Running Temporal Logical Queries on Knowledge Bases

Abstract

State of the art for model checking exploit computationally intensive solutions, bottlenecked by either repeated data access or suboptimal algorithmic implementations. Our solution outperforms the previous solutions while proposing novel temporal logic operators for accessing relational tables.

Keywords: Logical Artificial Intelligence, Knowledge Bases, Query Plan, Temporal Logic

Topics Physical Layout, Algorithms

1 Introduction

Conformance checking is an integral part of Artificial Intelligence bridging data mining and business process management [**?**]. It assesses whether a sequence of distinguishable events (i.e., a *trace*) conforms to the expected process behaviour represented as a *(process) model* [**?**]. Each event is associated with both an *activity label* describing the captured event, as well as payload data, either associated to the whole trace or to a specific event. When multiple distinct traces are considered in a log, model checking lists the traces satisfying the model [**?**]. Non-conforming traces are usually referred to as *deviant* [**?**]. *Declarative* models are composed of multiple human-readable *clauses* that should be jointly satisfied (i.e., *conjunctive query*) [**?**]; each of these is the instantiation of a specific behavioural pattern (i.e., *template*) expressing temporal correlations between actions being carried out thus linking preconditions to expected outcomes. Such correlations might data conditions, thus expressing Θ-joins between activated and targeted events. Models can be expressed as Finite State Machines [**?**, **?**] but, by doing so, each state will represent a possible state configuration the system might find itself in, for which we need to describe all the reasonable actions and data conditions the system might find itself in. This makes graph data-aware model checking as [**?**] rather inefficient, as the size of these graphs becomes exponential with respect to the original size of the declarative model. As a result, this increases the computational time required for conformance checking. Furthermore,

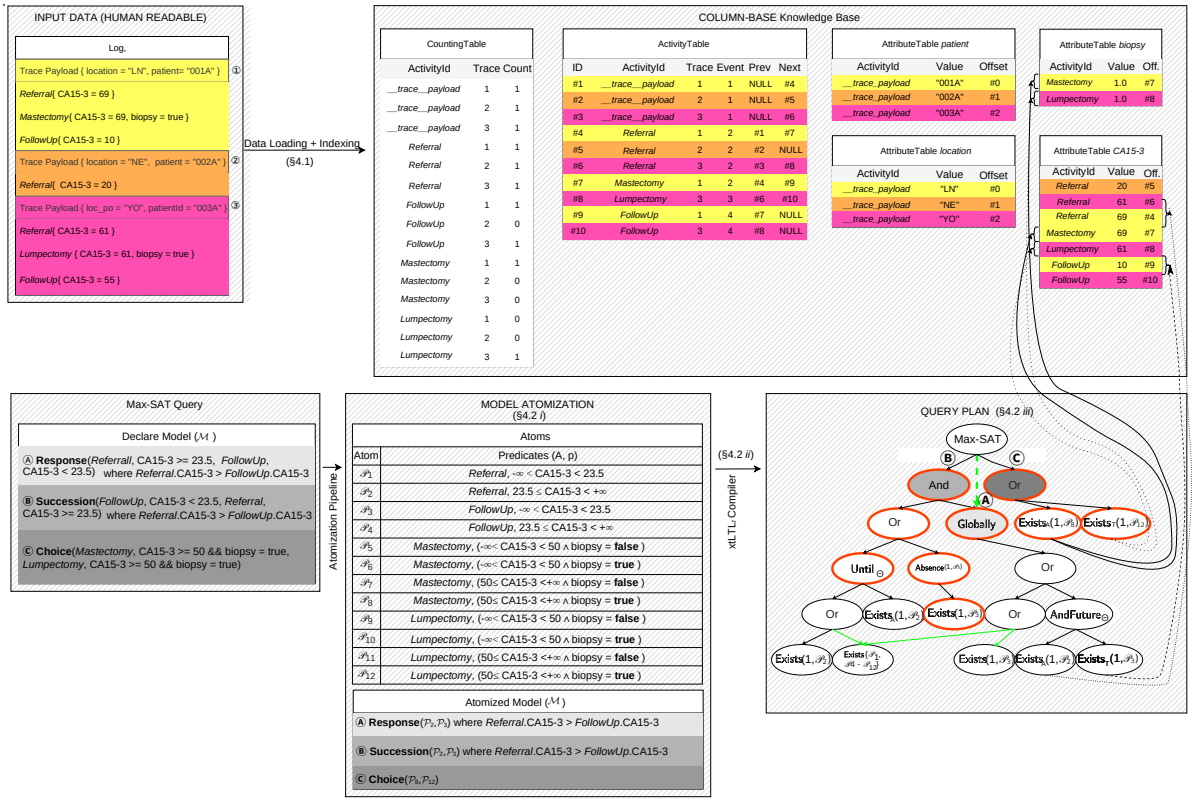


Figure 1: KnoBAB Architecture for Breast Cancer patients. Each trace ①-③ represents one single patient's clinical history, are represented with unique colouring. Please observe that the atomization process does not consider data distribution but rather partitions the data space as described by the data activation and target conditions. In the query plan, green arrows remark access to shared sub-queries as in [**?**], and thick red ellipses remark which operators are untimed.

such models are incapable of expressing Θ-correlation conditions on the data payload, thus limiting the models' expressiveness.

