DBToaster Documentation

Revision 2827

Table of Contents

1

Getting Started

The DBToaster compiler is used to generate incremental maintenance (M3) programs. M3 programs can be executed in the following ways:

- *Interpreter*: The DBToaster compiler includes an internal M3 interpreter. When the interpreter is used, the M3 program is evaluated and results are printed at the end of evaluation. The interpreter is not especially efficient and should not be used in production systems, but is useful for query development and testing.
- *Standalone Binaries*: The DBToaster compiler can produce standalone binaries that evaluate the M3 program. This requires invoking a second stage compiler (`g++` or `scalac`) to generate the final binary. This mode is far more efficient than the interpreter, but compilation is slower.
- *Source Code*: The DBToaster compiler can also produce source code that can be linked into your own binary.

For best performance, generate either Standalone Binaries or Source Code.

1.1 The DBToaster Interpreter

To use DBToaster to evaluate queries in its internal interpreter, invoke it with the `-r` flag and one or more SQL query files. The output of all queries in the file will be printed once all data has been processed. If any of the queries do not terminate (e.g., one or more data sources are sockets), then pressing control-c will terminate the process and print the most recent query results. For example, we can run the sample query RST (included in the distribution) on the interpreter.

```
$> cat examples/queries/simple/rst.sql
CREATE STREAM R(A int, B int)
  FROM FILE 'examples/data/simple/tiny/r.dat' LINE DELIMITED CSV;

CREATE STREAM S(B int, C int)
  FROM FILE 'examples/data/simple/tiny/s.dat' LINE DELIMITED CSV;

CREATE STREAM T(C int, D int)
  FROM FILE 'examples/data/simple/tiny/t.dat' LINE DELIMITED CSV;

SELECT sum(A*D) AS AtimesD FROM R,S,T WHERE R.B=S.B AND S.C=T.C;

$> bin/dbtoaster -r examples/queries/simple/rst.sql
Processing time: 0.0206508636475
---------- Results ------------ AtimesD: 306
```

1.2 Generating Standalone Binaries

To use DBTOASTER to create a standalone binary, invoke it with `-c [binary name]`. The binary can be invoked directly. Like the interpreter, it will print the results of all queries once all data has been processed.

Requirements: Note that in order to compile binaries, DBTOASTER will invoke `g++`. DBTOASTER relies on pthreads and several Boost libraries ("`program_options`", "`serialization`", "`system`", "`filesystem`", "`chrono`", and "`thread`"). These must all be in your binary, include, and respectively, library search paths. The `-I` and `-L` flags may be used to pass individual include and library paths (respectively) to `g++`, or the environment variables `DBT_HDR`, and `DBT_LIB` may be used to store a colon-separated list of search paths.

Additionally, if only the multi-threaded versions of the Boost libraries are available, as is the case with some Cygwin or MacPorts provided distributions, one also needs to add the `-d MT` flag when compiling queries to binaries.

The following command line will generate the rst executables

```
$> bin/dbtoaster examples/queries/simple/rst.sql -c rst
```

or if the `-d MT` flag is needed:

```
$> bin/dbtoaster examples/queries/simple/rst.sql -c rst -d MT
```

Running the `rst` executable will produce the following output:

```
$> ./rst
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="9">
  <ATIMESD>306</ATIMESD>
</boost_serialization>
```

To produce a Scala JAR file, invoke `dbtoaster` with `-l scala`, and the `-c [binary name]` flag as above. DBTOASTER will produce `[binary name].jar`, which can be run as a normal Scala program.

Requirements: Note that in order to produce JAR files, DBTOASTER will invoke the `scalac` compiler, which needs to be reachable through the binary search paths of the system. Also, on Windows, make sure to use `;` as the classpath separator instead of `:`.

```
$> bin/dbtoaster examples/queries/simple/rst.sql -l scala -c rst
$> scala -classpath "rst.jar:lib/dbt_scala/dbtlib.jar"
   org.dbtoaster.RunQuery
Run time: 0.261 s
<ATIMESD>306 </ATIMESD>
```

1.3 Generating Source Code

DBTOASTER'S primary role is the construction of code that can be linked in to existing applications. To generate a source file in C++ or Scala, invoke it with `-l [language]`, replacing `[language]` with `cpp` or `scala`. If the optional `-o` flag is used to direct the generated code into a particular file, the target language will be auto-detected from the file suffix ("`.scala`" for Scala, and "`.h`", "`.hpp`", or "`.cpp`" for C++).

```
$> bin/dbtoaster examples/queries/simple/rst.sql -o rst.cpp
$> bin/dbtoaster examples/queries/simple/rst.sql -o rst.scala
$> ls
rst.hpp      rst.scala
```

See the individual target language documentation pages for details.

2

Command-Line Reference

```
$> dbtoaster [options] <input file 1> [<input file 2> [...]]
```

2.1 Command Line Options

`-c <target file>`
: Compile the query into a standalone binary. By default, the C++ code generator will be used with `g++` to generate the binary. An alternate compiled target language (currently, C++ or Scala) may be selected using the `-l` flag.

`-l <language>`
: Compile the query into the specified target language (see below). By default, the query will be interpreted. The use of this flag overrides any previous `-l` or `-r`.

`-o <output file>`
: Redirect the compiler's output to the specified file. If used in conjunction with `-c`, the source code for the compiled binary will be directed to this file. The special output filename `'-'` refers to `stdout`. By default, output is directed to `stdout` or discarded if the `-c` flag is used.

`-r`
: Run the query (queries) in interpreter mode, overriding any target language previously specified by `-l`. This is the default.

`-F <optimization>`
: Activate the specified optimization flag. These are documented below.

`-O1 | -O2 | -O3`
: Set the optimization level to 1, 2, or 3 respectively. At optimization level 1, compilation is faster and generated code is (usually) easier to understand and follow. At optimization level 3, compilation is slower, but more efficient code is produced. Optimization level 2 is the default. Overrides any prior `-O` flags provided on the command line.

`-I <dir>`
: When invoking a second-stage compiler with the `-c` flag, add `dir` to the include file search path.

`-L <dir>`
: When invoking a second-stage compiler with the `-c` flag, add `dir` to the library file search path.

`-D <macro>`
: When invoking a second-stage compiler with the `-c` flag, define the preprocessor macro `macro`.

`-g <arg>`
: When invoking a second-stage compiler with the `-c` flag, pass through the argument `arg`.

`--depth <level>`
: Limit the compiler's maximum recursive depth. By default, DBToaster compiles queries with the depth set to infinity.

`--custom-prefix <prefix>`
: Prefix all DBToaster -generated symbols with this character string. Use this if DBToaster generates a symbol that conflicts with a symbol in user code. (Default: "__")

2.2 Supported Languages

Language	Command-line Name	Output Format	Description
DBT Relational Calculus	`calc`	output	DBToaster 's internal query representation. This is a direct translation of the input queries
M3	m3	output	A map-maintenance messages program. This is the set of triggers (written in DBT Relational Calculus) that will incrementally maintain the input queries and all supporting data structures
C++	cpp	output/compiled	A C++ class implementing the queries.
Scala	scala	output/compiled	A Scala class implementing the queries.

