A Query Language for Workflow Logs

YAN TANG, WEILONG CUI, and JIANWEN SU, University of California, Santa Barbara

A business process (workflow) is an assembly of tasks to accomplish a business goal. Real-world workflow models often demanded to change due to new laws and policies, changes in the environment, and so on. To understand the inner workings of a business process to facilitate changes, workflow logs have the potential to enable inspecting, monitoring, diagnosing, analyzing, and improving the design of a complex workflow. Querying workflow logs, however, is still mostly an ad hoc practice by workflow managers. In this article, we focus on the problem of querying workflow log concerning both control flow and dataflow properties. We develop a query language based on “incident patterns” to allow the user to directly query workflow logs instead of having to transform such queries into database operations. We provide the formal semantics and a query evaluation algorithm of our language. By deriving an accurate cost model, we develop an optimization mechanism to accelerate query evaluation. Our experiment results demonstrate the effectiveness of the optimization and achieves up to 50× speedup over an adaption of existing evaluation method.

CCS Concepts: • **Information systems** → **Query languages**; **Query languages for non-relational engines**; **Enterprise applications**

Additional Key Words and Phrases: Business workflow, query languages, log

ACM Reference format:

Yan Tang, Weilong Cui, and Jianwen Su. 2021. A Query Language for Workflow Logs. *ACM Trans. Manage. Inf. Syst.* 13, 2, Article 16 (December 2021), 28 pages.
<https://doi.org/10.1145/3482968>

1 INTRODUCTION

Process-centric business analytics plays a crucial role in the success of a business [6, 9, 17]. To support intelligent decision making, the underlying business workflow must constantly evolve. It is important to swiftly analyze past executions to discover bottlenecks, loopholes, and new patterns to improve workflow models and executions so that complexity can be reduced and efficiency and vital KPIs improved. Workflow analytics is typically based on detailed inspection of their past execution [18]. To facilitate analytics it is necessary to gather all traces of tasks and discover correlations of instances, and so on. Recent interests in the research and development communities have been on *process mining* [23, 30] that broadly aims at discover important relationships, especially temporal ones, between actions and events that have occurred in past executions through analyzing log records. At a more fundamental level, querying executed workflows [21] becomes an interesting research topic that may lead more structured and systematic approaches to process

Authors' address: Y. Tang, W. Cui, and J. Su, Department of Computer Science, University of California, Santa Barbara; emails: {yantang, cuiwl, su}@cs.ucsb.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2021 Association for Computing Machinery.

2158-656X/2021/12-ART16 \$15.00

<https://doi.org/10.1145/3482968>

Fig. 1. A framework for flexible process analytics.

mining and analytics. This article develops a query language to support ad hoc queries and exploration of workflow execution logs and optimization techniques for evaluating queries in this language.

Process mining is a practice to learn from completed workflow executions (enactments) to discover workflow/process models, perform conformance checking/audit, analyze performance, and predict and improve future process executions [23, 30]. In a broader sense, process mining can also reveal hidden information of and insights into the environments of business processes, including but not limited to organizational structures, social network (relations between performers), and more [30].

Workflow (execution) logs (or event logs) are often the key starting point for process analytics. There have been numerous efforts in the **business process management (BPM)** research community on developing concept, techniques, and tools in process mining over the last decade and a half [32]. An overwhelming amount of the past work has been on discovering/(re)constructing process models [32]. Interestingly, most process mining techniques are of the “one-stop shop” style: An algorithm would start from the input log and produce the final result (models, performance analysis, etc.) without interactions with, e.g., stakeholders of the workflows, business managers, dataset owners, and so on. It is only recently argued that exploration with interactive querying of the log [3, 4] can greatly enhance effectiveness of process analytics/mining, and there have been increased efforts over the last five or so years in the BPM community [16, 21, 22, 28, 29, 36].

A workflow log faithfully records actions and events that happened during the execution. In practice, most workflow management systems use (relational or XML) databases to store workflow log. For example, jBPM and Activiti record a workflow instance log, an activity log, and other logs into different tables in databases. For such systems, the framework can be easily extended with a log constructor that combines different pieces of the log into a single log based on timestamp and workflow instance ids (Figure 1).

Figure 1 immediately suggests interactive ad hoc queries over a workflow log [3, 22, 29]; this would allow the stakeholders to explore and discover potentially useful information. Log exploration allows the possibility to efficiently bridge semantic-rich high-level business queries and low-level operations, leading to effective workflow analytics. In practice, such flexible workflow analytics is often a precursor to developing systematic process mining algorithms [21, 29].

Reinkemeyer [23] presents a dozen use cases where process mining techniques were used to discover hidden regularities and relationships that subsequently led to organizational/process improvements. Execution logs were used in many of these use cases. The case descriptions also reveal

that business managers and executives are often heavily involved in the mining activities as they are among the key stakeholders. It is much desired to support these domain experts for their querying logs directly without software development/engineers to stand in the middle. Easy-to-use and expressive query languages for logs clearly deserve to be studied and developed.

An early step toward querying logs is the development of FPSPARQL [3, 4], an extension to SPARQL to query graphs that are already converted from workflow logs and models. Since a key kind of interesting relationships in the log to be queried about concerns timing among actions and events, temporal logic [22] was a natural choice for selection conditions. Temporal precedence may also be turned into edges between activities (as nodes) to allow temporal relationships expressed as “neighborhoods” [36]. The framework presented in Reference [16] assumes a workflow log in the native format, either relational or noSQL form, while allows queries expressed as a graph with node properties on individual activities linked with temporal precedence relations. Directly querying over the native log (i.e., without preprocessing such as in References [3, 4, 36]) as shown in References [16, 22] is an enabling technology, since it is immediately applicable as the process model discovery literature [32] has already demonstrated and it can also be easily used in complex event processing [1, 2, 10, 19]. In References [28, 29], we developed log query language based on BPMN-inspired “patterns” as examples to “match” traces of log records. At a fundamental level, querying logs is very interesting, since understanding log queries is not trivial in a number of dimensions including at least: usability (query languages, expressiveness, reasoning, inferencing) and efficiency (indexing, optimization).

Searching and querying logs arise also from complex event processing [1, 2, 10, 19]. Although tools such as SPL [8] allow search conditions involving, e.g., “activity name,” there is no additional semantics associated to such attributes other than the meaning in a natural language. Furthermore, there is a lack of support for directly reasoning about temporal relationships between activities, i.e., the user has to use the log record sequence in defining queries.

This article builds on the patterns developed in the work [28, 29] and further develops a query language IQL to directly act on workflow logs (Figure 1). IQL supports activities with conditions on data values before and after activity executions and allows the user to combine activities into “incidents,” which are fragments of workflow models within query expressions. By using an SQL-like query structure and workflow modeling primitives from BPMN, IQL queries can state properties of workflow execution directly. IQL is declarative and more suitable for workflow analytics.

IQL has a formal semantics. Intuitively, an IQL query retrieves from the log instances of the workflow formulated in the query expression. A workflow instance is a *sequence* of log records. Instead of scan-searching the log, we turn the control flow combination operators over workflow instances (log record sequences) into an algebra and develop an evaluation engine for this algebra. We further develop a cost model for the algebra and our IQL optimizer resembles SQL optimizer but acts on log record sequences instead of records or rows (note that the intermediate results during IQL evaluation are sets of log record sequences). Finally, experiments are conducted to show the validity of the cost model developed in the article and efficiency improvements. In particular, experimental results of queries on real datasets demonstrate the effectiveness of the cost model and achieve up to 50× speedup over naive evaluation.

1.1 Contributions of This Article

This article makes the following technical and scientific contributions:

- (1) We develop an approach to combine workflow modeling constructs and relational database query expressions into a single query language IQL. IQL allows direct and easier expression of queries for workflow instances.

- (2) We develop a cost model that accurately captures the relative cost of different evaluation paths and provide an optimized query evaluation algorithm.
- (3) We adopt and extend the conventional SQL evaluation and cost based optimization approach for relations to sets of (log) record sequences.

1.2 Related Work

This article is most related to the work on querying workflow logs using FPSPARQL [3, 4]. There are several areas that FPSPARQL and our work differ. Although both can express queries on workflow logs, the input for FPSPARQL queries are graphs (pre-constructed from logs) while ours are sequences of (native) log records. Also, queries in this article finds all workflow instances satisfying specified conditions, FPSPARQL queries formulate properties without focusing on individual instances. For temporal conditions, FPSPARQL uses regular expressions, while ours uses BPMN constructs. In terms of language style, both are declarative, but FPSPARQL is based on SPARQL while ours is based on SQL. While FPSPARQL relies on SPARQL, this article adapts the relational query optimization framework with a new cost model. The cost model faithfully reflects the evaluation cost (subject to the modeling accuracy) and allows for other potential optimization means. For example, it was used to guide the development of an incremental evaluation of log queries.¹ Finally, the performance study [3] aimed at understanding usability in terms of query evaluation time; this article attempts to an understanding of effectiveness of the query optimization techniques including the new cost model.

This article is also related to References [16, 22, 28, 29, 36]. [36] also queries a pre-converted graph from the log. Requiring preprocessing of workflow logs is double edged sword: It could be potentially very powerful if a rich set of relationships are represented in the graph; however, queries are restricted to only the extracted relationships and thus limits the usefulness.

In contrast, the logs in query frameworks reported in References [16, 22, 28, 29] and this article are in the native format (relational or noSQL), which makes it immediately and widely applicable in workflow analytics applications. However, a key omission in References [16, 22, 28, 29] is the limited ability to formulate constraints on data being manipulated by workflow (while they focus exclusively on temporal relationships between activities). They may be useful for, e.g., workflow model discovery, but not clearly as effective for log exploration. Reference [22] uses linear temporal logic to express such properties; Interestingly, queries in Reference [16] may involve aggregate functions, along with activity type, timestamps, and so on, along with temporal “precedence” relations. Reference [29] (and full version [28]) on which this article is based uses BPM gateways the key constructs to express temporal relations, aiming at ease of use by workflow modelers who must have already familiar with BPMN. This article significantly extends References [28, 29] by (1) allowing constraints on data associated with activities in the log, (2) an algebra and an evaluation algorithm based on the algebra, (3) the IQL language, (4) a cost model and optimization techniques, and (5) an experimental evaluation.

This article is also related to complex event processing [1, 2, 10, 19]. Search processing language SPL [8] is designed for the log management system Splunk, focussing on big data searching, filtering, and analysis, with basic operations like sorting, filtering, insertion, and deletion. In comparison, IQL in this article focuses more on complex temporal relationships between log records.

Also related are Online Analytic Processing, temporal relationships, and relational query optimization studied in the database community. The conventional framework for workflow analysis extracts data from various databases storing fragments of workflow log using ELT and loads data

¹R. Liu. *Incremental Evaluation for Querying Workflow Logs*, M.S. project presentation, Department of Computer Science, UC Santa Barbara, July 2020.

into a data warehouse for analysis [15, 35]. ETL aims at specific types of analysis queries centered around summaries over data cubes [5, 11, 12], it depends heavily on data selection/filtering. Composing queries then requires detailed knowledge of the underlying workflows' data structures and imposes a steep learning curve on the end users of the analytical process (e.g., project managers).

