# Running Temporal Logical Queries on Knowledge Bases

Samuel Appleby
s.appleby3@newcastle.ac.uk
Newcastle University
School of Computing
United Kingdom

Giacomo Bergami
ngb113@newcastle.ac.uk
Newcastle University
School of Computing
United Kingdom

Graham Morgan
graham.morgan@newcastle.ac.uk
Newcastle University
School of Computing
United Kingdom

## 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.

## CCS CONCEPTS

• **Information systems** → **Association rules**; **Data mining**; *Data scans*; *Data access methods*; **Query optimization**; **Database query processing**; • **Computing methodologies** → *Temporal reasoning*;

## KEYWORDS

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

**ACM Reference format:**
Samuel Appleby, Giacomo Bergami, and Graham Morgan. 2022. Running Temporal Logical Queries on Knowledge Bases. In *Proceedings of 26th International Database Application & Enginnerring Symosium, Budapest, HU, August 22–24, 2022 (IDEAS'22),* 10 pages.
https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

*Conformance checking* is an integral part of Artificial Intelligence bridging data mining and business process management [5]. It assesses whether a sequence of distinguishable events (i.e., a *trace*) conforms to the expected process behaviour represented as a *(process) model* [19]. 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 [6]. Non-conforming traces are usually referred to as *deviant* [5]. *Declarative* models are composed of multiple human-readable *clauses* that should be jointly satisfied (i.e., *conjunctive query*) [11]; 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 [2, 12] 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 [5] 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, 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 [18], 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 [4]. 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 1 target only breast cancer patients with successful therapies: Ⓒ two possible surgical operations for breast tumours are mastectomy or lumpectomy if the biopsy is positive and the CA-13.5 is way above (≥ 50) the guard level being 23.5 units per *ml*, and Ⓐ-Ⓑ any successful treatment should decrease the CA-13.5 levels, which should be below the guard level; such correlation data condition is expressed via a Θ condition (also indicated as where). A twinned negative model (not in Figure) might better discriminate healthy patients from patients where the therapy was unsuccessful. *Secondly*, conformance checking can also exploit such models for predicting which novel clinical situations represented as traces are likely to adhere to the expected clinical standards. Novel situations can be represented as a log: e.g., in Figure 1, we have three patients: ① a cancer patient with a successful mastectomy, ② a healthy patient, and ③ an unsuccessful lumpectomy, thus suggesting that the patient might still have some cancerous cells. Given the aforementioned model, patient ① will satisfy the model as the surgical operation was successful, ② will not satisfy the model because neither mastectomy nor lumpectomy was required, and ③ will not satisfy the follow-up condition. Our model of interest should only return the first patient as an outcome of the conformance checking process.

Real business use case scenarios usually require Θ-correlations. In a goods brokerage scenario [14], items are traded between producers (vendors) and retailers (customers): each transaction starts