2.3 Optimization Flags

These flags are passed to the DBTOASTER compiler with the `-F` flag. The `-O1` and `-O3` flags each activate a subset of these flags. `-O2` is used by default (no optimization flags active).

`HEURISTICS-ENABLE-INPUTVARS`

Enable experimental support for incremental view caches. Queries with joins (and correlations) on inequality predicates are implemented in a way that corresponds roughly to nested-loop one-way joins in stream processing (a tree-based implementation is in development). If this flag is on, the compiler will cache and incrementally maintain the results of this one-way join. This is typically a bad idea, since the cost of maintaining the cached values is often higher than the cost of the nested loop scan. However, if the domains of the variables appearing in the join predicate are small, this flag can drastically improve performance (e.g., for the VWAP example query). Future versions of DBTOASTER will include a cost-based optimizer that automatically applies this flag when appropriate. This optimization is not activated by default at any optimization level.

`HEURISTICS-AGGRESSIVE-INPUTVARS`

Enable experimental support for aggressive materialization of maps with input variables (view caches). It requires the `HEURISTICS-ENABLE-INPUTVARS` flag to be active. If the calculus optimizations are disabled (see `CALC-NO-OPTIMIZE`), this option might significantly prolong the compilation time. Note: the C++ backend might fail to compile certain classes of queries when this flag is on.

`HEURISTICS-PULL-OUT-VALUES`

Prevent value terms (variables and comparisons) from being materialized inside maps. In certain cases (e.g., `mddb/query2.sql`), this option reduces the number of generated maps and speed-ups the compilation time at the expense of doing more computation at runtime.

`EXPRESSIVE-TLQS`

By default, each user-provided (top-level) query is materialized as a single map. If this flag is turned on, the compiler will materialize top-level queries as multiple maps (if it is more efficient to do so), and only combine them on request. For more complex queries (in particular nested aggregate, and AVG aggregate queries), this results in faster processing rates, and if fresh results are required less than once per update, a lower overall computational cost as well. However, because the final evaluation of the top-level query is not performed until a result is requested, access latencies are higher. This optimization is not activated by default at any optimization level.

`IGNORE-DELETES`

Do not generate code for deletion triggers. The resulting programs will be simpler, and sometimes have fewer data structures, but will not support deletion events. This optimization is not activated by default at any optimization level.

`HEURISTICS-ALWAYS-UPDATE`
: In some cases, it is slightly more efficient to re-evaluate expressions from scratch rather than maintaining them with their deltas (for example, certain queries containing nested aggregates). Normally the compiler's heuristics will make a best-effort guess about whether to re-evaluate or incrementally maintain the expression. If this flag is on, the compiler will incrementally maintain all expressions and never re-evaluate.

`HASH-STRINGS`
: Do not use strings during evaluation. All strings are immediately replaced by their integer hashes (using each runtime's native hashing mechanism) as soon as they are parsed. This makes query evaluation faster, but is not guaranteed to produce correct results if a hash collision occurs. Furthermore, strings that would normally appear in the output are output as their integer hash values instead. This optimization is not activated by default at any optimization level.

`COMPILE-WITH-STATIC`
: Perform static linking on compiled binaries (e.g., invoke `gcc` with `-static`). The resulting binaries will be faster the first time they are run. This optimization is not activated by default at any optimization level.

`CALC-DONT-CREATE-ZEROES`
: Avoid creating empty relation terms during pre-evaluation. Empty relation terms are aggressively propagated throughout expressions in which they occur, and may result in expressions that do not need to be incrementally maintained (because they are guaranteed to be always empty). Activating this flag is only useful if you want to inspect the generated Calculus/M3 code by hand. This optimization is not activated by default at any optimization level.

`AGGRESSIVE-FACTORIZE`
: When optimizing expressions in DBTOASTER relational calculus, perform factorization as aggressively as possible. For some queries, particularly those with nested subqueries, this can generate much more efficient code. However, it makes compilation slower on some queries. This optimization is automatically activated by `-O3`.

`AGGRESSIVE-UNIFICATION`
: When optimizing expressions in DBTOASTER relational calculus, inline lifted variables wherever possible, even if the lift term can not be eliminated entirely. This can produce substantially tighter code for queries with lots of constants, but slightly increases compilation time. This optimization is automatically activated by `-O3`.

`DELETE-ON-ZERO`
: In generated code, when a map value becomes `0`, remove the value from the map. Resulting programs are more efficient over long stretches of insertions and deletions. This optimization is automatically activated by `-O3`.

`COMPILE-WITHOUT-OPT`
: Request that the second-stage compiler disable any unnecessary optimizations (e.g., by default, `gcc` is invoked with `-O3`, but not if this flag is active). This optimization is automatically activated by `-O1`.

`WIDE-TUPLE`
: Use nested tuples in generated C++ programs. This option is mostly used at lower compilation levels (depth 0 or 1) to overcome the Boost limitation that tuples may contain at most 50 attributes.

`WEAK-EXPR-EQUIV`
: When testing for expression equivalence, perform only a naive structural comparison rather than a (at least quadratic, and potentially exponential) matching. This accelerates compilation, but may result in the creation of duplicate maps. This optimization is automatically activated by `-O1`.

`CALC-NO-OPTIMIZE`
: Do not apply calculus optimizations that simplify delta expressions. This option prevents range restrictions from being propagated through expressions, which usually leads to significantly worse performance. The resulting code is close to what the naive recursive incremental algorithm would produce. This flag is not activated by default at any optimization level.

`CALC-NO-DECOMPOSITION`
: Do not apply query decomposition when computing deltas. This option is not activated by default at any optimization level.

`K3-NO-OPTIMIZE`
: Do not apply functional optimizations. This optimization is automatically activated by `-O1`.

`DUMB-LIFT-DELTAS`
: When computing the viewlet transform, use the delta rule for lifts precisely as described in the PODS10 paper. If this flag is not active, a post-processing step is applied to lift deltas, that range-restricts the resulting expression to only those tuples that are affected. This optimization is automatically activated by `-O1`.