Conformance checking with declarative models is a well-studied technology at the core of AI's temporal decision making. *Firstly*, conformance checking is adopted when mining a model from logs either containing only positive (or negative) traces [**?**], or on logs containing both, but where positive traces can be discriminated from the negative ones via behavioural or data conditions, thus allowing to generate both a positive and a negative model [**?**]. In a hospital scenario, a positive model might both describe a medical protocol and the desired patients' health condition at each step. E.g., For the positive models in Figure

Real business use case scenarios usually require Θ-correlations. In a goods brokerage scenario [**?**], items are traded between producers (vendors) and retailers (customers): each transaction starts with a vendor sending a sales quotation to a customer. If an offer is accepted and the order is confirmed, then the item is scheduled for delivery. When ready, a logistic operator collects it. In this scenario, deviant traces either do not reflect the company's rules or will potentially lead to retailers' complaints: e.g., a late delivery complaint can occur only if the date the product is received is greater than the agreed time to receive it as registered in a previous agreement event. This situation cannot be directly expressed as a temporal pattern, as we also need to test the timestamps associated in the data payload. Conformance checking can be applied to several unexplored non-business domains. Most recent **video games** exploit AI features [**?**]: existing state of the art exploits automata [**?**] for modelling Non-Player Character's behaviours. As Declarative models and automata are completely equivalent approaches, developers might exploit the former to compactly represent the latter. Furthermore, as debugging AI in video games is a crucial challenge [**?**], conformance checking solutions might be exploited for debugging unexpected behaviours. As AAA video games already track and log both players and NPC actions[1], it might be also possible to use game logs for distinguishing winning strategies from losing ones [**?**]. As a result, analysis of an ongoing trace at runtime might 'suggest' actions beneficial to the player based on the game state.

Given that conformance checking is at the heart of both trace validation and model mining, it is of crucial relevance to optimize such a task. Solutions enabling conformance checking via model mining through SQL queries [**?**, **?**] neither explicitly evaluate the satisfiability for every single trace, nor return the traces that satisfy them, but only associate support and confidence values to each of said clauses for model mining purposes. However, as shown in this paper, these queries can be extended to both evaluate satisfiability per trace and return the set of traces satisfying every single clause, thus adhering to the definition from conformance checking literature (§

Even state-of-the-art implementations explicitly engineered to solve the conformance checking problem without relying on a relational representation of traces, are not particularly efficient [**?**]. This solution, not being able to assemble the previously described LTL_f operators within

[1] `https://battlefieldtracker.com/`

Type	Exemplifying clause (c_l)	Natural Language Specification for Traces	LTL$_f$ Semantics ($[\![c_l]\!]$)
Simple	$\mathsf{Init}(A, p)$	The trace should start with an activation	$A \wedge p$
	$\mathsf{Exists}(A, p, n)$	Activations should occur at least n times	$\mathbf{F}(A \wedge p \wedge \mathbf{X}(\mathsf{Exists}(A, p, n-1)))$
	$\mathsf{Absence}(A, p, n+1)$	Activations should occur at most n times	$\neg\mathsf{Exists}(A, p, n+1)$
	$\mathsf{Precedence}(A, p, B, q)$	Events preceding the activations should not satisfy the target	$\neg(B \wedge p)\ \mathbf{W}\ (A \wedge p)$
(Mutual) Correlation	$\mathsf{ChainPrecedence}(A, p, B, q)$	The activation is immediately preceded by the target.	$\mathbf{G}(\mathbf{X}(A \wedge p) \Rightarrow (B \wedge q))$
	$\mathsf{Choice}(A, p, B, q)$	Either the activation or the target condition must appear.	$\mathbf{F}(A \wedge p) \vee \mathbf{F}(B \wedge q)$
	$\mathsf{Response}(A, p, B, q)$	The activation is either followed by or simultaneous to the target.	$\mathbf{G}((A \wedge p) \Rightarrow \mathbf{F}(B \wedge q))$
	$\mathsf{ChainResponse}(A, p, B, q)$	The activation is immediately followed by the target.	$\mathbf{G}((A \wedge p) \Rightarrow \mathbf{X}(B \wedge q))$
	$\mathsf{RespExistence}(A, p, B, q)$	The activation requires the existence of the target.	$\mathbf{F}(A \wedge p) \Rightarrow \mathbf{F}(B \wedge q)$
	$\mathsf{ExlChoice}(A, p, B, q)$	Either the activation xor the target happen.	$(\mathbf{F}(A \wedge p) \vee \mathbf{F}(B \wedge q)) \wedge \mathsf{NotCoExistence}(\mathsf{A}, p, \mathsf{B}, q)$
	$\mathsf{CoExistence}(A, p, B, q)$	RespExistence, and vice versa.	$\mathsf{RespExistence}(\mathsf{A}, p, \mathsf{B}, q) \wedge \mathsf{RespExistence}(\mathsf{B}, q, \mathsf{A}, p)$
	$\mathsf{Succession}(A, p, B, q)$	The target should only follow the activation.	$\mathsf{Precedence}(\mathsf{A}, p, \mathsf{B}, q) \wedge \mathsf{Response}(\mathsf{A}, p, \mathsf{B}, q)$
	$\mathsf{ChainSuccession}(A, p, B, q)$	Activation immediately follows the target, and the target immediately preceeds the activation.	$\mathbf{G}((A \wedge p) \Leftrightarrow \mathbf{X}(B \wedge q))$
	$\mathsf{AltResponse}(A, p, B, q)$	If an activation occurs, no other activations must happen until the target occurs.	$\mathbf{G}((A \wedge p) \Rightarrow (\neg(A \wedge p)\ \mathbf{U}\ (B \wedge q)))$
	$\mathsf{AltPrecedence}(A, p, B, q)$	Every activation must be preceded by an target, without any other activation in between	$\mathsf{Precedence}(\mathsf{A}, p, \mathsf{B}, q) \wedge \mathbf{G}((A \wedge p) \Rightarrow \mathbf{X}(\neg(A \wedge p)\ \mathbf{W}\ (B \wedge q))$
Not.	$\mathsf{NotCoExistence}(A, p, B, q)$	The activation nand the target happen.	$\neg(\mathbf{F}(A \wedge p) \wedge \mathbf{F}(B \wedge q))$
	$\mathsf{NotSuccession}(A, p, B, q)$	The activation requires that no target condition should follow.	$\mathbf{G}((A \wedge p) \Rightarrow \neg\mathbf{F}(B \wedge q))$