Query conditions often involve activities performed during workflow execution and the effects left by the activities, temporal relationships between two or more activities. Existing temporal data models and query languages [13, 26] focus on data with temporal aspects and temporal relationships between data values or rows in tables. Recent developments such as SPARQL-ST [20] also focus on RDF data elements and their temporal relationships. Languages to specify temporal properties for data-centric workflows combine first-order logic and temporal logic [7, 14], they state directly properties of data changes over time and are immediately usable for workflow analytics.

SQL database query optimization was studied extensively [25]. This article essentially adapts the cost-based optimization method to process centric query language. Although the optimization approach is similar, cost model is new and the experimental evaluation shows that there is a "new life" for the old relational optimization techniques, specifically for temporal operators.

The recent work [24] uses SQL constructs to inductively construct process models based on a set of source models. Although it is slightly related as far as research problems and methodology are concerned, an appealing problem could be to integrate (elements of) IQL in their model construction process, using the process instances in a log as source models.

This article is organized as follows. We introduce key notions of workflow log, "incident pattern," and "incident instance" in Section 2, and IQL in Section 3. We present the query evaluation algorithm in Section 4 and examine in Section 5 operator properties, derive the cost model, and discuss query optimization. We report experimental evaluation in Section 6; we conclude the article in Section 7.

2 LOGS AND INCIDENTS

In this section, we first define the log structure (notions of "log record" and "log") used throughout the article. Then we define "incident pattern" used to describe patterns of interest to the user in a log. Intuitively, an incident specifies temporal relationships between activities that already happened. Finally, we define "incident instance" that is a log record sequence that matches an incident pattern. We note here that the model for log records and log structure is identical to the one presented in Reference [28], but the notions for incident patterns and incident instances presented in this article extend the notions (with identical names) by allowing conditions on input/out (data) maps.

In this article, we assume that every workflow has a set of *workflow attributes* and a set of *activities*. During runtime, a workflow (model) can have multiple workflow instances, each of which has a unique identifier called a (*workflow*) *instance id*.

A log is generated at runtime by the **workflow management system (WMS)** executing one or more workflows. For the purpose of control, recovery, and analysis in WMS, the log contains the details of workflow executions, including workflow instance ids, activity names, and manipulated attributes along with their values. For instance, the BPM suite jBPM (Activiti) generates log records containing such information and stores them in a relational database. A health insurance system developed by Shandong University in China² also generates log records and stores them in a database as tables. (However, the data change history is not recorded.) Also, eXtensible Event Stream [33, 34] is an event log standard that contains details of instances, activities, and data and

²Private communications with Professor L. Kong, who lead the team in Shandong University.

is used for business analytics in ProM [31]. The information captured in the logs introduced above essentially coincides with that in logs in our framework.

To formulate the notion of a log, we start with reviewing two key definitions needed for the technical discussions: the notions of a “log record” (an entry in a log) and of a “log” originally introduced in Reference [28].

We assume the existence of the following pairwise disjoint countably infinite sets: $\mathcal{A}$ of *attribute names*, $\mathcal{T}$ of *activity names*, $\mathcal{D}$ of *values*, $\mathcal{V}_A$ of *attribute variables*, $\mathcal{V}_T$ of *activity variables*, and $\mathbb{N}$ ($\mathbb{N}^+$) of (respectively, *positive*) *natural numbers*. A (*data*) *map* is a mapping $\mathcal{A} \rightarrow \mathcal{D}$ whose domain is finite. If the domain is empty, then the map is called *empty* and denoted as $\emptyset$. Given a map, an attribute either has a value in $\mathcal{D}$ or is undefined.

Definition. A (*log*) *record* is a tuple $(lsn, wid, is-lsn, t, \alpha_{in}, \alpha_{out})$, where

- $lsn \in \mathbb{N}^+$ is a *log sequence number*,
- $wid \in \mathbb{N}^+$ is the *id* of a workflow instance (to which the log record belongs),
- $is-lsn \in \mathbb{N}^+$ is an *instance-specific log sequence number* (unique within each workflow instance),
- $t \in \mathcal{T}$ is an *activity name*,
- α_{in} is an *input map* (read by activity t), and
- α_{out} is an *output map* (written by activity t).

To simplify notations in technical discussions, we use the following functions to extract attributes of a log record: For each log record l , $lsn(l)$ is the log sequence number of l , $wid(l)$ the workflow instance id, $is-lsn(l)$ the instance specific log sequence number, $act(l)$ the activity name, $\alpha_{in}(l)$ the input map, and, finally, $\alpha_{out}(l)$ the output map. To differentiate attributes in a log record from workflow attributes, we call the former *log record attributes*.

Example 2.1. An example log record is $l = (10, 3, 2, register, \{(x, 1)\}, \{(x, 2), (y, 2)\})$. Here l is the 10th log record in the log ($lsn(l) = 10$), belongs to the workflow instance with id 3 ($wid(l) = 3$), is the 2nd activity executed in this workflow instance ($is-lsn(l) = 2$), and is the result of executing the activity *register* ($act(l) = register$). Finally, $\alpha_{in}(l) = \{(x, 1)\}$ means that the activity *register* reads attribute x having value 1, and $\alpha_{out}(l) = \{(x, 2), (y, 2)\}$ means that *register* updates values of both x and y to 2.

In addition, there are two special types of log records, “start” and “end.” A log record $(lsn, wid, is-lsn, t, \alpha_{in}, \alpha_{out})$ is the *start* record of a workflow instance if $is-lsn = 1$, $t = \text{START}$, and $\alpha_{in} = \alpha_{out} = \emptyset$. Similarly, it is the *end* record for an instance if $is-lsn > 1$, $t = \text{END}$, and $\alpha_{in} = \alpha_{out} = \emptyset$.

To simplify notations, for each pair of log records l, l' , we use $l <_{lsn} l'$ to denote $lsn(l) < lsn(l')$. Similarly, $l <_{is-lsn} l'$ denotes $is-lsn(l) < is-lsn(l')$, and $l =_{wid} l', l \neq_{wid} l'$ denote $wid(l) = wid(l'), wid(l) \neq wid(l')$, respectively.

Definition. A *log* L is a finite sequence $l_1 l_2 \cdots l_n$ ($n \geq 0$) of log records that satisfies all of the following:

- The log begins with a *start* record and the log sequence number increments by 1 for each subsequent log record: $lsn(l_1) = 1$, $act(l_1) = \text{START}$, and for each $i \geq 1$, $lsn(l_{i+1}) = lsn(l_i) + 1$,
- Each instance has exactly one *start* record: For each log record l_i , $is-lsn(l_i) = 1$ iff $act(l_i) = \text{START}$,
- Each workflow instance has consecutive *is-lsn*: For each l_i with $is-lsn(l_i) > 1$, there is an $l_j <_{lsn} l_i$ such that $l_j =_{wid} l_i$, $is-lsn(l_j) = is-lsn(l_i) - 1$, and $l_k \neq_{wid} l_i$ for each record l_k where $j < k < i$, and
- *end* is the last record for each workflow instance: For each log record l_i , if $act(l_i) = \text{END}$ then for each log record l_j where $i < j \leq n$, $l_j \neq_{wid} l_j$.

lsn	wid	$is-lsn$	t	α_{in}	α_{out}
1	1	1	START	-	-
2	2	1	START	-	-
3	1	2	GetRefer	-	hospital="Public Hospital", referId=034d1, referState=start, balance=1000
4	1	3	CheckIn	referId=034d1, referState=start, balance=1000	referState=active
5	2	2	GetRefer	-	hospital="People Hospital", referId=022f3, referState=start, balance=2000
6	3	1	START	-	-
7	3	2	GetRefer	-	hospital="Public Hospital", referId=048s1, referState=start, balance=1000
8	2	3	CheckIn	referId=022f3, referState=start, balance=2000	referState=active
9	1	4	SeeDoctor	referId=034d1, referState=active	-
10	1	5	PayTreatment	referId=034d1, referState=active	receipt1=560
11	1	6	SeeDoctor	referId=034d1, referState=active	-
12	1	7	PayTreatment	referId=034d1, referState=active	receipt2=460
13	2	4	SeeDoctor	referId=022f3, referState=active	-
14	2	5	UpdateRefer	referId=022f3, referState=active, balance=2000	balance=5000
15	1	8	GetReimberse	referState=active, balance=1000, receipt1=560, receipt2=460	amount=1020, balance=0, reimburse=1000
16	1	9	CompleteRefer	referState=active, balance=0	referState=complete
17	2	6	SeeDoctor	referId=022f3, referState=active	-
18	2	7	PayTreatment	referId=022f3, referState=active	receipt1=4560
19	2	8	TakeTreatment	referId=022f3, receipt1=4560	-
20	2	9	GetReimberse	referState=active, balance=5000, receipt1=6560	amount=6560, balance=0, reimburse=5000
⋮					

Fig. 2. An initial segment of a hospital referral log.

In the above definition, each workflow instance must have a *start* record in the log but may not have an *end* record (the instance is not yet completed). Also, this definition implies that *start* is the first record for each workflow instance.

Example 2.2. Clinics in Chinese colleges are only staffed with doctors for general practice and medical devices who use a “referral process” for complicated illnesses (such as cancer). The referral process for a student starts when she gets a referral from a doctor in college clinic. She then gets treatment at the referred hospital, makes payments, and gets receipts. She then go to the college for reimbursement (if her referral status is active). Also, a student can request to update her referral if the disease is diagnosed differently at the referred hospital. Then the new balance will cover every receipt produced after the update. Students can complete referral whenever they want.

Figure 2 shows an initial log segment for the referral processes. The log shows three active workflow instances. The column “*wid*” has three distinct values representing the ids for the three instances. Every workflow instance begins with a *start* log record. For each subsequent log record, attributes read and written are shown in the columns α_{in} and α_{out} .

One interesting query on the log in Figure 2 is “Do any students get reimbursement after they update referral?” This query concerns two activities *updateRefer* and *getReimburse* and their temporal relationship (*updateRefer* occurred before *getReimburse*). To express such queries, the notion “(incident) pattern” and “incident (instance)” were introduced based on four binary temporal operators [28]: consecutive $\ominus$, sequential $\odot$, exclusive $\otimes$, and parallel $\oplus$. An “(incident) pattern” is an expression for a pattern that occurs in workflow log. For the query mentioned above, the pattern is “*updateRefer* $\odot$ *getReimburse*.” An “incident (instance)” is an instance (a set of log sequence number) of an *incident* from a log. For the above pattern, the corresponding incident is $\{l_{14}, l_{20}\}$.

An abstract language with incidents and four operators introduced in References [28, 29] form an algebra. While this algebra is capable of expressing “sequencing” relationships among activities, an important weakness is its inability to select and constraint activities based on their data values in input/output maps. The algebra in References [28, 29] is in fact data agnostic. As our aim in this article is to develop query languages for practitioners (e.g., business managers, executives, stakeholders, etc.), an immediate task is (i) to extend this algebra [28, 29] by allowing constraints on log record attributes and (ii) to further develop a user-friendly SQL-like query language (Section 3). For this purpose, we extend the notions of “incident pattern” and “incident instance” in References [28] to allow constraints on data attributes.

