Constrained Data Base with Slow Streams

Stream Reasoning can be described as Reasoning Upon Rapidly Changing Information [1].

[1] (E. Della Valle; F. Heintz, 2018)

Linear temporal logic or linear-time temporal logic (LTL) is a modal temporal logic with modalities referring to time. In **LTL**, one can encode formulae about the future of paths, e.g., a condition will eventually be true, a condition will be true until another fact becomes true. [2]

[2] (https://en.wikipedia.org/wiki/Linear_temporal_logic)

- We will not revise extensively neither Stream Reasoning nor LTL here, please consider reading more about to further understand why **Constrained Data Base with Slow Streams** is different from all available stream reasoning methods, because **Constrained Data Base with Slow Streams** addresses another problem**.**

Stream Reasoning tries to put constraints on streams in *real time*, over a window of time (memory) so as to decide whether arriving data fits a logical rule or not. Stream Reasoning applies reason on arriving data and past data with a window memory.

LTL can be an expression language of Stream Reasoning [3]

[3] (https://www.diva-portal.org/smash/get/diva2:1043981/FULLTEXT02.pdf)

Context

- If we think of a database with a streaming data source, the problem of data consistency can be present. Generally, even if a database business and functional specifications consists of hundreds or thousands of logical rules (e.g.: table1.column1 > table1.column2 …), generally we only care about already arrived data, but not future arriving data, as long as data consistency is maintained in the database. We can ignore rejects, log them, or historize them, but we have no control over neither:
    - **"Anticipated" future rule changes,**

    nor

    - **Data changes in the future, even if we have knowledge of future changes.**

    Note: these two cases are common with real time applications (databases also) with multiple data sources and frequent changes from data providers.

- Data consistency in *real time databases* can be achieved with SQL TRANSACTION command, or beforehand in the ETL step (using other programming languages).
- Classical constrained data intends to control changes over domain definition of data.

- Constrained Data-Base with “Slow Streams” intends to control, by anticipation, future changes of:
    - Data schema.

    and

    - Business rules.
- Specifying a framework to force by imposed structures and rules acting on streaming data using Linear Time Logic combined with Constraint Logic Programming can make systems in this context more robust to change.

⇨ **When to consider Constrained Data-Base with Slow Streams**

In the case, an application has not full control of
- Schema of its sourcing feeds.
- Definition domain (values of columns).
- Changes are slow (humanly predictable) in contrast with constant unpredictable mutations of data feeds by and from another system.

If we are more concerned with a “consistent database” rather than a “consistent stream” (time window of arriving data), or if we have at least a minor vision on the future arriving data (we will talk about this later), we can use:
- **LTL language.**
- **Constraint Logic Programming language:**
  [SQL: https://arxiv.org/pdf/1907.10914.pdf ,
  OCL: https://en.wikipedia.org/wiki/Object_Constraint_Language]
- **Event-Condition-Action.**

All three in order to:
- Expect new changes of data definition in the future.
- Specify in advance known changes of rules in the future.
- Historize events and actions.

- **Why Slow Streams**

  Talking about Streams in academia, we find generally all subjects address performance problems, solutions to rapid (changing or not) of input data to a system. We find *reactive programming* for example based on *asynchronous* calls and *decoupling* of functionalities of a system, all benefiting *parallel computing*. We are not addressing this issue. *Slow streams* here emphasize that is not the subject. Rather, we focus on maintaining a system robust on rapid changes of business specifications.

  Normally data does not change at the same rate of its arrival speed, otherwise It would be complete random and non-useful data; Except when we are talking about unstructured data that is meant to be so to some extent.

  When we have changing sourcing data and some prior knowledge of how these changes occur (upon business specification changes), we can still have a consistence database Using **LTL, Constraint Logic Programming**, and an **event-condition-action** language. We can express near future “known changes”.

  With the described pattern and definitions, we hopefully can have a consistent database by pushing and changing rules in a defined **LTL-CONSTRAINT-EVENT-ACTION** layer in contrast with traditional ETL/Databases solutions.

⇨ **Specification**

A run (a version) can be described as a set of rules and actions.

Actions can be BACK-UP, IGNORE, HISTORISE, …

Conditions can target columns in different tables.

Action set

{

```
run1 : @cond WHEN tableA.ID == tableB.ID AND tableA.column1 > tableB.column2
        BACK-UP
Except LOG ('Rule 1 error')
Finally LOG ('Rule 1 chosen')

run2 : @cond WHEN tableA.column1 > 0
        DO NOT BACK-UP
Except LOG ('Rule 2 error')
Finally LOG ('Rule 2 chosen')

run3 : @cond WHEN exists(tableC.column1)
        BACK-UP tableC.column1
Except LOG ('Rule 3 error')
Finally LOG ('Seeing a new business change on sourcing data')
}
```

Rule set

{

```
ALWAYSE (run2 AND run1) UNTIL run3@cond
```

}

Can omit @cond and expect “run” as a condition when it is an operand in an LTL expression.

⇨ **LTL-C (LTL-Constraint) variant**

As described above, LTL is meant to express a *fact. H*owever, the expression highlighted in green, is different, because it is bound to an *event-action* statement; it is meant to impose itself on the system (if always run successfully, then it evaluates to true), otherwise the system raises an exception. The system *obeys* the LTL formulae choosing from possible actions so as to satisfy the abstract LTL formulae *as soon as possible*. In other words, the system is always searching for *a sub-domain* from the action set that satisfies all *LTL formulas in the rule set*. It is a form of constraint optimization paradigm expressed over LTL.

Analogy to Predicate logic:

First-order logic uses quantified variables over non-logical objects like *5+5 == 10*

LTL-C is a higher order logic where constraints expressed in a constraint language are the propositional variables for LTL formulas.

Example

ALWAYSE (run2 , run1) UNTIL run3@cond run3

In this example: run1, run2, and run3 are all self-conditioned runs, they execute only when their conditions are met, if their condition is not met, they don't necessarily fail. For instance: run1 can be run only if run1@cond is satisfied.

The query means the system always runs the conditioned actions run2 and run1 while checking also every time whether run3@cond is satisfied, as soon as run3@cond it is satisfied, then run3 is executed instead of run1 AND run2.

Discussion

Read more about the idea (in its early stage) in my posts on the internet:

https://stackoverflow.com/q/58199623/1951298

at the time, I talked about TDDA which is an open source library which only describes constraints on columns in the same table at that time. At that time, I thought it would be a sufficient solution as I did not picture the abstraction described above.

The second limit, is that constraints are expressed only over arriving data and cannot be extended to future changes like LTL would do.

LTL-C Memory engine (not yet specified)

The proposed LTL-EVENT-ACTION solution needs a memory. It caches actions “runs” to be able to handle future events. (Like “until”, “sometimes”, “eventually”, etc.).