3

DBT-SQL Reference

3.1 CREATE FUNCTION

Declare a forward reference to a user defined function in the target language.

```
create_function :=
  CREATE   FUNCTION   <name>   (   <arguments>   )   RETURNS   <type>   AS
<definition>

arguments := [<var_1> <type_1> [, <var_2> <type_2> [, ...]]]

definition := EXTERNAL '<external_name>';
```

Use create_`function` to declare a user-defined primitive-valued function in the target language. At this time, DBToaster does not create target-language specific declarations are created, so the function must be in-scope within the generated code. Once declared, a UDF may be used in any arithmetic expression within DBToaster. For example, the following block illustrates use of the `math.h` `cos()` and `sin()` functions for C++ targeted code.

```
CREATE FUNCTION cos ( x double ) RETURNS double AS EXTERNAL 'sin';
CREATE FUNCTION sin ( x double ) RETURNS double AS EXTERNAL 'cos';

SELECT r.distance * cos(r.angle) AS x,
       r.distance * sin(r.angle) AS y,
FROM RadialMeasurements r;
```

3.2 CREATE TABLE/STREAM

Declare a relation for use in the query.

```
create_statement :=
  CREATE { TABLE | STREAM } <name> ( <schema> )
         [<source_declaration>]

schema := [<var_1> <type_1> [, <var_2> <type_2> [, ...]]]

source_declaration := source_stream source_adaptor

source_stream :=
  FROM FILE '<path>' {
      FIXEDWIDTH <bytes_per_row>
    | LINE DELIMITED
    | '<delim_string>' DELIMITED
  }

source_adaptor :=
  <adaptor_name> (
    [<param_1> := '<value>' [, <param_2> := '<value>' [, ...]]]
  )
```

A create statement defines a relation named name with the indicated schema and declares a method for automatically populating/updating rows of that relation.

Each relation may be declared to be either a `Stream` or a `Table`:

- Tables are static data sources. A table is read in prior to query monitoring, and must remain constant once monitoring has started.
- Streams are dynamic data sources. Stream updates are read in one tuple at a time as data becomes available, and query views are updated after every update to a stream.

The source declaration allows DBTOASTER (either in the interpreter, or the generated source code) to automatically update the relation. The source declaration is optional when using DBTOASTER to generate source code. User programs may manually inject updates to relations, or manually declare sources during initialization of the DBTOASTER -generated source code.

A source declaration consists of stream and adaptor components. The stream component defines where data should be read from, and how records in the data are delimited. At present, DBTOASTER only supports reading tuples from files.

If the same file is referenced multiple times, the file will only be scanned once, and events will be generated in the order in which they appear in the file.

The adaptor declares how to parse fields out of each record. See below for documentation on DBTOASTER 's standard adaptors package.

3.2.1 Example

```
CREATE STREAM R(a int, b date)
FROM FILE 'examples/data/r.dat' LINE DELIMITED
CSV (fields := '|')
```

3.3 INCLUDE

Import a secondary SQL file.

```
include_statement := INCLUDE 'file'
```

Import the contents of the selected file into DBTOASTER. The file path is interpreted relative to the current working directory.

3.4 SELECT

Declare a query to monitor.

```
select_statement :=
  SELECT <target_1> [, <target_2> [, ...]]
  FROM <source_1> [, <source_2> [, ...]]
  WHERE <condition>
  [GROUP BY <group_vars>]

target := <expression> [[AS] <target_name>] | * | *.*
        | <source_name>.*

source := <relation_name> [[AS] <source_name>]
  | (<select_statement>) [AS] <source_name>
  | <source> [NATURAL] JOIN <source> [ON <condition>]

expression :=  (<expression>) | <int> | <float> | '<string>'
  | <var> | <source>.<var>
  | <expression> { + | - | * | / } <expression>
  | -<expression>
  | (SELECT <expression> FROM ...)
  | SUM(<expression>) | COUNT(* | <expression>)
  | AVG(<expression>) | COUNT(DISTINCT [var1, [var2, [...]]])
  | <inline_function>([<expr_1> [, <expr_2> [, ...]]])
  | DATE('yyyy-mm-dd')
  | EXTRACT({year|month|day} FROM <date>)
  | CASE <expression> WHEN <expression> THEN <expression> [, ...]
                      [ELSE <expression>] END
  | CASE WHEN <condition> THEN <expression> [, ...]
         [ELSE <expression>] END

condition := (<condition>) | true | false | not (<condition>)
  | <expression> { < | <= | > | >= | = | <> } <expression>
  | <expression> { < | <= | > | >= | = | <> } { SOME | ALL }
                  <select_statement>
  | <condition> AND <condition> | <condition> OR <condition>
  | EXISTS <select_statement>
  | <expression> BETWEEN <expression> AND <expression>
  | <expression> IN <select_statement>
  | <expression> LIKE <matchstring>
```

DBTOASTER SQL's SELECT operation differs from the SQL-92 standard. Full support for the SQL-standard SELECT is planned and will be part of a future release.

3.4.1 Aggregates

DBTOASTER currently has support for the SUM, COUNT, COUNT DISTINCT, and AVG aggregates. MIN and MAX are not presently supported. Also, see the note on NULL values below.

3.4.2 Types

DBTOASTER presently supports integer, floating point, string, and date types. The char and varchar types are treated as strings of unbounded length.

3.4.3 Conditional Predicates

DBTOASTER presently supports boolean expressions over arithmetic comparisons (=, <>, <, <=, >, >=), existential/universal quantification (SOME/ALL/EXISTS), BETWEEN, IN, and LIKE.

3.4.4 SELECT syntax

SELECT [FROM] [WHERE] [GROUP BY] queries are supported. The DISTINCT, UNION, LIMIT, ORDER BY, and HAVING clauses are not presently supported. The HAVING operator may be simulated by use of nested queries:

```
SELECT A, SUM(B) AS sumb FROM R HAVING SUM(C) > 2
```

is equivalent to

```
SELECT A, sumb FROM (
  SELECT A, SUM(B) AS sumb, SUM(C) as sumc FROM R
)
WHERE sumc > 2
```

3.4.5 NULL values

DBToaster does not presently support NULL values. The SUM or AVG of an empty table is 0, and not NULL. OUTER JOINS are not supported.

3.4.6 Incremental Computation with Floating Point Numbers

There are several subtle issues that arise when performing incremental computations with floating point numbers:

- When using division in conjunction with aggregates, be aware that SUM and AVG return 0 for empty tables. Once a result value becomes NAN or INFTY, it will no longer be incrementally maintainable. We are working on a long-term fix. In the meantime, there are two workarounds for this problem. For some queries, you can coerce the aggregate value to be nonzero using the LISTMAX standard function (See the example query `tpch/query8.sql` in the DBToaster distribution for an example of how to do this). For most queries, the `-F EXPRESSIVE-TLQs` optimization flag will typically materialize the divisor as a separate map (the division will be evaluated when accessing results).
- The floating point standards for most target languages (including OCaml, Scala, and C++) do not have well-defined semantics for equality tests over floating point numbers. Consequently, queries with floating-point group-by variables might produce non-unique groups (since two equivalent floating point numbers are not considered to be equal). We are working on a long-term fix. In the meantime, the issue can be addressed by using CAST_INT, or CAST_STRING to convert floating point numbers into canonical forms.

3.4.7 Other Notes

- DBToaster does not allow non-aggregate queries to evaluate to singleton values. That is, the query

  ```
  SELECT 1 FROM R WHERE R.A = (SELECT A FROM S)
  ```

  is a compile-time error in DBToaster (while such a query would instead produce a run time error if it returned more than one tuple in SQL-92). An equivalent, valid query would be:

  ```
  SELECT 1 FROM R WHERE R.A IN (SELECT A FROM S)
  ```

- Variable scoping rules are slightly stricter than the SQL standard (you may need to use fully qualified names in some additional cases).