Example 2.3. To illustrate the need for expressive languages to query workflow logs, consider again the college clinic example (Example 2.2). the clinic leadership plans to reorganize the departments to speed up referral workflow to serve an increased student population. Specifically, the directors want to find the number of cases where students updated their referral cases once or twice within a month but end with relatively small reimbursements. This count will give a good estimate of savings if all referrals are granted one or two automatic updates and also a possible budgetary impact.

In the current setting, the above task cannot be performed without the help of software engineers to write code to extract records from, e.g., a SQL database, and to check the activity sequencing, temporal, and data constraints. If the directors later would like to find cases with excessive referral updates (3 or more times) and their reimbursement amounts, then the software engineers need to again develop new code pieces to serve this new need. It will be much desired to bring query formulation task directly to the hands of the clinic management (domain experts) to not only reduce the need for software engineers but also the wait time to see the results due to the removal of software coding.

Consider the log discussed in Example 2.2, and the query “how many students every year get referrals with balance equal to \$1,000?” The query concerns the activity *getRefer* in some log records and the value of attribute balance in the output map of these log records. This query can be expressed by an “incident pattern” with data conditions in it. To introduce an “incident pattern,” we begin with the notion of “condition.”

Definition. Let $a \in \mathcal{A}$ be an attribute, $X, Y \in \mathcal{V}_A$ attribute variables, and $v \in \mathcal{D}$ a value. An *atom* is an expression of form: “ $\text{DEF}(a)$,” “ $a = X$,” “ $X = a$,” “ $X = v$,” “ $v = X$,” or “ $X = Y$.” A *negated atom* is of form “ $\neg c$ ” where c is an atom. A *condition* is an atom, negated atom, or one of the following, $\theta_1 \vee \theta_2$, $\theta_1 \wedge \theta_2$, $\neg \theta_1$, where θ_1, θ_2 are conditions.

The current formalism only includes equality; however, extensions to include other relational operators are rather straightforward and thus omitted for simplicity. Also we use Boolean constants *true* and *false* to denote conditions that are tautologies and contradictions, e.g., “ $X = X$ ” and “ $X \neq X$ ”.

Definition. Let β_1, β_2 be two conditions. The family of (incident) patterns is recursively defined as follows. If e_1 and e_2 be patterns with disjoint sets of variables, then the following are also patterns:

- $[\beta_1] x : t [\beta_2]$ where $x \in \mathcal{V}_T$ is an activity variable and $t \in \mathcal{T}$ an activity,
- $[\beta_1](e_1 \ominus e_2)[\beta_2]$ (consecutive),
- $[\beta_1](e_1 \odot e_2)[\beta_2]$ (sequential),
- $[\beta_1](e_1 \otimes e_2)[\beta_2]$ (exclusive choice), and
- $[\beta_1](e_1 \oplus e_2)[\beta_2]$ (concurrent).

If β_i ($i = 1, 2$) is *true*, then “ $[\beta_i]$ ” may be dropped, and the pattern $[\beta_1]e[\beta_2]$ is simply written as $[\beta_1]e$, or $e[\beta_2]$, or even e (note that there could still be conditions inside e).

We now define a key notion of an “incident instance” of a pattern. We start with a few notions concerning data attributes in workflow logs.

A “state” is a partial mapping from workflow instance id and attribute name pairs to values in $\mathcal{D}$, capturing the current attribute values within each workflow instance. Formally, a *state* is a (partial) mapping $s : \mathcal{D} \times \mathcal{A} \rightarrow \mathcal{D}$ with a finite domain. If $s(w, a) = v$, where w is a workflow instance id, a is an attribute, and v is a value, then it means that attribute a in the instance w has the value v . If $s(w, a)$ is undefined, then a is uninitialized in the workflow instance w . A state is *initial* if its domain is empty (all attributes are undefined and there are no workflow instances).

Let s be a state and l a log record. The state s is *input-consistent with* l if for each attribute a , $\alpha_{\text{in}}(l)(a) = v$ implies $s(\text{wid}(l), a) = v$. Similarly, the state s is *output-consistent with* l if for each attribute a , $\alpha_{\text{out}}(l)(a) = v$ implies $s(\text{wid}(l), a) = v$. Intuitively, s is input-(output-)consistent with l means that s agrees with the input (respectively, output) map of l .

Definition. Given a log $L = l_1 l_2 \dots l_n$, an *extended log* of L is an alternating sequence state-log records $s_0 l_1 s_1 l_2 \dots l_n s_n$ where s_0 is the initial state and for each $1 \leq i \leq n$, s_{i-1} is input-consistent with l_i , and s_i is output-consistent with l_i and differs from s_{i-1} only on attributes defined in $\alpha_{\text{out}}(l_i)$.

An extended log captures all attribute-value pairs that reflect the value changes recorded in the log. We use states in an extended log to evaluate conditions in patterns with the usual semantics.

Given a pattern and a log, we formulate in the following the notion of the log “satisfying” a pattern. The key notions “i-assignment” and “a-assignment” are introduced to map activity variables and attribute variables to log sequence number and values, respectively.

An *i-assignment* is a 1-1 mapping from $\mathcal{V}_T$ to $\mathbb{N}^+$. Given a log $L = l_1 l_2 \dots l_n$ and a pattern e , an i-assignment σ is an (L, e) -candidate if (1) for each variable $x \in \mathcal{V}_T$ occurring in e , $\sigma(x) \leq n$, and (2) for each pair of variables $x, y \in \mathcal{V}_T$ occurring in e , $l_{\sigma(x)} =_{\text{wid}} l_{\sigma(y)}$. Intuitively, an (L, e) -candidate maps all variables in e to log records in L from a single workflow instance.

An *a-assignment* is a mapping from $\mathcal{V}_A$ to $\mathcal{D}$. Given a pattern e , an a-assignment τ assigns each attribute variable in e a value in $\mathcal{D}$. Given a condition β and an a-assignment τ , $\tau(\beta)$ denotes formula obtained from β with each occurrence of an attribute valuable x replaced by the value $\tau(x)$. Given a state s , an a-assignment τ , and a condition β , $\tau(\beta)$ *evaluates* to true is defined in the usual manner, except that atoms of form $\text{DEF}(a)$ is satisfied if a has a value (i.e., is initialized) in s .

Definition. Let L be a log, $s_0 l_1 s_1 l_2 \dots l_n s_n$ an extended log of L , e a pattern, σ an (L, e) -candidate i-assignment, and τ an a-assignment. L *satisfies* e under σ and τ , denoted by $L \models_{\sigma, \tau} e$, is defined recursively as follows:

- $L \models_{\sigma, \tau} [\beta_1] x : t [\beta_2]$ if $\text{act}(l_{\sigma(x)}) = t$, $\tau(\beta_1)$ evaluates to *true* in the state $s_{\sigma(x)-1}$, and $\tau(\beta_2)$ evaluates to *true* in the state $s_{\sigma(x)}$.

- $L \models_{\sigma, \tau} [\beta_1](e_1 \ominus e_2)[\beta_2]$ if $L \models_{\sigma, \tau} [\beta_1]e_1$, $L \models_{\sigma, \tau} e_2[\beta_2]$, and $\min_{x \text{ in } e_2} \text{is-lsn}(l_{\sigma(x)}) = 1 + \max_{x \text{ in } e_1} \text{is-lsn}(l_{\sigma(x)})$.
- $L \models_{\sigma, \tau} [\beta_1](e_1 \odot e_2)[\beta_2]$ if $L \models_{\sigma, \tau} [\beta_1]e_1$, $L \models_{\sigma, \tau} e_2[\beta_2]$, and $\max_{x \text{ in } e_1} \text{is-lsn}(l_{\sigma(x)}) < \min_{x \text{ in } e_2} \text{is-lsn}(l_{\sigma(x)})$.
- $L \models_{\sigma, \tau} [\beta_1](e_1 \otimes e_2)[\beta_2]$ if $L \models_{\sigma, \tau} [\beta_1]e_1[\beta_2]$ or $L \models_{\sigma, \tau} [\beta_1]e_2[\beta_2]$.
- $L \models_{\sigma, \tau} [\beta_1](e_1 \oplus e_2)[\beta_2]$ if for some $i, j = 1, 2$ satisfying $\min_{x \text{ in } e_i} \sigma(x) < \min_{x \text{ in } e_{3-i}} \sigma(x)$ and $\max_{x \text{ in } e_j} \sigma(x) > \max_{x \text{ in } e_{3-j}} \sigma(x)$ such that $L \models_{\sigma, \tau} [\beta_1]e_i$, $L \models_{\sigma, \tau} e_{3-i}$, $L \models_{\sigma, \tau} e_j[\beta_2]$, and $L \models_{\sigma, \tau} e_{3-j}$.

Example 2.4. Consider the query “How many students every year get referrals with balance equal to \$1,000?” This query can be expressed as “A: *GetRefer*[balance= $X \wedge X = 1000$].” Based on the log example in Figure 2, the candidate i-assignments are $\sigma_1 = \{A \mapsto 3\}$ and $\sigma_2 = \{A \mapsto 5\}$ and $\sigma_3 = \{A \mapsto 7\}$. Let the a-assignments be $\tau = \{X \mapsto 1000\}$, (σ_1, τ) and (σ_3, τ) are incidents of the query.

The notions of (incident) pattern and incident (instance) are different from that in References [28, 29] in two aspects. First, their incident patterns [28, 29] do not include data attributes and conditions, while the notion in this section includes. Second, the notion of incident instance is extended to incorporate data attributes.

3 A QUERY LANGUAGE FOR INCIDENTS

In this section, we introduce an incident-based query language IQL for workflow logs. We first describe the overall design of IQL and then illustrate IQL with specific example queries from the hospital referral application.

IQL resembles SQL syntactically and semantically. However, IQL serves different purposes and focuses on log queries and temporal constraints. In particular, IQL queries assume that a set of records in a log is ordered as a sequence, while SQL queries assume that tuples in tables are not ordered. Also, IQL formulates temporal constraints over sequences, which SQL does not support directly. IQL uses the SELECT, FROM, WHERE components that play similar roles to the corresponding components in SQL and adds a FOR component that is similar to the FOR component in XQuery. These four components form a basic query block in IQL.

We start with the following key elements in IQL.

- (1) *Workflow names* whose (workflow) instances are queried. For example, the workflow name would be HospitalReferral, when the user queries the history of hospital referrals. The workflow name “INSTANCE” is used when the query focuses on all workflow instances regardless of workflow models.
- (2) *Log names* indicating the log to be queried on. For instance, the log name would be HospitalLog, when the user queries instances in the log.
- (3) *Activity names* identifying activities of interest in a query expression. For instance, *SeeDoctor* in the log example (Figure 2) is an activity name. The special activity name “ANY” matches every activity name (i.e., is a wildcard).
- (4) *Variables* that are divided into three kinds: *log variables*, *log record variables*, and *attribute variables* to hold workflow instances, log records, and attribute values, respectively.
- (5) *Log record attributes* to hold (workflow) instance ids (“wid”), activity names (“act”), and in/output maps (“in”/“out”). The expression “*LogRecordVar.AttName*” represents a log record attribute *AttName* in the log record held in *LogRecordVar*.
- (6) *States attributes* in a workflow. Recall that a state (Section 2) is a partial mapping from (workflow) instance id and attribute name pairs to values. We use *LogRecordVar.in* (or

LogRecordVar.out) to represent the state input-consistent (or out-consistent) with a log record *LogRecordVar*. Then *LogRecordVar.in.AttName* represents the value of workflow attribute *AttName* in the input-consistent state of the log record *LogRecordVar*. The syntax for attributes in output-consistent state is similar.