*IDEAS'22, August 22–24, 2022, Budapest, HU*

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

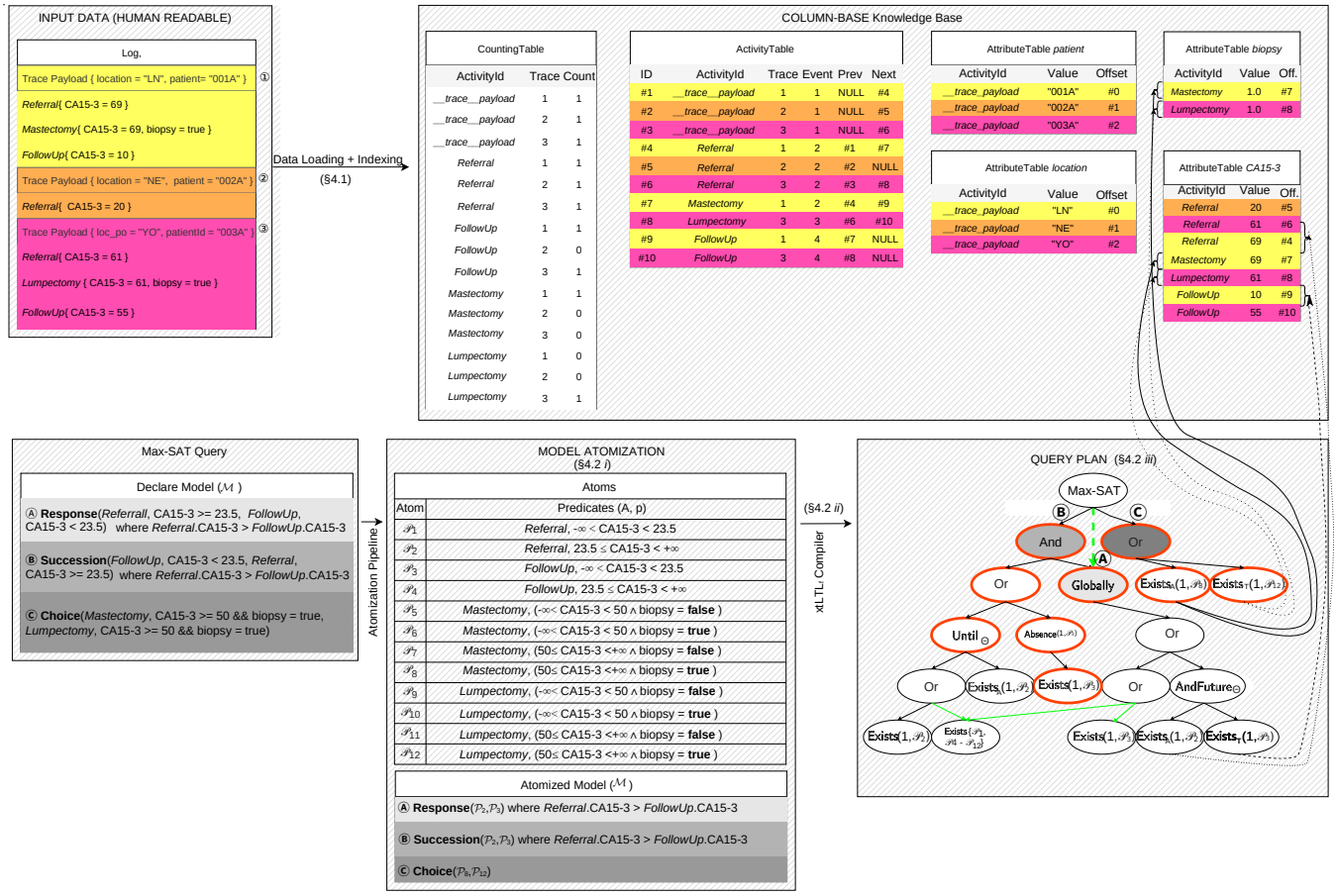


**Figure 1: KnoBAB Architecture for Breast Cancer patients. Each trace ①-③ represents one single patient's clinical history, are represented with unique colouring. Please observe that the atomization process does not consider data distribution but rather partitions the data space as described by the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

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 [10]: existing state of the art exploits automata [13] 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 [9], 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 [4]. 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 [20, 21] 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 (§2). In doing so, we are forced to introduce aggregation and nesting operations, which are not generally efficient. This fact is supported by experimental evidence (§5.1), where we also extend the relational representation of traces from [20, 21] (§4.1).

[1] https://battlefieldtracker.com/

Our specific contribution is then the provision of specific operators ($\texttt{xtLTL}_\text{f}$) rewriting existing $\text{LTL}_\text{f}$ operators for the relational model thus efficiently running conformance checking queries in Declare (§3.2). This is also possible through a query plan solution similar to [3] (§4.3), which proves to be more efficient than any solution relying solely on the SQL language.

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 [6]. This solution, not being able to assemble the previously described $\text{LTL}_\text{f}$ operators within 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 [3]. Further experimental shreds of evidence support such theoretical claims (§5.2): in the first instance, these show that our solution is already more efficient than the state of the art in the literature by two orders of magnitude (hundredths of a millisecond vs. tens of seconds). Furthermore, by using different Declare models composed of several clauses accessing the same activation and target conditions, except the data correlations, our solution exhibits an increase in running time only when new data is accessed and, otherwise, it preserves a constant running time with fewer temporal fluctuations.

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 (§4.1). In this instance, we then generate a conformance checking query plan generated from a declarative model (§4.2), be it positive or negative, so to compute desired properties associated with non-deviant traces (§4.3). As per previous remarks, declarative models represent temporal and data constraints that one would expect to hold as true in the non-deviant traces from the twinned system. As such, one can consider those traces returned by the query associated with the declarative model as correct, and the remainder as deviant. As a temporal representation of the declarative model provides a point-of-relativity in the context of correctness (i.e., time itself may dictate if traces maintain correctness throughout the unfolding of the associated events), the considerations of such temporal issues significantly increase the time spent for checking the meeting of the requirements. We also provide these ancillary contributions: first, we show how to extend the relational solution for representing logs from [20, 21] with a CountingTable and a column-based relational model for representing data payloads (§4.1, upper part of Figure 1). Second, we design a query compiler (§4.2) transforming each Declare model composed of multiple single clauses into a DAG query plan (lower part of Figure 1). Next, we designed an execution engine running the $\texttt{xtLTL}_\text{f}$ operators either sequentially or in parallel (§4.3).

## 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 $\sigma^i_j$ is a pair $\langle \mathrm{a}, p \rangle$, where $\mathrm{a} \in \text{Act}$ is an activity label, and $p$ is a payload, mapping each key to a single value. A *trace* $\sigma^i$ is a temporally-ordered and finite sequence of distinct events $\sigma^i = \sigma^i_1 \cdots \sigma^i_n$, 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 $\Sigma \subseteq \text{Act}$ 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, __trace_payload, at the beginning of the trace. This characterization [5] 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 [1].