See the DBT StdLib Reference for documentation on DBToaster's standard function library.

DBTOASTER maintains query results in the form of either multi-key dictionaries (a.k.a., maps, hashmaps, etc...), or singleton primitive-typed values. Each query result is assigned a name based on the query (see documentation for your target language's code generator for details on how to access the results).

- Non-aggregate queries produce a dictionary named "COUNT". Each entry in the dictionary has a key formed from the target fields of the SELECT. Values are the number of times the tuple occurs in the output (i.e., the query includes an implicit group-by COUNT(*) aggregate).

- Singleton (non-grouping) aggregate queries produce a primitive-typed result for each aggregate target in the SELECT. The result names are assigned based on the name of each target (i.e., using the name following the optional AS clause, or a procedurally generated name otherwise).

- Group-by aggregate queries produce a dictionary for each aggregate target. The non-aggregate (group-by) targets are used as keys for each entry (as for non-aggregate queries), and the value is the aggregate value for each group. The dictionaries are named based on the name of each aggregate target (as for singleton aggregate queries)

If multiple SELECT statements occur in the same file, the result names of each query will be prefixed with "`QUERY#_`", where `#` is an integer.

3.5 Examples

```
CREATE STREAM R(A int, B int);
CREATE STREAM S(B int, C int);
```

3.5.1 Non-aggregate query

```
SELECT * FROM R;
```

Generates a single dictionary named COUNT, mapping from the tuple "<R.A, R.B>" to the number of time each tuple occurs in R.

3.5.2 Aggregate query

```
SELECT SUM(R.A * S.C) AS sum_ac FROM R NATURAL JOIN S;
```

Generates a single constant integer named SUM_AC containing the query result.

3.5.3 Aggregate group-by query (one group-by var)

```
SELECT S.C, SUM(R.A) AS sum_a
FROM R NATURAL JOIN S
GROUP BY S.C;
```

Generates a dictionary named SUM_A mapping from values of S.C to the sums of R.A.

3.5.4 Aggregate group-by query (multiple group-by vars)

```
SELECT R.A, R.B, COUNT(*) AS foo FROM R GROUP BY R.A, R.B;
```

Generates a single dictionary named FOO, mapping from the tuple "<R.A, R.B>" to the number of time each tuple occurs in R.

3.5.5 Query with multiple aggregates

```
SELECT SUM(R.A) AS sum_a, SUM(S.C) AS sum_c
FROM R NATURAL JOIN S
GROUP BY S.C;
```

Generates two dictionaries named SUM_A and SUM_C, respectively containing the sums of R.A and S.C.

3.5.6 Multiple Queries

```
SELECT SUM(R.A) AS SUM_A FROM R;
SELECT SUM(S.C) SUM_C FROM S;
```

Generates two dictionaries named QUERY_1_SUM_A and QUERY_2_SUM_C, respectively containing the sums of R.A and S.C.

4

DBT StdLib Reference

DBToaster includes a standard library of functions that may used in SQL queries. Functions are called using a C-like syntax:

```
SELECT listmin(A, B) FROM R;
```

See the DBT-SQL Reference for information about calling functions.

4.1 listmin, listmax

Compute the minimum or maximum of a fixed-size list of values.

```
listmin (val1, val2, [val3, [val4, [...]]])
listmax (val1, val2, [val3, [val4, [...]]])
```

Input

`val1, val2, ...: any`
An arbitrary number of value expressions. All values in the list must be of the same type, or escalatable to the same type (e.g., Int -> Float), and must be comparable.

Output

Returns the minimum or maximum element of `val1, val2, ...`. The return type is the type of all of the elements of the list.

Notes

This library function is supported on the Interpreter, C++, and Scala runtimes.

4.2 date_part

Extract the parts of a date. Identical to EXTRACT(datepart FROM date).

```
date_part ('year', date)
date_part ('month', date)
date_part ('day', date)
```

Input

`datepart: string literal`

One of the string literals 'year', 'month', or 'day'. Dictates which part of the date to extract. datepart is case-insensitive.

`date: date`

The day to extract a component of.

Output

Returns the requested year, month, or date part of a date, as an integer.

Notes

This library function is supported on the Interpreter and C++ runtimes.

4.3 regexp_match

Perform regular expression matching.

```
regexp_match (pattern, string)
```

Input

`pattern: string`

A regular expression pattern (see Notes below).

`string: string`

The string to match against.

Output

Returns the integer 1 if the match succeeds, and the integer 0 otherwise.

Notes

This library function is supported on the Interpreter and C++ runtimes.

Note: Regular expressions are compiled and evaluated using each target runtime's native regular expression library. The Interpreter uses Ocaml's built-in Str module. The C++ code generator uses `regex(3)`.

4.4 substring

Extract a substring.

```
substring (string, start, len)
```

Input

`string: string`

The string to extract a substring from.

`start: int`

The first character of string to return. 0 is the first character of the string.

`len: int`

The length of the substring to return.

Output

Returns the substring of string of length `len`, starting at index `start`.

Notes

This library function is supported on the Interpreter and C++ runtimes.

4.5 cast_int,cast_float,cast_string,cast_date

Typecasting operations.

```
cast_int(val)
cast_float(val)
cast_string(val)
cast_date(val)
```

Input

`val: any`

The value to cast. The value's type must be castable to the target type.

Output

Returns the cast value of `val` as follows. If `val` is already of the target type, it is passed through unchanged.

From	To	Notes
int	float	The floating point representation of the input.
int	string	The string representation of the input, as determined by the runtime's standard stringification functionality.
float	int	The integer representation of the input. Non-integer values will be truncated.
float	string	The string representation of the input, as determined by the runtime's standard stringification functionality.
date	string	The string representation of the input, in the SQL-standard YYYY-MM-DD date format.
string	int	The input parsed as an integer, using the runtime's standard integer parsing functionality. (Ocaml: int_of_string, C++: atoi(3))
string	float	The input parsed as a float, using the runtime's standard integer parsing functionality. (Ocaml: float_of_string, C++: atof(3))
string	date	The input parsed as a date. The input string must be in SQL-standard YYYY-MM-DD date format, or a runtime error will be generated.

Notes

This library function is supported on the Interpreter and C++ runtimes.

5

DBT Adaptors Reference

DBTOASTER 's runtimes currently support the CSV and Order Book adaptors.

