

# DataRoller

*“Roll your own data”*

Version 1.0

User Guide

May 2012

<http://dataroller.sourceforge.net/>

*In a nutshell ...*

---

DataRoller is a Java application that will insert piles of good looking data into your favorite databases so you can move in and be happy.

Rich Alberth

# Table of Contents

1	<a href="#">For The Impatient</a>	6
2	<a href="#">Overview</a>	8
3	<a href="#">Project Files</a>	9
3.1	<a href="#">Comments</a>	11
3.2	<a href="#">Labels</a>	11
3.3	<a href="#">Tables</a>	11
3.4	<a href="#">Deleting Old Rows</a>	12
3.5	<a href="#">Parent Child Relationships</a>	14
3.5.1	<a href="#">Where Clause</a>	15
3.6	<a href="#">Unique and Unique Per Parent</a>	16
3.6.1	<a href="#">Data Exhaustion</a>	18
3.6.2	<a href="#">Example</a>	19
3.6.3	<a href="#">Independently Unique</a>	20
3.6.4	<a href="#">Sequences</a>	20
3.7	<a href="#">Columns</a>	21
3.8	<a href="#">Data Types</a>	23
3.8.1	<a href="#">Strings</a>	23
3.8.2	<a href="#">Integers</a>	24
3.8.3	<a href="#">Decimals</a>	25
3.8.4	<a href="#">Floats</a>	26
3.8.5	<a href="#">Dates and Times</a>	26
3.8.6	<a href="#">Booleans</a>	27
3.8.7	<a href="#">Nulls</a>	27
3.8.8	<a href="#">Unsupported Data Types</a>	27
3.8.9	<a href="#">Null Expressions</a>	28
3.9	<a href="#">Variables</a>	28
3.9.1	<a href="#">Assignment</a>	28
3.9.2	<a href="#">Variables and randomrow()</a>	30
3.10	<a href="#">Embedded SQL</a>	31
3.10.1	<a href="#">Embed SQL in DataRoller File</a>	32
3.10.2	<a href="#">Reference External SQL File</a>	33
3.11	<a href="#">A Note on File and Folder Names</a>	34
3.12	<a href="#">Lookup Tables</a>	35

4	<a href="#">Generators</a>	37
4.1	<a href="#">Random Data</a>	38
4.1.1	<a href="#">Random Integers</a>	38
4.1.2	<a href="#">Random Decimals</a>	39
4.1.3	<a href="#">Random Floats</a>	39
4.1.4	<a href="#">Random Dates and Timestamps</a>	39
4.1.5	<a href="#">Choice</a>	40
4.1.6	<a href="#">BLOB</a>	41
4.2	<a href="#">Structured Data</a>	42
4.2.1	<a href="#">Lorem Ipsum</a>	42
4.2.2	<a href="#">Sequences</a>	43
4.3	<a href="#">Column</a>	45
4.3.1	<a href="#">Current-Row Reference</a>	47
4.3.2	<a href="#">Separate Table Reference</a>	47
4.3.3	<a href="#">Parent Table Reference</a>	48
4.4	<a href="#">Lookup Data</a>	50
4.4.1	<a href="#">File Row Lookup</a>	50
4.4.2	<a href="#">Folder Contents Lookup</a>	50
4.4.3	<a href="#">XML File Lookup</a>	51
4.4.4	<a href="#">randomrow()</a>	53
4.4.5	<a href="#">Previous Row</a>	54
4.5	<a href="#">SQL</a>	57
4.6	<a href="#">Operators</a>	58
4.6.1	<a href="#">Types, Nulls and No Short-Circuit Logic</a>	60
4.6.2	<a href="#">Boolean Operators</a>	60
4.6.3	<a href="#">Equality Operators</a>	61
4.6.4	<a href="#">Comparison Operators</a>	62
4.6.5	<a href="#">Algebraic Operators</a>	63
4.6.6	<a href="#">String Operators</a>	64
4.6.7	<a href="#">Dates</a>	65
4.6.8	<a href="#">Operator Summary Table</a>	67
4.7	<a href="#">Conditionals</a>	67
4.7.1	<a href="#">If then else</a>	67
4.7.2	<a href="#">Case When</a>	68
4.7.3	<a href="#">Conditionals Example</a>	69
4.8	<a href="#">Raw()</a>	69
5	<a href="#">Functions</a>	71
5.1	<a href="#">String Functions</a>	72

5.1.1	<a href="#">Function pattern()</a>	74
5.1.2	<a href="#">Function guid()</a>	75
5.2	<a href="#">Numeric Functions</a>	76
5.2.1	<a href="#">Integral Functions</a>	76
5.2.2	<a href="#">Floating-point Functions</a>	76
5.3	<a href="#">Date and Timestamp Functions</a>	77
5.4	<a href="#">System Functions</a>	78
5.5	<a href="#">Cryptographic Functions</a>	79
5.6	<a href="#">Data-Type Conversion Functions</a>	79
6	<a href="#">Execution</a>	82
6.1	<a href="#">Command-line Switches</a>	82
6.2	<a href="#">User Preferences and Aliases</a>	84
6.3	<a href="#">Loading JDBC Drivers</a>	85
6.3.1	<a href="#">SQL Server</a>	86
6.3.2	<a href="#">Oracle</a>	86
6.3.3	<a href="#">DB2</a>	86
7	<a href="#">Speeding up DataRoller</a>	87
7.1	<a href="#">DataRoller Generator Relative Costs</a>	87
7.2	<a href="#">Rebuild Indexes</a>	89
7.3	<a href="#">Regenerate Statistics</a>	90
7.4	<a href="#">Lock Tables</a>	92
7.5	<a href="#">Disable Costly Constraints</a>	92
8	<a href="#">Extending DataRoller</a>	93
8.1	<a href="#">User-Supplied Functions</a>	93
8.2	<a href="#">Java Function</a>	94
8.3	<a href="#">Arguments and Return Types</a>	94
8.4	<a href="#">Function Alias</a>	95
8.5	<a href="#">Function Signature</a>	95
8.6	<a href="#">Invocation and Execution</a>	97
8.7	<a href="#">For Example</a>	97
9	<a href="#">Tips &amp; Tricks</a>	100
9.1	<a href="#">Using Delete for Row Partitioning</a>	100
9.2	<a href="#">Dealing with Artificial Primary Keys</a>	101
9.2.1	<a href="#">MySQL</a>	102
9.2.2	<a href="#">SQL Server</a>	103
9.2.3	<a href="#">Oracle Sequences</a>	103

9.3	<a href="#">Avoid Querying Unneeded Data</a>	104
9.4	<a href="#">Mutually Unique randomrow()</a>	105
10	<a href="#">For Reference</a>	108
10.1	<a href="#">Project Syntax Reference</a>	108
10.2	<a href="#">DataRoller Keywords</a>	110
10.3	<a href="#">Syntax Reference</a>	111
10.3.1	<a href="#">Lexical Elements</a>	111
10.3.2	<a href="#">Grammar</a>	111
10.4	<a href="#">JDBC URL Reference</a>	114
10.5	<a href="#">DataRoller License and Included Works</a>	116

# 1 For The Impatient

DataRoller inserts rows into a database based on a file describing your tables and columns.

Execute this against MySQL:

```
create database dr;
use dr;

create table invoices (
  invoice_id int,
  alt_key_uid varchar(50),
  name varchar(80),
  label varchar(30),
  status char(3)
);
```

Put this in a file called basic.txt:

```
table invoices
  insert 4 rows
{
  invoice_id sequence(),
  alt_key_uid guid(),
  name lorem(20..80),
  label pattern("UU-NNNN"),
  status choice("UNK","SHP","CLS",
              "OPN","INP")
}
```

This file defines how DataRoller should go about populating your tables and columns, such as how many rows to insert, and what value to put into each column. `sequence()` means 1, then 2, then 3, etc. Read <http://lipsum.com> for details on `lorem()`. `Pattern()` generates a string based on the codes you pass in. “U” for an upper-case letter, “N” for a number (0-9 digit). `choice()` just picks one of the values supplied at random.

Download and unzip DataRoller, open a command-window and type this:

```
dataroller -c jdbc:mysql://localhost/DRTEST -u ROOT -p mypasswd basic.txt
```

You’ll see this if you have everything installed and running correctly:

```
DataRoller 1.0
This program comes with ABSOLUTELY NO WARRANTY, is free software, and you
are welcome to redistribute it under certain conditions. See License.txt
for details. Copyright 2012 Rich Alberth.

Invoices [.....] 0s
Done (0s)
```

Run “select \* from invoices”:

```
+-----+-----+-----+-----+
| invoice_id | alt_key_uid | name | label | status |
+-----+-----+-----+-----+
| 1 | 03153499-f392-46a8-aa69-f9ff225971 | Lacus torq | XN-5814 | INP |
| 2 | afbc0751-ed8b-4773-9678-4bbf84aef7 | Dui maecen | RK-2669 | INP |
| 4 | 2686521e-cc37-4471-99ec-a2625c6d13 | Lobortis f | ZR-3740 | SHP |
| 5 | b8a73acc-a376-441e-bff2-9f8567f52d | Amet in no | DC-1536 | OPN |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

That's it! You write a text file with instructions on what should go into each table and column and execute it via a command-line tool. The input text file supports variables, complicated table relationships, user-supplied functions and much more.

## 2 Overview

---

DataRoller (motto “*Roll your own data!*”) is a Java application that will insert piles of good looking data into your favorite databases so you can move in and be happy.

Here are some good ideas:

***Load up thousands of rows of data, fire up your app and see what areas need tuning.***

Databases execute your SQL SELECT statements differently based on how much data is in each table. Real data means real tuning.

***Generate a bunch of values that look just like your production database for your next demo.***

Ever say this to a crowd of people “sorry this isn’t how the app will look in production, it’s just test data”?

***Throw some stand-in data together quickly for your testing team.***

Testers can’t find subtle bugs when they test a screen with only a single record they generated.

***Clear out all the rows in your database and reset it with some basic starting data.***

After a demo, how do you “reset” your box back to a developer mode?

***Rapid prototyping and requirements gathering.***

Fill up a new table with good-looking sample data, use a rapid prototyping tool to generate a skeleton UI and demo it to the users during a requirements meeting.

## 3 Project Files

---

A project file is the main input to DataRoller. This file documents every table that needs populating, what to put in each column, and how tables relate to each other. The syntax feels a little like SQL “create table” statements: each table is listed in the file in the order in which they should be populated. Within each table section, each column is listed with how data should be generated for it.

A guided tour through your first Project File:

```
table invoices
```

```
    insert 2000 rows
```

```
{
```

```
    invoice_id  sequence(),
```

```
    name        10rem(20..80),
```

```
    status      choice("UNK", "SHP", "CLS"),
```

```
    delivery    random(D'2009-4-1' ..  
                      D'today')
```

*List each table you want populated, in the order you want them populated in.*

*Number of rows to insert (duh). There are a lot of things you can add later in this section between the table name and the “{” for column definitions!*

*Braces group columns per table.*

*“sequence()” means start at 1, increment each time a new row is inserted.*

*Generate Lorem Ipsum text between 20 and 50 characters in length.*

*Pick one of these supplied values at random.*

*Pick a random date between April 1<sup>st</sup> 2009*

```
step /hour/),
```

*and today, with a time component rounded to the nearest hour (minutes and seconds are always zero).*

```
last_name filerow("names.txt"),
```

*“filerow()” means pick a random line from the text file listed (stripping line termination chars).*

```
ship_code pattern("UU-NNNN"),
```

*Generate a string based on the pattern where “U” is an upper-case letter and “N” is a number (digit).*

```
} ship_label folder("labels.zip")
```

*Pick a random file from the given folder, read it in, and use the contents of the file as the value for this column (good for “blob” and “text” columns).*

```
table invoice_items  
  child of invoices on  
    this.inv_id = parent.invoice_id  
  insert 2..15 rows  
{
```

*Child table: generate between 2 and 15 rows in invoice\_items **for every row** generated in table invoices. DataRoller makes the foreign keys work out the right way. There are no invoices with no invoice\_items, and no invoices with thousands of items by accident!*

```
  item_id sequence(10000),
```

*Start these values way above invoice\_id values just to keep them apart.*

```
  product column(products.product_id),
```

*Look-up table: pick a random value from table products, column product\_id.*

```
  quantity random(1..100),
```

*Pick a random integer between 1 and 100.*

```
  price random(1.00 .. 100.25  
             step 0.25)
```

*Pick a random decimal (not a real/float/double, this is specific precision)*

```
}
```

*between 1 and 100.25, divisible by 0.25.  
I.e., the price always ends in “.00”, “.25”,  
“.5” or “.75” (quarters).*

The rest of this chapter covers the language elements in a project file.

## 3.1 Comments

Comments: two slashes (“//”), two dashes (“--”), or C-style /\*...\*/

```
table mytbl // really an insertable view
  delete all
  insert 0 rows /* just clear it out */ {
  mycol sequence() -- no args to sequence() means start at 1
}
```

## 3.2 Labels

Labels are used to name tables and columns. Labels are:

- Any combination of letters and numbers, underscores (“\_”), hash marks (“#”) or dollar signs (“\$”) that are not a DataRoller keyword, or
- Any combination of characters enclosed in square brackets (except a right square bracket itself (“]”), a newline, carriage return or form feed character).

Note that there is no rule in DataRoller that a label starts with an alphabetic character. Both “12345g” and “\_\_#mytmp6” are valid labels. This is useful for temporary tables that start with “#”.

## 3.3 Tables

A DataRoller project file consists of a sequence of table definitions, with some extra, optional declarations at the top of the file.

A Table definition is made up of:

1. “table” and the name of the table to insert into. A database table can be listed more than once in a DataRoller project file.

2. Table prologue: a set of table-level things, such as:
  - a. Whether to delete everything before inserting new rows
  - b. How many rows to insert
  - c. Whether this is a child table of another definition in the project file.
3. Column definitions, each having:
  - a. Name of the column to insert into, and
  - b. An expression telling DataRoller how to construct values to insert.

Separate Column Definitions in a table with a comma.

The order that table definitions appear in the file is the order in which they are processed and inserted. If you list table products before invoices, and invoices references products, you're OK: you'll have data in products before you try to insert rows into invoices.

Anatomy of a simple table definition:

```
table invoices

    delete all

    insert 1000 rows

{
    pk sequence()
}
```

*Optionally include the keywords “delete all” to have DataRoller remove all existing rows before inserting new ones.*

*Number of rows to insert, as a fixed amount, or a range of integers. Zero is valid.*

*List all column definitions within curly braces. You do not need to use a comma to separate columns as in SQL “create table” statements.*

## 3.4 Deleting Old Rows

“delete all” in the example above is the simplest way to clear out old data before having DataRoller insert fancy new data. This removes all records from the target table.

DataRoller is meant to be run with only the most basic access to the target tables it operates over. The SQL “truncate” command is much faster than deleting rows from a table, but sometimes requires more

rights to the table than “delete”. DataRoller is designed to require no special permissions on the database other than basic select, insert, and delete.

DataRoller is sensitive to the relationships between tables. You would like table vendors earlier in the DataRoller Project file than products if products had a foreign key to vendors: DataRoller should populate the vendors table first, and then populate the products table.

DataRoller does not issue delete statements following this same logic: when executing a DataRoller Project file, the DataRoller Engine work in two passes:

- **Pass 1:** Start at the end of the file and work backwards, issuing a delete statement for every table that specifies “delete”.
- **Pass 2:** Start at the beginning of the file and work forwards, generating insert statements.

This way, if you have foreign keys referencing tables, you will see the following:

1. Delete from dependent tables
2. Delete from parent tables
3. Insert into parent tables
4. Insert into dependent tables

This processing is not obvious from the syntax of the DataRoller Project file. It was decided to put the “delete” clause inside the table definition so all facts about a table are defined together in a single location in the file. This avoids a lot of scrolling around while writing a DataRoller Project file. The DataRoller Project file is *declarative*: you specify what you want the results to be, and DataRoller will make it happen.

The delete statement has two forms:

1. “delete all”, which issues a plain “delete from \_\_\_” SQL statement
2. “delete where” followed by a string literal, which issues the same SQL as above with the string literal appended as a “where” clause.

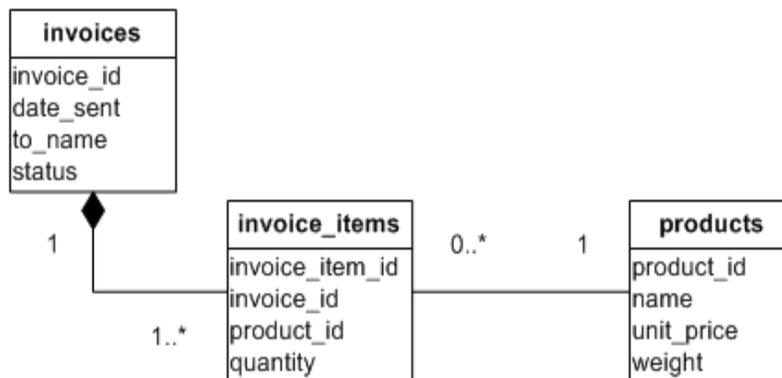
```
table foo
  delete where “bid < 7000”
{
  ...
}
```

This example will issue the SQL statement “delete from foo where bid < 7000”. Note that the argument to the “delete where” clause is a string, and not arbitrary SQL. The entire example above is not actual SQL: “delete where” are two DataRoller keywords chosen to look a lot like SQL as a memory aide. The thing after “where” must be a DataRoller string literal. DataRoller assembles the SQL delete statement based on these components.

Because a simple string literal is allowed, you cannot use string concatenation (“+”), call functions, or use generators or any other facility in DataRoller.

## 3.5 Parent Child Relationships

In database parlance, there is only the single “foreign key” concept to handle all design patterns. Database designers use higher-level concepts to capture relationships.



Examples:

- **Reference**: Plain foreign key to another table, such as `invoice_items.product_id` pointing to the `products` table, column `product_id`. This is the “standard” join that database vendors implement directly.
- **Composition**: A design concept not directly implemented by databases. In the example above, each Invoice is composed of one or more Invoice Items. The only way a database can approximate this is to provide a foreign key from `invoice_items` to `invoices`, but this is incomplete. There is no direct database structure to capture the fact that no Invoice can exist without at least one Invoice Item.

DataRoller treats these two types of association differently within a project file. References are handled on a per-column basis, since they just capture picking a random value from the lookup table and using it in the current row being generated.

Composition is represented in a DataRoller Project file with a clause at the table level to show how the parent and child tables relate.

Anatomy of a parent child relationship:

```
child of table on this.fk = parent.pk
```

“table” is the name of the parent table, “fk” is the column in the child table with the references constraint on it (the foreign key), and “pk” is the primary key column in the parent table. Note that “this” and “parent” are keywords in DataRoller and are included to make the line more readable, mimicking the syntax of a typical SQL JOIN clause.

Example of a child table definition:

```

table invoice_items
  delete all
  child of invoices on this.invoice_id = parent.invoice_id
  insert 1..30 rows
{
  ...
}

```

When specifying a child table in a parent-child relationship, you can specify a range of rows to generate (“insert 1..30 rows” in sample above). The range (or single value) is used by DataRoller to decide how many child rows to insert per parent row. In the above example, if the invoices table section has “insert 10 rows”, then the invoice\_items table will end up with between 10 and 300 rows total (1×10 .. 30×10).

### 3.5.1 Where Clause

The “child” clause also supports passing a SQL “where” clause to pick what set of rows should be affected. In the above example, all rows from invoice\_items are removed, and then every row from invoices receives between 1 and 30 rows in invoice\_items. This is a comprehensive action: everything removed, all parent rows used.

Using a “where” clause on the “child” portion will affect which rows from the parent table are used to attach child rows. Consider a database with a table of hospital patients, and a child table holding a record for all medications currently prescribed. Patients that are active (in the hospital) have between zero and 6 medications. Patients that have been discharged always have zero. The database should never have a discharged patient with meds:

```

table patients
  insert 800 rows
{
  patient_id      sequence(),
  first_name      filerow("classpath:firstnames.txt"),
  last_name       filerow("classpath:lastnames.txt"),
  status          choice("ACTIVE", "DISCHARGED")
}

table prescriptions
  child of patients on this.patient_id = parent.patient_id
                    where "status = 'ACTIVE'"
  insert 0..6 rows
{
  rx_code         pattern("UU-NN-UUUUU-NN")
}

```

Internally, DataRoller simply issues “select patient\_id from patients prnt” when starting to populate the prescriptions table since it has a “child” clause. It loops over these records and inserts between 0 and 6 rows per patient\_id returned. When a “where” clause is included like above, it appends it to the query.

In the above example, DataRoller executes “select patient\_id from patients prnt where status = ‘ACTIVE’”, and then generates between 0 and 6 rows for every returned patient\_id.

The “where” clause text is any arbitrary clause that can be put into a select statement. The “from” that is in effect is the parent table with table alias “prnt”. Check out the previous paragraph – it’s used there. The table alias “prnt” is included for “where” clauses that include sub-selected or otherwise needs access to the parent table being queried. For example, the following table only adds maintenance records for airplanes with over 10,000 scheduled kilometers.

```

table airplanes
{
  . . .
  plane_id . . .
}

table maintenance
  child of airplanes
  on this.plane_id = parent.plane_id
  where "10000" < (select sum(distance)
                  from routes r
                  where r.plane_id = prnt.plane_id)"

  insert 1 row
{
  . . .
}

```

### 3.6 Unique and Unique Per Parent

Random data by itself will serve for basic testing and simple scripts, but is insufficient for demonstration data and portions of the database that will cause application logic to fail with bad data. For example:

- A user screen listing different part numbers from a supply table with the same description
- An algorithm that calculates distances between addresses may divide by zero if it picks two records from the address table with the same data

Clauses “unique” and “unique per parent” can be added to certain generators below to prevent DataRoller from generating the same value more than once. For tables that have a “child of” clause, “unique” will never generate the same value more than once *across every row* in the table (regardless of parent row). “unique per parent” will never generate the same value for all rows in the child table *for a particular parent value*. “unique per parent” may generate the same value for more than one row, but only if such rows have different parent rows.

	Table without “child of”	Table with “child of”
random(1..10)	May generate duplicates	
random(1..10 unique)	No duplicates at all	
random(1..10 unique per parent)	Error (“per parent” not allowed without “child of”)	No duplicates within each group of rows with same parent row.

All Generators that support uniqueness (see below for “implicit” discussion):

Generator	Supports “unique”	Supports “unique per parent”
<code>random(integers)</code>	Yes	Yes
<code>random(decimals)</code>	No	No
<code>random(floating-point)</code>	No	No
<code>random(dates)</code>	Yes	Yes
<code>sequence()</code> <i>(all types)</i>	implicit	Yes
<code>column(mytbl.mycol)</code>	Yes	Yes
<code>mytbl.mycol</code>	No: must use <code>column()</code> syntax to use unique clauses	
<code>column(parent.mycol)</code>	No: refers to parent row of a table with “child of” clause.	
<code>column(mycol)</code> <code>mycol</code>	No: refers to column in the current row of the current table.	
<code>folder()</code>	Yes	Yes
<code>filerow()</code>	Yes	Yes
<code>xpath()</code>	Yes	Yes
<code>randomrow()</code>	Yes	Yes

### 3.6.1 Data Exhaustion

Be careful when using “unique” and “unique per parent”: if DataRoller does not have enough values to supply to the number of rows requested, DataRoller will issue an error message and stop.