Table 1: Declare templates illustrated as exemplifying clauses. $A \wedge p$ ($B \wedge q$) represents the *activation* (*target*) condition, A (B) denotes the activity label, and p (q) is the data payload condition.

a query plan, can neither minimize the access operations to the trace data nor minimize the re-computation of sub-expressions that appear frequently in the model as recently proposed by [**?**]. Further experimental shreds of evidence support such theoretical claims (§

Our proposed solution is then implemented in KnoBAB[2]: we are synthesising logs derived from a system (be it digital or real) to a column-store knowledge base ad-hoc implemented for conformance checking (§

2 Related Work

XES Log Model (Data) *payloads* are maps associating attributes (i.e., *keys*) to data values. Given a finite set of activity labels Act, an event σ_j^i is a pair $\langle \mathsf{a}, p \rangle$, where $\mathsf{a} \in \mathsf{Act}$ is an activity label, and p is a payload, mapping each key to a single value. A *trace* σ^i is a temporally-ordered and finite sequence of distinct events $\sigma^i = \sigma_1^i \cdots \sigma_n^i$, modelling a process run. All events within the same trace associate the same values to the same trace keys. A log $\mathcal{L}$ is a finite set of traces $\{\ \sigma^1, \ldots, \sigma^m\ \}$. We denote $\Sigma \subseteq \mathsf{Act}$ as the set of all the possible activity labels in the log. If traces also contain a payload, then this can be easily mimicked by adding an extra event containing such a payload, $\_\_\mathsf{trace\_payload}$, at the beginning of the trace. This characterization [**?**] is compliant with the eXtensible Event Stream (XES) format, which is the *de facto* standard for representing event logs within the Business Process Management community [**?**].

[2][URL REMOVED FOR DOUBLE-BLIND REVIEW].

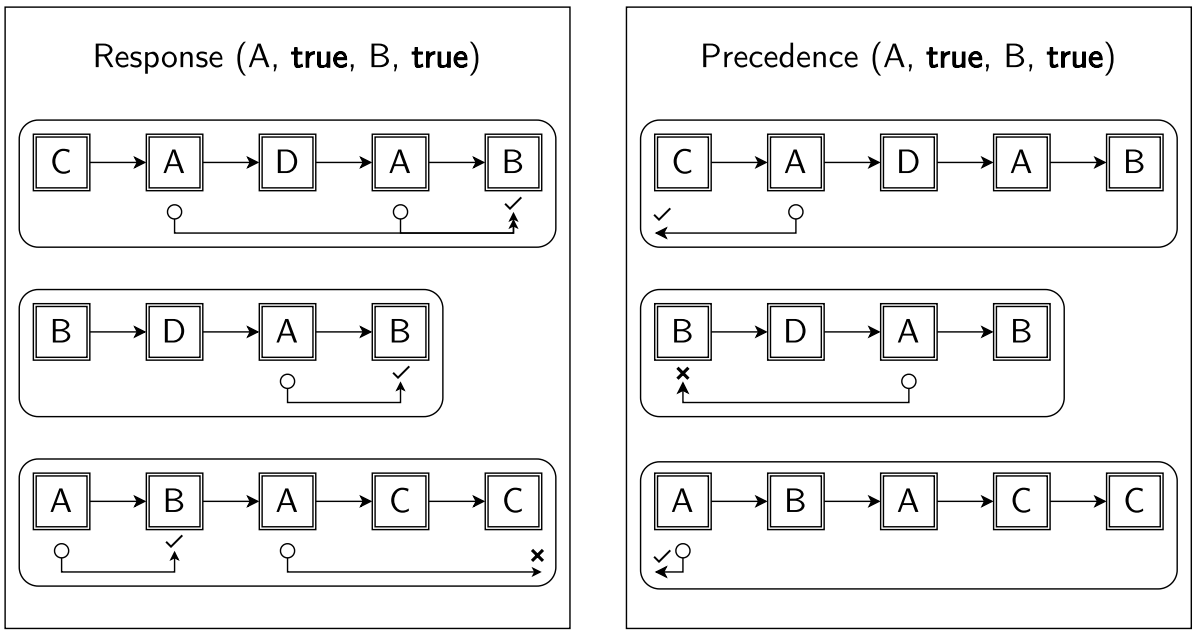


Figure 2: Traces describing the events generated by each hospital unit: those are temporally ordered events associated to *activity labels* (boxed). Activated (or targeted) events here circled (or ticked/crossed). Ticks (or crosses) indicate a (un)successful match of a target condition.