5.1 CSV

A simple string-delimited adaptor. Fields are separated using the delimiter passed in the *delimiter* parameter. If not provided, comma (",") will be used as a default delimiter.
The optional deletions parameter can be used to generate a single stream of both insertions and deletions. When set to "true", the input source is assumed to have an extra, leading column. When the value in this column is 0, the record is treated as a deletion. When the value is 1, the record is treated as an insertion. Fields are parsed based on the type of the corresponding column in the relation. Ints, floats, and strings are parsed as-is. Dates are expected to be formatted in the SQL-standard `[yyyy]-[mm]-[dd]` format.

```
CREATE STREAM R(A int, B int) FROM FILE 'r.dat'
LINE DELIMITED CSV (delimiter := '|');
```

5.2 Order Book

An adaptor that allows reading in stock trade historical data. It assumes that all the input records obey the following predefined schema: `<timestamp : float, message_id : int, action_id : char, volume : float, price : float>`.
Insertions and deletions are triggered for each record as follows:

- If `action_id` is `'b'`, and the orderbook adaptor was instantiated with the parameter `book := 'bids'`, an insertion will be generated for the record.
- If `action_id` is `'a'`, and the orderbook adaptor was instantiated with the parameter `book := 'asks'`, an insertion will be generated for the record.
- If `action_id` is `'d'`, and the orderbook had previously inserted a record with the same `message_id`, a deletion will be generated for the record.

Records will be instantiated into a relation with schema `<T float, ID int, BROKER_ID int, VOLUME float, PRICE float>`. All fields except `BROKER_ID` are taken directly from the input stream. `BROKER_ID`s are assigned in a range from 0 to the integer value of the brokers parameter. The value of `BROKER_ID` is assigned randomly, using `rand()` by default, or deterministically from the value of `ID` if the deterministic parameter is set to 'yes'.

```
CREATE STREAM bids(T float, ID int, BROKER_ID int, VOLUME float,
  PRICE float)
FROM FILE 'history.csv'
LINE DELIMITED orderbook (book := 'bids', brokers := '10',
  deterministic := 'yes');
```

5.3 Summary

Adaptor	Parameter	Optional	Description
CSV	delimiter	yes	A string delimiter used to extract fields from a record. If not specified, the default value is `','`.
	deletions	yes	If set to "true", use the first field of the input file to distinguish between rows for insertion and rows for deletion. A 0 in the first column triggers a deletion event. A 1 in the first column triggers an inertion event. The first column is stripped off of the record before further parsing is performed.
Orderbook	action_id	no	The value of this parameter may be 'bids' or 'asks', and determines for which orderbook events will be generated.
	brokers	yes	The number of brokers to simulate. By default, 10 brokers will be used.
	deterministic	yes	If the value of this parameter is 'yes', broker ids will be generated deterministically based on the message id. By default, broker ids will be generated randomly using the `rand()` system call or equivalent.

6

C++ Code Generation

6.1 Quickstart Guide

DBToaster generates C++ code for incrementally maintaining the results of a given set of queries if CPP is specified as the output language (`-l cpp` command line option). In this case DBToaster produces a C++ header file containing a set of data structures (`tlq_t`, `data_t`, and `Program`) required for executing the SQL program.

Let's consider the following SQL query:

```
$> cat examples/queries/simple/rs_example1.sql
CREATE TABLE R(A int, B int)
  FROM FILE 'examples/data/tiny/r.dat'
  LINE DELIMITED CSV (fields := ',');

CREATE STREAM S(B int, C int)
  FROM FILE 'examples/data/tiny/s.dat'
  LINE DELIMITED CSV (fields := ',');

SELECT SUM(r.A*s.C) as RESULT FROM R r, S s WHERE r.B = s.B;
```

The corresponding C++ header file can be obtained by running:

```
$> bin/dbtoaster examples/queries/simple/rs_example1.sql -l cpp
   -o rs_example1.hpp
```

Alternatively, DBToaster can build a standalone binary (if the `-c [binary name]` flag is present) by compiling the generated header file against lib/dbt_c++/main.cpp, which provides code for executing the SQL program and printing the results.

Requirements: Besides `g++`, the Boost header files and the following library binaries: `boost_program_options`, `boost_serialization`, `boost_system`, `boost_filesystem`, and `boost_chrono andboost_thread` have to be present on the system since the generated code makes use of them. If these cannot be found in the paths searched by default by `g++` then their location has to be explicitly provided to DBTOASTER. This can be done in one of the following two ways, either through the environment variables:

- `DBT_HDR` which should contain the path to Boost's `include` folder;
- `DBT_LIB` which should contain the path to Boost's `lib` folder.

```
$> export DBT_HDR=path-to-boost-include-dir
$> export DBT_LIB=path-to-boost-lib-dir
$> bin/dbtoaster examples/queries/simple/rs_example1.sql -l cpp
   -c rs_example1
```

or through the `-I` and `-L` command line flags:

```
$> bin/dbtoaster examples/queries/simple/rs_example1.sql -l cpp
   -c rs_example1 -I path-to-boost-include-dir
   -L path-to-boost-lib-dir
```

Additionally, if only the multi-threaded versions of the Boost libraries are available, as is the case with some Cygwin provided distributions, one also needs to add the `-d MT` flag when compiling queries to binaries.

```
$> bin/dbtoaster examples/queries/simple/rs_example1.sql -l cpp
   -c rs_example1 -d MT
```

Running the compiled binary will result in the following output:

```
$> ./rs_example1
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="9">
Initializing program:
Running program:
Printing final result:
<snap class_id="0" tracking_level="0" version="0">
    <RESULT>156</RESULT>
</snap>
</boost_serialization>
```

If the generated binary is run with the `--async` flag, it will also print intermediary results as frequently as possible while the SQL program is running in a separate thread.

```
$> ./rs_example1 --async
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="9">
Initializing program:
Running program:
<snap class_id="0" tracking_level="0" version="0">
    <RESULT>0</RESULT>
</snap>
<snap>
    <RESULT>0</RESULT>
</snap>
<snap>
    <RESULT>0</RESULT>
</snap>
<snap>
    <RESULT>0</RESULT>
</snap>
<snap>
    <RESULT>9</RESULT>
</snap>
<snap>
    <RESULT>74</RESULT>
</snap>
<snap>
    <RESULT>141</RESULT>
</snap>
Printing final result:
<snap>
    <RESULT>156</RESULT>
</snap>
</boost_serialization>
```

6.2 C++ API Guide

The DBTOASTER C++ code generator produces a header file containing three main type definitions in the `dbtoaster` namespace: `tlq_t`, `data_t`, and `Program`. Additionally, `snapshot_t` is pre-defined as a garbage collected pointer to `tlq_t`. What follows is a brief description of these types, while a more detailed presentation can be found in the Reference section.

- `tlq_t` encapsulates the materialized views directly needed for computing the results and offers functions for retrieving them.
- `data_t` extends `tlq_t` with auxiliary materialized views needed for maintaining the results and offers trigger functions for incrementally updating them.
- `Program` represents the execution engine of the SQL program. It encapsulates a `data_t` object and provides implementations to a set of abstract functions of the `IProgram` class used for running the program. Default implementations for some of these functions are inherited from the `ProgramBase` class while others are generated depending on the previously defined `tlq_t` and `data_t` types.