*Conformance Checking.* Temporal declarative languages pinpoint recurring temporal patterns in highly variable scenarios so as to describe them compactly for both machines and humans [16]. Every single temporal pattern is expressed through *templates* (i.e., an abstract parameterized property: Table 1 column 2), which are then instantiated on a set of real activation, target, or correlation conditions. We can then categorize each Declare template from [11] by means of these conditions and the ability to express correlations between two temporally distant events happening in one trace: simple templates (Table 1, rows 1-3) only involving activation conditions; (mutual) correlation templates (rows from 4 to 15), which describe a dependency between two activation and target conditions, thus including correlations between the two; and negative relation templates (last 2 rows), which describe a negative dependency between two events in correlation. Despite these templates may appear quite similar, but they generate completely different finite state machines, thus suggesting that these conditions are not interchangeable[3]. Figure 2 exemplifies the behavioural difference between two clauses differing only on the template of choice. As a semantics, Declare adopts Linear Temporal Logic over finite traces ($\text{LTL}_\text{f}$), which interprets formulae over an unbounded, yet finite linear sequence of states. Given a trace $\sigma^i$, the evaluation of a formula $\varphi$ is done in a given state (i.e., event id, or position) of the trace, and we use the notation $\sigma^i_j \vDash \varphi$ to express that $\varphi$ holds starting from the $j$-th event of the $i$-th trace. We also use $\sigma^i \vDash \varphi$ as a shortcut notation for $\sigma^i_0 \vDash \varphi$. This denotes that the entire trace $\sigma^i$ *satisfies* $\varphi$. Given that a Declare Model is composed of a set of clauses $\mathcal{M} = \{ c_l \}_{l \leq n, n \in \mathbb{N}}$ which have to be contemporarily satisfied in order to be true, we say that a trace $\sigma^i$ is *conformant* to a model if such a trace satisfies the $\text{LTL}_\text{f}$ semantics $[\![c_l]\!]$ associated to each clause[4] $c_l$. Therefore, the Maximum-SATisfiability problem (Max-SAT) for each trace counts the ratio between the satisfied clauses over the whole model size. An $\text{LTL}_\text{f}$ formula $\varphi$ is built by extending propositional logic with temporal operators in bold:

$$\varphi := \mathrm{A} \wedge p \mid \neg\varphi \mid \varphi \vee \varphi' \mid \varphi \wedge \varphi' \mid \mathbf{X}\varphi \mid \mathbf{G}\varphi \mid \mathbf{F}\varphi \mid \varphi \ \mathbf{U} \ \varphi'$$

where ne**X**t ($\mathbf{X}\varphi$) denotes that the condition $\varphi$ should occur from the next state, **G**lobally ($\mathbf{G}\varphi$) denotes that the condition has to hold on the entire subsequent path, **F**uture ($\mathbf{F}\varphi$) denotes that the condition should occur somewhere on the subsequent path, and **U**ntil as $\varphi \ \mathbf{U} \ \varphi'$ denotes that $\varphi$ has to hold at least until $\varphi'$ becomes true, either at the current or a future state. Generally, binary operators bridge activation and target conditions appearing in two distinct sub-formulæ. Some operators can be seen as syntactic sugar:

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

[3] http://ltlf2dfa.diag.uniroma1.it/

[4] More formally, $\sigma^i \vDash \mathcal{M} \Leftrightarrow \forall c_l \in \mathcal{M}. \sigma^i \vDash [\![c_l]\!]$.

**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.**

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

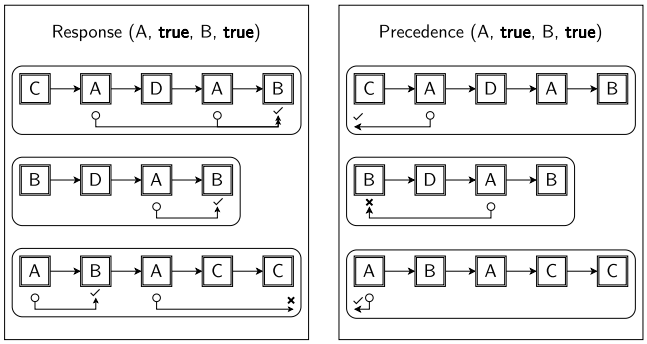


**Figure 2: Traces describing the events generated by each hospital unit: those are temporally ordered events associated to *activity labels* (boxed). Activated (or targeted) events here circled (or ticked/crossed). Ticks (or crosses) indicate a (un)successful match of a target condition.**

**W**eakUntil is denoted as $\varphi\ \mathbf{W}\ \varphi' := \varphi\ \mathbf{U}\ \varphi' \vee \mathbf{G}\varphi$, while the implication can be rewritten as $\varphi \Rightarrow \varphi' := (\neg\varphi) \vee (\varphi \wedge \varphi')$. Similarly to relational algebra, these operators also support equivalence rules, thus allowing to rewrite a given LTL$_f$ expression in an equivalent one that might be more efficient to compute.