Main structure of IQL includes the following four clauses:

- (1) The FOR clause specifies the workflow and log from which the log records are fetched.
- (2) The SELECT clause defines the query answers. The query answers can be a (workflow) instance id (“wid”), or other log record attributes.
- (3) The FROM clause indicates the log records in the instances used to answer the query (including both the query condition and the answer).
- (4) The WHERE clause states the conditions to be satisfied by the query answer.

We now illustrate IQL with queries on the log shown in Figure 2.

Example 3.1. Consider the query Q_1 to find the instance id where activity *CheckIn* occurs before activity *UpdateRefer*. The query is expressed in IQL as shown below.

```

Q1:  FOR    INSTANCE L IN HospitalLog
      SELECT X.wid
      FROM    CheckIn@L X, UpdateRefer@L Y
      WHERE   X << Y

```

The FOR clause in Q_1 indicates that the query targets the log named HospitalLog where INSTANCE indicates that all workflow instances are considered. In particular, the log variable L holds an arbitrary workflow instance in the log HospitalLog.

In the FROM clause, “UpdateRefer@L” and “CheckIn@L” specify that X and Y (respectively) hold log records for activities *CheckIn* and *UpdateRefer* (respectively) in the instance L.

The WHERE clause contains a single pattern “X << Y,” where “<<” denotes sequential operator. “X << Y” means that *ls-lsn* in X is less than *ls-lsn* in Y. In this example, it means that the activity *CheckIn* occurred before *UpdateRefer*.

Finally, the SELECT clause specifies the query answer. In this example, the query answer is the (workflow) instance id (“wid”). The SELECT component here plays a similar role as the SELECT clause in SQL; the difference is that in IQL output values specified by log records and attributes instead of columns in relations (tables). As shown in the example query, X holds log records and here “X.wid” means the log record attribute for (workflow) instance ids. For the log shown in Figure 2, the query result is 2.

The notation “<<” denotes the sequential operator in the pattern of the WHERE clause in Example 3.1. Other operators in IQL are denoted as “:” (consecutive), “|” (or), and “&&” (parallel).

Note that activities *UpdateRefer* and *CheckIn* are explicitly specified in Q_1 . IQL also allows two activity instances, i.e., log records with the same activity name. For example, listing each workflow instance that updated referral two or more times can be expressed similarly to Q_1 except that X, Y both have the same activity name “UpdateRefer.” Sometimes, the query condition may not specify individual activity name. Example 3.2 below illustrates the use of “ANY” for such situations.

Example 3.2. We now consider the second example query Q_2 in which we want to query the activity name and log sequence number of a log record immediately followed by a log record with activity name *CheckIn*,

```

Q2:  FOR    INSTANCE L IN HospitalLog
      SELECT X.act, X.lsn
      FROM    ANY@L X, CheckIn@L Y
      WHERE   X :: Y

```

Note that the expression “ANY@L X” means that variable X holds log records in the instance L regardless of its activity name. Also, the query returns both the activity name and lsn in the log record variable X.

In Example 3.3, we illustrate the use of attribute variables in (incident) patterns and query results.

Example 3.3. Query Q_3 finds the value of attribute *balance* right after the *GetRefer* activity, when the activity *GetRefer* occurs before *CheckIn* and *balance* is 500,

```

Q3:  FOR    INSTANCE L IN HospitalLog
      SELECT X.wid, X.out.balance
      FROM    GetRefer@L X, CheckIn@L Y
      WHERE   X[X.out.balance=500] << Y

```

In the query expression, “X[X.out.balance = 500]” is a pattern that means that the balance value when X completes is 500. The square brackets correspond to the output-consistent state of X, i.e., X.out. The sequential operator further constrains that this *GetRefer* activity is followed by a *CheckIn* activity.

In Example 3.4, we focus on workflow attribute representation in FROM clause, which is used to construct a condition in an (incident) pattern.

Example 3.4. In example query Q_4 , we want to find the ids for instances where the patient sees a doctor, then updates the referral, finally gets reimbursement, but has a different reimbursed amount from the balance after the referral update,

```

Q4:  FOR    HospitalReferral L IN HospitalLog
      SELECT X.wid
      FROM    SeeDoctor@L X, UpdateRefer@L Y,
              GetReimburse@L Z, Y.out.balance B
      WHERE   X << (Y << Z) [NOT(Z.out.amount=B)]

```

Note that in the FOR clause, the use of “HospitalReferral L” indicates that we are only interested in workflow instances of the workflow model HospitalReferral.

We can use workflow attribute variables in the FROM clause to shorten the length of patterns in the WHERE clause and make it easy to read. In this example, “Y.out.balance B” is used in the FROM clause so that “Y.out.balance” in the WHERE clause can be conveniently replaced by “B.”

Based on the log in Figure 2, there is only one log record l_{14} for Y with activity name *UpdateRefer*. Its *wid* is 2. In the same (workflow) instance, there is only one log record l_{20} for Z with activity name *GetReimburse*. In this case, B has value 5000 and Z.out.amount has value 6560, so the condition is satisfied. With log record l_{13} with name *SeeDoctor*, the only i-assignment to the pattern is $\sigma = \{X \mapsto 13, Y \mapsto 14, Z \mapsto 20\}$. Notice that an a-assignment mapping B to any value in $\mathcal{D}$ can be satisfied by the pattern.

IQI is implemented with the semantics of IQI is based on the formal model outlined in Section 2. The current version is rather limited but provides the basis for further extensions. For example, the FOR clause can be extended to allow multiple workflow instances over which conditions are formulated in the WHERE clause. It is even possible to have multiple logs (i.e., a web store log and warehouse shipping log). Also, it is very useful to have aggregation.

4 QUERY EVALUATION

In this section, we discuss our query evaluation framework. We first discuss algorithms for incident operators in Section 4.1 and then discuss a query evaluation strategy in Section 4.2.

4.1 Evaluation of Incident Operators

As query evaluation is based on operator evaluation methods, we first discuss the evaluations for the four operators. Due to the differences of notions between this article and Reference [28], the evaluation algorithms in this section are adapted and extended from the ones presented there. We summarize the complexity of evaluating each operator in Lemma 4.1.

LEMMA 4.1 ([28]). *Let L be a log and for $i = 1, 2$, e_i a patterns and $s_L(e_i)$ a set of incidents of e_i on L with $|s_L(e_i)| = n_i$. The following hold.*

- *The answer to $e_1 \ominus e_2$ can be computed in time $O(n_1 n_2)$ and has size at most $n_1 \times n_2$.*
- *The answer to $e_1 \odot e_2$ (or $e_1 \oplus e_2$) can be computed in time $O(n_1 n_2)$ and has size at most (respectively, exactly) $n_1 \times n_2$.*
- *The answer to $e_1 \otimes e_2$ can be computed in time $O(n_1 + n_2)$ and has size at most $n_1 + n_2$.*

Note that operators consecutive, sequential and parallel have the same complexity and result size bound in the worst case (first two items of Lemma 4.1), but usually the consecutive operator is cheaper than sequential operator, and sequential operator is cheaper than parallel operator. This is naturally reflected by the strictness of temporal constraint of the three operators. The consecutive operator has the most severe restrictions, though the worst case is when all incidents for the right operand are consecutive with (happen right after) all incidents for the left. An algorithm for the sequential operator can be optimized to stop earlier when the constraint does not satisfy and there is no need to look into the later part of the input. For the exclusive operator, there may be incidents that satisfy both e_1 and e_2 . The algorithms in this article postpone deduplication till a later step of extracting query answers.

The evaluation algorithm presented in this article extends our earlier algorithm [28] in two aspects. First, due to the fact that incident patterns in the earlier algebra [28] do not contain constraints on data values, the evaluation algorithm in Reference [28] only maintain and manipulate i-assignments (i.e., activity timestamps). The algorithm in this article deals with both i-assignments and a-assignments (for data values) simultaneously. Consequently, the algorithm adds steps that checks data constraints to only keep the a-assignments (and coupled i-assignments) only when the data constraints are satisfied. Second, the evaluation (algorithm) in this article does not produce incidents directly, and an additional step is used to generate every incident for a pattern if there is an exclusive operator evolved. Because i-assignment requires all variables in a pattern to be mapped to a log sequence number, the algorithm for exclusive operator won't produce a complete incident for a pattern (only one branch is chosen and the other is dropped). Consider the following queries as an example, "which students get 2,000 balance after they get a referral or update a referral, and see a doctor afterwards?" A pattern expressing this query is $e_1 = ((X: \text{GetRefer} \otimes Y: \text{UpdateRefer})[\text{balance} = A \wedge A = 2,000]) \odot Z: \text{SeeDoctor}$. An incident, according to Figure 2, is (σ, τ) where $\sigma = \{X \mapsto 5, Y \mapsto 14, Z \mapsto 13\}$ and $\tau = \{A \mapsto 2000\}$. However, the algorithm for exclusive operator does not generate a full assignment. It just produces part of the assignment $\{X \mapsto 5, Z \mapsto 13\}$ without assigning a log sequence number to Y , because if X is chosen as the execution branch, then Y is dropped. Also, evaluation in reference [28] is simpler, since patterns do not involve data.

Because the result generated by an operator evaluation algorithm may not be a complete incident, we call the result generated "witness assignment." A *witness assignment* is a subset of an

assignment that satisfies the temporal relationship and conditions defined in a pattern. Correspondingly, the complement set of the assignment is the mappings dropped when we choose a branch at an exclusive operator. In Section 4.2, we discuss how to generate an assignment from a witness assignment.

4.2 Evaluation of IQL Queries

Given an IQL query and a log, our evaluation framework consists of the following four steps. The first step parses the pattern in WHERE clause and turns the pattern into an “incident tree.” An incident tree represents an evaluation plan. In the second step, the incident tree produced in Step 1 is converted into an “optimal” one. Then the third step uses a bottom-up evaluation strategy to generate all incidents to the pattern using the log. Finally, the last step extracts the query answer from every incident.

With the use of witness assignments, the query evaluation steps can be reformulated as follows. In the third step, we generate all the witness assignments using a bottom-up strategy. In the final step, all incidents are constructed based on witness assignments, and query answers are extracted from every incident. For example, in the pattern e_1 described in the above section, the two witness assignments, based on the log example in Figure 2, are $\{X \mapsto 5, Z \mapsto 13\}$ and $\{X \mapsto 5, Z \mapsto 17\}$. To generate incidents, for variables that do not occur in witness assignments, map them to available values. Here are the corresponding incidents for e_1 , $\{X \mapsto 5, Y \mapsto 14, Z \mapsto 13\}$ and $\{X \mapsto 5, Y \mapsto 14, Z \mapsto 17\}$.

Before introducing our query evaluation framework, we define the notion of an incident tree.

Definition. An *incident tree* is a binary tree with three types of nodes, namely operator, condition, and activity, that satisfies all of the following:

- An *operator* node has both left and right child nodes, and is labeled with an incident operator,
- A *condition* node has only left child node, and is labeled with a *pre-condition* and a *post-condition*, and
- An *activity* node is a leaf node and is labeled with an activity variable and an activity name. The variables can be eliminated if every activity name only occurs once in an incident tree.

