safe but cannot be proved so by the current checks. In general, to enable this feature, the following needs to be added to the logic:

```
lang:compiler:disableError:AGGREGATE_RECURSION[]=true.
lang:compiler:disableWarning:AGGREGATE_RECURSION[]=true.
```

The computation for such programs procedes according to what we term a staged partial fixpoint semantics. For most examples, this should yield the intutively expected behavior. For instance, here is a program which computes the shortest paths in a directed graph using recursion through a `min` aggregate:

```
e(x,y,weight) -> int(x), int(y), int(weight).

path(x,y,d) <-
   e(x,y,d),
 ; e(x,z,d1), spath[z,y] = d2, d = d1+d2.

spath[x,y] = dist <-
   agg<<dist=min(d)>> path(x,y,d).

lang:compiler:disableError:AGGREGATE_RECURSION[]=true.
lang:compiler:disableWarning:AGGREGATE_RECURSION[]=true.
```

2.1.10 Hierarchical Import/Export

The hierarchical data import/export schema have changed; Re-running **proto2datalog** will regenerate fresh logic for protocol buffer messages. We advise users to simply include `.proto` files in project files, in which case **proto2datalog** will be invoked automatically.

2.2 Logic Optimization

In LogicBlox 3.x, programmers often apply hand optimizations to their logic in order to achieve better performance. These optimizations include various forms of creating alternative indices of predicates.

In LogicBlox 4, we **discourage** any hand optimization of logic. The query optimizer in LogicBlox 4 is showing strong promise. Indices are created on the fly, rendering hand-created indices unnecessary. In fact, any logic added by hand to optimize logic might have the reverse effect. Thus, existing applications might experience a performance improvement by removing these hand-applied optimizations.

2.3 Services Framework

- LogicBlox 4.0.5 introduced support for HTTP methods. Since CORS rules do not take HTTP methods into account and instead only operate on URLs, they can no longer be configured per service. Instead of specifying the name of the CORS rule in the service configuration, developers must now specify a prefix (or prefix and group) in the CORS rule itself. It is possible to wildcard prefixes to specify a group of services at once.

```
cors:rule_by_name["allow_foo_GET"] = r,
  cors:rule(r) {
    cors:allowed_origin("http://foo.com"),
    // prefixes
    cors:prefix("/foo-service"),
    cors:prefix("/bar/*"),
    cors:prefix_and_group("/baz","private-group")
  }.
```

- Modifications have also been made to the code organization the the ServiceBlox Java code. These changes only affect applications that use custom Java handlers. All service related classes (such as `ServiceMap`, `ServiceContext`, etc.) were moved from `com.logicblox.bloxweb` to `com.logicblox.bloxweb.service`. A few minor method signatures on `ServiceContext` and `ServiceMap` have also changed.

2.3.1 Measure Service

- *Aggregation methods are no longer enums:* In order to support named and composite aggregations, the `method` enum has been converted to a `message`. Existing uses of aggregation methods need to be rewritten to the new form.

```
"method": "TOTAL" ,
```

becomes

```
"method": { "primitive": "TOTAL" } ,
```

- *Types are no longer enums:* In order to ease expansion of types and eventually allow entity-typed metrics and attributes, `type` has been changed from being an `enum` to a `message`. Existing uses of `type` need to be rewritten to the new form.

```
"type": "STRING" ,
```

becomes

```
"type": { "kind": "STRING" } ,
```

- *Spreading kinds are no longer enum:* In order to support named and composite spreads, the `SpreadKind` enum has been converted to a `message`. Existing uses of spread kinds need to be rewritten to the new form.

```
"spread_kind": "RATIO" ,
```

becomes

```
"spread_kind": { "primitive": "RATIO" } ,
```

- *Ratio/even spreading semantics have changed* What was previously called "even" spreading is now called "delta". See what's new for more details.
- *Dimensions with more than one hierarchy are now required to have a default:* In previous releases, specifying a default hierarchy was optional. Now this is required if a dimension has more than one hierarchy.

Example of specifying a default hierarchy, when using the LogiQL measure configuration library:

```
+measure:config:dimension(dim) {
  +measure:config:dimension_hasName[]="MyDim",
  +measure:config:dimension_hasDefaultHierarchy[]="MyHierarchy"
}.
```

2.4 Developer Tools

2.4.1 Renaming

- To lower cognitive load on the developers, we have applied a consistent set of commands to all interactions with the LogicBlox database. Additionally, developers will benefit from better help messages and tab completion. All LogicBlox commands now start with `lb`, followed by the subcommand(s).