Despite this formulation has been already extended so to support correlation constraints [6], 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 §3.2 (xtLTL$_f$).

*Data-Aware Conformance Checking.* `Declare Analyzer`[5] [6] 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 1), that not only contain event information, but a payload associated to each event per clause. This approach is claimed to be more optimised than prior works [22]. Such solution does not exploit RDBMS's benefits where query optimisations enhance query running times. So, no possible performance gains by shared sub-queries is considered so to minimize the data access, e.g., by conveniently structuring queries in a query plan [3]. In addition, the authors scan all of the traces completely for each Declare clause, while our proposed solution minimizes the data access by only accessing the data relevant for running the model-checking query. As their solution does not exploit multiple queries running processes, sub-queries or entire clauses appearing multiple times in the model might be recomputed multiple times, thus tampering with the overall running time. As per their implementation of the LTL$_f$ operators, authors do not exploit efficient relational algebra operators when possible, as full-outer-theta-joins (or theta-joins) for unions (or conjunctions) with correlation conditions.

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 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$_f$.

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

*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 [21] does so via SQL queries [20] 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 $\langle$a, b, $s\rangle$ represents a candidate clause Response(a, **true**, b, **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 [3], 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 $\sigma^i_j$ in trace $\sigma^i$, 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 $\Theta$ predicate (see the next section). This is a sensible representation, as per declarative constraints, it may exist only one possible $\Theta$ predicate. Such triplets are sorted by trace id and event id, and operators manipulating those (§3.2) guarantee that only one triplet should appear per unique trace and event id. This guarantees efficient join operations across different intermediate results, as well as efficient counting of the satisfied conditions for each trace. E.g., Clause Ⓒ from Figure 1 requires access to just AttributeTables, as all of the activity labels are associated to data conditions. When we want to return events for which $\mathcal{P}_{12}$ holds, we need to only consider the data associated to `Lumpectomy` events having a positive biopsy and levels of CA15.3 greater than 50. This will require the intersection of the events related to biopsy with the ones related to CA15.3. The selected rows are then converted into the intermediate result representation ad intersected; in this situation, we only obtain $\{\ \langle 3, 2, \{A(2)\}\rangle\ \}$, as the only event meeting such requirements is the second from the third trace. As we are going to see in the next paragraph, A is the container of matched activation conditions. Similarly, $\mathcal{P}_8$ will return $\{\ \langle 2, 2, \{A(2)\}\rangle\ \}$, thus obtaining $\{\ \langle 1, 2, \{A(2)\}, \langle 3, 2, \{A(2)\}\rangle\rangle\ \}$ as a final result associated to Ⓒ: this remarks that only traces ① and ③ describe patients that underwent a surgical operation under such conditions.

Our proposed representation is different from the one provided by [6] 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 §2, this information is required for checking the satisfiability of $\varphi$ while jointly visiting both the trace (now represented as subsequent rows in the result representation) and the formula. In fact, authors exploit a hash map of hash maps, associating each trace to the collected activation conditions which, in turn, might be associated with further target conditions. This solution is even less efficient than exploiting sorted linear data structures.

## 3.2 eXTended LTL$_f$ operators

We now propose the extended LTL$_f$ operators (xtLTL$_f$) 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 [6], 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 1, Exists$(1, \mathcal{P}_3)$ is a shorthand for Exists$(1, \text{FollowUp}, -\infty < \text{CA-15.3} < 23.5)$, as each atom always associates an activity label to a payload condition. The operator associated to Absence$(1, \mathcal{P}_3)$ is untimed, while the Exists$(1, \mathcal{P}_3)$ descendant of Globally is timed. While the timed definition returns a tuple $\langle i, j, L\rangle$ for each possible event $\sigma^i_j$ within the trace $\sigma^i$ where the formula holds, the untimed specification only checks whether the formula holds at the beginning of the trace. E.g., untimed Exists (Absence) returns the first event trace if at least $n$ (at most $n-1$) events satisfy $A \wedge p$, while the timed version returns the events satisfying (not satisfying) $A \wedge p$ (always $n = 1$). All of these operators might be optionally marked as returning either an activation ($A$) or a target ($T$) condition, so that each $\langle i, j, L\rangle$ triplet has $L = \{A(j)\}$ or $L = \{T(j)\}$; when no mark is specified, $L$ is empty. To wrap up the previous example, the timed Exists$(1, \mathcal{P}_3)$ will list the events where $\mathcal{P}_3$ happened, $\{\ \langle 1, 3, \{A(3)\}\rangle\ \}$, while the untimed version will just list the traces where such event happened and collect the event of interests in $L$, $\{\ \langle 1, 1, \{3\}\rangle\ \}$.