Example 4.2. To visualize an incident tree, we use a rectangle to represent an activity node, a circle to represent an operator node, and a diamond to represent a condition node. Figure 3 shows two incident trees for patterns

$SeeDoctor \odot (UpdateRefer \odot GetReimburse[balance = X \wedge X = 0])$ (Figure 3(a))

$SeeDoctor \odot (UpdateRefer \odot GetReimburse)[balance = X \wedge X = 0]$ (Figure 3(b))

For condition nodes, we use the notation “(*pre-condition*, *post-condition*)” as a node label. In Figure 3, the condition nodes have *condition1* as post-condition that represents $balance = X \wedge X = 0$ and no pre-condition. (We will explain the numbers attached to the edges later.)

The first step of the evaluation framework is to convert a query into an incident tree. In an incident tree, every subtree corresponds to a pattern. In Step 3 of evaluating an incident tree, we use a bottom-up strategy, i.e., leaf nodes are processed first and the root is evaluated last. More specifically, we process nodes according to post-order traversal of the incident tree. The witness assignments for each subtree are generated and passed to its parent. The parent node then filters the witness assignments from its child nodes based on its node type and label. If the current node is an operator node, depending on the label of this node, then the corresponding algorithm (omitted in the article) is used to generate witness assignments. This process repeats and ends at the root

Fig. 3. Two incident trees.

Fig. 4. Illustration of bottom-up evaluation.

node. At the last step, for a set of witness assignments, every variable that is not in this set but is in the pattern is assigned a value. If the new mapping satisfies the definition of candidate assignment, then it is then an incident. Finally, query results are assembled from the final set of incidents according to the SELECT clause.

Example 4.3. Consider an evaluation of the incident tree in Figure 3 over the log in Figure 2. The evaluation process starts from the leaf nodes. A set of witness assignments for each of the leaf nodes is generated from the log. In Figure 4, we use dashed rectangles next to the nodes to show all witness assignments that we get from the log example (Figure 2). Here “(S, 9)” represents the i-(a)-assignment $\sigma = \{S \mapsto 9\}$.

After processing each activity node, the condition node with “ $balance = A \wedge A = 0$ ” has results from all of its children, it is then executed to filter the input with its conditions. For instance, in the log example Figure 2, both the log records l_{15} and l_{20} have $balance = 0$ in its output map,

so assigning 0 to A satisfies the post-condition. The witness assignments are shown next to the condition node.

The result generated at the condition node is then passed to the parent (a sequential operator in this case). This node takes the inputs as arguments and call the algorithm of sequential operator. The result is presented next to the node.

Finally, the results are passed to the root, also a sequential operator. The root does the same operation as what happens in its right child and finally generates all the witness assignments for the pattern. The final result are shown next to the root. In this example, there is only one witness assignment (σ, τ) where $\sigma = \{S \mapsto 13, U \mapsto 14, G \mapsto 20\}$, $\tau = \{A \mapsto 0\}$. The witness assignment (σ, τ) is also an incident.

THEOREM 4.4. *Let L be a log, m the length of L , and n the number of variables in a pattern e . Then the size of incidents for e is exponential in n in the worst case.*

The above (expression complexity) is not surprising. The average size of assignments generated for each activity is m/n . By Lemma 4.1, the operator $\oplus$ and $\odot$ have the largest result size among the four operators. Especially in the case that all the log records are from the same workflow instance, then the operator $\oplus$ has the result size $(m/n)^2$ exactly. For a pattern with all operators are $\oplus$, the size is $(m/n)^n$, which is exponential on n . For the other two operators $\ominus$ and $\otimes$, the result size is quadratic, respectively. With the data conditions, the parallel operator is still the most expensive.

5 QUERY OPTIMIZATION

Workflow logs are typically huge in size; for example, the housing management in Hangzhou, China with a population of 9 million handles 300,000 cases real estate transactions each year. The need for efficient IQL query evaluation is highly demanded. In this section, we focus on query optimization. We start with relevant properties of incident operators, then introduce a new cost model suitable for log queries, and finally provide heuristics to prune the search space of evaluation plans.

5.1 Properties of Incident Operators

An initial step for optimization is to examine properties concerning incident operators, including commutativity, associativity, distributivity on operator nodes and other properties on condition nodes. Such properties allows equivalent expressions to be considered for optimization plans. Some of the results concerning incidents presented in below extend from the results reported in References [28, 29]. However, patterns in References [28, 29] contain no data nor data conditions while the results provided in this section have data conditions. The “equivalence” ($\equiv$) notion is naturally extended from the dataless one [28].

THEOREM 5.1. *Exclusive operator $\otimes$ and parallel operator $\oplus$ are commutative, i.e., for each pair of patterns e_1 and e_2 , $e_1 \theta e_2 \equiv e_2 \theta e_1$ where $\theta \in \{\otimes, \oplus\}$.*

PROOF. We sketch a proof for operator $\otimes$ here. The goal is to prove that given arbitrary conditional incidents e_1, e_2 and a log L , for any i-assignment σ and a-assignment τ , $L \models_{\sigma, \tau} e_1 \otimes e_2$ iff $L \models_{\sigma, \tau} e_2 \otimes e_1$. Indeed, if $L \models_{\sigma, \tau} e_1 \otimes e_2$, then $L \models_{\sigma, \tau} e_1$ or $L \models_{\sigma, \tau} e_2$. Based on the definition of satisfaction, $L \models_{\sigma, \tau} e_2 \otimes e_1$. The proof for the other direction is quite similar, and a similar argument works for operator $\oplus$. The details for these cases are omitted. $\square$

Example 5.2. Consider the pattern “ $X: \text{UpdateRefer} \oplus Y: \text{GetReimburse}$.” It means that a patient updates referral and gets reimbursement, and the order of doing these does not matter. Based on the log example in Figure 2, the only incident is $\{X \mapsto 14, Y \mapsto 20\}$. By Theorem 5.1, “ $Y: \text{GetReimburse} \oplus X: \text{UpdateRefer}$ ” is an equivalent pattern, and indeed has the same incident.

THEOREM 5.3. *Each of consecutive $\ominus$, sequential $\odot$, exclusive $\otimes$, and parallel $\oplus$ is associative, i.e., given arbitrary patterns e_1, e_2, e_3 , $(e_1 \ominus e_2) \theta e_3 \equiv e_1 \theta (e_2 \theta e_3)$ for each $\theta \in \{\ominus, \odot, \otimes, \oplus\}$.*

Consider operator “ $\ominus$ ” as an example when e_1, e_2, e_3 are atomic, i.e., single activities x_1, x_2, x_3 . Assume $\sigma = \{x_1 \mapsto v_1, x_2 \mapsto v_2, x_3 \mapsto v_3\}$ and $\tau = \emptyset$, where $v_1, v_2, v_3 \in \mathcal{D}$ are LSN’s. If $L \models_{\sigma, \tau} (e_1 \ominus e_2) \ominus e_3$, then we have $l_{v_1} =_{\text{wid}} l_{v_2} =_{\text{wid}} l_{v_3}$, $\text{is-lsn}(l_{v_1}) + 1 = \text{is-lsn}(l_{v_2})$, and $\text{is-lsn}(l_{v_2}) + 1 = \text{is-lsn}(l_{v_3})$. According to the definition of i-assignment, it follows that $L \models_{\sigma, \tau} e_1 \ominus (e_2 \ominus e_3)$.

PROOF OF THEOREM 5.3: We sketch the proof of operator “ $\ominus$ ” in the following, proofs for other operators are similar and thus omitted. The goal is to prove that given arbitrary patterns e_1, e_2, e_3 and a log L , for each i-assignment σ and each a-assignment τ , $L \models_{\sigma, \tau} (e_1 \ominus e_2) \ominus e_3$ iff $L \models_{\sigma, \tau} e_1 \ominus (e_2 \ominus e_3)$.

Suppose $L \models_{\sigma, \tau} (e_1 \ominus e_2) \ominus e_3$. Then $L \models_{\sigma, \tau} (e_1 \ominus e_2) \ominus e_3$ and $L \models_{\sigma, \tau} e_i$, ($i = 1, 2, 3$). (Conditions may occur in e_i .)

$$\begin{aligned}
 & L \models_{\sigma, \tau} (e_1 \ominus e_2) \ominus e_3 \\
 & \Rightarrow 1 + \max_{x \text{ in } (e_1 \ominus e_2)} \text{is-lsn}(l_{\sigma(x)}) = \min_{x \text{ in } e_3} \text{is-lsn}(l_{\sigma(x)}) \text{ and } L \models_{\sigma, \tau} e_1 \ominus e_2 \\
 & \Rightarrow 1 + \max_{x \text{ in } e_1} \text{is-lsn}(l_{\sigma(x)}) = \min_{x \text{ in } e_2} \text{is-lsn}(l_{\sigma(x)}) \\
 & \Rightarrow 1 + \max_{x \text{ in } e_1} \text{is-lsn}(l_{\sigma(x)}) \leq \max_{x \text{ in } e_2} \text{is-lsn}(l_{\sigma(x)}) \\
 & \Rightarrow 1 + \max_{x \text{ in } (e_1 \ominus e_2)} \text{is-lsn}(l_{\sigma(x)}) = 1 + \max\{\max_{x \text{ in } (e_1)} \text{is-lsn}(l_{\sigma(x)}), \max_{x \text{ in } e_2} \text{is-lsn}(l_{\sigma(x)})\} \\
 & \quad = 1 + \max_{x \text{ in } e_2} \text{is-lsn}(l_{\sigma(x)}) = \min_{x \text{ in } e_3} \text{is-lsn}(l_{\sigma(x)}) \\
 & \Rightarrow L \models_{\sigma, \tau} e_2 \ominus e_3 \\
 & \Rightarrow \min_{x \text{ in } (e_2 \ominus e_3)} \text{is-lsn}(l_{\sigma(x)}) = \min\{\min_{x \text{ in } e_2} \text{is-lsn}(l_{\sigma(x)}), \min_{x \text{ in } e_3} \text{is-lsn}(l_{\sigma(x)})\} \\
 & \quad = \min_{x \text{ in } e_2} \text{is-lsn}(l_{\sigma(x)}) = 1 + \max_{x \text{ in } e_1} \text{is-lsn}(l_{\sigma(x)}) \\
 & \Rightarrow L \models_{\sigma, \tau} e_1 \ominus (e_2 \ominus e_3)
 \end{aligned}$$

Based on $L \models_{\sigma, \tau} e_1 \ominus (e_2 \ominus e_3)$ and $L \models_{\sigma, \tau} e_i$, ($i = 1, 2, 3$), $L \models_{\sigma, \tau} (e_1 \ominus e_2) \ominus e_3$.

The main idea of the above proof is to reason the temporal relationship (*is-lsn*) between the incidents. The proof of the other direction is essentially the same.

The following example illustrates that data conditions may move across expression boundaries (but not across activities) while maintaining equivalence.