```
table foo
  insert 1..10 rows
{
  . . .
}
table bar
  child of foo on this.id = parent.id
  insert 5..7 rows
{
  code    random(1..10)
}
```

There will never be a problem with column `bar.code`: since there is nothing unique about it, the 10 possible values may be used over and over again.

```
table foo
  insert 1..10 rows
{
  . . .
}
table bar
  child of foo on this.id = parent.id
  insert 5..7 rows
{
  code    random(1..10 unique)
}
```

Table `foo` will only have between one and ten rows total.

Table `bar` will have between five and seven rows *per row* in `foo`:

$$1 \times 5 - 10 \times 7$$
$$5 - 70$$

Table `bar` will contain between five and seventy rows! The `random()` generator does not have enough possible values to serve more than 10 rows.

```
table foo
  insert 1..10 rows
{
  . . .
}
table bar
  child of foo on this.id = parent.id
  insert 5..7 rows
{
  code    random(1..10 unique per parent)
}
```

Table `bar` may have up to seventy rows, but they are grouped into groups of five to seven. The ten available values in `random()` are more than enough to handle every group. DataRoller will not run out of possible values since it can reuse values across rows in `foo`.

## 3.6.2 Example

These rules apply for all generators below that support “unique” and “unique per parent”. For example, a company that leases large construction equipment has several repair Teams that fix broken equipment in the field. Each Team has a mix of labor categories on them. A Team does not have more than one person with the same labor category. Each Team is assigned a single Truck to haul their equipment. Obviously, the same Truck cannot be assigned to more than one Team.

```
table trucks
  insert 15 rows
{
  truck_id  sequence(1)
  . . .
}

table teams      // no need to make this a child of trucks:
                // some trucks are just not used
  insert 10 rows
{
  team_id    sequence(),
  truck_id   column(trucks.truck_id unique)
}

table team_members
  child of teams on this.team_id = parent.team_id
  insert 1..5 rows
{
  employee_id  column(employees.emp_id unique),
  labor_category  filerow("categories.txt" unique per parent)
}
```

- Every team needs a unique truck: no need for “unique per parent” because teams do not need to be a child of trucks just to link them together somehow. “unique” on trucks.truck\_id assures that no record in teams has the same truck\_id value.
- Each team\_members row represents a person assigned to a truck with their particular skill set. A single employee cannot work on two trucks, so employees.emp\_id is marked “unique” and not “unique per parent” so that every value for employee\_id in table team\_members is unique regardless of the truck they work on. If employee\_id was “unique per parent”, a single employee could work on more than one truck but never more than once on the same truck.
- Each team\_members row represents an assignment of an employee to a team. The labor\_category is the capacity they work under. The business does not staff a team with more than one employee with the same labor category (“welder”, “electrician”, “mechanic”), but any particular labor\_category may be on more than one team (trucks 5 and 7 both have mechanics on them). Column labor\_category is a row from file “categories.txt” such that no row appears more than once per team. If this filerow() was “unique” instead of “unique per parent”, every row from categories.txt could be assigned to only a single team (parent row) instead of being reused from team to team.

### 3.6.3 Independently Unique

Clauses “unique” and “unique per parent” apply only to the generator on which they are applied, even if two such generators appear on the same column or multiple columns on the same table. For example:

```
table code_parts
  insert 100 rows
{
  part      filerow("parts.txt" unique)
}
table codes
  insert 50 rows
{
  code      column(code_parts.part unique) + "-" +
            column(code_parts.part unique)
}
```

The generator “column(code\_parts.part unique)” appears twice in table codes, both for column code. The fact that they are both there and both use “unique” does not mean that they are *collectively unique*: if code\_parts contains “ABC” for column part, by chance DataRoller may generate “ABC-ABC” as a valid value for column code. All generators with “unique” are *independently unique*: they don’t talk to each other.

### 3.6.4 Sequences

Sequences always generate unique values, and so do not support “unique” explicitly (you can’t put “unique” on them: they are already unique). There are no non-unique sequences such as a “wrap-around” sequence that might start over from 1 after hitting a maximum value.

Sequences do support the “unique per parent” clause. A “unique per parent” sequence will start over and reset itself back to its starting value for each new parent row.

Consider a company that wants to generate purchase numbers based on the supplier’s code. The first P.O. for each supplier has a “1”, then a “2”, and so on. Any P.O. number will therefore have the supplier it applies to, and a sequence showing which P.O. it is for that supplier.

```

table suppliers
  insert 80 rows
{
  label "S" + format(sequence(1), "00")
}

table purchase_orders
  child of suppliers on this.id = parent.id
  insert 0..10 rows
{
  id      sequence(1),
  po_num  parent.label + "-" +
          format(sequence(1),
                "000")
}

```

Generates P.O. numbers:

S01-01  
S01-02  
S01-03

S02-04 *we wanted "-01" here!*  
S02-05  
S02-06

S03-07 *we wanted "-01" here!*  
S03-08  
S03-09

Not what we wanted.

```

table suppliers
  insert 80 rows
{
  label "S" + format(sequence(1), "00")
}

table purchase_orders
  child of suppliers on this.id = parent.id
  insert 0..10 rows
{
  id      sequence(1),
  po_num  parent.label + "-" +
          format(sequence(1 unique per parent),
                "000")
}

```

Generates P.O. numbers:

S01-01  
S01-02  
S01-03

S02-01  
S02-02  
S02-03

S03-01  
S03-02  
S03-03

Each supplier gets P.O. numbers with their label, starting at 1.

## 3.7 Columns

After a table definition (“table invoices delete all insert 100 rows”) are a collection of zero or more Column Definitions separated by commas, all within one set of curly braces (“{” and “}”). Each Column Definition has the name of a column to populate with data, and an expression that describes how DataRoller should generate data for that column. DataRoller Variables can be set as well, but see the Variables section below for details.

A sample Column Definition:

```
ncd_code      "AB-" + pattern("UUUNNNNNN") + format(sequence(1), "000")
```

“ncd\_code” is the name of the column and everything else is the Generator. The column name does not appear in double-quotes, but you can surround it with “[” and “]” if your column name is the same as a DataRoller keyword (like “sequence”).

“Generator” is a broad term in DataRoller for anything that can generate data suitable for inserting into the database. Generators each can generate dates, strings, and numbers following a random pattern, or

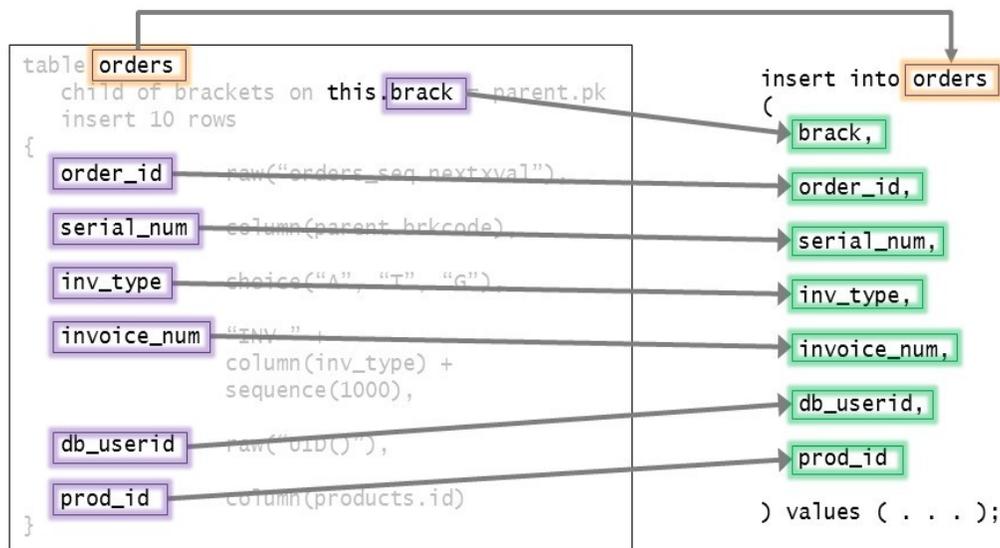
more organized logic. A Generator Expression is a bunch of Generators combined with things like “+” for string concatenation or numerical addition.

Column values can be broken down into two groups:

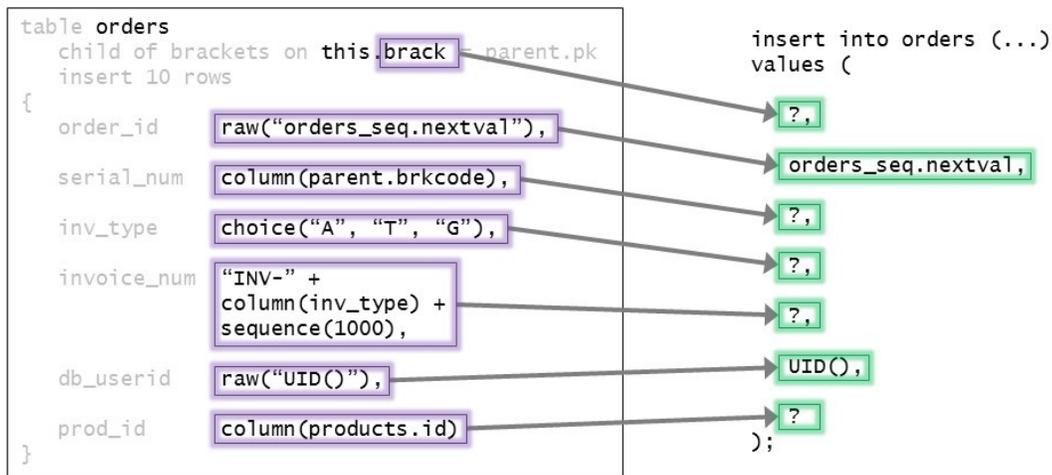
- The single syntax `raw()` used to modify the generated SQL `insert` statement used by DataRoller.
- All other Generators (see below) that generate values that are bound to the SQL `insert` statement.

DataRoller constructs a SQL `insert` statement for each table as follows:

1. Each column in the DataRoller file is added as a column name in the SQL `insert` statement. Note that the local foreign key to a parent table is also included in the insert statement.



- Each column contributes one “?” to the values clause of the SQL insert statement. All generators and parent table definitions contribute a “?” that is bound on a per-row basis to whatever the Generator generates. `raw()` is the exception: the argument to `raw()` is used directly in the SQL insert statement instead of a “?” for later binding.



Because `raw()` is used to construct the SQL `insert` statement directly, `raw()` cannot be combined with any Generator or appear in any fashion other than by itself as the only portion of the column.

## 3.8 Data Types

### 3.8.1 Strings

Strings are any sequence of characters surrounded by double quotes. The usual escapes are supported:

Escape	Value	Description
<code>\\</code>	<code>\</code>	Backslash escapes the next char, another backslash.
<code>\r</code>		ASCII 0x0D, the carriage return (non-printable).
<code>\n</code>		Newline
<code>\t</code>		Horizontal tab. There is no “\v” available for vertical tab. Does anyone use vertical tabs anymore?
<code>\"</code>	<code>"</code>	Escape the double-quote. There is no “\'” to escape a single quote. Use the single quote directly in the string.

Escape	Value	Description
<code>\u00a7</code> <code>\u2264</code>	§ ≤	Unicode characters can be included with “\u” followed by exactly four hex digits (case-insensitive).

Newlines (carriage returns and linefeeds) are valid characters in their own right, so you don’t have to use “\n” if you don’t want to. *Be careful*, because this means forgetting to put a closing double-quote at the end of a line does not generate an error. All text until the next double-quote character will be included in the string, leading to unexpected errors flagged nowhere near where the root cause is!

String Literal	Exact Value	Comments
<code>"Hello world!"</code>	Hello world!	Simplest string, nothing fancy going on.
<code>"line1\nline2\nline3"</code>	line1 line2 line3	The “\n” put actual newlines into the string literal.
<code>"line1 line2 line3"</code>	line1 line2 line3	A string may span multiple lines for embedding long SQL, for example. Newlines are preserved.
<code>"_'thing1'_"thing2\"_"</code>	<code>_'thing1'_"thing2\"_"</code>	Escaped double quotes. Single quotes do not need to be escaped.
<code>"Use \\\"t\" for tab"</code>	Use <code>"\t"</code> for tab	<code>\\</code> is a single <code>\</code> and the <code>t</code> is just a <code>t</code> .

### 3.8.2 Integers

Integral numbers are any whole number, negative, zero or positive. Internally, all integral numbers are stored as Java longs, so you have an effective range of a negative number of 19 digits to a positive number of 19 digits.

By default, all integer literals are assumed to be expressed in base 10 notation. Prefix a literal with “0x” to use hex (0-9 and A-F). Octal is supported by appending “\_8” and binary is supported by appending

“\_2”. The “\_” is reminiscent of a subscript notation from typesetting language T<sub>E</sub>X. There are no other bases supported with the “\_” notation currently. Note that “0123” is a decimal number, not octal in this scheme!

Integer Literal	Base 10 Value	Description
123	123	Numbers are base-10 by default.
-560000	-560000	Negative numbers as expected.
0xFF	255	“0x” is hex prefix. “F” and “f” are both 15.
-0xa8	-168	Negation comes before the “0x”.
123_8	83	$123_8 = 83_{10}$
-123_8	-83	Negatives are the same as base-10 above.
1010_2	10	$1010_2 = 10_{10}$
-1010_2	-10	Negatives are the same as base-10 above.

Note that Decimals and Floats below are only expressible in base-10 notation.

### 3.8.3 Decimals

Decimals are real numbers with a fixed number of decimal places (no repeating decimals like 0.3333333...). Decimals are expressed as 1 or more digits, a period and one or more digits, such as “1.0”

or “-559.0021837”. Decimals are used for database columns like `DECIMAL(10,5)` and `NUMBER(9,2)`: those types that have perfect precision and no round-off errors like typical floating point numbers.

Be careful with Decimals! They have no ability to represent irrational numbers or numbers with repeating decimal digits like the number “one third” ( $1 \div 3$ ). The number “1.3333...” is not a Decimal.

Decimals are internally represented by a 32-bit integer holding the mantissa and a 32-bit integer scale that holds where the decimal point is in relation to the whole number mantissa. The advantage of Decimals is that they do not suffer from rounding or inexact mathematics. Ever add 0.000001 to 0.000002 and get 0.000002999999999 with IEEE floating point types? ...not with Decimals!

### 3.8.4 Floats

Floats differ from Decimals in that they directly correspond to the internal floating point types of the Java language and the underlying operating system. Floats are only applicable for database types such as “float”, “real” and “double”. Using a Decimal DataRoller type for a database double column will result in potential rounding problems. Using a Float DataRoller type with a database `DECIMAL` column may result in over/underflows and inexact results.

Floats are represented using the scientific notation of a mantissa, “e” and the power-of-10 exponent, either potentially negative. The use of the “e” notation is the defining syntax for Floats. Examples: “0e0”, “0.0000005e189”, “1231321.77e-90”.

Mathematically, , so if you don’t like scientific notation, just add “e0” to the end of a number to make it a floating point literal. I.e., write 123.456 as a float in a DataRoller file as “123.456e0”.

### 3.8.5 Dates and Times

Date literals start with `D'` and end with a `'` character. Note that double-quotes are not used (they delimit strings). Values within the delimiters generally follow the ISO date and date-time structure of “yyyy-mm-dd”. Timestamps use the date format with “T” and the time in “hours:minutes” and optional “:seconds”. Hours are in 24-hour notation.

In addition to the above, the keywords `today`, `yesterday`, `tomorrow` and `now` are valid as a Date and Timestamp with the same delimiters as above. `D'now'` is literally this very second. `D'today'` is the date portion of `D'now'`. If you use `today` for a database column that supports a time component, the time component is midnight, meaning `today` always predates `now` except at exactly midnight on the wall clock, when they are equal.

There currently is no time-only data type supported.

Examples:

- `D'1900-1-1'`

- D'2011-12-24T23:59:59'
- D'1998-04-30T8:00'
- D'yesterday'

### 3.8.6 Booleans

Valid Boolean constants: "true", "false".

Example:

Given a PostgreSQL table definition	Fill it with values via
<pre>create table abc (     foo    Boolean,     bar    Boolean )</pre>	<pre>table abc {     foo    true,     bar    false }</pre>

Note that true and false are not enclosed in quotes: they are not strings, they are constant values of type Boolean in DataRoller.

### 3.8.7 Nulls

Not terribly interesting, but "null" is a keyword in the language and will insert a NULL database value when used explicitly.

### 3.8.8 Unsupported Data Types

There is no "bit", enumeration or MAC address or other data types within DataRoller. This does not mean that you cannot store them into a database. JDBC drivers and databases themselves are flexible in converting types when they are unambiguous.

For example, you can do this to a macaddr column and the database will convert the string into the internal representation:

```
insert into mytable (mymac) values ('08:00:2b:01:02:03')
```

There are two approaches to handle unusual circumstances, such as populating an Oracle Spatial database with columns of type `SDO_GEOMETRY`:

1. Write or use an existing stored procedure in the database to convert from a simple data type above, into the database-specific internal representation. For Oracle above, the PL/SQL function `SDO_GEOMETRY(v varchar2, number)` already exists.
2. Use a User-Supplied Java function in DataRoller to accept simple types from above, and return a vendor-specific internal object, such as an object of type `oracle.sql.ARRAY` for structured types in Oracle.

### 3.8.9 Null Expressions

Each Column Expression has an optional Suffix, which is of the form “`null ___%`” where “`___`” is an integer between 0 and 100. “`null 5%`” means approximately 5 out of every 100 rows generated by DataRoller will be NULL values. “`null 0%`” is the same as not specifying the suffix at all, and “`null 100%`” is the same as omitting the entire Column Expression.

DataRoller uses a simple random number generator to decide when to leave a column null. This means for small values of “`insert ___ rows`” on a table, DataRoller will not generate the correct percentage of null values.

## 3.9 Variables

Variables in DataRoller are un-typed places to store pre-computed values for use elsewhere in a DataRoller project file. To declare a variable, just assign it a value – there are no separate declaration statements. To reference a variable, just mention it. References to variables that have not yet been assigned are taken to have a value of `null`.

Variable names start with a dollar symbol (“\$”) and contain one or more letters or numbers. “`$12345`”, “`$abcde`” and “`$c3p0`” are all valid variable names. Labels used to name columns may also contain letters, numbers and optional dollar signs, so a column name that starts with a dollar sign must be escaped using the square bracket syntax: “`$abc`” is a variable name, regardless of where it appears. “`[$abc]`” is a label (such as a column name) regardless of where it appears.

### 3.9.1 Assignment

Assign values using the equals sign (“=”) anywhere within a table’s columns. DataRoller will evaluate each column’s values in the sequence they are listed in the project file, including the occurrences of variable assignments. DataRoller re-computes each column and variable assignment for each row it generates.

For example,

```
table testVars
  insert 3 rows
{
  $n = sequence(5),   col1  $n,
  $n = $n + 1,       col2  $n,
  $n = $n + 1,       col3  $n
}
```

will produce the following rows in table testVars:

col1	col2	col3
5	6	7
6	7	8
7	8	9

The first assignment to `$n` is `sequence(5)`, which will hand out numbers 5, 6, 7 and so on. `sequence()` is unrelated to any other assignment in this table: `DataRoller` simply returns one more than the last value `sequence()` returned from the last row it generated.

Anything you can assign to a column can be assigned to a variable, including the “`null 10%`” syntax at the end. This includes expressions like “`case...when...else`”, any function, any operator such as “`+`” or “`*`”, or any reference to any variable (including the one on the left-hand side of the assignment).

All of the following are valid variable assignments:

Assignment	Description
<code>\$a = \$b null 50%</code>	Sometimes whatever is in variable <code>\$b</code> , sometimes null.
<code>\$a = pattern("B" * random(1..6))</code>	

Assignment	Description
<code>\$a = products.prod_id</code>	Picks random values from table products column prod_id
<code>\$a = name != "root" &amp;&amp; group != "wheel"</code>	References columns name and group in the current row being generated
<code>\$a = if (haspreviousrow())     "DEFAULT"     else         column(policies.label unique per pa parent)     end</code>	Assuming this is in a table with a parent, this will pick a value for \$a of "DEFAULT" for the first row per parent table row, and then pick a random policy label (unique) for each subsequent record per parent row.

Variables hold a single, scalar value until they are assigned another value. There is no local scope to any variable, and they have an extent from when they are first assigned to the end of the entire DataRoller project file. Put another way, variables are global across table sections and hold their values from table to table.

For example, the following will always put a new value one greater than the previous into table tree, even though this table is included twice in the project file.

```
table tree
  insert 4..9 rows
{
  $seq = sequence(),
  id    $seq
}

table tree
  insert 10..20 rows
{
  $seq = $seq + 1, // has last value used in table above to start!
  id    $seq
}
```

### 3.9.2 Variables and randomrow()

The generator `randomrow()` picks a row at random from the table (or lookup table) specified. The actual value it returns is a DataRoller internal type that is generally not useful other than with the special variable syntax below. Assign a variable to `randomrow()`, and then reference columns from this random row via the `."` syntax:

```

table myproducts
  child of person on this.person_id = parent.id
  insert 5 rows
{
  $prod = randomrow(products),
  prod_id      $prod.id,
  prod_name    $prod.name,
  prod_price   $prod.price
}

```

DataRoller will pick a row at random from table `products` and store all columns in the `$prod` variable during the assignment statement. After this assignment statement, the variable `$prod` contains these column values until `$prod` is assigned another value (like when the next row in table `myproducts` is generated). This means that each use of variable `$prod` above (like `$prod.id`) will refer to columns from the *same row* in table `products`. This is a great way to coordinate values between two tables, such as in associative tables.

The “`bar`” in the “`$foo.bar`” syntax is a column label like any other. Escape a column that contains special characters using square brackets, such as “`$foo.[bar]`” to reference column “`bar`” in variable “`$foo`”. Be careful to include the “`.`” to separate the two portions. “`$foo[bar]`” is not a valid syntax in DataRoller: square brackets are used to escape labels, not indicate array reference.

The “`$foo.bar`” syntax cannot appear on the left-hand side of an assignment: DataRoller does not allow assignment into a random row picked via `randomrow()` via “`$foo.bar = 123`”.

## 3.10 Embedded SQL

Operational databases are often constructed to help the application maintain valid data “state” through things like check constraints and triggers. Any modification to data in the database outside the normal range of operation expected by the application can lead to real problems. For example, having triggers on table “`orders`” that insert history records into table “`orders_hist`” are effective, as long as the target application performs normal business modifications to the `orders` table.

There is no expectation that these triggers maintain the history when “`delete from orders`” executes. DataRoller can perform disruptive, wide-spread changes to a database outside the bounds of what application triggers were designed for.

Before running DataRoller, it is a good idea to disable or remove these types of database-resident code so the database is “opened up” to non-operational access. When DataRoller is complete, put the triggers back.

In a similar vein, deleting all data from multiple tables and reloading them with a data set of an entirely different character will render a database inefficient: table and column statistics are not always updated along with data changes, and a table that does not match its pre-determined statistics will likely produce inefficient queries.

To wit, an `orders` table with a million rows and a `products` table with 100 rows will likely start an inner-join on the `products` table first, and then perform index unique searches on the `orders` table. If DataRoller removes all records from `products` and `orders`, and then loads up a million `products` and only a few `orders`, inner joins will still start on the `products` table taking up dramatically more resources than needed.

Use DataRoller to replace statistics on tables after new data has been loaded.

### 3.10.1 Embed SQL in DataRoller File

The method to perform the above is by embedding SQL statements to execute as part of the DataRoller Project execution. Embedding SQL is easy: use keyword “`sql`” followed by the statement in double-quotes. Include a single statement per `sql` keyword.