The next two lines report the same operators described in §2 with the addition of the explicit correlation conditions over activation and target conditions for each binary operator. Algorithm 1 provides implementations of the timed versions of such operators, due to lack of space untimed versions are not provided, yet available in our codebase: please observe that Next($\phi$) keeps unaltered the activation and target conditions from $\phi$ and just returns the events where $\phi$ happens as a subsequent step. Any binary operator supports $\Theta$ conditions: And (and Or) can be expressed as a (full-outer-)$\Theta$-join algorithm over the activation and target conditions stored in $L$ associated with the same event. If at least one activation condition matches one target condition from the same event, those are expressed as a marked correlation condition $M(i, j)$ which is then returned by the join. Regarding the same Choice clause from Figure 1, the correlation condition $\Theta$ associated to Or is then computed for each activation/target match, and if the condition is passed, the resulting match is added to $L$.

The remaining operators merge multiple operators together when a specific implementation outperforms the execution of the operators separately: e.g., AndFuture$(\phi, \phi', \Theta)$ is equivalent to And$(\phi,$ 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 $\theta$-joins are usually more efficient than performing a join and a selection operation separately. On the other hand, Implies$(\phi, \phi', \Theta)$ is rewritten as Or(Not$(\phi)$, And$(\phi, \phi', \Theta)$, **true**). As per previous discussion, the left leaf of AndFuture$_\Theta$ in Figure 1 returns all of the referral events with CA 15-3 above the safeguard levels, $\{ \langle 1, 1, \{A(1)\}\rangle, \langle 3, 1, \{A(1)\}\rangle \}$, while the right leaf returns just the follow-up events below such levels, $\{ \langle 1, 3, \{A(3)\}\rangle \}$. The operator AndFuture$_\Theta$ will then return only $\{ \langle 3, 1, \{M(1, 3)\}\rangle \}$, as only the first trace will have a decrease below the safeguard levels from referral to follow-up.

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 (§4.3), that is, computing either the Max-SAT for each log trace, or the Confidence/Support associated to each model clause, or computing the traces satisfying all of the model clauses (*conjunctive query*).

### 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 [5]. 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 $\Sigma$ 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 $\sigma^j$; each row $\langle \mathsf{a}, j, i, q, q'\rangle$ from ActivityTable states that the $i$-th event of the $j$-th trace ($\sigma^j_i = \langle \mathsf{a}, p\rangle$) is labelled as a, while $q$ (or $q'$) is the pointer to the immediately preceding $\sigma^j_{i-1}$ (or following, $\sigma^j_{i+1}$) event within the

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

```
1:  function Future(φ)
2:      for all ⟨t, e, L⟩ ∈ φ do yield ⟨t, e, ⋃ { L' | ⟨t, e', L'⟩ ∈ φ and e' ≥ e }⟩
3:      end for
4:  function Globally(φ)
5:      for all ⟨t, e, L⟩ ∈ φ do
6:          E ← { e' | ⟨t, e', L'⟩ ∈ φ and e' ≥ e }
7:          if |E| = ℓ_t − e then yield ⟨t, e, ⋃ { L' | ⟨t, e', L'⟩ ∈ φ and e' ∈ E }⟩
8:          end if
9:      end for
10: function Next(φ)
11:     for all ⟨t, e, L⟩ ∈ φ s.t. e > 1 do yield ⟨t, e − 1, L⟩
12:     end for
13: function CommonJoin(φ, φ', Θ, isDisjunctive)
14:
15:     it ←Iterator(φ), it ←Iterator(φ')
16:     while it ≠ ∅ and it' ≠ ∅ do
17:         ⟨t, e, L⟩ ← current(it), ⟨t', e', L'⟩ ← current(it')
18:         if t = t' and e = e' then
19:             if L = ∅ then L'' ← L'
20:             else if L' = ∅ then L'' ← L
21:             else
22:                 L'' ← ∅
23:                 for all A(m) ∈ L and T(n) ∈ L' s.t. Θ(m, n) do
24:                     L'' ← L'' ∪ { M(m, n) }
25:                 end for
26:             end if
27:             yield ⟨t, e, L''⟩; next(it); next(it');
28:         else if t < t' or (t = t' and e < e') then
29:             if isDisjunctive then yield ⟨t, e, L⟩
30:             end if
31:             next(it)
32:         else
33:             if isDisjunctive then yield ⟨t', e', L'⟩
34:             end if
35:             next(it')
36:         end if
37:     end while
38: function And(φ, φ', Θ) CommonJoin(φ, φ', Θ, false)
39: function Or(φ, φ', Θ) CommonJoin(φ, φ', Θ, true)
40: function Until(φ, φ', Θ)
41:     for all t s.t. ⟨t, i', L'⟩ ∈ φ' do
42:         α ← 1; Map ← {}; i ← min_ι ⟨t, ι, L⟩ ∈ φ'; I ← max_ι ⟨t, ι, L⟩ + 1
43:         while i < I do
44:             if exists ⟨t, j, L⟩ ∈ φ s.t. j < i then
45:                 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
46:                     Map[i] ← { M(k, i) | k ∈ L_α ∪ ··· ∪ L_{i−1} }
47:                     i ← i + 1
48:                 else α ← α + 1
49:                 end if
50:             else α ← i
51:             end if
52:         end while
53:     end for
```