Example 5.4. The two patterns

“*SeeDoctor* $\odot$ (*UpdateRefer* $\odot$ *GetReimburse*[amount > reimburse])” (Figure 3(a)) and

“*SeeDoctor* $\odot$ (*UpdateRefer* $\odot$ *GetReimburse*)[amount > reimburse]” (Figure 3(b))

are equivalent. Intuitively, they both have the meaning that a patient gets referral updated after he/she sees a doctor and before he/she gets reimbursement. The incident for both of them is {13, 14, 20}.

THEOREM 5.5. *Let e_1, e_2, e_3 be patterns. The following are all true.*

- (1) $e_1 \ominus (e_2 \ominus e_3) \equiv (e_1 \ominus e_2) \ominus e_3$ and $e_1 \odot (e_2 \odot e_3) \equiv (e_1 \odot e_2) \odot e_3$.
- (2) For each $\theta \in \{\ominus, \odot, \oplus\}$, $e_1 \theta (e_2 \theta e_3) = (e_1 \theta e_2) \theta (e_1 \theta e_3)$ and $(e_1 \otimes e_2) \theta e_3 = (e_1 \theta e_3) \otimes (e_2 \theta e_3)$.
- (3) $[\alpha](e_1 \ominus e_2)[\beta] = ([\alpha]e_1) \ominus (e_2[\beta])$, $[\alpha](e_1 \odot e_2)[\beta] = ([\alpha]e_1) \odot (e_2[\beta])$, and $[\alpha](e_1 \otimes e_2)[\beta] = ([\alpha]e_1[\beta]) \otimes ([\alpha]e_2[\beta])$.

From the above theorem, (1) it does not matter which order we take to evaluate if the two operators are either consecutive or sequential, (2) distributivity rules permits alternative expressions that may have lower cost on evaluation. and (3) the conditions can always be pushed through into the lowest level if the operators are one type of consecutive, sequential, or exclusive.

We omit the detail proofs here but note that most of the proofs are similar to the ones in Reference [28] except the following. Item 3 of Theorem 5.5 holds because of the following. Given

Parameters	Description
N_a	Total number of log records reflecting the execution of activity a in log L
$B_{a,t}$	Number of log records with $is-lsn = t$ reflecting activity executions of a in log L
W_a	A set of <i>wids</i> satisfying that at least one log record of a occurs in the workflow instance with its <i>wid</i> in W_a

Fig. 5. Cost model parameters for each activity a .

three patterns e_1, e_2 , and e_3 , for each i-assignment σ and each a-assignment τ , $L \models_{\sigma, \tau} [\alpha](e_1 \odot e_2)[\beta] \Rightarrow L \models_{\sigma, \tau} [\alpha]e_1, L \models_{\sigma, \tau} e_2[\beta]$, and $1 + \max_{x \in e_1} is-lsn(l_{\sigma(x)}) = \min_{x \in e_2} is-lsn(l_{\sigma(x)}) \Rightarrow 1 + \max_{x \in [\alpha]e_1} is-lsn(l_{\sigma(x)}) = \min_{x \in e_2[\beta]} is-lsn(l_{\sigma(x)}) \Rightarrow L \models_{\sigma, \tau} ([\alpha]e_1 \odot e_2[\beta])$.

Example 5.6. Consider the incident trees in Figure 3. The numbers next to the edges represent the number of witness assignments to patterns that are constructed by the corresponding subtrees.

According to Lemma 4.1, the time cost for Figure 3(a) is roughly $100 + 122 \times 15 + 150 \times 20 = 4930$, where 100 is the cost for the condition node, 122×15 for the operator “ $\odot$ ” at the second level, and 150×20 for the root. Accordingly, the time cost for Figure 3(a) is roughly $122 \times 100 + 300 + 150 \times 20 = 15500$.

Because the patterns represented by the two incident trees are equivalent, the tree in Figure 3(a) is a cheaper evaluation plan than the one in Figure 3(b).

In the next subsection we introduce a cost model to estimate the evaluation cost of a pattern to serve as the basis for query optimization.

5.2 Cost Model

Although IQL uses process modeling constructs in formulating query conditions on log record sequences, there is a close resemblance between IQL query evaluation and SQL query evaluation. Naturally, we study IQL optimization using a cost-based method (similarly to SQL [25]). A core of this approach is to develop an accurate cost model.

Based on Lemma 4.1, the cost of each operator depends on the cardinality of incidents from its operands and their temporal relationship. To estimate the cost of each operand, we develop a cost model with three sets of parameters as shown in Figure 5. For each activity a , N_a is the total number of log records reflecting the execution of activity a in a log. It captures cardinality of atomic incidents. As *is-lsn* directly reflects the order of log records, we use another set of parameters to capture the temporal relationship based on *is-lsn*. For each activity a and a natural number t , $B_{a,t}$ is the number of log records with $is-lsn = t$ reflecting activity executions of a in a log L , i.e., the number of activity a executing as the t th activity in a workflow instance. For example, based on the hospital referral example, $B_{GetRefer, 2} = 3$ because *GetRefer* occurs as the 2nd activity in all three workflow instances. The last set of parameters is to reflect the instance information of each activity, because we are interest in incident within one workflow instance. For each activity a , W_a is a set of *wids*. The *wids* are the ids of instances where a occurs.

By accumulating $B_{a,t}$ with all numbers t , we get a histogram showing number of log records of activity a occurring as the t th activity over all the workflow instances. From this histogram, we can easily get the distribution of where an activity occurs within a workflow instance. For example, in the hospital referral example, *GetRefer* always starts as one of the early activities, so it always has small *is-lsn*. However, *GetReimburse* may probably starts with much larger *is-lsn*. These histograms of all activities are collected before any query arrives and are used in our optimizer for estimation.

In the following, we outline how we estimate the cardinality of incidents of a pattern. For now, we only consider the case where all log records within a log are from one workflow instance.

First, we introduce some notations to represent the properties of incidents that we would like to use to help estimation. We assume each incident is a random event, and each of the following notation is a random variable. (1) N_e , number of incidents of pattern e in the log L . (2) $B_e = \min_{x \in e} is-lsn(l_{\sigma(x)})$, begin of incident with pattern e , (3) $D_e = 1 + \max_{x \in e} is-lsn(l_{\sigma(x)}) - \min_{x \in e} is-lsn(l_{\sigma(x)})$, duration of incident with pattern e . Here, we also assume (2) and (3) are independent from each other, i.e., when an incident happens is independent from how long it lasts. And every atomic incident is independent from each other, i.e., when an activity happens is independent from when other activities happen. If we can estimate the distribution of (2) and (3), then we can estimate (1) by aggregation of (2) (or (3)) on t . Then we can use (1) to estimate the computation cost of an incident operator. For this purpose, these parameters are first estimated for atomic incidents (leaf level in an incident tree) and are then propagated to complex incidents. Finally, the size of a query answer and the total cost of evaluating a query can be estimated using a bottom-up traversal of an incident tree. We now discuss how to estimate (1), (2), and (3) for each incident.

To propagate the estimations to the root of an incident tree, we need to provide an estimation method for each type of nodes, activity node, operator node and condition node. For condition node, we use a fixed rate to filter the candidates here. Practically, this rate can be learned and tuned using machine learning techniques. For the remaining two cases, we start from activity node.

For an atomic pattern a , we use maximum likelihood estimation to estimate the distribution of its begin and duration. That is, use our observation from the current log to estimate the behavior of activity a on start. Therefore,

$$P(B_a = t) = B_{a,t} / N_a.$$

We assume the duration for each activity is 1, then

$$P(D_a = 1) = 1.$$

For an operator node with label $\ominus$, the corresponding pattern is $e_1 \ominus e_2$. The start can be estimated in the following:

$$P(B_{e_1 \ominus e_2} = t) = \sum_{t'} P(B_{e_1} = t, D_{e_1} = t', B_{e_2} = t+t') = P(B_{e_1} = t) \cdot \sum_{t'} P(D_{e_1} = t') \cdot P(B_{e_2} = t+t'). \quad (1)$$

By definition of $\ominus$, $B_{e_1 \ominus e_2} = t$ is equivalent to $B_{e_1} = t$ (e_1 has log sequence number t), $D_{e_1} = t'$ (e_1 has duration t'), $B_{e_2} = t + t'$ (e_2 happens immediately after e_1 finishes). This leads to the first step in Equation (1). At the second step, by our assumption, begin and duration are independent. Here we further assume B_{e_1} is independent from B_{e_2} . This holds if e_1 and e_2 have no overlaps on activity names. For simplicity, we still treat e_1 and e_2 to be independent when they have overlap. Note that each of the terms $P(B_{e_1} = t)$, $P(D_{e_1} = t')$, $P(B_{e_2} = t + t')$ is already estimated for e_1 and e_2 . If e_1 (or e_2) is atomic, then the estimation is done at the leaf node. Otherwise, the estimation is propagated from child nodes. Thus, $P(B_{e_1 \ominus e_2} = t)$ is estimated. Similarly, we can estimate the duration of $e_1 \ominus e_2$ by the following:

$$\begin{aligned} P(D_{e_1 \ominus e_2} = t) &= \sum_{t_2 < t} P(B_{e_1} = t_1, D_{e_1} = t_2, B_{e_2} = t_1+t_2, D_{e_2} = t-t_2) \\ &= \sum_{t_2 < t} P(B_{e_1} = t_1) \cdot P(D_{e_1} = t_2) \cdot P(B_{e_2} = t_1+t_2) \cdot P(D_{e_2} = t-t_2). \end{aligned} \quad (2)$$

The cardinality of $e_1 \ominus e_2$ is then estimated as

$$N_{e_1 \ominus e_2} = \Phi \cdot N_{e_1} \cdot N_{e_2},$$

where $\Phi = \sum_t P(D_{e_1 \ominus e_2} = t)$. Here Φ represents the likelihood of a pair of e_1 and e_2 satisfying the definition of $e_1 \ominus e_2$. For the remaining case of incident operators, the idea is similar. The only difference is the condition that represents the different temporal relationship between two incidents using start and duration. We give some explanations here and details are skipped. For sequential operator, the condition should satisfy that e_2 happens after e_1 , the start of $e_1 \odot e_2$ is the start of e_1 , and the duration of $e_1 \odot e_2$ is from the start of e_1 to the end of e_2 ; for exclusive operator, there is no temporal constraint on e_1 and e_2 , and the start and duration of $e_1 \otimes e_2$ are either the ones of e_1 or that of e_2 ; for parallel operator, there is no temporal constraint, the start of $e_1 \oplus e_2$ is the minimum of the start of e_1 and the start of e_2 , and the duration of $e_1 \oplus e_2$ is the duration from the minimum start to the maximum end.