```
sql "create function change_date (d date) returns date return d - 1 day"
sql "set database event log level 0"

sql "set table fireworks read write"
sql "drop index i_fireworks_code"

table fireworks
  insert 100 rows
{
  date_lit      random(D'2009-4-12' .. D'today' step /hour/),
  bang          sql("values change_date(?)" bind date_lit),
  product_code  sequence()
}

sql "create index i_fireworks_code on fireworks(product_code)"
sql "set table fireworks read only"

// continue on with other tables ...

sql "drop function change_date"
sql "set database event log level 2"
```

This uses embedded SQL to

1. Embed a function in the database that can help with inserting test data, and then remove it when DataRoller is done executing.
2. Turn logging in the database way down so the event logs do not fill up while every row is deleted and a new set is inserted.
3. Allow modifications to table `fireworks`, which is usually read-only.
4. Remove an index and rebuild it afterward to speed up execution of DataRoller.

## 3.10.2 Reference External SQL File

Embed simple SQL statements directly in a DataRoller project file with the `sql` keyword. Collect larger SQL statements, blocks of code or larger sequences of statements in a separate “.sql” file and reference this file from within the DataRoller project file.

- `sql` “update mytbl ...” will execute the string as a single SQL statement
- `sql file` “myfile.sql” will execute each statement in file `myfile.sql`

Each separate SQL statement is executed within an external SQL file. DataRoller does not parse the SQL syntax or otherwise understand the structure of an external SQL file. It makes rudimentary decisions about where one statement ends and another begins, and can only handle a limited set of comment placements without being confused.

Rules for assembling an external SQL file for DataRoller:

- C-style comments are not supported in any capacity “/\* ... \*/”
- “#”, “--” and “//” comments are supported if they have nothing but whitespace to their left (no executable SQL statements can occur on a line with a comment for DataRoller)
- A SQL statement may span multiple lines as long as there are no blank lines within it
- Use one or more blank lines to separate SQL statements, optionally with one or more lines with nothing other than “GO”

Example:

```
sql "insert into states (abbrev, name) values ('ca', 'california')"  
  
table states  
  insert 2 rows  
{  
  ...  
}  
  
sql file "cleanup.sql"
```

File `cleanup.sql`:

```
-- Fix any abbreviations that are lower-case  
update states  
  set abbrev = upper(abbrev)  
  
#  
# comment  
#  
create index i_states_abbrev on states(abbrev);
```

## 3.11 A Note on File and Folder Names

Several Generators require the name of a file or folder to read. The examples above show a simple file-based string like “C:\data\myfile.txt”. DataRoller has an internal syntax for describing files and folders that extend this to support looking in zip files and searching the Java classpath.

Basics of DataRoller Files and Folders:

1. If it starts with “classpath:”, everything else is searched in the Java classpath
2. If it contains an exclamation point (“!”), the left-hand side is a zip file to open, and the right side names the thing in the zip file to retrieve.
3. If DataRoller is expecting a folder, but a zip file is specified, DataRoller will open the zip file and consider all files in the zip archive to be the files in the folder to use.
4. If the above do not apply, just consider the string to be a file or folder on disk.

Things in DataRoller that work this way:

Construct	Context	Examples
lookup table	File	lookup table mytbl = “prod_dump.zip!recs.xls” Read recs.xls from the zip archive prod_dump.zip and use it as a pseudo table in DataRoller.
folder()	Folder	folder(“C:/mydata”) Open folder C:/mydata on disk and read in all files. folder(“C:/mydata.zip”) Open the zip file and consider its contents to be the files to process.
filerow()	File	filerow(“classpath:data.txt”) Search the classpath for file data.txt and read it in.
xpath()	File	xpath(“a.zip!b/c/d.xml”, “/root/child”) Read the file d.xml in folder b/c/d inside zip archive a.zip as a source of XML.

When using “!” to name a zip file and an internal entry, the zip file can be an absolute or relative path. Zip files with a relative path (or no path) are rooted in the same folder as the input project file. This way,

a project file can be kept with all zip and other files that it depends on. The entire directory of files can be moved around together.

## 3.12 Lookup Tables

The job of generating test data or demo data is rarely an exercise in filling random bits and bytes into columns. One of the simplest ways to fill a database with production-looking demo data is to go raid the production database! DataRoller has the ability to reference Excel spreadsheets as if they were tables.

For example:

- Export important lookup tables from a production database or other source and store them in Excel sheets.
- Reference the Excel “tables” in DataRoller via the “lookup table” syntax.
- Use these table aliases in Generators like `randomrow()` and `column()` below.

Create an alias within DataRoller called “mytbl” that refers to the first worksheet in file “people.xls” which resides in ZIP file “myfile.zip”:

```
lookup table mytbl = "myfile.zip!people.xls"
```

Because of the way other parts of the system work, don’t pick table aliases that are the same as real tables in the database. The lookup definition here will make the real database table unreachable in things like `column()` and `randomrow()`.

Excel files must be “.xls” format, not “.csv” or “.xlsx”. This is another way of saying the file must be in “Excel 97-2003 Workbook” format. The first worksheet will be used, and all other worksheets will be ignored. The first row of the worksheet should contain the column names, and all other rows should contain the data.

Because of a limitation in Excel internal format, DataRoller cannot easily tell the difference between floating-point non-precision real numbers, “decimal” precision numbers and integers. So, DataRoller takes the most liberal interpretation and reads in all numbers as floats. To override this behavior, suffix the header cell name with “#” and a code from the table below.

Suffix	Meaning
#D	Assumes each cell in the column is a number representing a Date or Date-time value.
#F	Default for numeric columns, uses a floating-point value for each cell in the column.
#I	Assume each cell in the column is an integer number.

If a cell is formatted in Excel as a Text cell, any suffixes above are ignored and the cell value is taken as a string.

## 4 Generators

---

Generators come in several flavors:

**Random Data:** Generator that picks values randomly from a range you specify.

- Random values picked from a list or explicit range of values
- Large binary data for BLOBs

**Structured Data:** Generator creates data following a plan you describe.

- Strings following a certain pattern, or Lorem Ipsum text
- Sequential numbers or dates

**Lookup Data:** Data pulled from another column, file on disk, or database location:

- Foreign key lookups into other database tables
- Random records (rows) picked from a text file

**Operators:** Infix symbol that combines two Expressions into a new Expression:

- “<”, “>=” and other comparators to compare values
- “+”, “-”, “\*”, “/”, and “%” for typical math operations

**Conditionals:** “if then else” logic to adjust processing based on conditions:

- “if (cond) ... else ... end” syntax similar to Java or C++
- “case when cond<sub>1</sub> then stmt<sub>1</sub> ... else stmt<sub>n</sub> end” syntax similar to SQL

**Functions:** Java code that creates new data from zero or more Generator Expressions:

- Math and number functions like `sin()`, `cos()` and `tan()`
- String functions like `upper()`, `lower()` and `replace()`

**User Functions:** Java code written by you and invoked by the DataRoller engine:

- Write your own functions in Java that generate strings, numbers, dates, etc.
- DataRoller will load your jar file and invoke your methods

## 4.1 Random Data

The hallmark of all random value Generators for basic types is “random()” with values using “..” syntax:

- `random(1..10)`
- `random(1.50 .. 9.25)`
- `random(-5.7e10 .. -1.2e10)`
- `random(D'1985-1-1' .. D'1985-12-31')`

For obvious reasons, the first term has to be less than or equal to the second term, or the range is invalid. Values are taken at random from the range, including both the lower and upper bound values (the range is inclusive). Duplicates are likely generated. The distribution of random values is approximately linear, given a large enough number of rows to generate. For small row counts, the distribution is likely non-linear, and many valid values in the range may not be generated while others are generated more than once.

There is no version of `random()` that supports Boolean types; use `choice(true, false)` instead.

### 4.1.1 Random Integers

In addition to the basic “`random(1..10)`” notation, a step can be included to change which values are generated. “`random(1..10 step 2)`” means 1, 3, 5, 7, and 9 are generated: start at the low bound, and only pick values that are reachable by 2s. Note that “step 2” has nothing to do with even, or “divisible by 2”: it is something like an increment value. “`random(2..10 step 2)`” generates random values in the set 2, 4, 6, 8, 10.

DataRoller will never generate a value outside of the low and high bounds, even if the step doesn't exactly land on the high bound. For example, “`random(6..14 step 3)`” generates 6, 9, 12 and never 15 because  $15 > 14$ .

The step can be any positive value (zero and negative numbers are not allowed), even one larger than the entire range. The expression “`random(89..226 step 10000)`” generates random values starting at 89 and all values 10000 from that initial value up to value 226. In other words, it always generates the single value 89 because  $89 + 10000 > 226$ . This has the side-effect that no value for step will ever lead to an empty set of values to pick from at random. The lower bound is always in the set of valid values.

If a random integer expression is suffixed with “unique”, it will never generate the same value twice. If the number of valid possible values is less than the number of rows to be generated for the current table, an error will result when this Generator runs out of values.

When both “step” and “unique” or “unique per parent” are included, “step” must appear first. The following are valid:

- `random(1..100 step 2)`
- `random(1..100 unique)`
- `random(1..100 step 2 unique)`

## 4.1.2 Random Decimals

Random decimal values are syntactically the same as integers above with decimal literals, except:

- The “step” clause is required, and
- The “unique” and “unique per parent” suffixes are not supported.

The `step` clause is required because it is not always obvious what the implied step is: “`random(1.00 .. 1.4)`” could be {1.0, 1.1, 1.2, 1.3, 1.4}, or something much larger such as {1.00, 1.01, 1.02, 1.03 .. 1.38, 1.39, 1.4}.

## 4.1.3 Random Floats

Due to the continuous nature of Float values, “step” is not supported. “`1.54e-106`” as a step could easily lead to round-off values, resulting in strange results. This is also against the general use of floating-point, continuous numbers.

Specify random float values with a simple low ... high range, such as “`random(1.1e4 .. 2.2e5)`”.

Both “unique” and “unique per parent” are not supported for floating-point ranges.

## 4.1.4 Random Dates and Timestamps

Dates and Timestamps contain discrete values and therefore support “step” and “unique” clauses, but their syntax is different than above. The date or Timestamp range is just a low value “.” high value. The range is a notation of which dimension is the “cut-off” for the set of valid values. I.e., “`step /hours/`” means generate values such that the minutes and seconds components are always zero. The forward slashes are required.

The default step is “`step /days/`”, meaning the time is always zero (midnight). “`random(D'1900-1-1' .. D'1900-12-31')`” means generate Timestamps between those two dates, with hours, minutes and seconds zero.

The valid values for step are “/years/”, “/months/”, “/days/”, “/hours/”, “/minutes/” and “/seconds/”. All values supplied in the low and high bound must not have a value. For example, “step /hours/” means you cannot specify a value for minutes or seconds in the low or high bound. Since some step values (like “/years/”) run into syntax problems with the date literals, “1” can be used for month and day.

Step	Template for Low and High bounds	Interpretation
Seconds	None – use any date, unrestricted	
Minutes	D' nnnn- <u>nn-<u>nn</u></u> ' D' nnnn- <u>nn-<u>nn</u></u> Tnn: <u>nn</u> ' D' nnnn- <u>nn-<u>nn</u></u> Tnn: <u>nn:00</u> '	Seconds must be zero when specified.
Hours	D' nnnn- <u>nn-<u>nn</u></u> ' D' nnnn- <u>nn-<u>nn</u></u> Tnn: <u>00</u> ' D' nnnn- <u>nn-<u>nn</u></u> Tnn: <u>00:00</u> '	Seconds and minutes must be zero when specified.
Days	D' nnnn- <u>nn-<u>nn</u></u> ' D' nnnn- <u>nn-<u>nn</u></u> T <u>00:00</u> ' D' nnnn- <u>nn-<u>nn</u></u> T <u>00:00:00</u> '	Entire time component must be zero when specified
Months	D' nnnn- <u>nn-<u>1</u></u> ' D' nnnn- <u>nn-<u>1</u></u> T <u>00:00</u> ' D' nnnn- <u>nn-<u>1</u></u> T <u>00:00:00</u> '	Because the syntax for a date does not allow omitting the day, set the day to 1. I.e., “May 12” is not a good bound when you only want months. The date returned always has a day-of-the-month set to 1.
Years	D' nnnn- <u>1-<u>1</u></u> ' D' nnnn- <u>1-<u>1</u></u> T <u>00:00</u> ' D' nnnn- <u>1-<u>1</u></u> T <u>00:00:00</u> '	Like above, use “1” for the month number. All dates returned have a day+month of January 1 <sup>st</sup> .

Don't forget that “today” always has a time component of all zeros. “now” is this exact point in time, and so probably has a non-zero value for hours, minutes and seconds. This means “random(D'today'..D'now' step /minutes/)” or any step other than “/seconds/” will fail because today will have at least one non-zero time component.

Both “unique” and “unique per parent” clauses are supported and may only appear after the step clause.

### 4.1.5 Choice

The choice() generator will pick a value at random from the list of explicit values included directly in the declaration. Specify a list of any valid values, and choice() will pick values at random. The default

distribution is linear: each value listed in `choice()` is picked with equal probability. Each item listed in `choice()` can also include a relative weight so some values will be chosen more frequently.

If no weight is included, it is assumed to be 1.

Examples:

<pre>choice( 0xff0000,         0x00ff00,         0x0000ff )</pre>	Pick any of the three values at random, each with equal probability.
<pre>choice( "open" weight 6,         "closed" )</pre>	"open" chosen around 6 times more frequently than "closed".
<pre>choice( "A" weight 8,         "B" weight 3 )</pre>	$8 + 3 = 11$ , so 8 times out of 11 "A" is chosen and 3 times out of 11 "B" is chosen.
<pre>choice( "Jan" weight 31,         "Feb" weight 28 )</pre>	Using this same approach, pick a month based on the distribution of possible days. January has 31 days in it, and February usually has 28. So, "Jan" is chosen slightly more frequently because there are more days in January than February.

## 4.1.6 BLOB

Database column types accepting binary data include `binary`, `long`, `long raw`, `blob`, `image` and others. The primary way to put binary data into these fields is to use `folder()` above with a folder full of non-text files. This approach leaves the database in a state where the application can still execute. Imagine the results of putting random bits into a column that the application considers an image! The application would not function correctly.

Use the `blob()` Generator for creating massive amounts of random bytes to store into database columns that you don't care about. This is a good approach for truly maxing-out a database in preparation for load testing: firing queries at a database with gigabytes of random data to chew on.

Pass the number of bytes to generate as a range to `blob()`.

Example:

```
blob (1000000 .. 1000000000) // generate between 1MB and 1GB of random data
```

**Implementation Note.** DataRoller does not stream random data into the database. It creates the entire blob byte array in memory first, and then sends it to the database. This means the complete generated BLOB value exists in memory. Asking for a BLOB of size 30 GB will likely crash your Java VM without special consideration.

## 4.2 Structured Data

### 4.2.1 Lorem Ipsum

Lorem Ipsum refers to generated nonsense words that are organized in a way that resembles a natural western language without diacritical characters. Sentences start with capital letters and end with periods. Sentence lengths tend to follow general western word counts. Lorem Ipsum is used to populate areas that should hold prose without being distracting to the eyes: the text does not intrigue the “reader” or pull their gaze from the overall appearance of the screen.

Sample Lorem Ipsum text:

*“Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut mauris augue, fringilla at lacinia vitae, convallis in est. Sed molestie dictum sem, eget pulvinar est fermentum sed. Vestibulum eros ipsum, interdum id pulvinar et, vehicula ac dolor. Integer viverra, libero vitae egestas scelerisque, odio lorem dictum lectus, id egestas nibh felis mollis augue.”*

Use “`lorem()`” to generate random strings to populate descriptive database columns. Pass a range to use for the generated lengths, such as “`lorem(50..900)`” to generate a string of between 50 and 900 characters that has sentences of random-looking text.

This Generator does not just spit out the same text each invocation. There are some nice subtle features in here:

- The industry-standard Lorem Ipsum text contains only a limited number of nonsense words. DataRoller has record of the complete set of Lorem Ipsum words, and the relative weighting for how often they appear in the standard text. `lorem()` uses these weights to construct a string of Lorem Ipsum text following the standard distribution without being a copy of existing text. Each row gets distinct text.
- `lorem()` first picks a random length to generate, and then adjusts the sentence lengths near the end of the generated string so it ends as close to the chosen length as possible without going over, and without having a final sentence too short (sentences are between 5 and 12 words each).

## 4.2.2 Sequences

Sequences serve to return a different value for each row, each value one greater than the last. For numbers, this is straight-forward. For Dates and Timestamps, it uses an increment time duration for the same effect.

Integer Sequences like “`sequence(100 step 10)`” start at 100 and count up with each subsequent value 10 larger than the last. The `step` is optional and defaults to 1. If both the starting value and `step` are omitted (“`sequence()`”), a starting value of 1 and `step` of 1 are used.

The `step` value may be positive or negative, but not zero.

Sequences do not “wrap” or stop at zero. “`sequence(3 step -1)`” returns {3, 2, 1, 0, -1, -2, -3, ...}.

Decimal and Float Sequences must always supply a `step`.

Because sequences generate a new number each invocation without wrapping, they are implicitly unique. The “`unique`” clause is not supported as it would be redundant.

The optional “`unique per parent`” clause is supported. A sequence with “`unique per parent`” will reset back to the starting value for each new parent row created.

### 4.2.2.1 Date Sequences

Date sequences must supply a starting Date or Timestamp and may optionally include a `step`. Steps specify a Time Duration that is added to each previous value to compute the next value. Quick example of a sequence with a Time Duration:

```
sequence(D'1920-7-19' step /7 days 6 hours/)
```

When omitted, the default Time Duration is “`/1 day/`”.

A Time Duration can specify a value for years, months, days, hours, minutes or seconds, or any combination of the above. Multiple entries must appear in coarsest to finest values. I.e., years before anything, days before hours and so on. “`/1 month/`” applied to `D'2000-1-1'` will yield `D'2000-2-1'`, `D'2000-3-1'`, `D'2000-4-1'`, `D'2000-5-1'`, and so on. Notice that the day component of the date remains the first day of each month! A month in general concept is not always the exact same duration. For DataRoller “`/1 month/`” means move forward to the next month with the same day. This same concept applies to years (not always 365 days).

When incrementing by “`/1 year/`” for example, there is no requirement that all finer components be zero. So, “`sequence(D'1994-2-14' step /1 year/)`” will return Valentine’s day on each subsequent year.

This works nice for things like “quarterly”: “sequence(D’2003-1-1’ step /3 months/)” returns January 1<sup>st</sup>, then April 1<sup>st</sup>, then July 1<sup>st</sup>, then October 1<sup>st</sup> each year (the first day of the next quarter).

A fully-specified Time Duration:

```
/ 1 year -2 months 3 days -4 hours 5 minutes -6 seconds/
```

To compute the next value in a sequence, the previous Timestamp is moved by each component of the Time Duration. That is, the previous value is moved forward 1 year, then back 2 months, then ahead 3 days, and so on.

All entries in a Time Duration must not cancel each other out. For example, the following are all *invalid* Time Durations because they will not have a net effect on the previous value when computing the next value:

Expression	Meaning
/1 year -12 months/	Always cancel each other out
/1 month -30 days/	Only works when you cross from one month to another that has 28, 29 or 31 days.  Causes an error for April 1 <sup>st</sup> : 4/1/2000 + 1 month = 5/1/2000 – 30 days = 4/1/2000. This would always generate the same date!
/-1 day 24 hours/	Always cancel each other out
/1 month -29 days/	This will generate values successfully until it encounters a leap year, where it will fail when it crosses February.
/-1 year 365 days/	Will fail on all common years (non-leap years).
/1 hour -59 minutes -60 seconds/	Will always fail: all three values always add up to a net of zero.

Hours, minutes and seconds always relate in 1:60:60 ratio, so there are no interesting cancellations like leap year above.

Note that the order in which components are applied is significant in a few miserable cases:

Specification	<code>sequence(D'2010-6-1' step /1 month -30 days/)</code>	<code>sequence(6/1/2010 step /-1 month 30 days/)</code>
Start with the first date	2010-6-1	2010-6-1
Apply the Month first	2010-7-1	2010-5-1
Now apply the Days	2010-6-1	2010-5-31
Result	Same as start! Error!	Not the same, no error

## 4.3 Column

Generator `column()` can be used to reference a value in another column, optionally in another database table or lookup table. `column()` is a collection of similar generators that all share a common syntax.

Column generators:

Column Generator	Use
<code>column(gate_num)</code> <code>gate_num</code>	<b><i>Current-Row Reference</i></b> Evaluates to the value in the current table in the current row that was just generated.  The <code>column()</code> can be omitted.
<code>column(gate_num.level1)</code> <code>gate_num.level1</code>	<b><i>Separate Table Reference</i></b> Evaluates to a random value from the column in a separate table or external file (lookup table).  The <code>column()</code> can be omitted.
<code>column(parent.level1)</code> <code>parent.level1</code>	<b><i>Parent Table Reference</i></b> Evaluates to the column in the specific row in the parent table. This is not the same as the Separate Table Reference above! See below for details.  The <code>column()</code> can be omitted.

As a shortcut because the above column references are so prevalent in DataRoller files, the “column()” syntax can be omitted as long as “unique” is not present. The column() syntax is required when the “unique” or “unique per parent” feature is used.