Conformance Checking Temporal declarative languages pinpoint recurring temporal patterns in highly variable scenarios so as to describe them compactly for both machines and humans [**?**]. Every single temporal pattern is expressed through *templates* (i.e., an abstract parameterized property: Table

Despite this formulation has been already extended so to support correlation constraints [**?**], such a solution is affected by the following two deficiencies: first, correlation conditions have to be represented alongside the target condition levels, thus hampering the exploitation of efficient relational database algorithms for correlation conditions via joins. Furthermore, these operators can only assess the validity of one trace at a time while, on the other hand, we might need to assess the satisfiability of multiple traces at the same time by composing partial results returned by every single operator. These operators cannot be directly exploited as query operators, where multiple traces are considered contemporarily. For this reasons, we propose a reformulation of such operators in §

Data-Aware Conformance Checking `Declare Analyzer`[3] [**?**] proposes one of the latest solutions for conformance checking over data-aware logs. Declare templates are decomposed into LTL_f expressions (as per the last column of Table

Last, each clause is completely hardcoded and, as they do not support novel templates via the definition of novel LTLf formulae, as we instead do. The addition of further Declare clauses would require an entirely new implementation. KnoBAB, on the other hand, supports

[3] `http://www.promtools.org/doku.php?id=prom611`

the definition of potential new Declare templates via configuration files loaded at warm-up, thus enabling a more general result that goes beyond the Declare language and that can be applied to any temporal specification exploiting LTL_f.

Process Mining through Conformance Checking Some approaches utilise conformance checking as a mechanism to mine declarative models from an event log: a scoring function tests the validity of each possible clause over each possible trace. SQLMiner [**?**] does so via SQL queries [**?**] where each specified declarative template is converted into a SQL query. E.g., given the SQL formulation for the Response template, the query returns a table (Activation,Target,Score) where each row $\langle \mathsf{a}, \mathsf{b}, s \rangle$ represents a candidate clause $\mathsf{Response}(\mathsf{a}, \mathbf{true}, \mathsf{b}, \mathbf{true})$, and s is the score associated to the candidate clause.

Each event log, as well as each activation and target activity label for generating the candidate Declare constraints to be tested, are stored in distinct relational tables. While the former are represented in Log(Id,Trace,ActivityId,Event), the latter are stored in Actions(ActivationId, TargetId). The authors consider Support and Confidence scoring functions to determine the precision and reliability of the calculation. Records which do not pass pre-determined Support and Confidence thresholds are filtered out from the data. While SQL also supports data constraints, this solution considers Declare clauses with neither activation, nor target, nor correlation ones with payload predicates.

Despite the authors exploit data perspectives in 'Resource Assignment Constraints' clauses, distinct from the Declare ones, only trace payload conditions are considered. Instead, KnoBAB supports payload information and predicate testing both *per trace* and *per event*, which could also be stored in a separate table as SQLMiner suggests, thus providing greater expressiveness per clause. SQLMiner queries can be chained together using **`SET UNION`**, though this provides no possibility for testing which are the clauses that are satisfied by the majority of the traces (Max-SAT). These query plans are not optimized as in [**?**], thus failing at both minimizing the data access and running multiple shared sub-queries only once. This is inferior to KnoBAB, which has the ability to process multiple declarative clauses from disparate templates.

3 Logical Model

3.1 (Intermediate) Result Representation

Within the computation pipeline, (intermediate) results are represented as a set of triplets $\langle i, j, L \rangle$ representing that, starting from event σ^i_j in trace σ^i, we might observe activation, target, or correlation conditions in L, an ordered vector. While for activation and target we preserve the matched event id, correlations keep track of both the activation and the target condition leading to the satisfaction of a given Θ predicate (see the next section). This is a sensible representation, as per declarative constraints, it may exist only one possible Θ predicate. Such triplets are sorted by trace id and event id, and operators manipulating those

(§

Our proposed representation is different from the one provided by [**?**] which cannot represent for each event within a trace all the possible activation, target, or join condition happening in the future, as it is impossible to represent single trace events that are not necessarily represented by activation or target conditions. As observed in §

3.2 eXTended LTL$_\mathbf{f}$ operators

We now propose the extended LTL$_\text{f}$ operators (xtLTL$_\text{f}$) directly exploited by our pipeline:

$$\begin{aligned}\phi := \ & \mathsf{Init}_{A/T}(A,p) \mid \mathsf{End}_{A/T}(A,p) \mid \mathsf{Exists}_{A/T}(n,A,p) \mid \mathsf{Absence}_{A/T}(n,A,p) \\ & \mid \mathsf{Next}(\phi) \mid \mathsf{Globally}(\phi) \mid \mathsf{Future}(\phi) \mid \mathsf{Not}(\phi) \\ & \mid \mathsf{Or}(\phi,\phi',\Theta) \mid \mathsf{And}(\phi,\phi',\Theta) \mid \mathsf{Until}(\phi,\phi',\Theta) \\ & \mid \mathsf{AndGlobally}(\phi,\phi',\Theta) \mid \mathsf{AndFuture}(\phi,\phi',\Theta) \mid \mathsf{AndNextGlobally}(\phi,\phi',\Theta)\end{aligned}$$

Operators in the first line filter traces' events and represent these into the previously-described result representation. Init (End) returns the events at the beginning (end) of each trace satisfying the condition $A \wedge p$. Similarly to [**?**], each of these operators might be expressed as either an untimed or as a timed specification. Any operator will be considered timed by default when appearing inside a timed operator, like Next, Globally, Future, Until, and any other composed operator from the last line. E.g., In Figure

The next two lines report the same operators described in §

The remaining operators merge multiple operators together when a specific implementation outperforms the execution of the operators separately: e.g., $\mathsf{AndFuture}(\phi,\phi',\Theta)$ is equivalent to
$\mathsf{And}(\phi,\mathsf{Future}(\phi'),\Theta)$, but preliminary experiments reveal that the former has a more efficient implementation than computing the latter. This choice was inspired by relational algebra, where θ-joins are usually more efficient than performing a join and a selection operation separately. On the other hand, $\mathsf{Implies}(\phi,\phi',\Theta)$ is rewritten as $\mathsf{Or}(\mathsf{Not}(\phi), \mathsf{And}(\phi,\phi',\Theta), \mathbf{true})$. As per previous discussion, the left leaf of $\mathsf{AndFuture}_\Theta$ in Figure

Each xtLTL$_\text{f}$ operator is going to both return and/or accept data in the result representation, thus making such operators closed on such format.