We now extend the estimation for cases where there are multiple workflow instances. Recall that in the definition of incidents, we also have the constraints that all the log records should have the same *wid*. To extend the above estimation, we add a factor that represents the constraint on *wid*. Given two pattern e_1, e_2 and their incidents respectively, $P(e_1 =_{wid} e_2)$ represents the probability of any two incidents from e_1 and e_2 respectively having the same *wid*. To estimate this probability, we assume that the incidents of a pattern are uniformly distributed over all the instances where the incidents occur, for example, if a pattern e has 10 incidents ($N_e = 10$) and all the incidents occur within workflow instance with *wid* = 2 and *wid* = 5. Then 5 incidents of e occur in the instance with *wid* = 2 and 5 other incidents of e occur in the instance with *wid* = 5. Then we add a new property to each incident. For a pattern e , W_e is a set of *wid*'s. The *wid*'s are ids of instances where the incidents of e occur. If e is an atomic pattern, then W_e is collected as the parameter in Figure 5. If e is $e_1 \theta e_2$, where $\theta \in \{\ominus, \odot, \otimes, \oplus\}$, then W_e is estimated as the conjunction of W_{e_1} and W_{e_2} . Then $P(e_1 =_{wid} e_2)$ can be estimated by $N_{e_1}/W_{e_1} * N_{e_2}/W_{e_2} * W_e/(N_{e_1} * N_{e_2}) = W_e/(W_{e_1} * W_{e_2})$. The corresponding extension to the estimation of begin and duration of e are as follows:

$$\begin{aligned} P^*(B_{e_1 \ominus e_2} = t) &= P(B_{e_1 \ominus e_2} = t) * P(e_1 =_{wid} e_2) \\ P^*(D_{e_1 \ominus e_2} = t) &= P(D_{e_1 \ominus e_2} = t) * P(e_1 =_{wid} e_2). \end{aligned}$$

Once we obtain the estimation of incident parameters, the next step is to estimate the cost of each operator. By extending Lemma 4.1, we define the following. Notice that not like other operators, the cost estimation for consecutive operator $\ominus$ is not consistent with Lemma 4.1. The reason is the complexity in the lemma is the worst case but it is not a common case. We assume that the incidents for e_1 and e_2 have random starts and ends, then the complexity is close to linear time (one scan through each set). To provide a more precise estimation for a common case, we use $N_{e_1} + N_{e_2}$ instead of $N_{e_1} * N_{e_2}$ for consecutive operator.

Definition. Given (incident) patterns e_1 and e_2 with estimated cardinality of incident (instances) N_{e_1} and N_{e_2} , the cost of a pattern e is

- $N_{e_1} + N_{e_2}$ if e is $e_1 \ominus e_2$,
- $N_{e_1} * N_{e_2}$ if e is $e_1 \odot e_2$ or $e_1 \oplus e_2$,
- $N_{e_1} + N_{e_2}$ if e is $e_1 \otimes e_2$.

5.3 Optimization

To estimate the cost of an incident tree, we use our cost model to evaluate the cost based on a post-order traversal. At each node, we consider the current node type and apply the cost estimation methods defined at the end of Section 5.2, finally the cost will be accumulated at the root. The details are shown in Algorithm 1.

ALGORITHM 1: Cost calculation algorithm**Input:** Incident tree *root*, Rule list *ruleList*;**Output:** Incident tree *optiTree*;

```

1: function GETTREECOST
2:   if root.type == Activity then
3:     return root;
4:   end if
5:   leftParas = getTreeCost(root.left);
6:   rightParas = getTreeCost(root.right);
7:   return estimateCost(root, leftParas.size, rightParas.size) + leftParas.cost + rightParas.cost;
8: end function

```

ALGORITHM 2: Optimal incident tree generation algorithm**Input:** Incident tree *tree*, Rule list *ruleList*;**Output:** Incident tree *optiTree*;

```

1: function GENERATEOPTIMALTREE
2:   frontier = {};
3:   visited = {};
4:   optiTree = null;
5:   frontier.add(tree);
6:   while !frontier.isEmpty() do
7:     curTree = frontier.remove();
8:     curCost = getTreeCost(curTree.root, ruleList);
9:     if optiTree != null or curCost < optiTree.cost then
10:      optiTree = curTree;
11:     end if
12:     for r in ruleList do
13:       candis = generateCandidate(curTree, r);
14:       frontier.addAll(candis);
15:     end for
16:   end while
17:   return optiTree;
18: end function

```

To enumerate the rewriting space of a pattern, we traverse its incident tree from the root. At each node, we enumerate all the rules introduced in Section 5.1 to generate new incident trees. New incident trees are added to frontier and will be processed later to generate more incident tree candidates. Among these trees, we pick the one with least cost as the optimal. The details are provided as Algorithm 2.

We conjecture that enumeration of all rewritings to a query is NP-hard. We analyze the properties of incident operators and build the foundation of our conjecture. Theorem 5.3 means that, for a query with only operator types $\ominus$ and $\odot$, we can execute the operators in any order to obtain the (identical) results. For a query with n such operators, there are $n!$ different incident trees. So the search space is $O(n!)$. Obviously, the search space grows when $\otimes$ and $\oplus$ are involved. Because the order of the operands can be changed based on Theorem 5.1.

To derive the “optimal” evaluation plan for a query, we provide some heuristics here to help avoid searching the whole space of the problem. In the following, we go over the rules introduced in Section 5.1, and discuss what heuristics to apply when one of the rules can be applied. Let e_1, e_2, e_3 be arbitrary patterns, for the incident rule in

Dataset	# Patients	# Log Records
2007	1,629	273,982
2008	3,337	328,160
2009	5,033	334,657
2010	6,695	493,325
2011	1,946	109,795

Fig. 6. Trauma patient datasets.

- (1) Theorem 5.3, i.e., $(e_1\theta e_2)\theta e_3$ vs. $e_1\theta(e_2\theta e_3)$, for $\theta \in \{\ominus, \odot, \otimes, \oplus\}$, either query is preferred depends on the cost model, because the size of different assignments may significantly affect the results.
- (2) Theorem 5.1, i.e., $e_1\theta e_2$ vs. $e_2\theta e_1$ for $\theta \in \{\otimes, \oplus\}$, the rule should be applied together with distributive rules introduced in Theorem 5.5(2).
- (3) Theorem 5.5(1), i.e., $e_1\ominus(e_2\odot e_3)$ vs. $(e_1\ominus e_2)\odot e_3$, always choose the query that executes “ $\ominus$ ” first. The intuition is that operator “ $\ominus$ ” has more tight constraint when constructing assignments, so it is more likely to have a smaller size of intermediate assignments, and reduce the cost of $\odot$ executed later.
- (4) Theorem 5.5(2), i.e., $e_1\theta(e_2 \otimes e_3)$ vs. $(e_1\theta e_2) \otimes (e_1\theta e_2)$, for $\theta \in \{\ominus, \odot, \oplus\}$, if θ is $\ominus$ or $\odot$, use cost model to determine which query to choose (which plan is cheaper depends on the size of input), otherwise always choose the first query that executes operator $\otimes$ first, because operator $\oplus$, is likely to increase the size of the result significantly, which would also increase the cost of $\otimes$.
- (5) Theorem 5.5(3), always choose to push the conditions to lower levels. The intuition here is that the condition is likely to filter more candidate assignments, which will reduce the cost for later incident operators.

To make the optimizer more efficient, during the search process, when an incident rule can be applied, we use the above corresponding heuristics to determine if we should stop expanding on the current candidate.

6 EXPERIMENTAL EVALUATION

We implemented our algorithms as presented in the previous section [27]. In this section, we evaluate our query evaluation strategy and query optimization model. The experiment setup is as follows:

- Intel Core i5, 3.20 GHz, and 16.0 GB DRAM.
- Dataset: National Trauma Data Bank. These data contain treatment logs that patients received from the time patients went to the hospital to the time they were discharged. There are five datasets from 2007 to 2011, respectively, with number of patients ranging from 1,629 to 6,695 (109,795 to 493,325 records). Figure 6 shows the datasets.
- Queries: There are three types of queries.
 - T1: single activity, e.g., *TakeVital*.
 - T2: one operator with two activities, e.g., *TakeVital* $\ominus$ *SentToICU*.
 - T3: multiple mixed operators with multiple activities, e.g., *TakeVital* $\ominus$ *SentToICU* $\odot$ *Discharge*.

To construct queries, we randomly pick activity names for query type T1 to T3, and randomly choose operators for query type T3. The correctness of query results is verified manually. In the following, we conduct experiments from three aspects. First, we study the factors that affect the efficiency of our evaluation strategy, and then we evaluate the effectiveness of our cost estimation.

Fig. 7. Time of queries T1 on dataset of year 2007.

Finally, we discuss the effect of our optimizer. The running time of each query type was measured by 10 runs of each query on the corresponding dataset and the average of the running time of these 10 queries is used as the final result.

6.1 Factors of Query Evaluation Cost

In this section, we discuss several factors affecting query evaluation cost: the number of the log records for an activity, the operator type, and the number of operators. We conducted three experiments regarding each of the factors one by one.

The first experiment is conducted by evaluating query type T1 with varying numbers of log records for that activity. In Figure 7, the x -axis denotes the number of log records, and the y -axis means the evaluation time for the query. Figure 7 shows the evaluation time of the single activity query increase linearly in the number of log records. The linearity is expected, because a linear scan is used on the log to collect the log records for an atomic pattern.

The second focus is on incident operator types. Figure 8 shows the evaluation time on five datasets of queries with one operator and two activities (T2), respectively. We randomly generated 10 queries for each operator type. The average evaluation time is shown as the y -axis.

From the performance on each dataset, we can tell that consecutive operator has the best performance and parallel operator has the worst performance, which is consistent of Lemma 4.1. Sequential operator has better performance than parallel operator. On most datasets, consecutive operator has better performance than exclusive operator.

In Lemma 4.1, sequential operator has the same complexity as parallel operator. But the former has much better performance in Figure 8. This is because there are expected pruning based on time constraints in sequential operator. But in parallel operator, there is no constraint on log sequence number. The performance gap between consecutive operator and exclusive operator differs in different datasets, because the running time is dependent on the size of operands. We suspect this advantage will be enlarged for more complex queries, since there would be much less intermediate results generated for consecutive operator.

Finally, we study how number of operators affects the evaluation performance. We randomly generate the operands for each query and 10 queries for each operator type. The 10 queries are evaluated on dataset 2007 for 10 times, respectively. The average of the running times are shown in Figure 9.

Figure 9 shows that the running times of the three operators, consecutive, sequential, and exclusive, increases slowly as the number of operators grows. However, the parallel operator has exponentially growing running time based on the number of operators.

Fig. 8. Time of queries T2 on different datasets.

# Operators	Consecutive	Sequential	Exclusive	Parallel
1	2.71	3.54	3.45	4.97
2	3.52	7.89	4.16	20.49
3	4.24	8.32	4.99	36.78
4	4.62	12.48	5.66	614.04
5	5.24	25.97	8.22	5067.94

Fig. 9. Evaluation time (ms) of queries T2 on different number of operators on dataset 2007.

This observation is reasonable, because the operators consecutive and sequential have constraints on log sequence number, and exclusive operator has linear evaluation complexity, but the parallel operator does not filter candidates and has quadratic evaluation complexity. The exponential increase demonstrates Theorem 4.4.

6.2 Accuracy of Cost Estimation

When a query becomes complex, there is a big difference between different evaluation plans in terms of actual cost. In this section, we discuss the accuracy of cost estimation.

We evaluate the cost estimation accuracy on different query evaluation plans on the 2007 dataset. (The results for other datasets are quite similar.) Figure 10 shows the comparison between estimate cost and actual cost measured by normalized complexity. To show the general trend comparison between estimate and actual costs, we randomly generate four queries. Figure 10(a), (b), and (c) show the costs of evaluation plans for a query with five operators, respectively, the query used in Figure 10(d) has six operators. For each query, the evaluation plans are evenly placed on the x -axis ordered by the real cost incrementally.