```
table products
  child of names on . . .
  insert 10 rows
{
  order_id      sequence(),
  inv_type      pattern("U"),
  invoice_num   "INV-" + inv_type, // inv_type references col above
  prod_id       apps.id,           // id col in table apps
  customer      parent.last_name  // last_name column from names table
}
```

A column name or table name that clashes with a DataRoller reserved keyword can be escaped with square brackets:

```
table [sequence]
  child of ...
  insert 10 rows
{
  [bind]        sequence(),
  key           "A" + format([bind], "000"),
  customer      parent.[folder]
}
```

### In Diagram Form

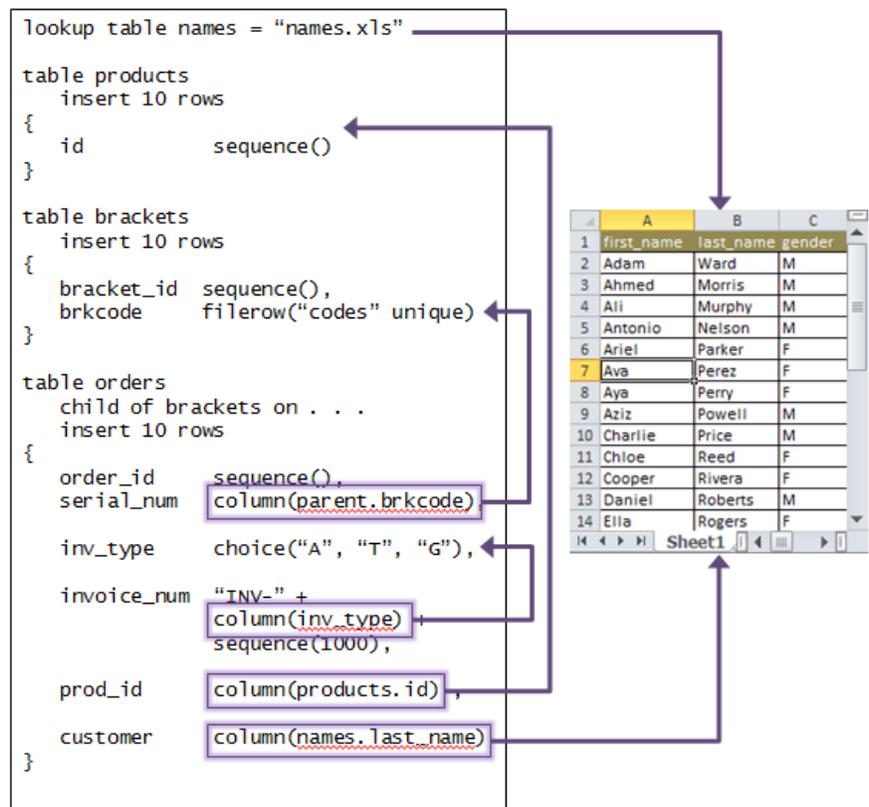
Table names refers to an external Excel sheet.

parent.brkcode references the single parent row in table brackets for every row in orders, column brkcode.

inv\_type by itself refers to the inv\_type column in the current row being generated.

products.id refers to a random value for column id in table products.

names.last\_name refers to a random value from Excel sheet names (see lookup table at top) column last\_name.



### 4.3.1 Current-Row Reference

The simplest invocation `column()` is to name another column in the table currently being processed. This will simply copy the value from that column in the current row being assembled by DataRoller. I.e., this value exists in DataRoller and will soon be in the database after DataRoller has assembled all values for the current row. Because of this, do not name a column that appears in the current table further down the DataRoller project file: the value for subsequent columns have not been generated at the time that `column()` is processed. Obviously, `column()` cannot refer to itself.

Neither “unique” nor “unique per table” are supported since a Current-Row Reference applies only to the current row about to be inserted.

### 4.3.2 Separate Table Reference

To reference a column in another table, invoke `column()` passing both the table and target column with a “.” separator. For example, `column(products.name)` refers to column name in the products table. When invoking `column()` with a table and column name, `column()` picks a random row from the target table and uses that column’s value.

The table name can be the alias for a lookup table defined at the top of the DataRoller project file, or a real table in the database. When the first invocation is constructed, the entire target table is read into memory, and this memory copy is used for the rest of the DataRoller project run. This means referencing very large Excel files (lookup tables) or database tables with a large number of rows or large columns (LOBs) may lead to memory problems.

Because tables are completely read into memory upon first need, self-referencing tables will not work the way you might think. Consider this:

```
table employees
  delete all
  insert 100 rows
{
  emp_id  sequence(),
  name    filerow("names.txt"),
  boss_id column(employees.emp_id) // doesn't work the way you want this to!
}
```

This is an attempt to build a random hierarchy by putting a previously-generated primary key into the `boss_id` field. While building the first row to be inserted into the database, DataRoller will cache the entire `employees` table in memory, which currently has zero rows (note the “delete all” above!) and so will cause an error.

The `sql()` generator might be considered as a possible solution too:

```
table employees
  delete all
  insert 100 rows
{
  emp_id      sequence(),
  name        filerow("names.txt"),

  boss_id     // probably not what you want:
              sql("select emp_id from employees order by RAND() limit 1")
}
```

This too is problematic. DataRoller will batch up insert statements and send them to the database in groups. The script above will only consider rows already in the database. DataRoller will create the first fifty rows all of which have null for `boss_id` since no data has been written to the database yet. When DataRoller creates row number fifty-one, there will then be fifty rows in the table for consideration.

Both “unique” and “unique per parent” are supported. When using one of these clauses, the `column()` syntax must be used. “`column(model_types.name unique)`” is correct; just “`model_types.name unique`” is a syntax error.

### 4.3.3 Parent Table Reference

A table definition that has a “child of” clause may use “`column(parent.foo)`” where `foo` is the name of a column in the table listed in the “child of” clause. “parent” is a DataRoller keyword and is required.

Any column in the parent table may be used, even columns not present in the DataRoller project file! DataRoller will execute a separate SQL `select` statement to retrieve a value for every row in the child table. This is significant if the parent table has triggers that create or alter data in the database table, or if there is a SQL statement after the parent table in the DataRoller project file that alters column values.

Consider the following example:

```
table accounts
  insert 100 rows
{
  username      pattern("L") + filerow("names.txt"),
  password_plain pattern( "L" * random(8..25) )
}

sql "update accounts set password_cipher = PASSWORD(password_plain)"

table login_probs
  child of accounts on this.username = parent.username
  insert 0..1 rows
{
  expected_passwd parent.password_cipher,
  actual_passwd   pattern( choice("L","U","N") * random(8..10) )
}
```

The `accounts.password_cipher` column is computed by calling a database-resident function `PASSWORD()` on the `password_plain` column (maybe because DataRoller does not have this ability). Table `login_probs` references this via the `parent.password_cipher` reference, even though column `password_cipher` does not appear as a column in the DataRoller file above. DataRoller simply issues a SQL `select` similar to the following:

```
select password_cipher from accounts where username = ?
```

Using the `parent` keyword is not the same as just using the actual table name of the parent table. “`parent.foo`” is nothing at all like “`names.foo`” even when the current table has clause “`child of names`”. The `parent` keyword means that DataRoller will follow the foreign key in the child table to the parent key in the parent table, and select the column in question. Using the actual table name of the parent table will select a value from a *random row* in the parent table, regardless of the foreign key in the child table.

In the example at right, `parent.person` and `pledges.person` appear to reference the same thing in parent table `pledges`. They do both name the same column in the same table, but “`parent`” keyword has different semantics from just using the name of the table.

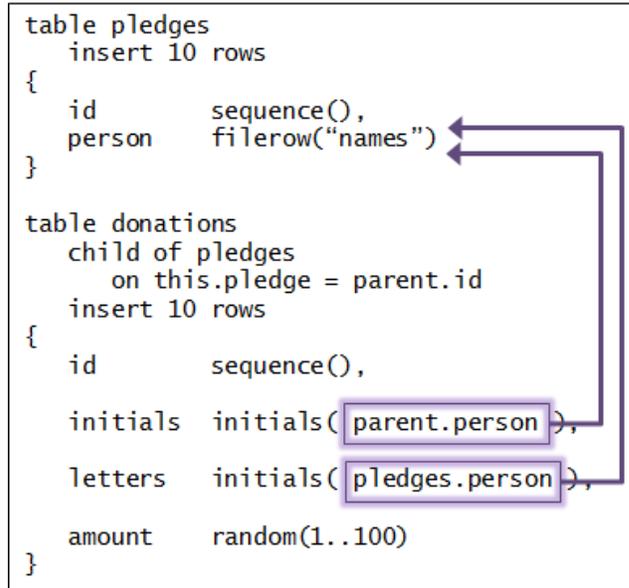
`parent.person` copies the value from the parent table, and then `initials()` converts the full name to just the 2 or three starting letters.

`pledges.person` just picks a random value from table `pledges`.

Also note that “`initials`” is the name of a function, but function names are not DataRoller keywords, so the column named “`initials`” does not need to be escaped as “`[initials]`”.

```
table pledges
  insert 10 rows
  {
    id      sequence(),
    person  filerow("names")
  }

table donations
  child of pledges
  on this.pledge = parent.id
  insert 10 rows
  {
    id      sequence(),
    initials initials( parent.person ),
    letters initials( pledges.person ),
    amount  random(1..100)
  }
```



DataRoller processes each table in the project file to completion before starting the next table in the project file. This rule also applies to tables with “`child of`” clauses: DataRoller will complete all processing for the parent table first, and then continue with each child table separately. This is important if you directly reference a parent table in a child table via the parent table name (the “`pledges.person`” above instead of “`parent.person`”). Using “`pledges.person`” has every row from table `pledges` available to it.

A table in a DataRoller project file may have more than one child table in the project file.

Both “`unique`” and “`unique per table`” clauses are not supported for Parent Table References since they directly reference a single row in the parent table.

## 4.4 Lookup Data

### 4.4.1 File Row Lookup

`FileRow()` will read in the entire contents of a file passed in, and pick rows at random to insert into the database. The file specified must be a text file with typical line breaks (UNIX or DOS format).

`FileRow()` is typically used with a large file of “boilerplate” text or common words and phrases like last names, cities, countries and so on.

Both “unique” and “unique per parent” clauses are supported for `fileRow()`.

### 4.4.2 Folder Contents Lookup

The `folder()` Generator will retrieve the contents of files on disk and use them as values to insert into database columns. This is a good choice for columns that store images, Microsoft Word or Excel documents, large XML strings, or other structured data that you might have on hand. The Generators above are designed to create data from nothing based on a structure you put into the DataRoller Project file. Folder Contents Lookup and other Generators below are designed to use content you already have on hand.

Pass the name of the folder on disk where target files are kept. The `folder()` generator will pick a file at random for every row inserted. The value inserted into the database is the entire contents of the file (not the name of the file in question).

Data is retrieved from the target files without any byte manipulation, line-termination conversion or national character set changes for files not ending with “.txt”. Files ending with “.txt” are treated differently. “.txt” files are read in as strings and inserted into the database as strings (with line-termination characters intact).

Use a folder containing only “.txt” files to populate columns of type `clob`, `long varchar` and such.

Use a folder containing no “.txt” files for `blob`, `raw`, `long raw`, `binary` and such.

### C:\data\images

 monalisa.jpg  
 starrynight.jpg  
 waterlilies.jpg

### C:\data\pamphlets

 monalisa.docx  
 starrynight.docx  
 waterlilies.docx

### C:\data\specs

 monalisa.xml  
 starrynight.xml  
 waterlilies.xml

```
create table christies_inventory (  
  forsale_id    number(8) primary key,  
  picture       image    not null,  
  pamphlet     long raw  not null,  
  spec_xml     clob      not null  
)
```

```
table christies_inventory  
  delete all  
  insert 3 rows  
{  
  forsale_id    sequence(),  
  picture       folder("C:/data/images"),  
  pamphlet     folder("C:/data/pamphlets"),  
  spec_xml     folder("C:/data/specs")  
}
```

This is a simple example to illustrate use. The table includes values for all three folders on disk, but “folder()” as used above does not correlate anything: a single row might have monalisa.jpg, starrynight.docx and waterlilies.xml in it. Folder() chooses values at random, so some values may be used more than once and others not used at all.

Folder() supports both “unique” and “unique per parent” clauses.

## 4.4.3 XML File Lookup

Like filerow() above, xpath() will read in a file and serve out contents randomly. xpath() reads in an XML file and will retrieve strings based on a supplied XPath expression. The XPath expression must match text elements directly, or Nodes that have text values (such as “<name>Malcolm Reynolds</name>”).

Consider the following XML snippet passed to xpath():

```

<bill session="107" type="sj" number="23">
  <titles>
    <title type="popular">Go get em!</title>
    <title type="official">Military Force Authorization</title>
  </titles>
  <sponsor id="300031" />
  <actions>
    <vote date="1000440000" how="roll" roll="281" where="s" />
    <vote date="1000520280" how="without objection" where="h" />
    <enacted law="107-40" date="1000785600"/>
  </actions>
  <subjects>
    <term name="Defense policy"/>
    <term name="Air piracy"/>
  </subjects>
  <summary>Authorizes the President to go get 'em.</summary>
</bill>

```

Expected behavior for various XPath expressions:

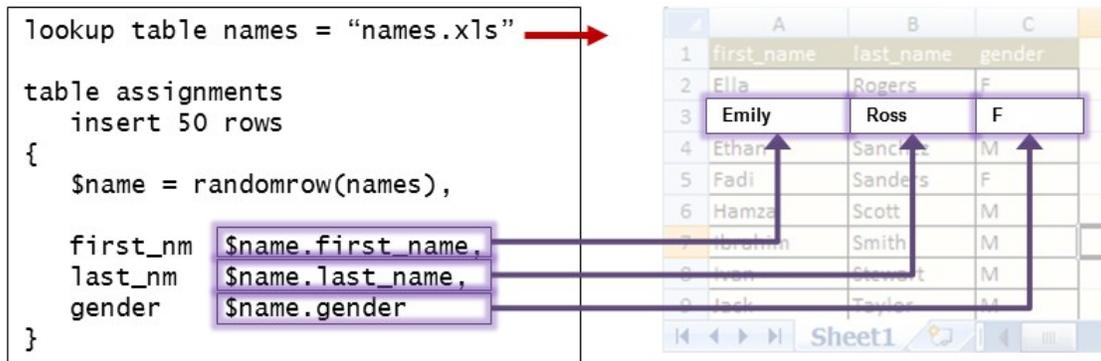
XPath	Sample Matched Value	xpath() value inserted into database
/bill/@session	107	107
titles	(nearly everything)	Error – need to match text or a text node.
/bill/summary	<summary> Authorizes the President to go get 'em. </summary>	Authorizes the President to go get 'em.
term/@name	"Defense policy" and "Air piracy"	"Defense policy" or "Air piracy"
/bill/sponsor	<sponsor id="300031" />	Error – no text matching this xml element

Both “unique” and “unique per parent” clauses are supported for xpath().

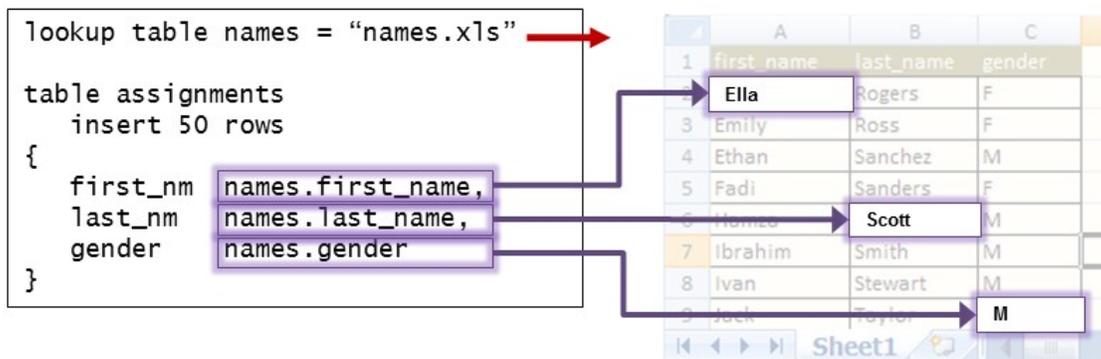
### 4.4.4 randomrow()

randomrow() will pick a row at random from the named table (or lookup table) and return a DataRoller-internal structure to hold the value for each column. This returned value is useful only with the “\$foo.bar” variable syntax. This is typically used when multiple values from another table need to be included and they all need to be from the same row. Use column() to pick multiple values from another table when the values do not need to be from the same row in the other table.

*Using randomrow() assures all values are from the same row:*



*Each column reference picks random values independently:*



The table named in randomrow() can be an alias for a lookup table defined at the top of the DataRoller file or the name of an actual database table. Because SQL does not have a built-in database-independent way to efficiently pick a random row from a database table, and because there is the potential for multiple tables that might need to access a table, the table named in randomrow() is read completely into DataRoller memory. Random rows are served from this in-memory cache of the table. Like column() above, this means that using randomrow() to refer to a table that is undergoing changes via DataRoller may lead to problems: the table is cached on first need. If DataRoller subsequently changes the in-database table, the in-memory cache of the table is not updated. Using randomrow() with a very large table, or a table containing large values for LOB columns may lead to memory exhaustion.

Both “unique” and “unique per parent” clauses are supported for `randomrow()`. Each occurrence of `randomrow()` with “unique” or “unique per parent” select their rows independently of each other, so adding a “unique” clause does not guarantee that both `randomrow()` occurrences are *mutually* unique.

Put another way, the following does not guarantee that a single row in table `race` won't have the same values for `$start` and `$end`. Variables `$start` and `$end` may both point to the same row in `location`, both coincidentally picked at random.

```
table race
  insert 50 rows
{
  $start = randomrow(location unique),
  $end   = randomrow(location unique),

  start_lat  $start.latitude,
  start_lng  $start.longitude,

  end_lat    $end.latitude,
  end_lng    $end.longitude
}
```

## 4.4.5 Previous Row

`DataRoller` maintains the last row it inserted to the current table so its values can be used by the subsequent row it inserts. There are two functions to accomplish this:

- `haspreviousrow()` which returns true if `DataRoller` has already inserted at least one row to this table (but see below for child tables)
- `previousrow(mycol)` which returns the value in column `mycol` from the previous row

As a best practice, use an “if” statement to separate what `DataRoller` should do on the first row it inserts, and for every other row it inserts:

```
table race_findings
  insert 3 rows
{
  place  sequence(), // 1st place, then 2nd place, ...
  person filerow("names.txt"),
  points if (haspreviousrow())
         previousrow(points) - random(1..5) // runners up below by 1 to 5
         else
         random(100 .. 120) // first place scored 100+ points!
         end
}
```

In the above example, a table holds the points scored by the top 3 contestants. To make the scores realistic, the first place holder has the most points, and each runner-up has between 1 and 5 fewer points. By using data keyed off of the previous row inserted, this assures that the third place winner doesn't have more points than the second or first place winners. Each value is computed to be a little less than the previous row `DataRoller` inserted.

DataRoller always resets the previous row to empty when it is about to insert the first row into any table. Put another way, the data from the last row of the previous table is not available to the first row of the next table in the DataRoller script file!

The real power of `previousrow()` is how it operates on a table with a “child of” clause: DataRoller resets every time the parent table row changes. Function `haspreviousrow()` is false exactly once for every unique row in the parent table.

To extend the example above, consider `race_findings` with a parent `race_schedule` table that contains a record for every race held. Function `haspreviousrow()` is false for the first row inserted into `race_findings` for each record in `race_schedule`.

```
table race_schedule
  insert 50 rows
  {
    race_id sequence()
  }

table race_findings
  child of race_schedule on this.race_id = parent.race_id
  insert 3 rows
  {
    place sequence(1 unique per parent), // 1st place, then 2nd place, ...
    person filerow("names.txt"),
    points if (haspreviousrow())
           previousrow(points) - random(1..5) // runners up below by 1 to 5
           else
           random(100 .. 120) // first place scored 100+ points!
  }
end
}
```

That’s it! Function `haspreviousrow()` is false for each row in `race_schedule` because DataRoller resets the previous row internally to empty whenever it moves to another parent record.

The table below contains data from the above DataRoller script. DataRoller resets the previous row to empty each time it begins a new race in table `race_findings` it

race_id	place	person	points
1	1	Chloe	101
1	2	Mary	99
1	3	Abdul	96
2	1	Chris	115
2	2	Arlene	111
2	3	Greg	98
3	1	Nate	105

previous row reset: first row

previous row reset: new parent

previous row reset: new parent

race_id	place	person	points
3	2	George	100
3	3	Simon	94

A great use for the DataRoller previous row feature is generating routing data for use in geospatial applications. Consider a data set of tagged elk that wander around the general Calgary area. The data set has a table for each tagged elk and tracks their position via a GPS signal in their collar tag. Their position is recorded each day for a two week period, and the results are charted on a map.

```

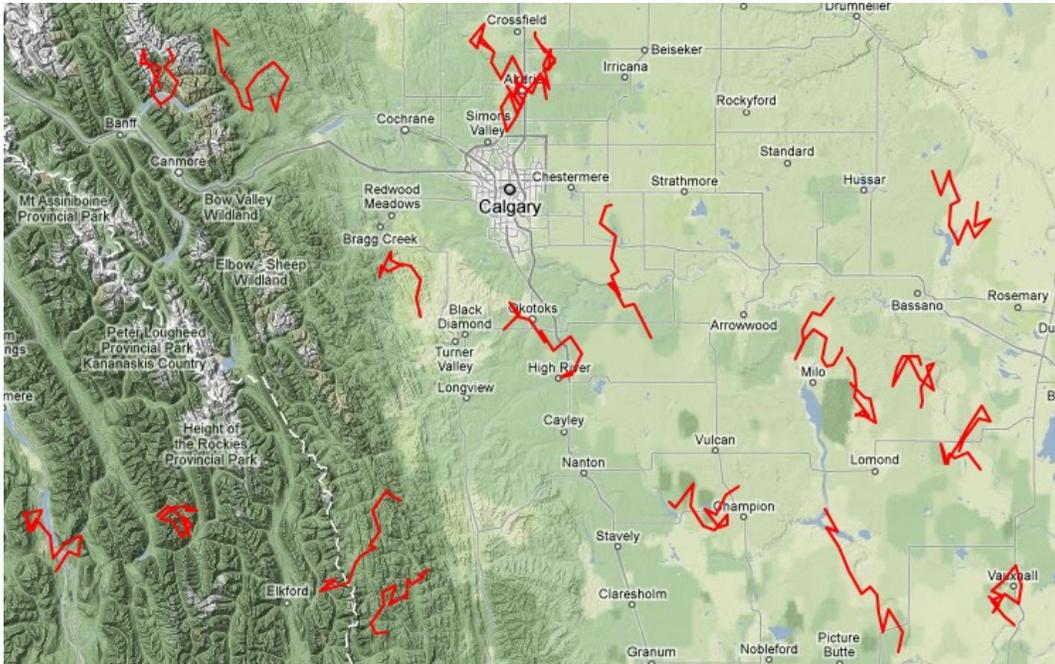
table elk_routes
  child of elk on this.elk_id = parent.elk_id
  insert 14 rows
{
  route_seq  sequence(1 unique per parent),
  latitude   if (!haspreviousrow())
             random( 50.0e0 .. 51.46e0)
             else
             previousrow(latitude) + random(-0.06e0 .. 0.06e0)
             end,
  longitude  if (!haspreviousrow())
             random(-116.2e0 .. -112.11e0)
             else
             previousrow(longitude) + random(-0.06e0 .. 0.06e0)
             end
}

```

Column `route_seq` holds numbers 1 through 14 for each tagged elk (parent table). This is the tracking day. Latitude and longitude for the first row of each elk is computed randomly near Calgary in the “else” clause above. Since DataRoller returns false for `haspreviousrow()` for the first row of each parent table row. For each subsequent day, DataRoller picks a latitude and longitude value that is up to 0.06 degrees away from the place that the elk was yesterday (previous row).

“`previousrow(longitude) + random(-0.06e0 .. 0.06e0)`” means pick the previous row’s longitude value and move up to 0.06 degrees west or east.

If you plot the `e1k_routes` table as lines on Google Maps, you would see something similar to this:



Map image generated from <http://maps.google.com>.  
Check out their excellent API at <http://code.google.com/apis/maps/>

## 4.5 SQL

The other Generators in DataRoller attempt to provide added value and abstract away implementation details to make writing project files fun for all. The `sql()` generator is the “trap door” of Generators: you pass in an arbitrary SQL statement to send to the database completely untouched by DataRoller.