6.2.1 Executing the Program

The program execution can be controlled through the functions: `IProgram::init()`, `IProgram::run()`, `IProgram::is_finished()`, `IProgram::process_streams()` and `IProgram::process_stream_event()`.

- *`virtual void IProgram::init()`*
  Loads the tuples of static tables and performs initialization of materialized views based on that data. The definition of this functions is generated as part of the `Program` class.
- *`void IProgram::run( bool async = false )`*
  Executes the program by invoking the `Program::process_streams()` function. If parameter `async` is set to `true` the execution takes place in a separate thread. This is a standard function defined by the `IProgramclass`.
- *`bool IProgram::is_finished()`*
  Tests whether the program has finished or not. Especially relevant when the program is run in asynchronous mode. This is a standard function defined by the `IProgram` class.
- *`virtual void IProgram::process_streams()`*
  Reads stream events from various sources and invokes the `IProgram::process_stream_event()` on each event. Default implementation of this function (`ProgramBase::process_streams()`) reads events from the sources specified in the SQL program.
- *`virtual void IProgram::process_stream_event(event_t& ev)`*
  Processes each stream event passing through the system. Default implementation of this function (`ProgramBase::process_stream_event()`) does incremental maintenance work by invoking the trigger function corresponding to the event type `ev.type` for stream `ev.id` with the arguments contained in `ev.data`.

6.2.2 Retrieving the Results

The `snapshot_t IProgram::get_snapshot()` function returns a snapshot of the results of the program. The query results can then be obtained by calling the appropriate `get_TLQ_NAME()` function on the snapshot object as described in the reference of `tlq_t`. If the program is running in asynchronous mode it is guaranteed that the taken snapshot is consistent.
Currently, the mechanism for taking snapshots is trivial, in that a snapshot consists of a full copy of the `tlq_tobject` associated with the program. Consequently, the time required to obtain such a snapshot is linear in the size of the results set.

6.2.3 Basic Example

We will use as an example the C++ code generated for the `rs_example1.sql` program introduced above. In the interest of clarity some implementation details are omitted.

```
$> bin/dbtoaster examples/queries/simple/rs_example1.sql -l cpp
   -o rs_example1.hpp

#include <lib/dbt_c++/program_base.hpp>

namespace dbtoaster {

  /* Definitions of auxiliary maps for storing materialized views. */
     ...
     ...

  /* Type definition providing a way to access */
  /* the results of the SQL program */
  struct tlq_t{
    tlq_t() { }

    ...

    /* Functions returning / computing the results of */
    /* the top level queries */
    long get_RESULT(){ ... }

  protected:

    /* Data structures used for storing/computing top level queries */
    ...
  };

  /* Type definition providing a way to incrementally maintain */
  /* the results of the SQL program */
  struct data_t : tlq_t{
    data_t() { }

    /* Registering relations and trigger functions */
    void register_data(ProgramBase<tlq_t>& pb) { ... }

    /* Trigger functions for table relations */
    void on_insert_R(long R_A, long R_B) { ... }

    /* Trigger functions for stream relations */
    void on_insert_S(long S_B, long S_C) { ... }

    void on_delete_S(long S_B, long S_C) { ... }

    void on_system_ready_event() { ... }

  private:
    /* Data structures used for storing materialized views */
    ...
  };


  /* Type definition providing a way to execute the SQL program */
  class Program : public ProgramBase<tlq_t> {

  public:
    Program(int argc = 0, char* argv[] = 0) :
        ProgramBase<tlq_t>(argc,argv) {
      data.register_data(*this);
      ...
      /* Specifying data sources */
      ...
    }

    /* Imports data for static tables and performs view */
    /* initialization based on it. */
    void init() {
      process_tables();
      data.on_system_ready_event();
    }

    /* Saves a snapshot of the data required to obtain the results */
    /* of top level queries. */
    snapshot_t take_snapshot() {
      return snapshot_t( new tlq_t((tlq_t&)data) );
    }

  private:
    data_t data;
  };

}
```

Below is an example of how the API can be used to execute the SQL program and print its results:

```
#include "rs_example1.hpp"

int main(int argc, char* argv[]) {
  bool async = argc > 1 && !strcmp(argv[1],"--async");

  dbtoaster::Program p;
  dbtoaster::Program::snapshot_t snap;

  cout << "Initializing program:" << endl;
  p.init();

  cout << "Running program:" << endl;
  p.run( async );
  while( !p.is_finished() ) {
    snap = p.get_snapshot();
    cout << "RESULT: " << snap->get_RESULT() << endl;
  }

  cout << "Printing final result:" << endl;
  snap = p.get_snapshot();
  cout << "RESULT: " << snap->get_RESULT() << endl;

  return 0;
}
```

6.2.4 Custom Execution

Custom event processing can be performed on each event by overriding the virtual function `IProgram::process_stream_event(event_t&)` while still delegating the basic processing task of an event to `Program::process_stream_event()`.

Example: Custom event processing

```
namespace dbtoaster {

  class CustomProgram_1 : public Program {
    public:
      void process_stream_event(event_t& ev) {
        cout << "on_" << event_name[ev.type] << "_";
        cout << get_relation_name(ev.id) << "(" << ev.data << ")"
             << endl;

        Program::process_stream_event(ev);
      }
  };
}
```

Stream events can be manually read from *custom sources* and fed into the system by overriding the virtual function `IProgram::process_streams()` and calling `process_stream_event()` for each event read.

Example: Custom event sourcing

```
namespace dbtoaster {

  class CustomProgram_2 : public Program {
    public:
      void process_streams() {

        for( long i = 1; i <= 10; i++ ) {
          event_args_t ev_args;
          ev_args.push_back(i);
          ev_args.push_back(i+10);
          event_t ev( insert_tuple, get_relation_id("S"), ev_args);

          process_stream_event(ev);
        }
    }
  };
}
```

6.3 C++ Generated Code Reference

6.3.1 `struct tlq_t`

The `tlq_t` contains all the relevant data structures for computing the results of the SQL program, also called the top level queries. It provides a set of functions named `get_TLQ_NAME` that return the top level query result labeled `TLQ_NAME`. For our example the `tlq_t` produced has a function named `get_RESULT` that returns the query result corresponding to `SELECT SUM(r.A*s.C) as RESULT ...` in `rs_example1.sql`.

6.3.1.1 Queries Computing Collections

In the example above the result consisted of a single value. If however our query has a GROUP BY clause its result is a collection and the corresponding `get_RESULT` function will return either a `boost::multi_index_container` or a `std::map`.