trace if any. NULLs from Figure 1 in ActivityTable highlight the start (finish) event of each trace, where there is no possible reference to past (future) events. Trace payload information is injected (as an event) before the first event, which is also contained: all trace payload events contain NULL as Prev.

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 [8], thus representing all of the values $v$ associated to a payload key $k$ within the rows from AttributeTable$k$. In our implementation, each row $\langle a, v, i\rangle$ 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 1, all the attributes are stored in distinct tables. The value column can contain multiple data types, but each attribute is associated to only one type. When decomposed atoms are used for a query, the tables associated with the query are then accessed.

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 $\ell_j$ for each trace $\sigma^j$, 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 $\Sigma$ in each trace by directly accessing the rows within the CountingTable in Figure 1 are within the range $[|\mathcal{L}| \cdot (i-1)+1,\ |\mathcal{L}| \cdot i]$. The offsets for accessing the *Mastectomy* activity label in CountingTable is $[3 \cdot (4-1)+1, 3 \cdot 4] = [10, 12]$. Given that this counting table computes only for untimed operations, the intermediate result for untimed $\text{Exists}_A(1, \textit{Mastectomy}, \textbf{true})$ is $\{\ \langle 1, 1, \emptyset\rangle\ \}$, as only ① contains such an event.

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 [5], 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 [3] also at the data accessing level. Correlation conditions do not undergo this rewriting step. The atomized model in Figure 1 replaces the non-correlation data predicates with the outcome of the atomization process as in [5].

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 subexpression appears at most once as in [3] 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 1, despite Response might be considered a subquery of Succession, the Max-SAT is still going to retrieve the output provided by the associated sub-expression. Green arrows remark operators' output shared among operators. Please also observe that operators with the same name and arguments but marked either with activation, target, or no specification are considered different as they provide different results, and therefore are not merged together. This also includes distinctions between timed and untimed operators.

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 1 depicts this as an arrow starting from the ancestor. To enforce that, we perform a lexicographical order over the DAG, through which we compute the maximum depth level associated with each node of the graph. After doing so, we represent the query graph as a stack of depth levels, where each operator on it can be run in parallel alongside its siblings. This proves that the computation of Declare Clauses can be reduced into an embarrassingly parallel problem, as the layered execution guarantees that no thread communication needs to happen, and that multiple threads could access contemporary the partial results associated with the immediately-descendant operators, as the former will return all of the events where the condition happened, while the latter will just return the trace event satisfying such condition alongside the required activations/targets listed in $L$. Furthermore, the proposed parallelization ensures minimizing the data access for computing the query. The DAG Figure 1 depicts a query plan.

### 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 $\phi_i$ 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 $\phi_k$ 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 **true** as a $\Theta$ condition. Max-SAT will count, for each log trace $\sigma_i$, the intermediate results $\phi_k$ 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 $\text{ActLeaves}(\phi_k)$ 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 $\phi_k$ and the overall traces containing an activation condition. Dividing the total number of traces returned by $\phi_k$ by the total log traces returns the Support. Once each $\phi_k$ per clause $c_k$ is computed, the aggregation functions can be then expressed as follows:

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

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

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

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

As the user in Figure 1 wanted to return the ratio between satisfied clauses over the model size, the query plan exhibits a Max-SAT aggregation.

## 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.

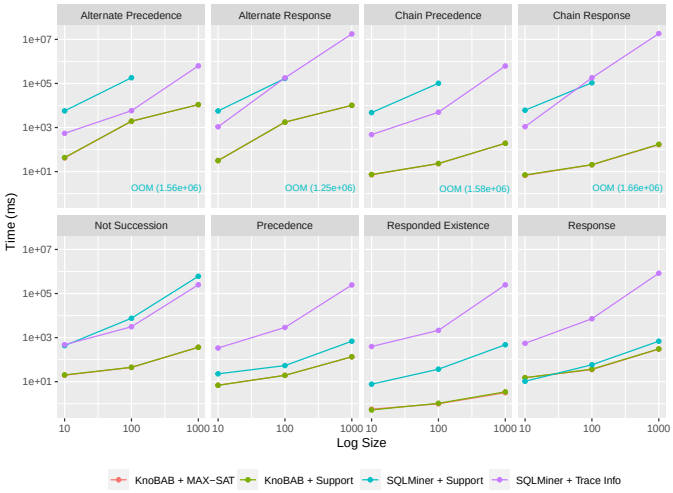


**Figure 3: KnoBAB vs SQLMiner Performance for 25 clauses with frequent activity labels with Support and Trace Information. OOM indicates Out of Secondary Memory for logs containing $10^3$ traces (followed by the time taken for an exception to occur).**