```
table whatever
  insert 5 rows
{
  mycol    sql("select count(*) from othertable")
  ...
}
```

The statement itself can be any valid SQL statement provided that it returns a result set with tabular data (not an XML result set or a stored procedure that cannot be invoked with “run this query” semantics).

The `sql()` Generator will pick the first row returned and return the first column in the result set. If there are no rows returned, `sql()` returns a NULL value.

The `sql()` Generator can parse queries requiring bind variables, and bind value from the current row. Call `sql()` with a string query, “bind”, and a list of columns to bind:

```
table person
  insert 18 rows
{
  zip_code    column(rates.zip),

  age         random(21..90),
  price       sql("select price
                  from rates
                  where zip = ? and age_low <= ? and age_high >= ?"
                  bind zip_code, age, age)
}
```

In this example, imagine a `rates` table that has a record for every combination of zip code and age range, such as an insurance company might keep. A record of zip 90120 with age range 21-30 would have a record with a high rate.

Populate the `price` column for a person by looking up their rate in this table. Bind the current zip and age being generated so the value chosen is realistic.

`sql()` only supports using current column names in the table being processed for the “bind” clause. Variables may not be used in the bind clause.

Note: unlike other Generators, `sql()` will execute the query once for every single row inserted, with no regard for caching data. This Generator exists as a “bail-out” to take over manual control of retrieval when no other Generator will suffice. Because of this, `DataRoller` does not attempt to optimize or otherwise improve the execution time of `sql()`.

## 4.6 Operators

Operators are punctuation symbols used in infix notation to combine Generators together. They look and (mostly) act just like other typical programming languages, so just code what you think works and you’re probably right.

The following table lists all operators in precedence order with a quick reference description:

Precedence Order	Operator	Quick Description
1	<code>  </code>	Logical OR on Booleans only
2	<code>&amp;&amp;</code>	Logical AND on Booleans only

Precedence Order	Operator	Quick Description
3	!	Logical NOT on Booleans only
4	=== !== == !=	String equals (ignores case and whitespace) String not equals Logical equals between arbitrary types Logical not-equals between arbitrary types
5	< <= > >=	Less than Less than or equals Greater than Greater than or equals
6	+ - &	Addition for numbers, strings or dates Subtraction for numbers, strings or dates String concat with intervening whitespace
7	* / %	Multiplication or string repetition Division (integer or real-values) Modulus (valid only for integer numbers)

A few other important notes:

- Everything is left-associative:  $6 - 4 - 1$  is 1, not 3.
- Use parenthesis if you want to change the order of operations:  $6 - (4 - 1)$  is 3, not 1.

Operators with higher Precedence numbers from the table above will bind lower on the parse tree. Examples:

- $1 > 4 \ \&\& \ 5 \leq 6$  means  $(1 > 4) \ \&\& \ (5 \leq 6)$
- $! a \ \&\& \ b \ || \ ! c$  means  $( (!a) \ \&\& \ b) \ || \ (!c)$
- $1 + 2 - 3$  means  $(1 + 2) - 3$  since they are the same precedence level, the left-associativity rule applies to disambiguate them

If you're a real geek, check out the BNF reference at the end of this document for the gory details. DataRoller is a LL(1) grammar.

## 4.6.1 Types, Nulls and No Short-Circuit Logic

Operators do not follow short-circuit behavior. Each operator will evaluate all parts first, and then apply the operation. For example, “!a && b” will compute the value for a and b, then negate a, then apply the “&&” operator. Short-circuit logic is not used because DataRoller is particular about data-types and will not allow operators to operate with data types that are not valid. For example, “true || 5” is invalid, even though the left-hand side is a valid Boolean value and a short-circuit evaluation would have led to a final value of true.

For user-supplied functions, this lack of short-circuit evaluation may have side-effects when the right-hand side of a comparison is evaluated unnecessarily.

DataRoller is picky about data types applied to each operator. DataRoller does not perform type casting to any value to allow an operator to work correctly and will not implicitly convert any type to any other type. Using invalid values for an operator are flagged at run-time.

All operators have specific rules for handling null values. In general, DataRoller will operate over nulls when the operation has an obvious interpretation, such as “5 + null” (evaluates to 5) and will flag anything else as an error, such as “now - null”. See below for a table listing all null and other border cases.

## 4.6.2 Boolean Operators

Logical &&, || and ! (“not”) operate strictly on Boolean values and act exactly as you would expect them to when all portions are not null. All three operators do not permit nulls.

Examples:

Expression	Evaluates To	Comments
true && true true && false false && false	true false false	Exactly what you expect.
true    true true    false false    false	true true false	
! true ! false	false true	
null && true true    null ! null	Invalid operations	

### 4.6.3 Equality Operators

The DataRoller Equality Operators:

- == Equals
- != Not equals
- === String Equals
- !== String not equals

The equality operators “==” and “!=” operate over all data-types, including strings, Booleans and nulls. Unlike databases which consider “null = null” to be *false*, in DataRoller “null == null” is *true* so we don’t need a separate “is” operator to handle null cases.

Operator “===” is true only when both sides are strings and the contents are equal without regard for null, character case, leading or trailing whitespace.

Be careful when deciding between “==” and “===” and deciding between “!=” and “!==” (note the “=” count in each operator). The non-string operators “==” and “!=” are sensitive to string case and whitespace and consider null to not be the same as an empty string. The string operators “===” and “!==” might be a better choice for string comparison.

For floating point numbers, “==” and “!=” don’t generally get along that well. Like in many programming languages, round-off and approximations often will result in two numbers not being *precisely* equal. It is not uncommon for a number such as 123.456 being represented as 123.4555555999999973645 internally, and so they are not considered equal according to the “==” operator.

A better alternative to using “==” and “!=” operators directly, is to use the built-in function `fequals()`. Function `fequals()` will return true when the two arguments passed in are within an epsilon of each other. Given “`fequals(a, b, c)`” will compute the *distance* between a and b and return true when this difference is less than c. It just returns “`absolutevalue(a-b) < c`”.

Expression	Evaluates To	Comments
<pre> null == null null == 12 now == null </pre>	<pre> true false false </pre>	Null is equal to itself and nothing else.
<pre> 123 == 123 123 == 123.00 </pre>	<pre> true error </pre>	<p>Integer comparisons are obvious.</p> <p>Cannot compare an integer 123 with a decimal 123.00 values.</p>
<pre> " AB7t " == "ab7T" " AB7t " === "ab7T"  "Firefly" == "RICHMOND" " Firefly" === "FIREFLY"  "" == null "" === null </pre>	<pre> false true  false true  false true </pre>	<p>“===” can be used with strings, but they must be precisely equal, including case and whitespace.</p> <p>“===” is a good choice for strings for values that are codes, labels, passwords and other precise values.</p> <p>“====” is good for things like product codes.</p>
<pre> now == today </pre>	<pre> false (other than at precisely midnight) </pre>	Date comparison includes all portions of the date and time down to the second.

## 4.6.4 Comparison Operators

The order comparison operators “<”, “<=”, “>” and “>=” will operate over any data type including nulls, dates and Booleans. Null values are generally considered to be “below” non-null values, such as “null < -900”, “null < false” and “null < today” (for any value of today). “false” is less-than “true” for Booleans.

Like operators “==” and “!=”, floating point values may be compared but may fall prey to the same round-off and approximation errors. Negative infinity is less than every other floating-point number and positive infinity is greater than every other floating-point number other than the strange “not a number” (NaN) value. NaN is greater than every other floating-point number, including infinity.

Examples:

Expression	Evaluates To	Comments
<code>1 &lt; 86</code> <code>1.56 &lt; 1.57</code> <code>123.45 &lt;= 123.45</code>	<code>true</code> <code>true</code> <code>true</code>	Simple numeric comparison
<code>D'today' &lt; D'today'</code> <code>D'today' &lt; D'now'</code> <code>D'now' &lt; D'now'</code> <code>D'now' &gt;= D'now'</code>	<code>false</code> <code>true</code> <code>false</code> <code>???</code>	Today means midnight today, now means this very second.  Since “now” is computed to the second, be careful with any expression with more than one “now” in it! The first occurrence may not be precisely the same as the last!
<code>false &lt; true</code>	<code>true</code>	Although not generally considered to have an intrinsic ordering, <code>false</code> is less-than <code>true</code> .
<code>"hi" &lt; "hi there"</code> <code>"123" &lt; "56"</code> <code>"DEF" &lt; "abc"</code>	<code>true</code> <code>true</code> <code>true</code>	Case-sensitive and whitespace-sensitive comparisons.

## 4.6.5 Algebraic Operators

DataRoller supports the typical operators for performing arithmetic, including:

- “+” for addition
- “-” for subtraction
- “\*” for multiplication
- “/” for division
- “%” for modulus

Addition, subtraction, multiplication and division are supported for all numeric types with exactly what you would expect. Modulus is supported for integers and floating point numbers, but not decimals.

In addition to the above, there are special cases for dates and strings, and additional operations for different combinations of types:

Expression	Meaning	Example
$123 / 10$ $123.4 / 10.0$ $1.2e2 / 1e2$	$12$ $12.34$ $1.2e1$	<p>Division for integers throws out the decimal or remainder portion.</p> <p>Division for decimal numbers is precise and maintains all decimal digits.</p> <p>Division for floating point numbers maintains decimal values, but is inherently an approximation.</p>
$1.0 / 3.0$	Error	Decimal division that results in an infinitely-repeating value is not permitted: the Decimal data type is not an approximation. It must maintain a fixed number of decimal places, all of which are significant.
$123 \% 10$ $12.34 \% 10.0$ $12.34 \% -10.0$	$3$ $2.34$ $2.34$	<p>Integer MOD is just the remainder after division.</p> <p>Same for floating point numbers, but the remainder itself can have a decimal portion.</p> <p>On the left, note that the remainder after dividing by -10 is a positive number.</p>
$12 / 0$ $12.34 / 0.0$ $0 / 0$ $0.0 / 0.0$	Error	Like any other system, you cannot divide by zero.

### 4.6.6 String Operators

Strings are a fundamental type in wide use in databases, and so DataRoller defines several operators to make working with strings more readable than relying on separate functions to perform manipulation. In DataRoller, the following operators have special meaning for strings:

Expression	Meaning	Example
$string == string$ $string != string$	Precise, exact equality or inequality	$"ABC" == "Abc"$ is false $"ABC" == "ABC "$ is also false (note the trailing space character)

Expression	Meaning	Example
string === string string !== string	Equality/inequality, ignoring boundary whitespace and letter case	<pre> " ABC" === "abc " is true "a b c" === " a b c" is false </pre> (note internal whitespace differences)
string + string	Concatenation, preserving boundary whitespace	<pre> " 12 3 " + " abc </pre> evaluates to <pre> " 12 3 abc " </pre>
string & string	Concatenation after trimming all boundary whitespace from both sides and adding a single space between operands.	<pre> " 12 3 " &amp; " abc </pre> evaluates to <pre> "12 3 abc" </pre>
string - string	Remove the RHS string from the LHS string, if present	<pre> "cookies" - "cook" is "ies" "cookies" - "cake" is "cookies" "axbxcxdx" - "x" is "abcd" " h t " - " " is "ht" </pre>
string * integer	Repeat the string <i>n</i> times ( <i>n</i> is RHS value)	<pre> "ab" * 0 is "" "ab" * 1 is "ab" "ab" * 4 is "abababab" "ab" * -2 is "" </pre>
string < string string <= string string > string string >= string	All order comparison operators are simple lexicographic ordering. I.e., “A” before “B”, etc. according to UTF-16.	<pre> "foo" &gt; "bar" is true "" &lt; "1" is true "ABC" &lt; "abc" is true "Garcon" != "Garçon" is true "Niña" &gt; "Nina" is true </pre>

### 4.6.7 Dates

Only a few operators are supported for Dates, since most combinations of two dates with an operator are not generally self-evident. For example, subtracting January 6<sup>th</sup> from July 10<sup>th</sup> has no widespread, understood meaning. In general, rely on functions to operate over date values.

## Supported Operators for Dates:

Expression	Meaning	Example
date == date date != date	<p>Precisely the same moment in time down to the second (if present).</p> <p>If an operand has no time component, midnight is assumed.</p> <p>Be careful comparing a database value that has no time component to another database value that has a time component.</p>	<p>D'today' == D'today' is true (both assume midnight)</p> <p>D'today' == D'now' is false (other than at precisely midnight when the time component of "now" is also midnight)</p>
date + date date - date date & date date * date date / date date % date	Error	January 6th can't be added to June 10th.
date + integer date - integer	Add or subtract days to a date	D'today' + 1 means tomorrow D'today' - 1 means yesterday
date < date date <= date date > date date >= date	All order comparison operators are simple comparisons based on date or date time. "Less-than" (<) means the LHS occurs in time before the RHS.	D'today' < D'today' is false D'now' >= d'today' is true

## 4.6.8 Operator Summary Table

	Null	String	Integer	Decimal	Float	Date	Boolean
&&    !	Error	Normal logic					
== !=	Nothing is == to null except null	Exact equality including whitespace	Equality	Same date down to the second	Normal logic		
=== !==	null === null is true	Case- insensitive, whitespace- insensitive string equality	Error				
< <= > >=	null is less- than everything	Lexographic comparison	Typical order comparison				false < true

## 4.7 Conditionals

### 4.7.1 If then else

The “if” statement is patterned after typical programming languages with an expression enclosed in parenthesis that evaluates to a Boolean value, a “then” portion and an optional “else” portion. All “if” statements must be terminated with a trailing “end” keyword.

Keyword “then” may optionally appear after the Boolean expression, but has no intrinsic meaning.

The “else” keyword and following expression is optional. When omitted, it has a default value of null.

In DataRoller the “if” clause acts as an expression itself which can be combined with other expressions and operators to assemble new values. Because an “if” statement can be combined with other expressions, every “if” must have a matching “end”. Without such a terminator, it would be unclear when an “if” portion ended and subsequent operators and expressions began:

Consider the following incorrect DataRoller expression:

```
if (a == null) 5 else 6 + 8 // incorrect! Missing trailing "end"
```

It is unclear which of the following expressions the above was meant to be:

```
( if (a == null) 5 else 6 ) + 8 // evaluates to 13 or 14
if (a == null) 5 else (6 + 8) // evaluates to 5 or 14
```

The “end” is a syntactic marker that makes it clear where the “if” statement stops and any additional operators begin. The two expressions above can be written correctly as:

```
if (a == null) 5 else 6 end + 8 // evaluates to 13 or 14
if (a == null) 5 else 6 + 8 end // evaluates to 5 or 14
```

A note on side-effects: the conditional always evaluated, the “then” or “else” that isn’t applicable is not executed: this is important for user-supplied functions that might have side-effects or run-time considerations.

## 4.7.2 Case When

The “case” statement is patterned after the SQL “case” expression and follows the same general syntax and meaning. The “case” statement begins with keyword “case” and ends with keyword “end”. It must have one or more “when” clauses and an optional “else” clause at the end before the “end” keyword. “when” clauses start with keyword “when”, have a Boolean expression, keyword “then” and a closing value expression. There is no keyword to end a “when” clause. Like “if” statements, the “else” clause begins with keyword “else” followed by a value expression.

A note on side-effects: execution starts with the first “when” clause, which is always evaluated. If true, the “then” is evaluated. If the first “when” is false, the “then” is not evaluated. Execution starts with the first “when then” clause and continues executing only until it finds a “when” clause that evaluates to true. No further when clauses or then expressions are evaluated. If no when clauses evaluate true and there is an else clause, it is evaluated. If no when clauses are true and there is no else clause, the entire case statement evaluates to null.

This specific sequence of evaluation is important for user-supplied functions that might have side-effects or run-time considerations. In short, a case expression uses short-circuit evaluation.

For example:

```
table projects
  insert 80 rows
{
  $val = 42,
  col   case
        when $val > 89 then "R" // not true, try next "when"
        when $val == 42 then "S" // true, so use value "S"
        when $val > 6 then "T" // Not evaluated (even though it happens
                                // to be true).
        else "V" // Not evaluated: a "when" clause was true
  end
}
```

## 4.7.3 Conditionals Example

Conditionals in action:

```
table hospital_users ...
{
  username  column(people.name unique),
  role      choice("admin", "doctor", "nurse", "orderly"),

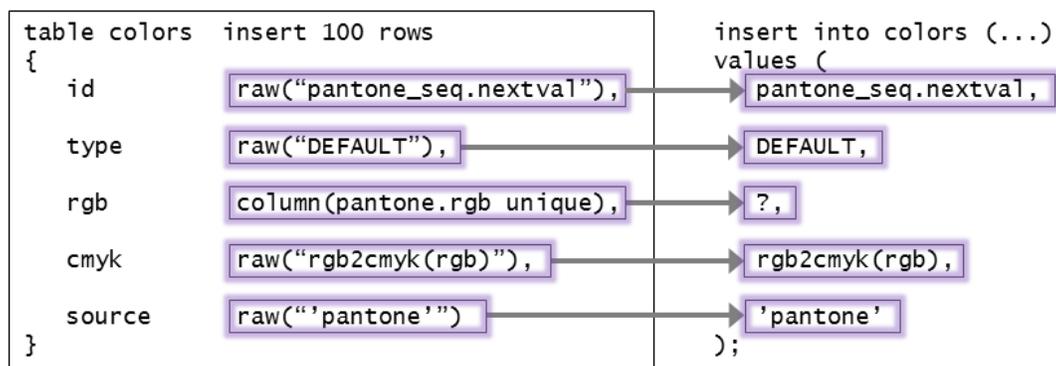
  // Only doctors and nurses have licenses
  lic_num   if (role == "doctor" || role == "nurse") then
             pattern("UU-NNNNN-U")
             end, // no else means null
  lic_exp   if (lic_num != null) then
             random(D'1985-1-1' .. D'today')
             end,

  emp_num   // Employee nums: 1st char is role abbrev, suffix is 1st 2 chars of lic
             substring(role, 0, 1) +
             "-" +
             format(sequence(), "00000") + "-" +
             case
               when role == "admin" then "00"
               when lic_num != null then substring(lic_num, 0, 2)
               else "x7"
             end
}
}
```

## 4.8 Raw()

Syntactically, `raw()` appears with other column definitions in a table section of a DataRoller project file, but it is handled differently: The string passed to `raw()` is used once, directly to build the SQL INSERT statement. All other Generators generate values that are bound to “?” placeholders in the SQL INSERT statement.

For example:



Whatever is passed to `raw()` appears directly in the SQL INSERT statement. In the example above:

- An Oracle sequence is used to generate unique values for `id`

- `DEFAULT` is a keyword in T-SQL that asks the database to use the default value defined for the column
- Database function `rgb2cmyk()` is called to convert the value in column `rgb`. Note that DataRoller does not allow “bind” in `raw()` like it does in `sql()`: `raw()` is used once to create the insert statement, so no generated values are available to it. Note that you can refer to columns in the current row, though!
- Constants can be passed to `raw()`, such as ‘pantone’ above. The single quotes are necessary for a string literal inside the SQL `SELECT` statement.
  - `raw(“pantone”)` won’t work because the double-quotes are DataRoller syntax elements. The word `pantone` would be taken as a column name by the database most likely.
  - `raw(“\”pantone\””)` won’t work either because some databases take double-quotes to mean column names, and `pantone` isn’t a column.

No other generators or expressions can be combined with `raw()`. `raw()` must be used solely by itself. For example, “`raw(“func(a, b)”) * 56`” is not a valid column expression because `raw()` is being combined with other elements.

## 5 Functions

---

Functions are transforms that operate over other generators, constants or other functions themselves. A function is a label with all arguments (potentially none) enclosed in parenthesis. Each argument is a constant or any Generator Expression. A function works the same way in DataRoller as most programming languages.

Example using function “upper()” that takes a single string and returns all letters in upper-case:

```
upper(trim(code) + “-ab-” + initials(xpath(“names.xml”, “/names/full”)))
```

Components of the above example:

Expression	Comment
code	Generator that looks up the value in another column for the current row.
trim(code)	Call function trim(), that removes leading and trailing whitespace to the column value above.
+ “-an-” +	Concatenate “-an-” with the results of trim() and initials().
xpath(“names.xml”, “/names/full”)	Generator that reads XML file names.xml and retrieves all text matching “/names/full”.
initials( xpath(“names.xml”, “/names/full”))	Function that takes an input string and returns the first letter of each word based on the usual word boundary characters. This takes the full name from xpath(), and retrieves only the initial letter of each word.

Some typical uses:

Example	Comments
<pre>first_name  filerow("firsts.txt"), middle_init pattern("U"), last_name  filerow("lasts.txt"), initials   initials(             first_name &amp;             middle_init &amp;             last_name)</pre>	Populate a column of a person's initials by pulling the values from the constituent name columns and calling <code>initials()</code> .
<pre>description  lorem(90..1000), title        substring(description,                     1,                     80)</pre>	Populate a large description column with a big generated string, and make the corresponding title column be just the first 80 characters to accommodate "title varchar(80)" setup.
<pre>release_year random(1994..2010), year_seq     sequence(), label       "NNP-" +             format(release_year, "000 0")             + "-" +             format(year_seq, "00")</pre>	Create "NNP-1999-01" and so on by combining the individual parts. If <code>year_seq</code> is 5, use <code>format</code> to convert it into a string with leading zeros. This converts 5 into "05".
<pre>table circ   insert 16 rows {   t  sequence(0.0e0 step 3.927e-1),   x  cos(t),   y  sin(t) }</pre>	Generate the Unit Circle. Parameter $t$ moves from 0 through $2\pi$ . 16 points over $2\pi$ radians, so step $t = 2\pi/16$ .

## 5.1 String Functions

`upper(string)`: string

Convert all lower-case letters to upper-case letters, leaving everything else as-is in the input string.

`lower(string)`: string

Same as `upper`, but convert all to lower-case.

`trim(string)`: string

Remove all white space characters at the beginning and end of the input strings, leaving all other white space alone.

`length(string)`: integer

Return the number of characters in the input string.

`replace(string, string, string): string`

Given the input string in the first argument, `replace()` searches for all occurrences of the second string. For all such matches, `replace()` substitutes the third string in the input string and returns the result of all such substitutions. The search string is a regular expression.

I.e., `replace("hooyaa", "[ao]", "ar")` returns "hararyarar" (I like pirates).

`substring(string, long, long): string`

Return the substring of the first argument, starting at the position of the second argument and ending at the third argument. The first character in the input string is position 0, so

`substring("smiles", 1, 4)` returns "mile".

This is permissive with the two string indexes. If the start is after the end of the string, this returns "". If end index is past the end of the string, the returned string just stops with the last character of the input string. If the start index is negative, it just assumes you meant 0.

`abbreviate(string, long): string`

Reduce the string passed in so its length is the second argument, or less if it is already smaller than the second argument.

`abbreviate()` adds "..." to the end of a string if it reduces its size.

`abbreviate("hithere", 10)` returns "hithere"

`abbreviate("abcdefg", 6)` returns "abc..."

If the second argument is  $\leq 0$ , this returns null.

There is a strange boundary condition when the second argument is 0 through 3. The abbreviation set of three periods cannot fit into a string of length 0 through 2, and abbreviating a string to exactly length 3 would just be "..."—a nonsense value. So, if the second argument is 0 through 3 no abbreviation symbol is used and the input string is truncated instead..

`capitalize(string): string`

Make the first letter a capital. `capitalize("hi")` returns "Hi". `capitalize("HEY")` returns "HEY"

`countMatches(string, string): integer`

Return the number of times the second argument was found in the first argument. The second argument is a plain string, not a regular expression.

`defaultIfEmpty(string, string): string`

Return the first string if it is not "", and the second string if it is.

`deletewhitespace(string): string`

Return the input string after removing every white space character.

`deletewhitespace(" a b c ")` returns "abc"

`initials(string): string`

Return a string by concatenating the first letter of each word in the input string.

`soundex(string): string`

Compute a single string that represents the general phonetic sequence of the input string. See “soundex” in various database guides.

Some important points:

Unpronounceable strings have a soundex defined, but they are not terribly useful, such as `soundex(“kxtgp”) = “K321”`.

Non-word strings return “” such as “123”.

`soundex()` operates over the whole string, not just the first word.

`nthtoken(string, string, long): string`

Break the first string into multiple tokens by splitting it via the regular expression in the second string. Once split, return the token at the index passed in the third argument (0 is first index).