4 KnoBAB Architecture

The methodology behind its design systematically follows the major architectural components of a relational database, with the only bespoke characteristics of tailoring such solution to the specific problem that we intend to solve (§

Algorithm 1 xtLTL$_\mathrm{f}$ pseudocode implementation for the basic timed operators

```
function Future(φ)
   for all ⟨t, e, L⟩ ∈ φ do yield ⟨t, e, ⋃{ L′ | ⟨t, e′, L′⟩ ∈ φ and e′ ≥ e }⟩
   end for
function Globally(φ)
   for all ⟨t, e, L⟩ ∈ φ do
      E ← { e′ | ⟨t, e′, L′⟩ ∈ φ and e′ ≥ e }
      if |E| = ℓ_t − e then yield ⟨t, e, ⋃{ L′ | ⟨t, e′, L′⟩ ∈ φ and e′ ∈ E }⟩
      end if
   end for
function Next(φ)
   for all ⟨t, e, L⟩ ∈ φ s.t. e > 1 do yield ⟨t, e − 1, L⟩
   end for
function CommonJoin(φ, φ′, Θ, isDisjunctive)

   it ←Iterator(φ), it ←Iterator(φ′)
   while it ≠ ∅ and it′ ≠ ∅ do
      ⟨t, e, L⟩ ← current(it), ⟨t′, e′, L′⟩ ← current(it′)
      if t = t′ and e = e′ then
         if L = ∅ then L″ ← L′
         else if L′ = ∅ then L″ ← L
         else
            L″ ← ∅
            for all A(m) ∈ L and T(n) ∈ L′ s.t. Θ(m, n) do
               L″ ← L″ ∪ { M(m, n) }
            end for
         end if
         yield ⟨t, e, L″⟩; next(it); next(it′);
      else if t < t′ or (t = t′ and e < e′) then
         if isDisjunctive then yield ⟨t, e, L⟩
         end if
         next(it)
      else
         if isDisjunctive then yield ⟨t′, e′, L′⟩
         end if
         next(it′)
      end if
   end while
function And(φ, φ′, Θ) CommonJoin(φ, φ′, Θ, false)
function Or(φ, φ′, Θ) CommonJoin(φ, φ′, Θ, true)
function Until(φ, φ′, Θ)
   for all t s.t. ⟨t, i′, L′⟩ ∈ φ′ do
      α ← 1; Map ← {}; i ← min_ι ⟨t, ι, L⟩ ∈ φ′; I ← max_ι ⟨t, ι, L⟩ + 1
      while i < I do
         if exists ⟨t, j, L⟩ ∈ φ s.t. j < i then
            if ⟨t, α, L_α⟩, ⟨t, α + 1, L_{α+1}⟩, . . . , ⟨t, i − 1, L_{i−1}⟩ ∈ φ,
                 and Θ(α, j) for all T(j) ∈ L_α ∪ ··· ∪ L_{i−1}        then
               Map[i] ← { M(k, i) | k ∈ L_α ∪ ··· ∪ L_{i−1} }
               i ← i + 1
            else α ← α + 1
            end if
         else α ← i
         end if
      end while
   end for
```

4.1 Data Loading

The data loading phase loads logs serialized in multiple formats, thus including the XML-based XES standard, a tab-separated events' activity labels, and the HUMAN READABLE LOG FORMAT firstly introduced in [**?**]. We use different data parsers, which are still linked to the same data loading primitives.

If the log does not contain data payloads, the entire log can be represented into two relational tables, CountingTable(ActivityId,Trace,
Count) and ActivityTable(ActivityId,Trace,Event,Prev,Next). While the former counts the occurrence of each activity label in Σ for each trace, the latter lists all of the possible events similarly to SQLMiner. Both tables compactly represent the initial three columns as a 64-bit unsigned integer, which is also used to sort the tables in ascending order. A row $\langle \mathsf{a}, j, h \rangle$ from CountingTable states that there are h events exhibiting the activity label a in the trace σ^j; each row $\langle \mathsf{a}, j, i, q, q' \rangle$ from ActivityTable states that the i-th event of the j-th trace ($\sigma_i^j = \langle \mathsf{a}, p \rangle$) is labelled as a, while q (or q') is the pointer to the immediately preceding σ_{i-1}^j (or following, σ_{i+1}^j) event within the trace if any. NULLs from Figure

If, on the other hand, the log is associated to either trace or event payloads, we exploit the query and memory-efficient column-based model [**?**], thus representing all of the values v associated to a payload key k within the rows from AttributeTable$k$. In our implementation, each row $\langle \mathsf{a}, v, i \rangle$ from AttributeTable$k$(ActivityId,Value,Offset) represents a value v associated to the key k, where i determines the location where the event containing the accessed value is located in ActivityTable; this provides the trace id and event id required for the intermediate representation. To perform payload-based queries efficiently, the table is sorted in ascending order by the three columns. As each data condition is always associated with a given activity label, those can be effectively run as data range queries run via binary search algorithms. From Figure

CountingTable is mainly accessed for existential and Exists and Absence templates where no data payload is specified, while ActivityTable is used for either returning all of the events within the log associated to a given activity label or returning all of the events happening at either the beginning or at the end of a trace. Each table AttributeTable$k$, on the other hand, will return all the events satisfying a given condition associated with a specific key k.