  A summary of the changes in the command names can be found in the table below. For a complete overview of all the LogicBlox commands, please refer to the Reference Manual.

Old Command	Replaced By
bloxbatch	lb
bloxcompiler	lb compile
connectblox-server	lb server
BloxPagerDaemon	lb pager
bloxweb	lb web-server
bloxweb-client	lb web-client
lb-services	lb services
lb-config	lb config
lbunit and bloxunit	lb unit

- Configuration and log files have been renamed, to reflect the changes in the names of our components. The tables below summarize the changes.

Config Files	
bloxcompiler.config	lb-compiler.config
bloxweb.config	lb-web-server.config
bloxweb-client.config	lb-web-client.config

Log Files	
bloxcompiler.log	lb-compiler.log

Log Files	
bloxweb.log	lb-web-server.log
bloxweb-server-access_*.log	lb-web-server-access_*.log
connectblox-server.log	lb-server.log
BloxPagerDaemon.log	lb-pager.log

- The environment variables below have been renamed:

Old Name	Replaced By
BLOXWEB_HOME	LB_WEBSERVER_HOME
BLOXWEB_MEASURE_SERVICE_HOME	LB_MEASURE_SERVICE_HOME

2.4.2 Enhancements

- The LogicBlox 4 release introduces the following enhancements to the **lb** command:
  - The new command **lb status** can be used to query whether the server is alive and to obtain information about workspaces. It also supports listing requests that are currently being processed by the server: When `--active` or `--all` are given, then the currently active (active and queued, respectively) requests are printed as a string-representation of the Request protobuf message.

Example 2.16

```
$ lb status foo bar baz ws
lb server is ON
---
Workspace 'foo' is processing 2 request(s).
Workspace 'baz' is idle.
Workspace 'ws' is closed.
Workspace 'bar' does not exist.
```

  - The new command **lb export-protobuf-schema** can be used to retrieve protocol buffer descriptors. The command also supports flags `-i` (for include) and `-e` (for exclude) that can be used to specify regular expressions for including and excluding descriptors to be output.
  - It is now possible to give predicate names as parameters to `--print`. This allows the printing of selected predicates' contents at the end of a transaction. This also works for local and pulse predicates.
  - The output of the **lb popcount** command is improved and the command is extended with the flags `--include` and `--exclude` that can be used to specify regular expressions for including and excluding predicates that are part of the popcount census.
  - The **lb query** command can be used to evaluate rules after stage `final`. Changes to the database by **lb query** blocks are discarded. This is similar to the `after-fixpoint` feature of ConnectBlox. It can, for example, be used to check whether a constraint holds at the end of a transaction. If the constraint does not hold, the transaction is not committed.

- By default, transactions are soft-committed; a field inside the Transaction protocol buffer message can be used to request a hard commit. During soft commits, a response is sent to the client as soon as the transaction has been committed to memory. Disk commit will occur immediately afterwards. With hard commits, the response is not sent until the transaction has been committed to disk.

3 Migration of Numerics

To aid in the migration of the changes in the numerics, we provide the following *sed rewrite scripts*. Each rewrite has the form `/oldcode/newcode/` where `oldcode` is a regular expression for old types, paths, or literals; `newcode` is the updated type or path without bit widths, or the updated literal with a disambiguating suffix. Several scripts can be applied to a file `oldfile.logic` to produce an updated file `newfile.logic` by running:

```
sed -e "s/oldcode1/newcode1/g" -e "s/oldcode2/newcode2/g" ... oldfile.logic ↩
    > newfile.logic
```

Important
These scripts have been tested to port code to LogicBlox 4 and there are several known caveats. The rewrites do not insert explicit type conversions between numerics, and they will replace text in comments and string literals which may be unintended. Users are advised to apply the rewrites and verify that there are no unintended changes between `oldfile.logic` and `newfile.logic`, then try recompiling the latter to determine where type conversions may be needed.

Integer types (e.g., `int[64]`) and standard library paths (e.g., `int64`) can be replaced by `int` with:

```
/\([^a-z]\)\(u\|\)int\[\(8\|16\|32\|64\)\]/\1int/
/\([^a-z]\)\(u\|\)int\(8\|16\|32\|64\)/\1int/
```

Float types (e.g., `float[64]`) and standard library paths (e.g., `float64`) can be replaced by `float`, and float literals can be replaced with a suffix `f`.