### 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 $\text{LTL}_\text{f}$ operators are exploited but a table similar to `ActivityTable` is exploited. Given that the SQL provided in [20, 21] might only return the Support associated with each candidate Declare clause (`SQLMiner+Support`), we provided the least possible changes to also associate each candidate 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 [21]. 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 [20]. 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

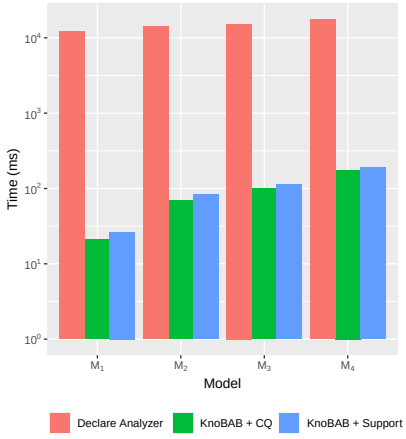


**Figure 4: KnoBAB vs Declare Analyzer Performance for Different Models.**

| 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**) where O_SELECTED.org:resource $\neq$ O_CANCELLED.org:resource |
| $M_4 = M_3+$ | $q_{16}$ := 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**

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 3, where each plot represents the running time associated with models containing the same *elected template*. In the worst-case scenario (Response), we exhibit similar query running times to SQLMiner. Even so, we are always providing trace information, and in the case of Response, altering the SQL to provide this causes over an order of magnitude increase in complexity. In the best-case scenario, we outperform SQLMiner by at most 5 orders of magnitude. This is because our query plan minimizes the access to the data queries and our computation avoided explicit computations of aggregations. This was achieved by sorting the intermediate results, and, as our operators' implementations guarantee that (intermediate) results are always sorted, counting operations are just linear scans of the intermediate result representation. Our solution never exceeded the 16GB of primary memory while, for some more complex queries (top row of Figure 3), SQLMiner exceeded it, thus proving that our solution is also memory efficient. One of the outstanding examples is RespExistence, where we are greatly more efficient than SQLMiner. This is a clear indicator of the potential gains from utilising our proposed CountTable, summarizing the appearance of activity labels in events per trace by counting their instances. The original SQL query is required to scan the whole `Log` table (similar to our `ActivityTable`), which contained all of the trace events. We also remind the reader that the CountTable can be efficiently created while scanning the whole log dataset, so no super-linear overhead is added at loading time. This further validates that an adequate tabular representation twinned with `xtLTL`$_f$ operators extending the LTL$_f$ specification for tabular data provides a suitable solution. Last, the running time of the Max-SAT problem and SUPPORT for KnoBAB exhibited similar running times, while in PostgreSQL those exhibit huge variations depending on the query-plan rewriting performed by the PostgreSQL query engine. For some elected templates, our proposed `SQLMiner+TraceInfo` formulation proved also to be more efficient than the `SQLMiner+Support` queries originally proposed by [20] (which contain no trace information).

## 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 [6], and we slightly edited the queries in the same paper as illustrated in Table 2, where all the models $M_i$ and $M_j$ with $i < j$ are always the former a subset of the latter, while $M_{i+1}$ provides a 5-fold increase from $M_i$. For Declare Analyzer, we chose to load the dataset via MapDB[6], thus aligning the authors' hash map representations of the dataset with a representation similar to the one for relational databases.

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 1 with reference to the model from Table 2. With each increase in model size, entirely new event labels and data payloads are considered (albeit the conditions within each sub-model are similar). As KnoBAB thrives when data access can be limited, the addition

[6] https://mapdb.org/

of new data requires more decomposition within the atomization pipeline, and, as more atoms are now considered, querying will also suffer as more data is going to be accessed. As a result, the complexity increase is worse than examples tailored to benefit data access limiting as in the previous scenarios where queries were sharing multiple and frequent activity conditions. Still, Declare Analyzer will always completely scan all the events by design despite, for some queries, we might exclude scanning irrelevant trace events.

## 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 $\text{LTL}_\text{f}$ 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 $\texttt{xtLTL}_\text{f}$ 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 [15].

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 [7].

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 [17] 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 $\texttt{xtLTL}_\text{f}$ operators.

## ACKNOWLEDGMENTS

Samuel Appleby's work is supported by Newcastle University.