After loading the whole dataset, the number of the traces within the log $|\mathcal{L}|$, the length ℓ_j for each trace σ^j, and the number of distinct activity labels $|\Sigma|$ is known. Given this, we can get the number of occurrences of each i-th activity label from Σ in each trace by directly accessing the rows within the CountingTable in Figure

On the other hand, the loading and indexing phase generates an ActivityTable associated with two indices, a primary index and a secondary index. While the former allows to effectively return all of the events associated with a specific activity label, the latter is used to access either the first and to the last event in a trace. If required, pointers associated with each record allow temporally scanning of the traces as a double linked list. Loading and indexing

algorithms are omitted due to the page limits.

4.2 Query Compiler

The query compiler is structured into three main phases. *(i)* The *atomization pipeline* rewrites the data predicates associated with each activity label as a disjunction of mutually exclusive data conditions. We can tune KnoBAB to always atomize each possible activity label if it exists any Declare Constraint associating it to a data condition as in [**?**], or we can choose to provide such an interval decomposition only to the Declare constraints exhibiting data conditions. While the former approach will maximise the access to the AttributeTables, the latter will maximise the access to the ActTable. By doing so, we can ensure that the data satisfying some given properties can be visited at most once, thus guaranteeing the assumptions from [**?**] also at the data accessing level. Correlation conditions do not undergo this rewriting step. The atomized model in Figure

We *(ii)* rewrite each Declare constraint as a $\texttt{xtLTL}_\text{f}$ formula, where the activations (and the potential target) conditions are instantiated with either just activity labels or also with associated data conditions as per the previous atomization step. Each sub-expression appears at most once as in [**?**] by representing every single node in the query plan at most once: this is ensured by an internal query manager cache. The resulting query plan considering the simultaneous execution of multiple queries can be represented as a Direct Acyclic Graph (DAG). For each declarative clause appearing more than once (e.g., $m > 1$), the associated $\texttt{xtLTL}_\text{f}$ expression will be computed at most run once, while its resulting data is going to be accessed m times by the final aggregator: as per Figure

Given that our execution engine provides the possibility of running a query plan in either a parallel or a sequential mode, we need an additional step. *(iii)* The previous DAG represents a dependency graph, where a link between an ancestor and one of its descendants implies that the latter has to be computed before the former, thus suggesting an execution order. Figure

4.3 Execution Engine

At the time of the writing, KnoBAB supports four different types of model aggregation queries: Conjunctive Query, Max-SAT, Confidence, and Support. As we will see at the end of the subsection, these will not require a change on the query plan, but just a different way to integrate the intermediate representation ϕ_i returned by each declarative clause c_i.

First, the execution engine takes the relational database resulting from the data loading (the DAG returned by the query compiler) and uses the leaf nodes from the latter to access the former. By query plan construction, all of the relevant data parts are going to be accessed at most once and then transformed into the expected intermediate result representation. *Second*, the intermediate results are propagated from the leaves towards each root node associated with a declarative clause c_k. Any intermediate representation is always associated with each operator returning it as a temporary primary-memory cache. Each intermediate cache might be

completely freed if we are not computing a CONFIDENCE query and if the furthest ancestor has already accessed it, or if it is a cache non-associated to an activation required by CONFIDENCE and the furthest ancestor has already accessed it. *Third*, when the computation will finish running the shallowest DAG depth level containing the $\texttt{xtLTL}_\text{f}$ root associated with the entry-point of each declarative clause c_k, each of these operators will have an intermediate result ϕ_k stating all the traces satisfying c_k.

The Conjunctive Query will return the traces satisfying all of the Declare clauses via the intersection of all of the clauses via And and $\mathbf{true}$ as a Θ condition. Max-SAT will count, for each log trace σ_i, the intermediate results ϕ_k associated with each clause c_k containing it, and then provide the ratio of such value over the total number of the model clause $|\mathcal{M}|$. By denoting as $\mathsf{ActLeaves}(\phi_k)$ the untimed union of the intermediate results returned by the activation conditions for the declare clause $c_k \in \mathcal{M}$, the CONFIDENCE for c_k is the ratio between the total number of traces returned by ϕ_k and the overall traces containing an activation condition. Dividing the total number of traces returned by ϕ_k by the total log traces returns the SUPPORT. Once each ϕ_k per clause c_k is computed, the aggregation functions can be then expressed as follows:

$$\text{ConjQuery}(\phi_1, \ldots, \phi_n) = \mathsf{And}(\phi_1, \ldots \mathsf{And}(\phi_{n-1}, \phi_n, \mathbf{true}), \mathbf{true})$$

$$\text{Max-SAT}(\phi_1, \ldots, \phi_n) = \left(\frac{|\{\, k \mid \exists j, L.\, \langle i, j, L\rangle \in \phi_k \,\}|}{|\mathcal{M}|} \right)_{\sigma^i \in \mathcal{L}}$$

$$\textsc{Confidence}(\phi_1, \ldots, \phi_n) = \left(\frac{|\{\, i \mid \exists j, L.\, \langle i, j, L\rangle \in \phi_k \,\}|}{|\mathsf{ActLeaves}(\phi_k)|} \right)_{c_k \in \mathcal{M}}$$

$$\textsc{Support}(\phi_1, \ldots, \phi_n) = \left(\frac{|\{\, i \mid \exists j, L.\, \langle i, j, L\rangle \in \phi_k \,\}|}{|\mathcal{L}|} \right)_{c_k \in \mathcal{M}}$$