Let's consider the following example:

```
$> cat examples/queries/simple/rs_example2.sql
CREATE STREAM R(A int, B int)
  FROM FILE 'examples/data/tiny/r.dat'
  LINE DELIMITED CSV (fields := ',');

CREATE STREAM S(B int, C int)
  FROM FILE 'examples/data/tiny/s.dat'
  LINE DELIMITED CSV (fields := ',');

SELECT r.B, SUM(r.A*s.C) as RESULT_1, SUM(r.A+s.C) as RESULT_2
FROM R r, S s
WHERE r.B = s.B
GROUP BY r.B;
```

The generated code defines two collection types `RESULT_1_map` and `RESULT_2_map`, and two corresponding entry types `RESULT_1_entry` and `RESULT_2_entry`. These entry structures have a set of key fields corresponding to the GROUP BY clause, in our case `R_B`, and an additional value field, `__av`, storing the aggregated value of the top level query for each key in the collection. Finally, `tlq_t` contains two functions `get_RESULT_1` and `get_RESULT_2` returning the top level query results as `RESULT_1_map` and `RESULT_2_map` objects.

```
/* Definitions of auxiliary maps for storing materialized views. */
struct RESULT_1_entry {
  long R_B;
  long __av;
  ...
};
typedef multi_index_container<RESULT_1_entry, ... > RESULT_1_map;
```

```
...

struct RESULT_2_entry {
  long R_B;
  long __av;
  ...
};

typedef multi_index_container<RESULT_2_entry, ... > RESULT_2_map;

...

/* Type definition providing a way to access */
/* the results of the SQL program */
struct tlq_t {
  tlq_t() {}

  /* Serialization Code */
  ...

  /* Functions returning / computing the results of */
  /* the top level queries */
  RESULT_1_map& get_RESULT_1(){
    ...
  }

  RESULT_2_map& get_RESULT_2(){
    ...
  }

protected:
  /* Data structures used for storing / computing top level queries */
  RESULT_1_map RESULT_1;
  RESULT_2_map RESULT_2;

};
```

If the given query has no aggregates the COUNT(*) aggregate will be computed by default and consequently the resulting collections will be guaranteed not to have any duplicate keys.

6.3.1.2 Partial Materialization

Some of the work involved in maintaining the results of a query can be saved by performing partial materialization and only computing the final results when invoking `tlq_t`'s `get_TLQ_NAME` functions. This behavior is especially desirable when the rate of querying the results is lower than the rate of updates, and can be enabled through the `-F EXPRESSIVE-TLQS` command line flag.

Below is an example of a query where partial materialization is indeed beneficial.

```
$> cat examples/queries/simple/r_lift_of_count.sql
CREATE STREAM R(A int, B int)
FROM FILE 'examples/data/tiny/r.dat'
LINE DELIMITED csv ();

SELECT r2.C FROM (
  SELECT r1.A, COUNT(*) AS C FROM R r1 GROUP BY r1.A
) r2;
```

Generated `tlq_t` without `-F EXPRESSIVE-TLQS`: We can see that `get_COUNT()` simply returns the materialized view of the results.

```
$> bin/dbtoaster examples/queries/simple/r_lift_of_count.sql -l cpp

  ...
  /* Type definition providing a way to access */
  /* the results of the SQL program */
  struct tlq_t {
    tlq_t() {}

   ...

    /* Functions returning / computing the results of */
    /* the top level queries */
    COUNT_map& get_COUNT(){
      COUNT_map& __v_1 = COUNT;
      return __v_1;
    }

  protected:

    /* Data structures used for storing / computing */
    /* the top level queries */
    COUNT_map COUNT;
  };

...
```

Generated `tlq_t` with `-F EXPRESSIVE-TLQS`: We can see that `get_COUNT()` performs some final computation for constructing the end result in a temporary `std::map` before returning it. We should remark that `tlq_t` no longer contains the full materialized view of the results `COUNT_map COUNT`, but a partial materialization `COUNT_1_E1_1_map COUNT_1_E1_1` used by `get_COUNT()` in computing the final query result.

```
$> bin/dbtoaster examples/queries/simple/r_lift_of_count.sql -l cpp
   -F EXPRESSIVE-TLQS

   ...
  /* Type definition providing a way to access */
  /* the results of the SQL program */
  struct tlq_t {
    tlq_t() { }

    ...

    /* Functions returning / computing the results of */
    /* the top level queries */
    map<long,long> get_COUNT() {
      map<long,long> __v_41;
      /* Result computation based on COUNT_1_E1_1 */
      return __v_41;
    }
  protected:

    /* Data structures used for storing / computing */
    /* the top level queries */
    COUNT_1_E1_1_map COUNT_1_E1_1;

  };
...
```

6.3.2 `struct data_t`

The `data_t` contains all the relevant data structures and trigger functions for incrementally maintaining the results of the SQL program.

For each stream based relation `STREAM_X` present in the SQL program, it provides a pair of trigger functions named `on_insert_STREAM_X` and `on_delete_STREAM_X` that incrementally maintain the query results for insertion/deletion of a tuple in `STREAM_X`. For the query presented above (`rs_example1.sql`), the produced `data_t` has the trigger functions `on_insert_S(long S_B, long S_C)` and `on_delete_S(long S_B, long S_C)`.

For static table based relations only the insertion trigger is required and will get called when processing the static tables in the initialization phase of the program.

6.3.3 `class Program`

Finally, `Program` is a class that implements the `IProgram` interface and provides the basic functionalities for reading static table tuples and stream events from their sources, initializing the relevant data structures, running the SQL program and retrieving its results.

7

Scala Code Generation

7.1 Quickstart Guide

Prerequisites:

- DBToaster Beta1 r2827
- Scala 2.9.2
- JVM (preferably a 64-bit version)

Note: The following steps have been tested on Fedora 14 (64-bit) and Ubuntu 12.04 (32-bit). The commands may be slightly different for other operating systems.

We start with a simple query that looks like this:

```
CREATE TABLE R(A int, B int)
  FROM FILE 'examples/data/tiny/r.dat'
  LINE DELIMITED CSV (fields := ',');

CREATE STREAM S(B int, C int)
  FROM FILE 'examples/data/tiny/s.dat'
  LINE DELIMITED CSV (fields := ',');

SELECT SUM(r.A*s.C) as RESULT FROM R r, S s WHERE r.B = s.B;
```

This query should be saved to a file named `rs_example.sql`.

To compile the query to Scala code, we invoke the DBToaster compiler with the following command:

```
$> bin/dbtoaster -l scala -o rs_example.scala rs_example.sql
```

This command will produce the file `rs_example.scala` (or any other filename specified by the `-o [filename]` switch) which contains the Scala code representing the query.

To compile the query to an executable JAR file, we invoke the DBToaster compiler with the `-c [JARname]` switch:

```
$> bin/dbtoaster -l scala -c rs_example rs_example.sql
```

Note: The ending `.jar` is automatically appended to the name of the JAR.