```
nthtoken(“a:b:c”, “:”, -1) is “”  
nthtoken(“a:b:c”, “:”, 0) is “a”  
nthtoken(“a:b:c”, “:”, 1) is “b”  
nthtoken(“a:b:c”, “:”, 2) is “c”  
nthtoken(“a:b:c”, “:”, 3) is “”
```

If either string is null or empty, a null value is returned. If the third parameter is negative or larger than the number of tokens, an empty string is returned.

### 5.1.1 Function `pattern()`

Function `pattern()` takes a string describing how it should generate values, where certain letters indicate what the Generate should create. Upper case letters matching the template letters below are substituted. Every other character is output as-is. To escape a template letter, precede it with a backslash (“\”). To output a backslash, escape it with another backslash (“\\”).

Alternately, use single-quotes to enclose a series of characters that should be output without consideration for replacement. Leaving the input string without a matching closing single-quote isn't a concern. To output a single quote, escape it with a backslash.

Character	Meaning	Valid Values
U	Uppercase letter	A – Z
L	Lowercase letter	a – z
A	Alpha letter	A – Z and a – z
N	Number (digit)	0 – 9
B	Letter or number (“both”)	A – Z, a – z, 0 – 9
X	Hex digit	0 – 9, A – F

Note that nothing here handles NLS characters like “ñ”, or “ç”. This Generator is not designed to be used to generate words from a particular language. It is designed for generating codes and labels, which typically are restricted to basic letters, numbers and symbols in ASCII-7 only and tend to be fixed lengths.

Using `pattern(“UU-NNN”)` would generate “AB-304”, “ZU-011”, “GL-956” or “SS-270”.

To generate strings of variable length, use the “\*” operator to repeat a string a certain number of times, such as

```
pattern( “B” * random(5..9) )
```

which would generate between 5 and 9 characters, each a letter or digit.

## 5.1.2 Function guid()

Function `guid()` returns a value that will never be returned from any other call to `guid()` in the future on any other machine.

Typically, database tables have identifiers that are not unique beyond each specific table (like “`uniqueidentifier`” for SQL Server primary keys, or a separate sequence object for each table in Oracle. For cases where uniqueness needs to be global, across all table rows everywhere, a simple “1-up” integer will not suffice. GUIDs (a.k.a. UUIDs or uniqueidentifiers) are typically used.

The data-type actually generated by DataRoller is a *string* since each database supporting this concept has a vendor-specific type. For example, the Microsoft SQL Server JDBC driver handles `uniqueidentifier` as a `CHAR` and the JTDS SQL Server JDBC driver handles `uniqueidentifier` as a JTDS-specific internal type (`net.sourceforge.jtds.jdbc.UniqueIdentifier`) objects.

## 5.2 Numeric Functions

### 5.2.1 Integral Functions

`max(integer, integer): integer`  
Return the larger of the two arguments.

`min(integer, integer): integer`  
Return the smaller of the two arguments.

### 5.2.2 Floating-point Functions

`sin(float): float, cos(float): float, tan(float): float`  
Standard sine, co-sine and tangent trigonometric functions. Arguments are in radians, returning float values.

`asin(float): float, acos(float): float, atan(float): float`  
Arc-sine, etc. Accept float number and return result in radians.

`sinh(float): float, cosh(float): float, tanh(float): float`  
Hyperbolic trig functions. If you don't know what these are, one sentence here isn't going to help.

`exp(float): float`  
Returns e raised to the argument passed in where e is somewhere near 2.71828.

`pow(float, float): float`  
Returns the first argument raised to the second power. `pow(x,y)` is  $x^y$ .

`log(float): float`  
Returns the natural log (log base e) of the argument.

`log10(float): float`  
Returns the base-10 log of the argument.

`sqrt(float): float`  
Returns the square root of the argument.

`cbirt(float): float`  
Returns the cube-root of the argument

`ceil(float): float`  
Returns the next whole number larger than the argument, or the argument if it has no fraction.

`floor(float): float`  
Returns the greatest whole number smaller than the argument, or the argument if it has no fraction.

`max(float, float): float`  
Returns the large of the two arguments.

`min(float, float): float`  
Returns the smaller of the two arguments.

`abs(float): float`  
Return the absolute value of the argument (sign removed).

`signum(float): float`  
Returns -1 if value is < 0, 0 if value is == 0 and +1 if value is > 0.

`round(float): integer`  
Unlike `ceil()` and `floor()` above, `round()` will return an integer value (good for converting floating point numbers to integers!).

`toRadians(float): float`  
Returns the degrees argument converted to radians.

`toDegrees(float): float`  
Returns the radians argument converted to degrees. Convert from radians to degrees and degrees to radians.

## 5.3 Date and Timestamp Functions

`isSameDay(date, date): boolean`  
Returns true only if both arguments are not null and have the same year, month and day portions. Use the “==” operator to check if two dates have the same year, month, day, hour, minute and second.

`addYears(date, integer): date`  
Return the date passed in after moving it forward a number of years (or backward in time if the second argument is negative). Moving a date by years has no effect on the day, month, hour, minute or second components. Adding a year is not the same thing as adding 365 days: it will add 366 across leap years.

`addMonths(date, integer): date`  
See `addYears()` above: this returns the input date moved the number of months in the second argument without modifying the day, hour, minute or second components (the year may change).

`addWeeks(date, integer): date`  
See `addYears()` above: this returns the input date moved the number of weeks in the second argument. `addWeeks(x, y)` is the same thing as `addDays(x, y * 7)`.

`addDays(date, integer): date`

See `addYears()` above: this returns the input date moved the number of days in the second argument without modifying the hour, minute or second components (year or month may change).

Note that `addDays(d, 6)` is the same as using the “+” operator: `d + 6`.

`addHours(date, integer): date`

See `addYears()` above: this returns the input date moved the number of hours in the second argument without modifying the minute or second components (day, month and year may change).

`addMinutes(date, integer): date`

See `addYears()` above: this returns the input date moved the number of minutes in the second argument without modifying the second component (the other date parts may change).

`addSeconds(date, integer): date`

See `addYears()` above: this returns the input date moved the number of seconds in the second argument.

`truncateDate(date, string): date`

Return the input date after setting a portion of date or time fields to zero. Valid values for the second parameter are `year`, `month`, `day`, `hour`, or `minute`. Truncating to `minute` means setting the minute and second portions of the input date to zero. Truncating to `hour` means setting the hour, minute and second portions of the input date to zero (and so on). A null or empty string as the second argument is the same as “`day`”.

`roundDate(date, string): date`

Round the time portion of the date passed in. Put another way, return midnight of the date passed in if it is before noon, and midnight of the next day if the time is at noon or later.

## 5.4 System Functions

`systemProperty(string): string`

Return the result of calling `System.getProperty()`. Used mainly when passing arguments via “-D” to the command-line.

`getenv(string): string`

Return the environment variable (OS-dependent) with the name passed in, or null if not found (or the argument itself is null).

## 5.5 Cryptographic Functions

`encodeBase64(string): string`

Take the input string and compute an ASCII-7 “base-64” representation of it, returned as a string.

`decodeBase64(string): string`

Return a string from `encodeBase64()` above into the original string.

`md5(string): byte[]`

Return the MD5 digest as an array of bytes.

`md5Hex(string): string`

Call `md5()` and hex-encode the result, returned as a string.

`sha256(string): byte[]`

Similar to `md5()` above, compute the SHA digest of the input string and return it as a byte array.

`sha256Hex(string): string`

Call `sha256()` and Hex-encode the result, returned as a string.

## 5.6 Data-Type Conversion Functions

`format(any, string): string`

Take a raw number or date and convert it to a string via a spec passed in the second argument. I.e., take 1234567 (integer) and make it “1,234,567.00”. See below for details on how to construct a format string.

If 1<sup>st</sup> argument is null, `format()` returns null.

If the 2<sup>nd</sup> argument is null and the 1<sup>st</sup> is a date, `format()` returns the date formatted as an ISO date time (“yyyy-MM-dd'T'HH:mm:ss”).

If both arguments are null, this returns “”.

`date2long(date): Long`

Convert a date data into an integer value (Java type “long”) per Java’s `java.util.Date.getTime()` function. The value returned is the number of milliseconds since 1/1/1970 at midnight, UTC.

`long2date(long): date`

Convert an integer value into a Date data-type. The argument is the number of milliseconds since 1/1/1970 at midnight, UTC. Calling `long2date(null)` returns null.

`string2long(string): Long`

Convert the string passed in into a long. If the input string is empty or null, return 0.

`string2double(string): Double`

Convert the string passed in into a floating-point number (“double” in Java parlance). If the input string is empty or null, return 0.

`string2decimal(string): argh`

Convert the string passed in into a long. If the input string is empty or null, return 0.

`long2string(long): string`

Returns the argument as a base-10 string with optional leading hyphen if negative.

`double2string(double): string`

Returns the argument as a string with one or more digits, a period (“.”) and one or more digits. A leading hyphen is included if the argument is negative.

`decimal2string(decimal): string`

Returns the argument as a string using the same format as `double2string()` above.

`decimal2float(decimal): double, float2decimal(double): decimal`

Convert between floating-point numbers and (fixed-point) decimal numbers. Passing null to either function returns null.

Floating point numbers can be larger than the available precision for decimals or smaller than the available decimal precision. In these cases,  $+\infty$  or  $-\infty$  will be returned.

`string2date(string): Date`

Convert an input string in ISO8601 date-time or date format.

Date-only format: `yyyy-MM-dd`

Date-time format: `yyyy-MM-dd'T'HH:mm:ss`

The ‘T’ in the middle is a literal “T” character.

Year must be exactly 4 digits long.

Month, day, hour, minute and seconds can be a single or two-digit number only.

Hours are in 24-hour military time.

“2011-02-03T14:23:09” is February 3<sup>rd</sup> 2011 at hour 14 (2 PM), with 23 minutes and 9 seconds past the hour.

Passing a null or empty string will return null.

To convert strings in different date formats, use `parseDate()` instead.

`date2string(date): string`

The opposite of `string2date()` above, this formats a date passed in in ISO8601 format.

Passing a null date will return an empty string.

To format a date with a different format pattern, use function `format()` instead.

`parseDate(string, string): Date`

Like `string2date()` above, `parseDate()` parses an input string and returns a `Date` object according to a format. Function `string2date()` uses ISO8601 and `parseDate()` accepts a custom format string as its second argument. The format argument is passed to Java's `SimpleDateFormat` object. In summary:

Format	Meaning
YYYY	4-digit year
YY	2-digit year
MM	Month number
MMM	Month as three letter abbrev.
w	Week number in year
W	Week number in month
DDD	3-digit day in year
dd	2digit day of month
F	Day of week in month
EEE	Weekday as three letter abbrev.
a	AM/PM
HH	Hour in 00-23 notation
hh	Hour in 1-12 notation (use with "a" above for AM/PM)
mm	Minute in hour
SS	Second in minute
SSS	3-digit milliseconds in second

`int2long(Integer): Long`

DataRoller uses a Java "long" internally for nearly all computations and internal representations. However, in some cases, DataRoller reads data directly from a database to perform work. Use `int2long()` to convert a Java `int` into the more expressive `long` data-type.

There is no simple `long2int()` function provided by DataRoller, since this "narrowing" conversion may lose precision.

## 6 Execution

DataRoller is shipped with a basic Windows batch file “dataroller.bat” and a Unix shell script “dataroller”.

```
C:\dataroller> dataroller fireworks.txt
DataRoller 1.0

This program comes with ABSOLUTELY NO WARRANTY, is free software, and you
are welcome to redistribute it under certain conditions. See License.txt
for details. Copyright 2012 Rich Alberth.

suppliers           [.....] (8s)
materials          [.....] (14s)
fireworks          [.....] (30s)
firework_details   [.....] (27s)
Done (1m 19s)
```

In its simplest form, just pass the name of the DataRoller Project file to read in and execute. DataRoller will connect to the default database (more on that below), parse and load the Project file, load ancillary things like defined Lookup Tables and insert data into the database.

There is no special file type or extension for DataRoller files, and the command-line client does not care what you use.

DataRoller is written in Java, and a suitable Java virtual machine is not included with the distribution. DataRoller will use whatever “java” resolves to, based on your operating system. For example, it will search %PATH% on Windows and any alias or \$PATH on Unix. DataRoller will execute with any version of Java at 1.6 or newer. DataRoller has not been tested on Java 1.5 or any previous version.

### 6.1 Command-line Switches

If you ever forget, just type “dataroller -h”. The system will give you a dump of what you need to do.

Simple	Long name	Argument	Description
-h	--help		Print help message and do nothing else.
-d	--driver	Java class name	Fully-qualified name of a class already present in the classpath to load via “Class.forName()”. This is useful for loading database drivers not supported by DataRoller.

Simple	Long name	Argument	Description
-c	--url	JDBC connection string	Full JDBC connection string (different for each database vendor) holding all information necessary to reach the (possibly remote) database.
-u	--username	Username	Username to send to the database, if needed.
-p	--password	Password	Password for the user specified above, if needed.
-b	--batch	“true” or “false”	When true, DataRoller holds onto 50 rows and sends them to the database in one command to avoid being to “chatty” on the network.
-D	--property	name=value	Set a Java system property that can be used by a loaded JDBC driver class, or referenced within the DataRoller Project file itself.
-t	--trace	filename	Produce a huge, detailed trace log file with all the details about what DataRoller did and what it generated. Be careful when using this when your project file inserts a log of records: the trace file can grow to hundreds of megabytes or larger. DataRoller adds several trace rows for every column of every row of every table it generates.

A fully-specified command-line argument for a remote DB2 database:

```
dataroller -d com.ibm.db2.jdbc.app.DB2Driver -c jdbc:db2://dbsrv/MYDB;convertNull=1
-u rich56 -p DyerMaker -b false -Ddbtype=PROD myfile.txt
```

Project files that start with a dash are problematic since DataRoller will try to parse them as a command-line switch. To get around this, put “--” before the project filename. Every command line argument after “--” is taken as a filename. “dataroller -- --trace foo” tries to open file “--trace” and “foo” (causing an error since DataRoller will only process one project file at a time).

## 6.2 User Preferences and Aliases

There are a bunch of command-line switches that are difficult to remember and that you need to pass most of the time. To alleviate this problem, you can store all your often-used command-line arguments in a user preferences file and DataRoller will read this in automatically for you.

On Microsoft Windows platforms, the file is one of the following, depending on your OS:

- C:\Documents and Settings\*username*\DataRoller.prefs
- C:\Users\*username*\DataRoller.prefs

Use `~/ .dataroller` for Unix-based operating systems.

Contents of a user preferences file are “name=value” lines, blank lines or comment lines that start with a hash (“#”) symbol. Valid user preferences are “driver”, “url”, “username” and “password”. When included as-is, they provide defaults for their corresponding command-line arguments (long-form). When included with an alias prefix, they are only in effect when mentioned with the “-a” command-line argument. For example, “alias.prod.password=secret” is only in effect if “-a prod” is on the command-line.

DataRoller always takes arguments on the command-line first. If an argument does not appear on the command-line, it takes a user preference as specified in a “-a” argument. If no such preference exists, it falls back to a default user preference.

Consider the following `~/ .dataroller` file:

```
# Production server
alias.prod.url=jdbc:jtds:sybase://prodsrvr
alias.prod.password=boogieshoes

# Test server
alias.test.url=jdbc:jtds:sybase://testsrvr
alias.test.password=brickhouse

# default to laptop
url=jdbc:jtds:sybase://localhost
username=goofy
password=wordup
```

The command line:

```
$ dataroller -a prod -p secret myfile.txt
```

results in:

- The in-effect URL is “jdbc:jtds:sybase://prodsrvr” because it is not specified explicitly on the command-line and is specified in alias prod
- The in-effect username is “goofy” because it is not on the command-line and not part of the active prod alias, but is set as a default.

- The in-effect password is “secret” because it is set explicitly on the command-line. This overrides anything found in the user preferences file (alias or default entry).

The command line:

```
$ dataroller myfile.txt
```

results in “jdbc:jtds:sybase://localhost”, “goofy” and “wordup” being used since there is no alias (“-a”) and nothing is specified on the command line.

## 6.3 Loading JDBC Drivers

For DataRoller to connect to the database, it needs to know the driver class to load into memory (“--driver” option above) and the connection string (“--url” above). If these two do not match, DataRoller will not be able to load the database vendor’s Java code into memory and successfully connect to the database.

DataRoller ships with several database vendors supported, and understands how to connect to many others.

Database	Beginning of URL	Driver preloaded?
SQL Server	jdbc:jtds:sqlserver:	Yes
Sybase	jdbc:jtds:sybase:	Yes
HSQLDB	jdbc:hsqldb:	Yes
MySQL	jdbc:mysql:	Yes
PostgreSQL	jdbc:postgresql:	Yes
Derby	jdbc:derby:	Yes
Oracle	jdbc:oracle:	No
SQL Server	jdbc:sqlserver:	No
DB2	jdbc:db2:	No

To use a database above with a preloaded driver, simply use a URL based on the prefix above. For example, to connect to a MySQL database, use “jdbc:mysql:” as the start of the connection string, and DataRoller will load the MySQL driver Java code for you. See the reference section at the end of this document for details on JDBC URLs.

To use SQL Server or DB2, you will need to acquire the JDBC driver code from the vendor (we are not allowed to package it as part of the DataRoller product), put it into the lib folder in your DataRoller installation, and use it in your connection string directly. DataRoller knows to load the correct Java code based on the URLs listed above.

To connect to a database not listed above:

1. Download and put the driver jar file in the DataRoller lib folder
2. Use --driver to name the correct JDBC driver class (see documentation from the JDBC driver provider)
3. Use --url according to the driver documentation

### 6.3.1 SQL Server

There are two JDBC drivers for SQL Server in widespread use. Microsoft publishes their own, and there is a project on SourceForge.org called “jTDS” that provides a driver. The jTDS driver is included as part of DataRoller, and the Microsoft driver is not. To connect to SQL Server using the Microsoft driver, download a copy from <http://msdn.microsoft.com/en-us/sqlserver/aa937724.aspx> and use the URL prefix “jdbc:sqlserver:”.

### 6.3.2 Oracle

Oracle does not permit its JDBC driver to be embedded in other projects, so you will need to download the driver separately and put it into the lib folder. Get the driver with your OTN account (free sign-up) at <http://www.oracle.com/technetwork/database/enterprise-edition/jdbc-112010-090769.html>

### 6.3.3 DB2

Similar to SQL Server and Oracle above, download the driver from the IBM web site at <http://www.ibm.com/developerworks/data/library/techarticle/dm-0512kokkat/>. This page has a pointer to information on the Universal Database on where to download the Java Developer Kit for DB2.

## 7 Speeding up DataRoller

---

Databases generally are happiest when multiple transactions each make relatively small changes to the database. The database can maintain indexes, manage disk space, extend files as necessary and generally tidy up as it goes. DataRoller makes dramatic changes to the contents of a database, usually with few to no other concurrent users. Because of this, there are several things done by the database that can be adjusted to make DataRoller faster.

- Use less-costly DataRoller Generators
- Do not maintain all indexes after every single `insert` statement
- Regenerate statistics for tables that changed drastically
- Lock tables to prevent contention and/or delay writing certain data until the end
- Disable costly constraints

### 7.1 DataRoller Generator Relative Costs

There is a large difference between Generators in both run-time cost and amount of memory consumed. Generators that have the largest impact on speed and memory cost are discussed below. In general, DataRoller takes the stance that memory is cheap and long run-time is annoying. DataRoller will use memory to pre-compute, index or cache values in memory to make run-time faster.

#### *In general, think small*

For generating test data, there is rarely need to push the database to its limits by populating each column to their maximum values. For example, “`pattern(“u” * 4000)`” is probably overkill to populate a “`product_code varchar(4000)`”. Choose less data to populate if your goal is only to have some data for a test or demo.

Also, “`null n%`” is your friend: it takes no time or memory to support, and speeds up inserts by sending less data to the database. If you have a nulls-allowed column, slap on a “`null 25%`”.

#### *Avoid large range with unique or unique per parent with random()*

DataRoller pre-computes all possible values in the range, stores them in memory, and marks them used as they are retrieved for use. This means the memory is proportional to all values in the stated range. There is no speed degradation using `unique` or `unique per parent with random()`. Using `unique per`

parent with `sequence()` is of no concern. All sequences use a small, fixed amount of memory and have the same run-time cost.

“`random(1.0 .. 100.0 step 1.0 unique)`” needs to store 100 values in memory.

“`random(1.0 .. 100.0 step 0.1 unique)`” needs to store 1,000 values in memory.

“`random(1.0 .. 100.0 step 0.0001 unique)`” needs to store 1,000,000 values in memory.

### *Use `blob()`, `lorem()` only with small upper bounds*

The `blob()` and `lorem()` generators do not stream their values to the database. The entire value is computed in memory and sent to the database for each row.

Similarly, large files for `folder()` are not streamed. Each file is read into memory in entirety and sent to the database for each row.

### *Do not use excessively large look up files*

`folder()`, `filerow()`, `xpath()` and column references to tables via “lookup table” all read their file into memory in entirety. Using a very large file reference will take up a very large amount of memory.

For Excel files used in lookup tables, it is tempting to grab an entire large data-set (lots of rows and lots of columns) into a single, large Excel file, and use it across many DataRoller files. Unless all columns are used in each file, there is a waste. It takes much less room in memory to cache a file that contains only the columns needed for a particular DataRoller file.

Any reference file with dramatically more rows than the number of DataRoller table rows it is used for is a waste: look up data that isn't used.

For `xpath()`, XML files with dramatically more structure (attributes, element trees) are similarly a waste. For very large lookup XML files, DataRoller will run faster if the XML files are preprocessed with a XSLT transform to filter out unneeded elements and attributes. Beyond reducing the number of disk blocks to read and cache in memory, XPath queries run faster on simpler XML structures.

In general, it is better for quick execution to have a “home” folder with original versions of all look-up data, and pared-down versions of these files in each separate folder with a DataRoller project: this way, each project runs quickly.

### *Use `sql()` as a last resort*

The `sql()` generator looks powerful because it can encode any database logic by way of `select` statements. Unfortunately, because it has a user-supplied SQL `select` statement, DataRoller does not

cache any past-computed values, and will execute the SQL statement once for every row being inserted. This is a “bail-out” generator that relies on the database to compute each value independently.

Try to use a combination of other DataRoller Generators and functions to accomplish the same thing or simplify the logic being implemented.

As an alternative, consider “wrapping” the SQL statement within a User Supplied Function. A User-Supplied Function has an opportunity to combine both SQL logic and Java logic, and can pre-compute or cache values to speed up execution.

Alternately, compute all needed values in a SQL statement and store in a temporary table, and then reference the temporary table via `column()` instead of using `sql()`.

## 7.2 Rebuild Indexes

Databases maintain indexes by rebalancing B-Trees on disk and removing unneeded entries upon every database `insert`, `update` and `delete` statement. `select` statements execute dramatically faster at the price of mildly slower `insert`, `update` and `delete` statements. In a general-purpose database with many concurrent sessions, this is the best balance to keep the database efficient. When DataRoller runs against a database that is not under heavy load, there is no need to constantly manage indexes.

Dropping an index is a fast operation. Rebuilding an index from scratch after populating data is generally much faster than maintaining the index across every single `insert` statement. For B-Tree based indexes, there is no rebalancing that needs to occur: the entire tree is built in one-shot.

Database	Relevant SQL Statements
MySQL	<p>For non-unique keys in MyISAM tables:</p> <pre>alter table _____ disable keys alter table _____ enable keys</pre> <p>For other types, drop indexes and recreate them after DataRoller.</p>
Oracle	<p>For general indexes on a per-index basis:</p> <pre>alter index _____ unusable alter index _____ rebuild parallel nologging</pre> <p>For function-based indexes:</p> <pre>alter index _____ disable alter index _____ enable</pre> <p>Use <code>nologging</code> to avoid clogging up the redo logs since indexes do not contain domain data. Use <code>parallel</code> to rebuild indexes in parallel.</p>
SQL Server	<p>All indices on a particular table can be disabled and rebuilt together:</p> <pre>alter index all on _____ disable alter index all on _____ rebuild</pre>

Database	Relevant SQL Statements
DB2 PostgreSQL Sybase	Drop and recreate each index after DataRoller is done.

## 7.3 Regenerate Statistics

Some databases maintain rough metrics of tables and indexes so they can build efficient query execution plans. For example, it might be dramatically faster to loop over all records in a small table and look-up a matching row in a large table to execute an inner join clause. Looping over the large table and looking up matching entries in the smaller table might be an order or magnitude slower!

Unlike indexes, table and index statistics are generally not maintained automatically after every insert, update or delete statement. They are updated via scheduled jobs. If DataRoller is used to make dramatic changes to a database, the existing statistics might be a terrible match to reality. Use a DataRoller SQL statement to regenerate the statistics.

Database	Relevant SQL Statements
MySQL	<p>For MyISAM, BDB, InnoDB, and NDB tables:</p> <pre>analyze local table _____</pre> <p>The local keyword speeds up generation by not writing to the binary log. This is fine unless you need statistics replicated to other instances.</p>
Oracle	<p>Generate statistics on all tables and all indexes in an entire schema at a time (cascade means do the indexes too):</p> <pre>EXEC dbms_stats.gather_schema_stats(   'myschema', cascade =&gt; TRUE)</pre> <p>Be careful to only use this in a SQL statement after the last table in a DataRoller file since it applies to all tables and indexes in the schema.</p>
SQL Server	<p>TO generate statistics for a table and all indexes:</p> <pre>update statistics _____ with fullscan</pre>
PostgreSQL	<p>Analyze all aspects of a single table:</p> <pre>analyze _____</pre> <p>Analyze all tables in the current database:</p> <pre>analyze</pre>

As a general rule, read the documentation for your database to understand the implications of messing with statistics. For SQL Server, for example, regenerating statistics invalidates all execution plans: they will each need to be regenerated upon next use. SQL Server will regenerate statistics automatically when it needs, so this may be unneeded.

Statistics serve another important purpose during DataRoller execution: Letting the database take advantage of indexes that have been dramatically updated by DataRoller itself. When DataRoller loads a large amount of data into a table that previously had only a small number of rows, the statistics will typically show that the table has the previous small number of rows. Even if indexes are present and rebuilt, they may not be used if the database considers a full table scan of a small table to be more efficient.

Consider the following:

```
/* table products starts out with:
 *   100 rows
 *   single index on columns product_id and active
 */
table products
  insert 100000 rows
{
  product_id  sequence(),
  . . .
}

table items
  insert 10 rows
{
  product_id  products.product_id,
  active_prod_id  sql("select case active = 1 then product_id else null end
                      from products
                      where product_id = ?"
                      bind product_id)
}
```

Even though DataRoller will issue “select product\_id from products” once at the start of populating table `items`, this will likely be implemented as a single full table scan (which fetches all column data for every row) instead of a simple full index scan which is much faster. They both return the same data to DataRoller, but the latter reads dramatically less disk blocks into the database memory to fulfill the request.

Column `active_prod_id` is filled by executing the SQL query once for each of the 10 rows in table `items`. If the database thinks table `products` only has a few rows, it will perform a full table scan for each request. Once statistics are in place for table `products`, it will likely do an index unique scan and avoid the table entirely since the index has both `product_id` and `active` values in it!

As a general rule, if you are dramatically changing a table that is used later in the same DataRoller file, generating statistics might be a good idea.

## 7.4 Lock Tables

Check the documentation for your database for hints on lock management. This is database-specific, and not necessarily a good idea if there are any other connections possible when DataRoller is executing.

By way of example, inserts for non-transactional tables in MySQL are faster with:

```
LOCK TABLES _____ WRITE
...
UNLOCK TABLES
```

The index buffer is flushed to disk only once at the “unlock tables”.

Insert performance for transactional tables in MySQL can be improved simply by using explicit transactions:

```
START TRANSACTION
...
COMMIT
```

## 7.5 Disable Costly Constraints

Data integrity controls that are enforced by a database are a great safety net for applications to verify that all data remains consistent. They come with a run-time maintenance cost, of course. Disabling integrity constraints before loading data into a table and then re-enabling them after data loading is complete may not be a savings, however. Unlike index maintenance, incurring the cost of constraint maintenance for every insert statement may not be more than the cost for all rows at one time after all inserts are complete.

Typical integrity constraints:

- **Unique**: row inserts or updates look-up values in the current table (usually with index help)
- **Referential Integrity**: row inserts or updates look-up values in another table (hopefully designed with an index!)
- **Check**: Database-resident functions or general expressions limit acceptable values

With integrity constraints enabled, every row will evaluate each constraint and fail the insert if it does not succeed. If integrity constraints are disabled, then data is loaded, and then the constraints are enabled, the database likely will simply evaluate each constraint for every record in the table. If DataRoller deleted all rows before inserting new rows, both approaches likely will execute with the exact same amount of time. If DataRoller did not delete all rows before inserting new rows, you may be in real trouble: Enabling a disabled constraint will evaluate every row in the table, not just the new rows DataRoller inserted!

So, be careful about disabling any integrity constraints without thinking them through completely.

## 8 Extending DataRoller

---

### 8.1 User-Supplied Functions

If the myriad of wonderful generators and functions above would not be enough for every conceivable need, you can write your own functions and reference them inside a DataRoller project file.

In a nutshell:

1. Write a public static Java function and compile it.
2. Define where the class file can be found in the DataRoller project file.
3. Use your function like any other built-in function above.

Straight to a simple example: File “MyClass.java”:

```
public class MyClass {  
    public static String triple(String s) {  
        return s + "," + s + "," + s;  
    }  
}
```

File func.txt:

```
function triplestring = "mystuff.jar!MyFunction.triple(string)"  
table channel  
  insert 25 rows  
{  
  number      sequence(154),  
  code        pattern("UUU"),  
  label       triplestring(number + "-" + code)  
}
```

If number is 154 and code is “ABT”, label would invoke `MyClass.triple()` with argument “154-ABT”, which would return “154-ABT,154-ABT,154-ABT”.

## 8.2 Java Function

A function to be invoked by DataRoller must be:

- In a public class
- Be “public static”
- Be named using a combination of only letters, numbers, “\_” and “\$”
- Declare formal arguments that DataRoller can supply (see below)
- Not use the “...” formal parameter declaration
- Return a value usable by DataRoller (see below)

DataRoller will load the class and invoke the method using Java reflection. Any exceptions thrown while loading the jar file, loading the Class definition, or invoking the method will result in an error being generated by DataRoller, and the execution of DataRoller will stop.

## 8.3 Arguments and Return Types

Methods may, in fact, declare any Java type in their formal parameter list, and return any Java type they choose, even types unknown to DataRoller or JDBC. If the method being invoked is itself not being used as an argument to another generator or function, then the value returned from the method is used directly in inserting data into the database.

This unrestricted use of types is by design: one function may return an object that cannot be inserted into a database (the underlying JDBC driver cannot handle it). But, this value may be passed as a parameter to a different user-supplied method that does return a type JDBC can handle.

Eventually, the last function to be invoked that returns a value that DataRoller sends to the database must be one of the following types:

- `java.lang.String`
- `java.util.Date` (and `java.sql.Date`, which is a subclass of `java.util.Date`)
- `java.math.BigDecimal`
- `int` or `java.lang.Integer`
- `long` or `java.lang.Long`
- `double` or `java.lang.Double`
- `byte[]`

## 8.4 Function Alias

Each user-supplied function to be invoked by DataRoller must be declared at the top of the Project file. The function declaration includes an alias you use in the Project file, and the location where the method can be found (jar file and class it lives in).

Sample:

```
function tripleStr = "mystuff.jar!MyFunction.triple(java.lang.String)"
```

“tripleStr” is the alias that can be used inside DataRoller Project files. Because the name of a Function is just a label, any functions that might conflict with DataRoller keywords (supplied or user-created) can be enclosed in square brackets to distinguish them. This is the same behavior for table names and column names above. No built-in functions have the same name as a DataRoller keyword.

For example:

```
function myfunc = "a.jar!MyCls.myFunc(java.lang.String) // No prob, not a keyword
function xpath = "a.jar!MyCls.xpath(java.lang.String) // nope, xpath is a keyword
function [xpath] = "a.jar!MyCls.xpath(java.lang.String) // brackets make it a label
function myxpath = "a.jar!MyCls.xpath(java.lang.String) // or pick a different name
```

If you feel the need, the same function signature after the “=” can be listed more than once in a DataRoller Project file as long as they have different aliases. In fact, every function declared in a DataRoller Project file must be unique. This extends to built-in functions too, so for example you cannot declare a function with alias “pow” since there is already a built-in function named “pow”.

## 8.5 Function Signature

Everything after the equals (“=”) is the location where DataRoller will look for the file.

The location is split by a “!”: everything on the left-hand side is a Java archive file to load. Everything on the right-hand side is the full package, class name and method name signature, usable by Java reflection. A signature is:

1. Package name
2. Class name (case counts!)
3. Method name
4. List of argument types separated by commas, surrounded by parenthesis. If the function takes no arguments, use “()”.

This syntax is similar to the file path strings used in generators like `filerow()`, `folder()` and `xpath()`. The left-hand side of the “!” cannot be a zip file—only a jar file. The right-hand side is a package-class-method signature, not the name of a file within the jar file.

The hardest part to get right is the argument types in the parentheses. The formal argument list must be the exact data types declared in the function itself, including the complete package name of each argument. “String” is invalid, but “java.lang.String” is not.

Several commonly-used Java types have DataRoller shortcuts available:

If the formal Type is:	You can use the following shortcut instead to save space:
<code>java.lang.Object</code>	<code>object</code>
<code>java.lang.String</code>	<code>string</code>
<code>java.util.Date</code>	<code>date</code>
<code>java.math.BigDecimal</code>	<code>decimal</code>
<code>java.lang.Boolean</code>	<code>boolean</code>
<code>java.lang.Byte</code>	<code>byte</code>
<code>java.lang.Short</code>	<code>short</code>
<code>java.lang.Integer</code>	<code>integer</code>
<code>java.lang.Long</code>	<code>long</code>
<code>java.lang.Character</code>	<code>character</code>
<code>java.lang.Float</code>	<code>float</code>
<code>java.lang.Double</code>	<code>double</code>
<code>byte[]</code>	<code>byte[]</code>

Note the “byte[]” syntax is not what you might think. In the eyes of Java Reflection, an array of bytes does not have a string type of “byte[]”. The actual name of a byte array is “L[byte”, which is pretty cryptic. Since `blob()` returns a `byte[]` value, DataRoller supports the shortcut “byte[]” in a function signature. No other array types are supported in this way, so to declare a function taking an array of strings, you cannot use “string[]” or “java.lang.String[]”. This must be included using the “raw” notation of “L[java.lang.String”.

Primitive types can be referenced by name directly: the following are all permitted, “boolean”, “byte”, “short”, “int”, “long”, “char”, “float” and “double”. Similarly, object wrappers for primitives can be used unqualified: “Boolean”, “Byte”, “Short”, “Integer”, “Long”, “Character”, “Float” and “Double”.

## 8.6 Invocation and Execution

The class named in the function declaration is loaded once into memory, which means any class-level static initializers will be executed once before the method is called the first time, and then never again. Since methods named in function declarations must be static, no constructors or instance-level initializers will be executed.

User-supplied functions do not need to be thread-safe since DataRoller executes completely within a single thread.

However, there is no reason why a user-supplied function cannot take advantage of multi-threading if it will speed up overall execution. User functions may create and manage their own threads, connect to external resources, or do anything else they like. Be cautious with threads: non-daemon threads will not be stopped automatically when DataRoller wants to quit, and the JVM process will remain running. For safety, create all threads with `setDaemon(true)`.

## 8.7 For Example

Consider an application that performs operations on small images, similar to Gimp or Picasa. Images are stored in the database in `BINARY` columns. You need to give a demo of the application and need to generate lots of rows with images and various transforms of them.

You have a Java library that can perform manipulations (maybe the application you are demoing itself!).

DataRoller is not built to manipulate images, but you have the code to do it. The library operates over `java.awt.Image` objects, which JDBC cannot understand, even though it will persist these images as `BINARY` values (byte arrays in Java).

The library has lots of methods, including these:

```
public class vendorUtil {
    public static Image removeRedeye(Image i)           { ... }
    public static Image desaturate(Image i, float percent) { ... }
    ...
}
```

All of these methods can be directly invoked by DataRoller because they meet the criteria above, but they operate over object types (“`java.awt.Image`”) that JDBC does not understand.

Solution:

1. Write methods to convert from the database types for images (“byte[]”) into `java.awt.Image` and back so the library can be invoked, and
2. Rely on the feature of DataRoller that a user-supplied function can accept any type and return any type, provided the final invocation results in something JDBC can understand.

File `MyImgUtil.java`:

```
public class MyImgUtil {
    public static Image bytesToImage(byte[] data) { ... }
    public static byte[] imageToBytes(Image i) { ... }
}
```

To use these conversion functions in DataRoller, you “wrap” them in one of your conversion routines above. DataRoller will happily pass around Image objects between function invocations:

File `gimpish.txt`:

```
function removeRedeye = "vendor.jar!VendorUtil.removeRedeye(java.awt.Image)"
function desaturate    = "vendor.jar!VendorUtil.desaturate(java.awt.Image, float)"
function bytesToImage = "mystuff.jar!MyImgUtil.bytesToImage(byte[])"
function imageToBytes = "mystuff.jar!MyImgUtil.imageToBytes(java.awt.Image)"

table user_pics
  insert 10 rows
{
  username      pattern("L") + lower(filerow("lastnames.txt")),
  raw_img       folder("samplepics.zip"),
  cleaned_img   imageToBytes(
                  desaturate(
                    removeRedeye(
                      bytesToImage(
                        column(raw_img))),
                    63),
                  5.6e1))
}
```

Here’s the debrief:

1. The vendor library operates on Image objects, we write some conversion functions to convert to/from byte[] which JDBC can persist in BINARY columns.
2. `raw_img` is pulled from actual images stored in a “demo” zip file called `samplepics.zip`. `folder()` is a Generator that stores the file contents in DataRoller as byte[].
3. `column(raw_img)` just retrieves this byte[] used in the other column.
4. `bytesToImage(column(raw_img))` converts this byte[] from `raw_img` into a `java.awt.Image` object, which DataRoller happily keeps in temporary memory and hands it immediately to `removeRedeye()` below.
5. `removeRedeye()` is passed the Image object from above and 63 and returns another Image object, which is passed to `desaturate()` below.

6. `desaturate()` is passed the `Image` from above and `5.6e1` and returns a new `Image`, which is passed to `imageToBytes()` below.
7. `imageToBytes()` takes this final image and converts it to a `byte[]`, which `DataRoller` uses as the value for column `cleaned_img`. A `byte[]` can be persisted by `DataRoller`.

## 9 Tips & Tricks

---

- List a table multiple times to match your inheritance hierarchy.
- Have a skeleton file with placeholders, and run it through a macro processor like `cpp` or `m4`. Pass this pre-processed file to DataRoller.
- “Sanitized” production data: “`select mycol from mytable`” from your production database into a text file, use it to generate sample data: each column has things that look darn close to production without actually being production data: “production data” is usually the combination of values on a particular row that contain business value, not the individual values themselves.
- Hit up the Internet for publicly available datasets and reference them as-needed...no sense in generating your own values.
- Call DataRoller from a Scheduled Job or cron job. Get really fancy and link the DataRoller Java code as an external procedure / data blade / whatever to the database back-end, and invoke it via Agent or other database-resident scheduling mechanism.
- Run DataRoller and then “script” the database so you have offline insert statements you can replay later on systems without DataRoller.
- Many applications have a data-services layer in them (like a Service or DAO interface) that performs low-level data services like making sure a denormalized database is consistent. Load methods from this layer as functions in DataRoller and invoke them directly when inserting new rows!
- There are plenty of folders on your local system with large, binary files you can use for databases. I point DataRoller at my music folder: each file is around 1-6 MB and certainly binary!
- Use database linking in your database to point to another database. Reference these foreign tables as lookups for DataRoller.
- Create a “wrap-around” sequence that counts up to 100 and then starts over at 1:

```
sequence(0) % 100 + 1
```
- In a pinch when other constructs do not work, you can always use `choice()` with booleans to execute a block only around 33% of the time.

```
if (choice(true, false weight 2))  
...
```

### 9.1 Using Delete for Row Partitioning

The usual use for DataRoller is to fill an empty database so you can “move in” and do things with it. This calls for “`delete all`” on the tables so you can just run the DataRoller Project file over and over again, always resetting back to something you love, with no trace of what was there before.

DataRoller can be setup to leave existing data alone and only insert and remove data generated by DataRoller. Consider a typical database with artificial primary keys in “int” columns. Extract production data and insert into your testing database, and augment it with DataRoller data:

1. Figure out the highest, reasonable primary key value across all tables. Assume 30,000 for now.
2. Pick a value much higher than this as the “base” for all DataRoller data, such as 100,000.
3. Insert new rows starting at this base number with “sequence(100000)” so there will not be any number clashes, and so it will be easy to tell these apart from bona fide rows.
4. Use “delete where” with a clause that deletes all values at the base or higher, such as:  
delete where “item\_id >= 100000”

Use a “where” clause on the “child” clause to only attach child table rows to parent rows that came from DataRoller:

```
child of widgets on this.widget_id = parent.widget_id
where “widget_id >= 100000”
```

## 9.2 Dealing with Artificial Primary Keys

Passport numbers and car V.I.N. numbers are examples of natural keys. Using a natural key as a primary key of a database table makes sense on the surface. However, databases rarely allow a primary key to be altered, since there usually are foreign keys in other tables that prevent it. Natural keys generally don’t change, but it is not generally wise to forbid this by the design of a database. To get around this, a best practice is to make all primary keys be numbers that have no relation to the data in the table, and mark the natural keys as unique. Each database vendor has a vendor-specific way of handing out unique numbers for artificial keys in a way that two concurrent inserts on the same table result in different values.

Since most database vendors support automatically generating artificial primary key values by omitting the primary column in insert statements, DataRoller scripts may simply omit the primary key entirely to let the database do the work. However, the generated primary key value will not be available within DataRoller while generating other column values for each row.

If the database is well-designed with no relationships between an artificial primary key and any other column value, this limitation is rarely a nuisance. However, if the primary key is automatically generated but not an artificial key, there may be a need to use the generated primary key value in computing other columns in each row. Each database vendor has a separate syntax to support this reference.

The vendor-specific sections below each discuss the following common topics:

- How to rely on the database to generate key values directly
- How to bypass automatic database key generation
- How to reset the database if the automatic generation mechanism is bypassed

Below are examples of the patterns outlined above.

Implicit Key Generation: no access to generated column:

```
table passports
  insert 100 rows
{
  // pass_id omitted, so database will generate it automatically
  prefix substring(pass_id, 0, 3) // Error! "pass_id" not defined in
DataRoller!
```

Explicit Use of Database Key Generator: DataRoller insert statement calls key generation functions:

```
table passports
  insert 100 rows
{
  pass_id raw("passports_seq.nextval") // invoke DB-resident object to gen
erate key
  prefix substring(pass_id, 0, 3) // Error! Pass_id not available
here
```

Explicit Key Generation: DataRoller access to generated column:

```
table passports
  insert 100 rows
{
  pass_id sequence(100000), // explicitly generated, bypassing
// DB generation
  prefix substring(pass_id, 0, 3) // works!
```

When bypassing the database-provided automatic key generation mechanism, the database may need to be “reseeded” with the next available value for subsequent insert statements to work correctly.

## 9.2.1 MySQL

Adding property `AUTO_INCREMENT` to a column will automatically generate a unique number when the column is omitted from an `insert` statement. The simplest way to use this is to not include the `AUTO_INCREMENT` column in DataRoller and let the DB handle it directly.

If a column is listed explicitly in an `insert` statement that has `AUTO_INCREMENT`, the database will use the supplied values instead of constructing values implicitly. There is no special syntax or consideration to turn off `AUTO_INCREMENT`.

If DataRoller explicitly supplies all values for a column marked `AUTO_INCREMENT`, consider resetting the `AUTO_INCREMENT` value for a table so subsequent inserts (possibly outside DataRoller entirely) will work correctly:

```
ALTER TABLE employees AUTO_INCREMENT = 100
```

Obviously pick a number much greater than the largest value DataRoller would generate.

```

table employees
  insert 400 rows
{
  emp_id    12345 + sequence(),
  . . .
}
sql "alter table employees auto_increment = 12746" // 12345 + 400 + 1 extra

```

## 9.2.2 SQL Server

Adding keyword `IDENTITY` to a column will automatically generate a unique number when the column is omitted from an `insert` statement. The simplest way to use this is to not include the `identity` column in DataRoller and let the DB handle it directly.

Explicitly listing a column in a SQL `insert` statement that is marked `IDENTITY` will cause an error. For SQL Server, the `IDENTITY` mechanism must be turned off for a table before the column can be used in insert statements with the `SET IDENTITY_INSERT` SQL statement. For safety, disable `IDENTITY_INSERT` only while inserting rows with explicit key values, and re-enable it when done:

```

sql "set identity_insert employees off" // allow explicit key values
table employees
  insert 400 rows
{
  emp_id    12345 + sequence()
  . . .
}
sql "set identity_insert employees on" // re-enables key generation

```

There is no need to re-seed the identity mechanism for SQL Server tables. The database will automatically generate a unique value for `IDENTITY` columns after re-enabling `IDENTITY_INSERT`.

## 9.2.3 Oracle Sequences

Unlike MySQL and SQL Server, Oracle uses a separate database object to generate unique values: sequences.

Typical use: create a Sequence along with each table to supply artificial primary key values:

```

create sequence cars_seq;
create table cars(
  car_id    number(8) primary key, // use cars_seq to supply values
  ...
);

```

Use function "nextval" on the sequence in each `insert` statement:

```

insert into cars (car_id, ...) values (cars_seq.nextval, ?, ?, ? . . .)

```

Because a separate database object is used to supply unique values outside the table itself, use `raw()` to invoke the sequence directly.