Note
Please remember that it may be preferable to change floats to decimals for accuracy, efficiency, and safety.

```
/\([^a-z]\)float\[\(32\|64\)\]/\1float/
/\([^a-z]\)float\(32\|64\)/\1float/
/\([0-9]+\(\.[0-9]+[eE][\+-]?[0-9]+\|\.[0-9]+\|[eE][\+-]?[0-9]+\)\)\([^0-9 ↩
    eEf\"]\)/\1f\3/
```

Decimal types (e.g., `decimal[64]`) and standard library paths (e.g., `decimal64`) can be replaced by `decimal` with:

```
/\([^a-z]\)decimal\[\(64\|128\)\]/\1decimal/
/\([^a-z]\)decimal\(64\|128\)/\1decimal/
```

Replacing types and paths from **float to decimal** can be done as follows (but shouldnâ€™t be combined with the above float-to-float rewrite):

```
/\([^a-z]\)float\[\(32\|64\)\]/\1decimal/
/\([^a-z]\)float\(32\|64\)/\1decimal/
```

4 Known Limitations and Discontinued Features

4.1 Not Supported but Planned

The following features are not supported, but are planned for subsequent releases:

- Mathematical programming and machine learning extensions.
- Default values predicates.
- Ordered entities.
- Replace and remove block.
- `choice` and `ambig` predicate-to-predicate mappings.
- String concatenation aggregations.
- `argmin` and `argmax`. To get the argument, an additional join can be used.
- There is limited support for user-level logging. More detailed logging levels will be supported in subsequent releases.
- Predicates can have arity at most 64. Post-4.0 versions will remove this limit.
- A few primitive predicates are missing. These include: `boolean:hash`, `int:hash`, `float:hash`, `datetime:hash`, `float:isFinite`, and `float:round2`.
- We do not detect whether transactions that are marked as read-only are indeed read-only. Possible changes by transactions marked as read-only are most of the times not committed into the database; however, once in a while we do commit read-only marked transactions (so that we can make subsequent use of indices created in the transaction). Until we verify that read-only marked transactions do not change the database, it is the responsibility of the user to do so.
- For functional edb predicates, retractions are performed based on key match only, rather than on key-value match. For example, `-a["foo"]="bar"` removes `a["foo"]="quz"`; i.e., it behaves like `-a["foo"]=_`. This will be corrected in a future release.

4.2 Permanently Removed

The following features are removed, and will not be supported in subsequent releases:

- Signed and unsigned integer types (`uint[8]`, `uint[16]`, `uint[32]`, `uint[64]`, `int[8]`, `int[16]`, `int[32]`, `int[64]`) are replaced by a single signed integer `int` type equivalent to the old `int[64]` type.
- Binary floating-point types (`float[32]`, `float[64]`) are replaced by a single floating-point `float` type equivalent to the old `int[64]` type.
- Floating-point decimal primitive types (`decimal[64]` and `decimal[128]`) are replaced by a single fixed-precision `decimal` type.

- The update operator ('*') is no longer supported.
- Primitive type `color` is not supported.
- Meta-predicates such as `system:Predicate` are not supported. Future versions of meta-predicates will likely differ substantially from how meta predicates are handled in 3.X.
- MoReBlox. While continuing to be supported in the LogicBlox 3.x series, support for generic programming in future releases is undergoing a redesign process and will be different enough that existing MoReBlox programs will require a rewrite.
- MochaBlox and WrappedControls. While continuing to be supported in the LogicBlox 3.x series, we encourage application programmers to choose the user interface framework of their choice, and communicate with the workspace over the service interface.
- **bloxbatch** and **bloxunit** are no longer supported. The **lb** and **lb unit** commands should be used instead.
- Measure service related features:
  - *Metadata service is no longer supported:* The previously deprecated `ModelRequest` and `ModelResponse` messages have been removed, and the `MetadataService` no longer exists. The model may be retrieved via the `MODEL_QUERY` kind of the `Request` message.
  - *Filter comparisons only use expressions instead of terms or expressions:* The previously deprecated `term` field has been removed from the `Comparison` message. Filtering is now done exclusively using the `expr` field of the `Comparison` message. Because `Terms` are a subset of `MeasureExprs`, this only requires a minimal change in queries.

For example, the comparison

```
"comparison": [ {
     "op": "EQUALS",
     "term": {
       "kind": "CONSTANT",
         "constant": { "string_constant": "Atlanta, GA" }
      }
 } ]
```

would be rewritten to

```
"comparison": [ {
     "op": "EQUALS",
     "expr": {
       "kind": "TERM",
       "term": {
         "kind": "CONSTANT",
           "constant": { "string_constant": "Atlanta, GA" }
        }
     }
} ]
```

  - *Legacy measure expression syntax is no longer supported:* The previously deprecated field `measure_text` has been removed from the `Binding` and `InstallRequest` messages.