Overall, Figure 10 shows that estimate costs have similar trend with actual costs in all queries studied. In Figure 10(a), estimates and actual costs overlap with tiny differences (the estimate cost is higher in the tail, i.e., when the actual cost is high). In Figure 10(b), both costs have the same general trend, though estimate cost starts lower than real cost. Figure 10(c) shows few places where estimate costs are lower than the actual ones. Figure 10(d) shows a gap between estimate and real costs at the tail. Note that there are significant jumps in the plan costs; this is due to 1 or 2 operator(s) whose placement(s) in the plan is(are) significant in affecting the overall plan cost.

The error margin of cost estimation is less desirable. However the error margin does not seem to reduce the effectiveness of our query optimization, because it can always choose the best evaluation plan. We discuss the effectiveness of query optimization in the next subsection. Overall,

Fig. 10. Estimated and actual costs have same trend on T3 query eval. Plans on dataset 2007.

although the cost estimation is not accurate for some evaluation plans, the cost estimation has the same trend as real cost. It means that the best plan with lowest actual cost would probably have lowest estimate cost. Thus the optimizer can utilize the estimate cost to select optimal evaluation plans.

6.3 Effectiveness of Query Optimization

We also conducted experiments to evaluate optimizer performance. We evaluated the optimizer on two aspects. First, we study how likely we can get a best plan with the optimizer. Then we show the impact of our optimizer on reducing the evaluation time for a query.

In our experiments, we randomly generated 150 queries with three, four, and five operators, respectively. We then used performance tolerance to loosen the constraint on “best” evaluation plan. Here, performance tolerance is defined as the ratio $\frac{CurRealCost - LowestRealCost}{LowestRealCost}$. If a query evaluation plan has an actual cost that is within performance tolerance, then we consider it as a best evaluation plan. For example, if performance tolerance is set to 0.1 and the lowest real cost is 10.0, then a query evaluation plan with real cost 10.5 is considered as a best evaluation plan. The optimizer accuracy is measured by the percentage getting the best evaluation plan with different performance tolerance for 150 queries with different number of operators. Figure 11 shows the results.

From Figure 11, given a performance tolerance, the accuracy for queries with three operators is the highest, and the accuracy for queries with 5 operators is the lowest. When the performance tolerance increases, the accuracy also increases. Note that when performance tolerance is 0, the

Fig. 11. Accuracy of optimizer on queries T3 on dataset 2007.

Fig. 12. Percentage of reduced evaluation time of queries T3 with optimization on dataset 2008.

average accuracy is 0.72; when performance tolerance is 0.01, the average accuracy reaches 0.88; when performance tolerance reaches 0.5, the accuracy rises to 0.93.

The accuracy for queries with lower number of operators has higher accuracy. This is because with more operators, the incident tree gets deeper and the estimation error propagates more. Also, Figure 11 suggests that setting performance tolerance to 0.2 would yield a high accuracy on average.

Finally, we analyze the effectiveness of optimizer on reducing query evaluation time. Figure 12 shows the box plot of reduced evaluation time for queries with different number of operators. We randomly generate 20 queries for each number of operators on trauma patient dataset 2008. Each query is evaluated 10 times without and with optimization, and the average evaluation time is treated as the evaluation time of a query. The effect of optimizer is evaluated with reduced percentage of evaluation time on each query, defined as the ratio $\frac{\text{TimeWithoutOptimizer} - \text{TimeWithOptimizer}}{\text{TimeWithoutOptimizer}}$ with each box for the 20 queries with corresponding number of operators.

Figure 12 shows up to 70% evaluation time reduction as a result of optimization. The largest percentage 98.06% is with the queries of four operators. And on average we have around 20% evaluation time reduced.

7 CONCLUSIONS

This article presents the design and implementation of the language IQL to query business process workflow log. IQL extends earlier work by allowing workflow models to be formulated and used in query conditions independent of the workflow models that guided the execution recorded in

the log. Such a flexibility is not only novel but also allows business analysts to formulate their “ad hoc” queries from high-level business perspectives. More importantly, querying logs occurs in many other fields including event processing, cyber-physical systems, IOT, and so on. The work reported in this article is an early step toward a promising approach for analytics in these fields. It is interesting to extend IQL to deal with executions across different workflow instances and to allow aggregates.

ACKNOWLEDGMENTS

The authors are grateful to reviewers who provided numerous comments and suggestions on earlier versions that led to significant improvements of the article.

REFERENCES

- [1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. 2005. The design of the borealis stream processing engine. In *Proceedings of the Conference on Innovative Data Systems Research*, Vol. 5. 277–289.
- [2] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. 2003. Aurora: A new model and architecture for data stream management. *Int. J. Very Large Data Bases* 12, 2 (2003), 120–139.
- [3] S.-M.-R. Beheshti, B. Benatallah, H. R. M. Nezhad, and S. Sakr. 2011. A query language for analyzing business processes execution. In *Proceedings of the International Conference Business Process Management (BPM’11)*. 281–297.
- [4] S.-M.-R. Beheshti, S. Sakr, B. Benatallah, and H. R. M. Nezhad. 2012. Extending SPARQL to support entity grouping and path queries. arXiv:1211.5817. Retrieved from <http://arxiv.org/abs/1211.5817>.
- [5] A. Berson and S. J. Smith. 1997. *Data Warehousing, Data Mining, and OLAP*. McGraw-Hill, Inc.
- [6] T. Bucher, A. Gericke, and S. Sigg. 2009. Process-centric business intelligence. *Bus. Process Manage. J.* 15, 3 (2009), 408–429.
- [7] D. Calvanese, G. De Giacomo, and M. Montali. 2013. Foundations of data-aware process analysis: A database theory perspective. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS’13)*. 1–12.
- [8] D. Carasso. 2012. *Exploring SPLUNK*. CITO Research.
- [9] M. Castellanos, K. A. de Medeiros, J. Mendling, B. Weber, and A. J. M. M. Weitjers. 2009. Business process intelligence. In *Handbook of Research on Business Process Modeling* (2009), 456–480.
- [10] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. 2003. TelegraphCQ: Continuous dataflow processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 668–668.
- [11] S. Chaudhuri and U. Dayal. 1997. An overview of data warehousing and OLAP technology. *SIGMOD Rec.* 26, 1 (1997), 65–74.
- [12] S. Chaudhuri, U. Dayal, and V. Narasayya. 2011. An overview of business intelligence technology. *Commun. ACM* 54, 8 (2011), 88–98.
- [13] J. Chomicki. 1994. Temporal query languages: A survey. In *Proceedings of the International Conference on Temporal Logic (ICTL’94)*, Lecture Notes in Computer Science, Vol. 827. Springer, 506–534.
- [14] E. Damaggio, A. Deutsch, R. Hull, and V. Vianu. 2011. Automatic verification of data-centric business processes. In *Proceedings of the International Conference on Business Process Management (BPM’11)*. 3–16.
- [15] U. Dayal, M. Castellanos, A. Simitsis, and K. Wilkinson. 2009. Data integration flows for business intelligence. In *Proceedings of the International Conference on Extending Database Technology (EDBT’09)*. 1–11.
- [16] B. Fazzinga, S. Flesca, F. Furfaro, E. Masciari, L. Pontieri, and C. Pulice. 2015. A framework supporting the analysis of process logs stored in either relational or NoSQL DBMSs. In *Proceedings of the 22nd International Symposium on Foundations of Intelligent Systems (ISMIS’15)* Lecture Notes in Computer Science, Vol. 9384. Springer, 52–58. https://doi.org/10.1007/978-3-319-25252-0_6
- [17] D. Grigori, F. Casati, M. Castellanos, U. Dayal, M. Sayal, and M.-C. Shan. 2004. Business process intelligence. *Comput. Industr.* 53, 3 (2004), 321–343.
- [18] Gartner Group. 2011. Gartner Newsroom. Retrieved from <http://www.gartner.com/it/page.jsp?id=1740414>.
- [19] Y. Mei and S. Madden. 2009. ZStream: A cost-based query processor for adaptively detecting composite events. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 193–206.
- [20] M. Perry, P. Jain, and A. P. Sheth. 2011. *SPARQL-ST: Extending SPARQL to Support Spatiotemporal Queries*. Springer, Chapter 3, 61–86.
- [21] A. Polyvyanyy, C. Ouyang, A. Barros, and W. M. P. van der Aalst. 2017. Process querying: Enabling business intelligence through query-based process analytics. *Decis. Supp. Syst.* 100 (2017), 41–56.

- [22] M. Räim, C. Di Ciccio, F. M. Maggi, M. Mecella, and J. Mendling. 2014. Log-based understanding of business processes through temporal logic query checking. In *On the Move to Meaningful Internet Systems: Proceedings of the OTM 2014 Conferences—Confederated International Conferences: CoopIS'14 and '14 Lecture Notes in Computer Science*, Vol. 8841. Springer, 75–92.
- [23] L. Reinkemeyer (Ed.). 2020. *Process Mining in Action*. Springer.
- [24] H. Scholta, M. Niemann, P. Delfmann, M. Räckers, and J. Becker. 2019. Semi-automatic inductive construction of reference process models that represent best practices in public administrations: A method. *Inf. Syst.* 84 (2019), 63–87.
- [25] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 23–34.
- [26] R. T. Snodgrass (Ed.). 1995. *The TSQL2 Temporal Query Language*. Springer.
- [27] Y. Tang. 2017. Workflow Log Query Evaluation. Retrieved from <https://github.com/yantangucsb/IncidentMS>.
- [28] Y. Tang, I. Mackey, and J. Su. 2018. Querying workflow logs. *Information* 9, 2 (2018).
- [29] Y. Tang and J. Su. 2017. Querying workflow logs. In *Proceedings of the IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW'17)*. 370–375.
- [30] W. M. P. van der Aalst. 2016. *Process Mining—Data Science in Action* (2nd ed.). Springer.
- [31] W. M. P. van der Aalst, B. F. van Dongen, C. W. Günther, A. Rozinat, E. Verbeek, and T. Weijters. 2009. ProM: The process mining toolkit. In *Proceedings of the Business Process Management Demonstration Track (BPMDemos'09)*.
- [32] W. van der Aalst et al. 2012. Process mining manifesto. In *Proceedings of the Business Process Management Workshops (BPM'11)*, Lecture Notes in Business Information Processing, F. Daniel, K. Barkaoui, and S. Dustdar (Eds.). Springer.
- [33] H. M. W. Verbeek, J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst. 2010. XES, XESame, and ProM 6. In *Information Systems Evolution—CAiSE Forum 2010*. 60–75.
- [34] W3C. 2016. XES Standard. Retrieved from <http://www.xes-standard.org/>.
- [35] R. Wrembel and C. Koncilia. 2007. *Data Warehouses and OLAP: Concepts, Architectures, and Solutions*. IGI Global.
- [36] K. Yongsiriwit, N. N. Chan, and W. Gaaloul. 2015. Log-based process fragment querying to support process design. In *Proceedings of the 48th Hawaii International Conference on System Sciences (HICSS'15)*. IEEE Computer Society, 4109–4119.

Received July 2019; revised July 2021; accepted August 2021