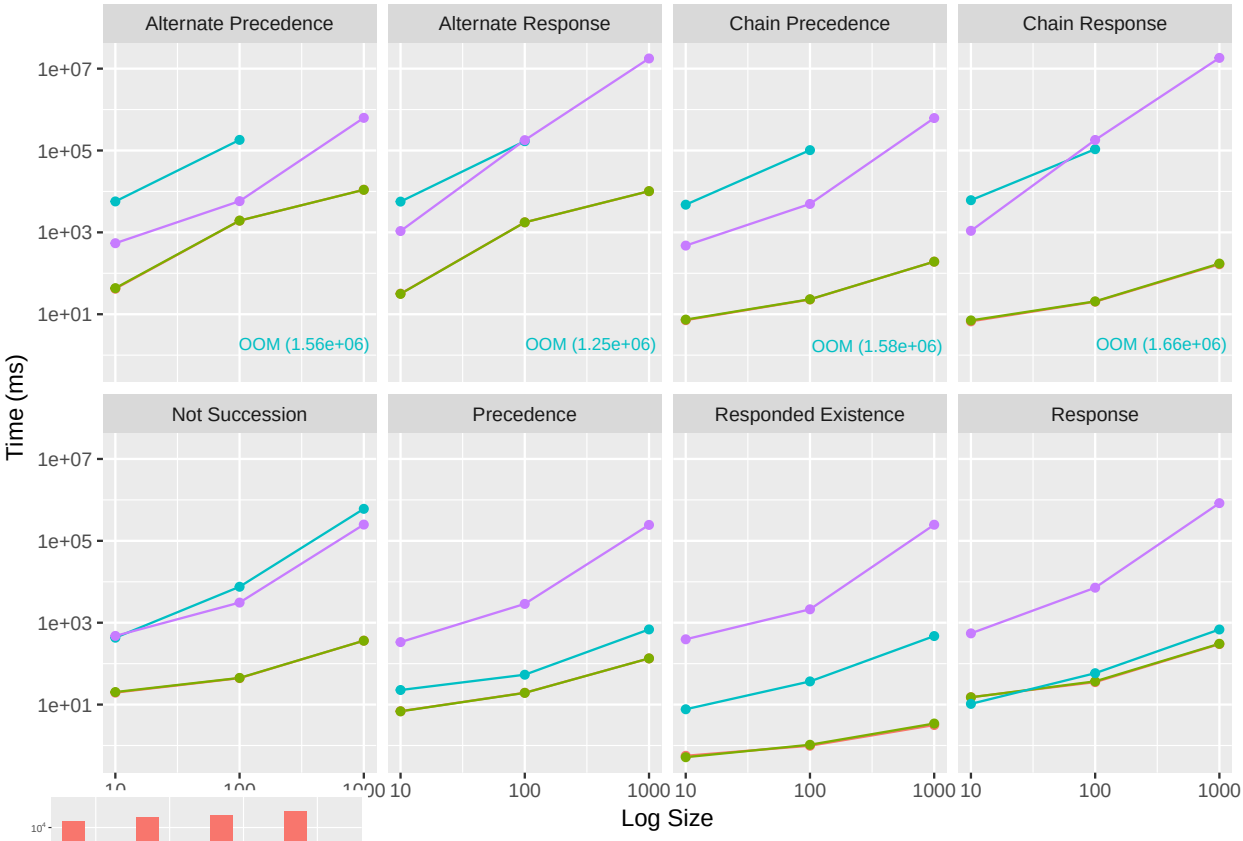
As the user in Figure

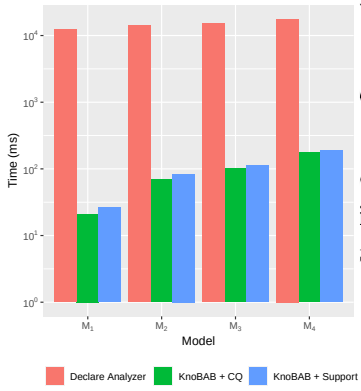
5 Experimental Analysis

Our benchmarks exploited a Razer Blade Pro on Ubuntu 20.04: Intel Core i7-10875H CPU @ 2.30GHz - 5.10 GHz, 16GB DDR4 2933MHz RAM, 180GB free disk space.

5.1 SQLMiner

These experiments want to test our working hypotheses for the possibility of engineering a tailored relational database architecture that can outperform process mining through conformance checking running on traditional relational databases. In the latter, no LTL_f operators are exploited but a table similar to `ActivityTable` is exploited. Given that the SQL provided in [**?**, **?**] might only return the SUPPORT associated with each candidate Declare clause (`SQLMiner+Support`), we provided the least possible changes to also associate each candidate





Miner Performance for 25 clauses with frequent activity labels with
ion. OOM indicates Out of Secondary Memory for logs containing
ime taken for an exception to occur).