The resulting JAR contains a main function that can be used to test the query. It runs the query until there are no more events to be processed and prints the result. It can be run using the following command assuming that the Scala DBToaster library can be found in the subdirectory `lib/dbt_scala`:

```
$> scala -classpath "rs_example.jar:lib/dbt_scala/dbtlib.jar" \
   org.dbtoaster.RunQuery
```

After all tuples in the data files were processed, the result of the query will be printed:

```
Run time: 0.042ms
<RESULT>156 </RESULT>
```

7.2 Scala API Guide

In the previous example, we used the standard main function to test the query. However, to make use of the query in real applications, it has to be run from the application itself. The following example shows how a query can be run from your own Scala code. Suppose we have a the following source code in `main_example.scala`:

```
import org.dbtoaster.Query

package org.example {
  object MainExample {
    def main(args: Array[String]) {
      Query.run()
      Query.printResults()
    }
  }
}
```

The code representing the query is in the `org.dbtoaster.Query` object. This program will start the query using the `Query.run()` method and output its result after it finished using the `Query.printResults()` method. To retrieve results, the `getRESULTNAME()` method of the `Query` object can be used.

Note: The `getRESULTNAME()` functions are not thread-safe, meaning that results can be inconsistent if they are called from another thread than the query thread. A thread-safe alternative to retrieve the results is planned for future versions of DBTOASTER.

The program can be compiled to `main_example.jar` using the following command (assuming that the query was compiled to a file named `rs_example.jar`):

```
$> scalac -classpath "rs_example.jar" -d main_example.jar \
   main_example.scala
```

The resulting program can now be launched with:

```
$> scala -classpath \
   "main_example.jar:rs_example.jar:lib/dbt_scala/dbtlib.jar" \
   org.example.MainExample
```

The `Query.run()` method takes a function of type `Unit => Unit` as an optional argument that is called every time when an event was processed. This function can be used to retrieve results while the query is still running.

Note: The function will be executed on the same thread on which the query processing takes place, blocking further query processing while the function is being run.

7.3 Generated Code Reference

The DBTOASTER Scala code generator generates a single file containing an object `Query` in the package `org.dbtoaster`. For the previous example the generated code looks like this:

```
...

package org.dbtoaster {
  // The generated object
  object Query {
    // Declaration of sources
    val s1 = createInputStreamSource(
      new FileInputStream("'examples/data/tiny/r.dat"), ...
    )
     ...

    // Data structures holding the intermediate result
    var RESULT = SimpleVal[Long](0)
    ...
    // Functions to retrieve the result
    def getRESULT():Long = {
      RESULT.get()
    }

    // Trigger functions
    def onInsertR(var_R_A: Long,var_R_B: Long) = ...
    ...
    def onDeleteS(var_S_B: Long,var_S_C: Long) = ...

    // Functions that handle static tables and system initialization
    def onSystemInitialized() = ...
    def fillTables(): Unit = ...

    // Function that dispatches events to the appropriate
    // trigger functions
    def dispatcher(event: DBTEvent,
                   onEventProcessedHandler: Unit => Unit): Unit = ...

    // (Blocking) function to start the execution of the query
    def run(
        onEventProcessedHandler: Unit => Unit = (_ => ())): Unit = ...

    // Prints the query results in some XML-like form (for debugging)
    def printResults(): Unit = ... } }
```

When the `run()` method is called, the static tables are loaded and the processing of events from the declared sources starts. The function returns when the sources provide no more events.

7.3.1 Retrieving Results

To retrieve the result, the `getRESULTNAME()` functions are used. In the example above, the `getRESULTNAME()` method is simple but more complex methods may be generated and the return value may be a collection instead of a single value.

7.3.1.1 Queries computing collections

Consider the following query:

```
CREATE STREAM R(A int, B int)
  FROM FILE 'examples/data/tiny/r.dat'
  LINE DELIMITED CSV (fields := ',');

CREATE STREAM S(B int, C int)
  FROM FILE 'examples/data/tiny/s.dat'
  LINE DELIMITED CSV (fields := ',');

SELECT r.B, SUM(r.A*s.C) as RESULT_1, SUM(r.A+s.C) as RESULT_2
FROM R r, S s WHERE r.B = s.B
GROUP BY r.B;
```

In this case two functions are being generated that can be called to retrieve the result, each of them representing one of the result columns:

```
def getRESULT_1():K3PersistentCollection[(Long), Long] = ...
def getRESULT_2():K3PersistentCollection[(Long), Long] = ...
```

In this case, the functions return collections containing the result. For further processing, the results can be converted to lists of key-value pairs using the `toList()` method of the collection class. The key in the pair corresponds to the columns in the GROUP BY clause, in our case `r.B`. The value corresponds to the aggregated value for the corresponding key.

7.3.1.2 Partial Materialization

Some of the work involved in maintaining the results of a query can be saved by performing partial materialization and only computing the final results when invoking `tlq_t`'s `get_TLQ_NAME` functions. This behavior is especially desirable when the rate of querying the results is lower than the rate of updates, and can be enabled through the `-F EXPRESSIVE-TLQS` command line flag.

Below is an example of a query where partial materialization is indeed beneficial.

```
CREATE STREAM R(A int, B int)
  FROM FILE 'examples/data/tiny/r.dat'
  LINE DELIMITED csv ();

SELECT r2.C FROM (
  SELECT r1.A, COUNT(*) AS C FROM R r1 GROUP BY r1.A
) r2;
```

When compiling this query with the `-F EXPRESSIVE-TLQS` command line flag, the function to retrieve the results is much more complex, unlike the functions that we have seen before. It uses the partial materialization `COUNT_1_E1_1` to compute the result:

```
$> bin/dbtoaster -l scala -F EXPRESSIVE-TLQS  \
   examples/queries/simple/r_lift_of_count.sql

  def getCOUNT():K3IntermediateCollection[(Long), Long] = {
    (COUNT_1_E1_1.map((y:Tuple2[(Long),Long]) =>
    ...
    )
  }
```

7.3.2 Using Queries in Java Programs

Since Scala is compatible with Java, it is possible to use the queries in Java applications. In order to use a query in Java, the Java application has to reference three libraries:

- The library containing the generated code for the query
- The Scala library
- The Scala DBTOASTER library

The following code snippet illustrates how a query can be executed from within a Java application:

```
import org.dbtoaster.dbtoasterlib.*;
import org.dbtoaster.*;

public class MainClass {
  public static void main(String[] args) {
    final Query q = new Query();

    QueryInterface.DBTMessageReceiver rcvr =
      new QueryInterface.DBTMessageReceiver() {

      public void onTupleProcessed() {
        // do nothing
      }

      public void onQueryDone() {
        // print the results
        q.printResults()
      }
    }

    QueryInterface.QuerySupervisor supervisor =
      new QueryInterface.QuerySupervisor(q, rcvr);
    supervisor.start();
  }
}
```

This program executes the referenced query and prints the result once there are no more tuples to be processed.

7.3.2.1 Compiling in Eclipse

To compile the previously presented program in eclipse (with the official Scala plugin installed), we can create a new Java project with a single `.java`-file containing the program. In the Java Build Path section of the project's properties the previously listed libraries have to be added. The Scala library can be added by clicking the "Add Library" button, while the other libraries can be added as external JARs. After adding the libraries to the project, the project should compile and execute the query once run.