## REFERENCES

[1] Giovanni Acampora, Autilia Vitiello, Bruno Di Stefano, Wil van der Aalst, Christian Gunther, and Eric Verbeek. 2017. IEEE 1849: The XES Standard: The Second IEEE Standard Sponsored by IEEE Computational Intelligence Society. *IEEE Comp. Int. Mag.* 12, 2 (2017).

[2] Simone Agostinelli, Giacomo Bergami, Alessio Fiorenza, Fabrizio Maria Maggi, Andrea Marrella, and Fabio Patrizi. 2021. Discovering Declarative Process Model Behavior from Event Logs via Model Learning. In *ICPM 2021*. IEEE, 48–55.

[3] Ladjel Bellatreche, Mohamed Kechar, and Safia Nait Bahloul. 2021. Bringing Common Subexpression Problem from the Dark to Light: Towards Large-Scale Workload Optimizations. In *IDEAS*. ACM.

[4] Giacomo Bergami, Chiara Di Francescomarino, Chiara Ghidini, Fabrizio Maria Maggi, and Joonas Puura. 2021. Exploring Business Process Deviance with Sequential and Declarative Patterns. *CoRR* abs/2111.12454 (2021).

[5] Giacomo Bergami, Fabrizio Maria Maggi, Andrea Marrella, and Marco Montali. 2021. Aligning Data-Aware Declarative Process Models and Event Logs. In *Business Process Management*. 235–251.

[6] Andrea Burattin, Fabrizio Maria Maggi, and Alessandro Sperduti. 2016. Conformance checking based on multi-perspective declarative process models. *Expert Syst. Appl.* 65 (2016), 194–211.

[7] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. 2016. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. In *ISCA*. IEEE Computer Society, 153–165.

[8] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.* 35, 1 (2012), 40–45. http://sites.computer.org/debull/A12mar/monetdb.pdf

[9] Nathan John, Jeremy Gow, and Paul Cairns. 2019. Why is debugging video game AI hard?. In *Proc. AISB AI & Games symposium*. 20–24.

[10] Jingrui Li, Kento Goto, and Motomichi Toyama. 2021. SSstory: 3D data storytelling based on SuperSQL and Unity. In *IDEAS*. ACM, 173–183.

[11] Jianwen Li, Geguang Pu, Yueling Zhang, Moshe Y. Vardi, and Kristin Y. Rozier. 2020. SAT-based explicit LTLf satisfiability checking. *Artificial Intelligence* 289 (2020), 103369. https://doi.org/10.1016/j.artint.2020.103369

[12] Felix Mannhardt, Massimiliano de Leoni, Hajo A. Reijers, and Wil M. P. van der Aalst. 2016. Balanced multi-perspective checking of process conformance. *Computing* 98, 4 (2016).

[13] Youichiro Miyake. 2017. Current Status of Applying Artificial Intelligence in Digital Games. In *Handbook of Digital Games and Entertainment Technologies*. Springer Singapore, 253–292.

[14] André Petermann, Martin Junghanns, Robert Müller, and Erhard Rahm. 2014. FoodBroker - Generating Synthetic Datasets for Graph-Based Business Analytics. In *WBDB'14*.

[15] Jose Picado, John Davis, Arash Termehchy, and Ga Young Lee. 2020. Learning Over Dirty Data Without Cleaning. In *SIGMOD Conference*. ACM, 1301–1316.

[16] Paul Pichler, Barbara Weber, Stefan Zugal, Jakob Pinggera, Jan Mendling, and Hajo A. Reijers. 2011. Imperative versus Declarative Process Modeling Languages: An Empirical Investigation. In *BPM Workshops*. 383–394.

[17] Artem Polyvyanyy. 2022. Process Query Language. In *Process Querying Methods*, Artem Polyvyanyy (Ed.). Springer, 313–341.

[18] Marcella Rovani, Fabrizio Maria Maggi, Massimiliano de Leoni, and Wil M. P. van der Aalst. 2015. Declarative process mining in healthcare. *Expert Systems with Applications* 42, 23 (2015), 9236–9251.

[19] Anne Rozinat and Wil M. P. van der Aalst. 2008. Conformance checking of processes based on monitoring real behavior. *Inf. Syst.* 33, 1 (2008).

[20] Stefan Schönig. 2015. SQL Queries for Declarative Process Mining on Event Logs of Relational Databases. *CoRR* abs/1512.00196 (2015).

[21] Stefan Schönig, Andreas Rogge-Solti, Cristina Cabanillas, Stefan Jablonski, and Jan Mendling. 2016. Efficient and Customisable Declarative Process Mining with SQL. In *CAiSE*. Springer.

[22] W. M. P. van der Aalst, H. T. de Beer, and B. F. van Dongen. 2005. Process Mining and Verification of Properties: An Approach Based on Temporal Logic. In *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*.