```
table cars
  insert 400 rows
{
  car_id    raw("cars_seq.nextval"), // use cars_seq to supply values
  ...
}
```

This is the simplest invocation to generate primary keys with Oracle. Unlike other databases that use a marker on the actual primary key column such as `IDENTITY` for SQL Server, Oracle has no table-enforced constraint. So, any value may be used for a primary key column even if there is a sequence created for the purpose. In Oracle, sequences are not tied to primary key columns to forbid clients from inserting their own explicit values at will.

If the database sequence is not used (no `raw("____.nextval")` invocation), the next application to try to insert a new row will cause a primary key violation since the Sequence will likely generate a number that DataRoller already inserted. When done inserting new rows via DataRoller, consider setting the Sequence so it will continue to generate unique values. There is no support for altering a Sequence and setting the next value it will return, so the simplest solution is to drop the sequence and recreate it. Note that DataRoller is often executed with an account that does not have DDL permission (permission to alter database objects), so this may be a concern.

Reset a sequence after explicitly inserting new data:

```
table employees
  insert 400 rows
{
  emp_id    12345 + sequence() // no need to use employees_seq here
  ...
}

sql "drop sequence employees_seq"
sql "create sequence employees_seq minvalue 12746" // 12345 + 400 + 1 extra
```

## 9.3 Avoid Querying Unneeded Data

When faced with a parent table (`"child of"` clause) or a reference to another table such as `column(tbl.col)` or `randomrow()`, DataRoller caches the entire target table in memory for efficiency. The advantage to this approach is that DataRoller can execute faster with all data in memory. The obvious disadvantage is the amount of memory needed to cache large referenced tables.

To make this as efficient as possible:

- Reduce the number of rows available in the target table
- Only query and cache the columns referenced in the DataRoller file.

For example, consider the following table and related DataRoller file:

```
create table periodicals (
  id          int,
  title       varchar(80),
  publisher_id int,
  product_id  int,
  issue_number int,
  issue_in_pdf BLOB
);
```

Each periodicals record has a few small foreign key columns plus a huge JPEG value of the entire issue in PDF form. Obviously avoiding the BLOBs would speed up DataRoller.

```
table orders
  child of periodicals on
    this.periodical_id = parent.id
  insert 10 rows
{
  title      parent.title
  ...
}
```

DataRoller will pull all records from periodicals table above, loop over each, and insert 10 records into orders for each. DataRoller pulls every column even though only id and title are referenced above.

To avoid caching every cover image and issue in PDF format, create a temporary view that only selects the columns used in the DataRoller file:

```
sql "create view periodvw as
  select id, title
  from periodicals"

table orders
  child of periodvw on
    this.periodical_id = parent.id
  insert 10 rows
{
  ...
}

sql "drop view periodvw"
```

DataRoller will pull all columns from the view instead of from the periodicals table, which is much faster and requires less memory to complete since the BLOB values are not retrieved.

If the database user does not have permission to create objects such as views, consider a temporary table built from a similar select statement:

```
sql "create temporary table periodvw
  select id, title
  from periodicals"
```

## 9.4 Mutually Unique randomrow()

A table that tracks motorcycle patrols from a local police station has a row for every patrol. A patrol has two motorcycles from the vehicle pool and two officers, referred to as the “lead” and “tail” riders. Obviously one officer cannot ride two bikes, and both officers cannot share a single bike. A patrol cannot be the same officer twice.

```

table motorcycle_patrols
  insert 50 rows
{
  patrol_id          sequence(),
  lead_cycle_vin     column(motorcycles.vin unique), // Not mutually unique!
  tail_cycle_vin     column(motorcycles.vin unique),
  $leadRider = randomrow(officers unique),           // Not mutually unique!
  $tailRider = randomrow(officers unique),
  lead_officer_id    $leadRider.id,
  lead_officer_badge $leadRider.badge,
  tail_officer_id    $tailRider.id,
  tail_officer_badge $tailRider.badge
}

```

The above first attempt looks good, but there is a problem: each Generator with “unique” works independently from all other Generators. So, `lead_cycle_vin` might be the same value as `tail_cycle_vin` for the same row and `$leadRider` may be the same row in `officers` as `$tailRider` for a particular row in `motorcycle_patrols`.

DataRoller has no support for assuring that two columns are mutually-unique. One way to get around this problem is to have separate views into table `motorcycles` and `officers` for each portion that has to be mutually unique. Define the views such that they partition the underlying table roughly into two equal portions such that someone looking at the data wouldn’t be able to see the algorithm obviously (like putting all male officers into one partition and all females into the other).

For the above examples, the badge numbers and ID fields can be used with a simple algorithm like “divisible by 2”. Since VIN is a character field, look at the last character instead.

Better solution:

```
/*
 * PARTITION the officers table into two groups with no items in both and no items
 * in neither. Same for motorcycles table.
 * For officers, just MOD the numeric ID.
 * For bikes, pull the last VIN character, convert it to ASCII, and MOD it.
 */
sql "create view officer1 as
      select id, badge
      from officers
      where mod(id, 2) = 0"

sql "create view officer2 as
      select id, badge
      from officers
      where mod(id, 2) = 1"

sql "create view bikes1 as
      select vin
      from motorcycles
      where mod(ascii(right(vin, 1)), 2) = 0"

sql "create view bikes2 as
      select vin
      from motorcycles
      where mod(ascii(right(vin, 1)), 2) = 1"

table motorcycle_patrols
  insert 50 rows
  {
    patrol_id          sequence(),
    lead_cycle_vin     column(bikes1.vin unique),      // Not mutually unique!
    tail_cycle_vin     column(bikes2.vin unique),
    $leadRider = randomrow(officer1 unique),          // Not mutually unique!
    $tailRider = randomrow(officer2 unique),
    lead_officer_id    $leadRider.id,
    lead_officer_badge $leadRider.badge,
    tail_officer_id    $tailRider.id,
    tail_officer_badge $tailRider.badge
  }

sql "drop view officer1"
sql "drop view officer2"
sql "drop view bikes1"
sql "drop view bikes2"
```

To DataRoller, this “fake out” just looks like four separate tables that it reads in and caches in memory separately. The randomrow() and column() entries serve out values from each table uniquely.

# 10 For Reference

## 10.1 Project Syntax Reference

Data Source	Syntax	Meaning
Integer	12 random(5..20) random(0 .. 10000 interval 10) random(5..20 unique) sequence(5 step 8) sequence(1) sequence()	Literal 12 for every row Random value, step 1 implied Random value divisible by 10 Random, unique Sequence from 5 up by 8s Sequence from 1 up by 1s Special case: “sequence(1)”
Real	-90.77 random(5.6 .. 7.9 step 0.1) random(5.6 .. 7.9 step 0.1 unique) sequence(7.4 step 0.001)	Real literal Random value in a range Same as above, remove dups Sequence of values (step required)
Date	D'2010-4-6T11:00' random(D'1900-1-1' .. D'2010-1-1' step /2 days/) random(D'1900-1-1' .. D'2010-1-1' step /2 days/ unique) sequence(D'1974-6-12' step /1 month/)	Date literal (with time component) Random value spaced 2 days Same, no dups Incremental dates, every month
List of values (not just strings!)	choice(1, 2, 3) choice("a", "b", "c") choice( D'today'      weight 2, D'yesterday'  weight 3, D'today' - 2  weight 4, D'today' - 3  weight 5 )	Random values from pre-supplied list There's no “sequential choice” type. There's no “unique” qualifier.

Data Source	Syntax	Meaning
Files from folder	<code>folder("mydir")</code>	Pick a file at random from the folder specified, return the file's contents.
Lines from file	<code>filerow("abc.txt")</code>	Pick a random row from the text file specified, without any trailing newline characters.
XPath from XML	<code>xpath("a.xml", "//topLv1")</code> <code>xpath("a.xml", "//topLv1" unique)</code> <code>xpath("a.xml", "//topLv1" unique per parent)</code>	Run the supplied XPath query on the input XML file, cache all values in memory, and pick one at random for each row.
SQL query	<code>sql("select ...")</code> <code>sql("select ... a == ?" bind \$var)</code>	Execute the supplied SQL statement and return the first column from the first row of the result set. Optionally bind to "?" parameters.
SQL column	<code>column(tbl.colnm)</code> <code>column(tbl.colnm unique)</code> <code>column(tbl.colnm unique per parent)</code> <code>tbl.colnm</code>	Use a random value from the supplied table and column (from database table or "lookup table" Excel file).

## 10.2 DataRoller Keywords

all	file	null	sql
bind	folder	of	step
blob	function	on	table
case	haspreviousrow	parent	then
child	hour	per	this
choice	hours	previousrow	TRUE
column	if	random	unique
day	insert	randomrow	weight
days	lookup	raw	when
delete	lorem	row	where
else	minute	rows	xpath
end	minutes	second	year
FALSE	month	seconds	years
filerow	months	sequence	

## 10.3 Syntax Reference

The text below expresses the grammar for DataRoller project files in modified BNF form. Parenthesis indicate grouping. Parenthesis with “+” suffix denotes one or more repetitions. Parenthesis with “\*” suffix denotes zero or more repetitions. Parenthesis with “?” suffix indicate an optional clause. Words in angle brackets denote keywords (lower case). Literals are enclosed in double-quotes (the double-quotes are not part of the literal). Non-terminals appear in italics.

### 10.3.1 Lexical Elements

```
<DATE>           : "D\" ([\"A\"-\"Z\", \"a\"-\"z\", \"0\"-\"9\", \"-\", \":\", \" \", \"/\"])+ \"\

<INT>            : (\"-\")? (
                    ([\"0\"-\"1\"])+ \"_2\" |
                    ([\"0\"-\"7\"])+ \"_8\" |
                    ([\"0\"-\"9\"])+ |
                    \"0x\" ([\"0\"-\"9\", \"A\"-\"F\", \"a\"-\"f\"])+
                    )

<DECIMAL>        : (\"-\")? ([\"0\"-\"9\"])+ \".\" ([\"0\"-\"9\"])+

<FLOAT>          : (\"-\")? ([\"0\"-\"9\"])+ \".\" ([\"0\"-\"9\"])+
                    [\"E\", \"e\"] ([\"+\", \"-\"])? ([\"0\"-\"9\"])+

<VAR>            : \"$\" ( [\"A\"-\"Z\", \"a\"-\"z\", \"0\"-\"9\", \"_\"] )+

<LABEL>          : ([\"A\"-\"Z\", \"a\"-\"z\", \"0\"-\"9\", \"_\", \"#\", \"$\"])+
                    | \"[\" (~[\" \", \"\\n\", \"\\r\", \"\\f\"])+ \"]\"

<STR>            : \"\

                    ~[\" \", \"\\n\", \"\\r\", \"\\f\"] |
                    \"\

                    [\"n\", \"t\", \"r\", \"\\n\", \"\\r\", \"\\f\"] |
                    \"u\" ([\"0\"-\"9\", \"a\"-\"f\", \"A\"-\"F\"]){4}
                    )* \"\


```

### 10.3.2 Grammar

#### 10.3.2.1 Project Files

```
project           := functions lookupTables projectActions

functions         := ( <FUNCTION> <LABEL> \"=\" <STR> ) *

lookupTables     := ( <LOOKUP> <TABLE> <LABEL> \"=\" <STR> ) *

projectActions   := ( ( <SQL> ( <STR> | <FILE> <STR> ) ) | table ) +
```

### 10.3.2.2 Tables

```
table := <TABLE> <LABEL>
      ( <DELETE> ( <ALL> | <WHERE> <STR> ) )?
      ( parentTable )?
      <INSERT> integerRange <ROWS> "{" columns "}"

parentTable := <CHILD> <OF> <LABEL>
              <ON> <THIS> "." <LABEL> "=" <PARENT> "." <LABEL>
              ( <WHERE> <STR> )?

columns := ( column ( "," column )* )?

column := ( <LABEL>
           (
             <RAW> "(" <STR> ")"
             |
             generatorExp ( <NULL> <INT> "%" )?
           )
         | <VAR> "=" generatorExp ( <NULL> <INT> "%" )? )
```

### 10.3.2.3 Expressions

```
generatorExp := andExp ( "||" andExp )*
```

```
andExp := notExp ( "&&" notExp )*
```

```
notExp := ( "!" )? eqExp
```

```
eqExp := relationExp (
         "==" relationExp
         |
         "!=" relationExp
         |
         "==" relationExp
         |
         "!=" relationExp
       )*
```

```
relationExp := additiveExp (
              "<" additiveExp
              |
              "<=" additiveExp
              |
              ">" additiveExp
              |
              ">=" additiveExp
            )*
```

```
additiveExp := multExp (
              "+" multExp
              |
              "-" multExp
              |
              "&" multExp
            )*
```

```
multExp := primaryExp (
           "*" primaryExp
           |
           "/" primaryExp
           |
           "%" primaryExp
         )*
```

```
primaryExp := ( generatorTerm | "(" generatorExp ")" )
```

### 10.3.2.4 Generators

```
generatorTerm      := (
    primitiveValue
  | randomGenerator
  | sequenceGenerator
  | <VAR> ( "." <LABEL> )?
  | <FOLDER> "(" <STR> ( uniqueType )? ")"
  | <FILEROW> "(" <STR> ( uniqueType )? ")"
  | <XPATH> "(" <STR> "," <STR> ( uniqueType )? ")"
  | <CHOICE> "(" weightedRandomList ")"
  | <SQL> "(" <STR> ( <BIND> labellist )? ")"
  | <BLOB> "(" integerRange ")"
  | <LOREM> "(" integerRange ")"
  | <RANDOMROW> "(" <LABEL> ( uniqueType )? ")"
  | <HASPVIOUSROW> "(" ")"
  | <PREVIOUSROW> "(" <LABEL> ")"
  | columnOrFunction
  | conditionalGenerator
)

randomGenerator    := <RANDOM> "(" (
    <INT> ".." <INT> ( <STEP> <INT> )? ( uniqueType )?
  | <DECIMAL> ".." <DECIMAL> <STEP> <DECIMAL>
  | <FLOAT> ".." <FLOAT>
  | <DATE> ".." <DATE> ( <STEP> "/" timeDimension "/" )?
    ( uniqueType )?
) ")"

sequenceGenerator := <SEQUENCE> "(" (
    <DECIMAL> <STEP> <DECIMAL> ( <UNIQUE> <PER> <PARENT> )?
  | <DATE> ( <STEP> duration )? ( <UNIQUE> <PER>
<PARENT> )?
  | <FLOAT> <STEP> <FLOAT> ( <UNIQUE> <PER> <PARENT> )?
  | ( <INT> ( <STEP> <INT> )? ( <UNIQUE> <PER>
<PARENT> )? )?
) ")"

columnOrFunction   := (
    <COLUMN> "(" (
        <LABEL> ( "." <LABEL> ( uniqueType )? )?
      | <PARENT> "." <LABEL>
    ) ")"
  | <LABEL> ( "." <LABEL> | "(" functionArgs ")" )?
  | <PARENT> "." <LABEL>
)

conditionalGenerator := (
    <IF> "(" generatorExp ")" ( <THEN> )?
    generatorExp ( <ELSE> generatorExp )? <END>
  | <CASE> ( <WHEN> generatorExp <THEN> generatorExp )+
    ( <ELSE> generatorExp )? <END>
)
```

### 10.3.2.5 Primitives

```
uniqueType        := <UNIQUE> ( <PER> <PARENT> )?

duration          := "/" ( <INT> timeDimension )+ "/"
```

```

timeDimension      := ( <YEARS> | <MONTHS> | <DAYS> | <HOURS>
                       | <MINUTES> | <SECONDS> )

functionArgs       := ( generatorExp ( "," generatorExp )* )?

integerRange       := <INT> ( ".." <INT> )?

labellist          := <LABEL> ( "," <LABEL> )*

weightedRandomList := primitiveValue ( <WEIGHT> <INT> )? (
    "," primitiveValue ( <WEIGHT> <INT> )?
)*

primitiveValue     := ( <STR> | <INT> | <DECIMAL> | <FLOAT>
                       | <DATE> | <TRUE> | <FALSE> | <NULL> )

```

## 10.4 JDBC URL Reference

“jdbc:” is a common prefix to all connection strings.

The second portion is a label. This label is used to cross-reference with drivers available to DataRoller.

Database	JDBC Type 4 Connection Information
SQL Server (Microsoft)	<p>Simplest connection to a local instance on port 1433 with the default database of the connecting user:</p> <pre>jdbc:sqlserver://localhost</pre> <p>Connection to instance SALES on host DBSRVR port 1234 and use database TESTDB:</p> <pre>jdbc:sqlserver://DBSRVR\SALES:1234;database=TESTDB</pre> <p>For reference:  <a href="http://msdn.microsoft.com/en-us/library/ms378672%28v=sql.90%29.aspx">http://msdn.microsoft.com/en-us/library/ms378672%28v=sql.90%29.aspx</a></p>
SQL Server (jTDS)	<p>Simplest connection to a local instance on port 1433 with the default database of the connecting user:</p> <pre>jdbc:jtds:sqlserver://localhost</pre> <p>Connection to the default instance on host DBSRVR port 1234 and use database TESTDB:</p> <pre>jdbc:jtds:sqlserver://DBSRVR:1234/TESTDB;instance=SALES</pre> <p>For reference:  <a href="http://jtds.sourceforge.net/faq.html#urlFormat">http://jtds.sourceforge.net/faq.html#urlFormat</a></p>

Database	JDBC Type 4 Connection Information
Sybase	<p>Simplest connection to a local instance on port 7100 with the default database of the connecting user:</p> <pre data-bbox="440 302 1385 354">jdbc:jtds:sybase://localhost</pre> <p>Connection to host “DBSRVR” port 1234 and use database TESTDB:</p> <pre data-bbox="440 428 1385 480">jdbc:jtds:sybase://DBSRVR:1234/TESTDB</pre> <p>For reference:  <a href="http://jtds.sourceforge.net/faq.html#urlFormat">http://jtds.sourceforge.net/faq.html#urlFormat</a></p>
HSQLDB	<p>Connecting to local HSQLDB database server:</p> <pre data-bbox="440 657 1385 709">jdbc:hsqldb:hsqldb://localhost</pre> <p>Loading HSQLDB files directly (when there is no HSQLDB server using them):</p> <pre data-bbox="440 783 1385 835">jdbc:hsqldb:file:C:/db/SALESDB</pre> <p>In the sample above, folder C:/db will open or create files such as SALESDB.log and SALESDB.properties.</p> <p>For reference:  <a href="http://hsqldb.org/doc/guide/guide.html#rgc_connecting_db">http://hsqldb.org/doc/guide/guide.html#rgc_connecting_db</a></p>
MySQL	<p>Connecting to local MySQL database server (no hostname defaults to 127.0.0.1) on port 3306:</p> <pre data-bbox="440 1161 1385 1213">jdbc:mysql://</pre> <p>Connection to host DBSRVR, port 1234, database TESTDB and compress all network traffic:</p> <pre data-bbox="440 1318 1385 1371">jdbc:mysql://DBSRVR:1234/TESTDB;useCompression=true</pre> <p>For reference:  <a href="http://dev.mysql.com/doc/refman/5.6/en/connector-j-reference-configuration-properties.html">http://dev.mysql.com/doc/refman/5.6/en/connector-j-reference-configuration-properties.html</a></p>

Database	JDBC Type 4 Connection Information
PostgreSQL	<p>Connect to local database TESTDB default port 5432:  <input data-bbox="440 264 1385 317" type="text" value="jdbc:postgresql:TESTDB"/></p> <p>Connect to host DBSRVR default database on default port 5432:  <input data-bbox="440 390 1385 443" type="text" value="jdbc:postgresql://DBSRVR"/></p> <p>Connection to host DBSRVR, port 1234, database TESTDB and compress all network traffic:  <input data-bbox="440 558 1385 611" type="text" value="jdbc:postgresql://DBSRVR:1234/TESTDB"/></p> <p>For reference:  <a href="http://jdbc.postgresql.org/documentation/head/connect.html">http://jdbc.postgresql.org/documentation/head/connect.html</a></p>
Oracle	<p>Connect to service TESTDB on the local TNS listener on default port 1521:  <input data-bbox="440 783 1385 835" type="text" value="jdbc:oracle:thin:@//localhost/TESTDB"/></p> <p>Connection to host DBSRVR, port 1234, database TESTDB:  <input data-bbox="440 909 1385 961" type="text" value="jdbc:oracle:thin:@//DBSRVR:1521/TESTDB"/></p> <p>For reference:  <a href="http://docs.oracle.com/cd/B14117_01/java.101/b10979/urls.htm">http://docs.oracle.com/cd/B14117_01/java.101/b10979/urls.htm</a></p>
DB2	<p>Connect to local database TESTDB on the default port (likely 446, 6789, or 50000):  <input data-bbox="440 1140 1385 1192" type="text" value="jdbc:db2:TESTDB"/></p> <p>Connection to host DBSRVR, port 1234, database TESTDB:  <input data-bbox="440 1266 1385 1318" type="text" value="jdbc:db2://DBSRVR:1234/TESTDB"/></p> <p>For reference:  <a href="http://www.ibm.com/developerworks/data/library/techarticle/dm-0512kokkat/">http://www.ibm.com/developerworks/data/library/techarticle/dm-0512kokkat/</a></p>

## 10.5 DataRoller License and Included Works

DataRoller is made available under the GNU General Public License. A copy of this license is provided in the DataRoller distribution in file “License.txt”.

Software used by DataRoller:

Software Group	Software Provided
The Apache Software Foundation ( <a href="http://www.apache.org/">http://www.apache.org/</a> )	BCEL CGLIB Commons CLI Commons Codec Commons Collections Commons DBCP Commons IO Commons Lang Commons Logging Commons Pool DbUtils Log4j OGNL Poi Xalan Xerces
The Werken Company ( <a href="http://jaxen.werken.com/">http://jaxen.werken.com/</a> )	Jaxen
The JDOM Project, Jason Hunter & Brett McLaughlin ( <a href="http://www.jdom.org/">http://www.jdom.org/</a> )	JDOM
The XOM Project, Elliott Rusty Harold ( <a href="http://www.xom.nu/">http://www.xom.nu/</a> )	XOM
The ICU Project from IBM ( <a href="http://site.icu-project.org/">http://site.icu-project.org/</a> )	Icu4j
John Cowan ( <a href="http://ccil.org/~cowan/XML/tagsoup">http://ccil.org/~cowan/XML/tagsoup</a> )	Tagsoup
The DOM4J Project ( <a href="http://www.dom4j.org">http://www.dom4j.org</a> )	Dom4j
The jTDS Project ( <a href="http://jtds.sourceforge.net/">http://jtds.sourceforge.net/</a> )	jTDS
The TestNG Project ( <a href="http://testng.org/">http://testng.org/</a> )	TestNG
The DBUnit Project ( <a href="http://www.dbunit.org/">http://www.dbunit.org/</a> )	DBUnit

Software Group	Software Provided
Joe Walnes, Henri Tremblay, Leonardo Mesquita ( <a href="http://objenesis.googlecode.com/">http://objenesis.googlecode.com/</a> )	EasyMock Objenesis
The Hypersonic SQL Group( <a href="http://www.hsqldb.org/">http://www.hsqldb.org/</a> )	HSQldb
The MySQL Group ( <a href="http://www.mysql.org">http://www.mysql.org</a> )	MySQL
The Unitils Project ( <a href="http://unitils.org/">http://unitils.org/</a> )	Unitils
The W3C consortium ( <a href="http://www.w3c.org">http://www.w3c.org</a> )	XML APIs
The SAX project ( <a href="http://www.saxproject.org">http://www.saxproject.org</a> )	SAX
Scott Hudson, Frank Flannery, C. Scott Ananian ( <a href="http://www.cs.princeton.edu/~appel/modern/java/CUP">www.cs.princeton.edu/~appel/modern/java/CUP</a> )	CUP Parser Generator