Model	Queries
$M_1 =$	$q_1 :=$ Response(A_SUBMITTED, **true**, A_ACCEPTED, **true**) $q_2 :=$ Response(A_SUBMITTED, AMOUNT_REQ $\geq 10^3$, A_ACCEPTED, **true**) $q_3 :=$ Response(A_SUBMITTED, AMOUNT_REQ $< 10^3$, A_ACCEPTED, **true**) $q_4 := q_1$ where A_SUBMITTED.org:resource $=$ A_ACCEPTED.org:resource $q_5 := q_1$ where A_SUBMITTED.org:resource $\neq$ A_ACCEPTED.org:resource
$M_2 = M_1+$	$q_6 :=$ Response(W_Completeren aanvraag, **true**, W_Valideren aanvraag, **true**) $q_7 :=$ Response(W_Completeren aanvraag, **true**, O_CANCELLED, **true**) $q_8 := q_6$ where W_Valideren aanvraag.org:resource $\neq$ W_Valideren aanvraag.org:resource $q_9 :=$ Response(W_Valideren aanvraag, AMOUNT_REQ $= 5 \cdot 10^3$, O_CANCELLED, **true**) $q_{10} := q_9$ where W_Valideren aanvraag.org:resource $=$ O_CANCELLED.org:resource
$M_3 = M_2+$	$q_{11} :=$ Response(O_SELECTED, **true**, O_CANCELLED, **true**) $q_{12} := q_{11}$ where O_SELECTED.org:resource $=$ O_CANCELLED.org:resource $q_{13} :=$ Response(O_SELECTED, AMOUNT_REQ $< 8 \cdot 10^3$, O_CANCELLED, **true**) $q_{14} := q_{13}$ where O_SELECTED.org:resource $=$ O_CANCELLED.org:resource $q_{15} :=$ Response(O_SELECTED, AMOUNT_REQ $> 10^3$, O_CANCELLED, **true**)
$M_4 = M_3+$	$q_{16} :=$ Response(A_PARTLYSUBMITTED, **true**, A_DECLINED, **true**) $q_{17} := q_{16}$ where A_PARTLYSUBMITTED.org:resource $=$ A_DECLINED.org:resource $q_{18} :=$ Response(A_PARTLYSUBMITTED, AMOUNT_REQ $> 2 \cdot 10^4$, A_DECLINED, **true**) $q_{19} :=$ Response(A_PARTLYSUBMITTED, AMOUNT_REQ $> 2 \cdot 10^4$, A_CANCELLED, **true**) $q_{20} := q_{18}$ where A_PARTLYSUBMITTED.org:resource $=$ A_DECLINED.org:resource

Table 2: Declare Models with their Respective Clauses

clause with the set of all the traces satisfying it. This was achieved by both extending the

activation condition expressed in SQL and using `array_aggr` included in **PostgreSQL 14.2** to list such traces (`SQLMiner+TraceInfo`). For comparing the same settings in KnoBAB, we run both Max-SAT and SUPPORT queries with the difference that, in our case, both of these implementations will always return, per intermediate result specification, the trace information satisfying each possible model clause. For our experiments, we exploited the same hospital log dataset from [**?**]. To test the scalability of the solutions, we recorded the query's runtime with increasing log size: we randomly sampled the log with three sub-logs containing 10, 100 and 1000 traces, while guaranteeing that each sub-log is always a subset of the greater ones. For each sub-log, we generated 8 distinct models as benchmarked in [**?**]. Each model consists of 25 clauses instantiating the same Declare template (*elected template*) with different activation and target conditions. Those did not consider payload conditions and were only considering the most frequent activity labels appearing in the sub-log. Models of greater size than this caused an exponential increase in required secondary memory for SQLMiner (on the order of TB), justifying our approach for a sampled model. In their approach, each model was queried by running the SQL query corresponding to the *elected template*, and the specific activation and target conditions from the model's clauses were distinct rows in the Action table.

The outcome of such experiments is represented in Figure

5.2 Declare Analyzer

The set of experiments on Declare Analyzer have the aim of comparing our proposed solution against a solution tailored for solving Declare Conjunctive Queries over logs running exclusively in primary memory. We exploited the BPI 2012 Challenge dataset, also used in [**?**], and we slightly edited the queries in the same paper as illustrated in Table

Our experiments indicate that, overall, we are 2-3 times orders of magnitude more performant than DeclareAnalyzer. The conjunctive query denoted as `KnoBAB+CQ` demonstrates greater performance that `KnoBAB+Support`, as the calculations required for the support values per clause are more costly for smaller models. Though this is only within the order of the milliseconds. For an increase in model size, Declare Analyzer has a much greater time increase than KnoBAB (the best case for Declare Analyzer is over an order of magnitude greater than that of KnoBAB). While the linear interpolation of Declare Analyzer provides a slope of $3.47 \cdot 10^2$ *ms* per model size, KnoBAB provides a slope of 10^1, thus providing an inferior overall growth rate. To explain the abrupt time increase from M_1 to M_2, we encourage the reader to refer back to the query plan from Figure

6 Conclusions and Future Works

We propose KnoBAB, a fully relational database architecture for computing Conformance Checking via conjunctive queries, as well Max-SAT and clause CONFIDENCE/SUPPORT functions. KnoBAB consists of a data loader and indexer, query compiler, and an execution engine,

thus fully matching the architecture of a relational database. This solution was enabled by the extension of the traditional LTL_f operators, providing algebraic semantics to declarative temporal models, so as to support data operations over tuples representing trace events. Our solution is not limited to one single declarative language of choice, as it might support any possible model that can be expressed via $\texttt{xt}\text{LTL}_\text{f}$ operators. Based on the latest solutions in current database literature, the query plan was also designed to minimize the data access by running the common sub-queries at most once. KnoBAB outperforms state of the art solutions both tailored to the specific dataset or based on traditional relational databases running SQL queries. This solution will enable us to learn models exploiting abductive reasoning rather than traditional mining techniques, thus also providing safety guarantees and models that are inconsistency free. These will be also extremely useful on noisy data [**?**].

Future works will provide extensive benchmarks for bigger log datasets and will provide speed-up results for the parallelized execution of the resulting query plan: despite this being already implemented, we postpone those results due to the lack of space in the present paper. For the time being, the logs available from the research community are quite compact, and therefore the whole dataset is well fit in primary memory. Dealing with actual big data solutions or bigger models will require us to migrate the data store location to secondary memory, thus requiring the adoption of Near-Data Processing techniques [**?**].

Last, the adoption of relational database representations and query plans will enable us to support incremental updates for model checking in the eventuality that traces might be extended at runtime: this is still an open research problem [**?**] that can be now promptly solved by extending our query plan to support incremental updates by transferring the approaches knew for relational databases to the novel $\texttt{xt}\text{LTL}_\text{f}$ operators.