EdiFlow: data-intensive interactive workflows for visual analytics

Véronique Benzaken, Jean-Daniel Fekete, Wael Khemiri, Ioana Manolescu

EdiFlow: data-intensive interactive workflows for visual analytics

Véronique Benzaken[2] Jean-Daniel Fekete[1] Wael Khemiri[1,2]
Ioana Manolescu[1,2]

[1]INRIA Saclay–Île-de-France, firstname.lastname@inria.fr

[2]LRI, Université de Paris Sud-11, firstname.lastname@lri.fr

Abstract

La visualisation analytique (*visual analytics*) vise à combiner la visualisation des données avec des tâches d'analyse et de fouille. Etant donnée la complexité et la volumétrie importante des données scientifiques (par exemple, les données associées à des processus biologiques ou physiques, données des réseaux sociaux, etc.), la visualisation analytique est appelée à jouer un rôle important dans la gestion des données scientifiques.

La plupart des plates-formes de visualisation analytique actuelles utilisent des mécanismes mémoire pour le stockage et le traitement des données, ce qui limite le volume de données traitées. En outre, l'intégration de nouveaux algorithmes dans le processus de traitement nécessite du code d'intégration ad-hoc. Enfin, les plate-formes de visualisation actuelles ne permettent pas de définir et de déployer des processus structurés, où les utilisateurs partagent les données et, éventuellement, les visualisations.

Ce travail, issu d'une collaboration entre des chercheurs en visualisation analytique interactive et en bases de données, appporte deux contributions. (i) Nous proposons une architecture générique pour déployer une plate-forme de *visual analytics* au-dessus d'un système de gestion de bases de données (SGBD). (ii) Nous montrons comment propager les changements des données dans le SGBD, au travers des processus et des visualisations qui en font partie. Notre approche a été implantée dans un prototype appellé EdiFlow, et validée à travers plusieurs applications. Elle pourrait aussi s'intégrer dans une plate-forme de workflow scientifique à usage intensif de données, afin d'en augmenter les fonctionnalités de visualisation.

Mots-clé : visualisation analytique des données, workflow scientifique

1 Introduction

The increasing amounts of electronic data of all forms, produced by humans (e.g., Web pages, structured content such as Wikipedia or the blogosphere) and automatic tools (loggers, sensors, Web services, scientific programs or analysis tools) lead to a situation of unprecedented potential for extracting new knowledge, finding new correlations, and interpreting data. Visual analytics is a new branch of the information visualization / human-computer interaction field [17]. Its aim is to enable users to closely interact with vast amounts of data using visual tools. Thanks to these tools, a human may detect phenomena or trigger detailed analysis which may not have been identifiable by automated tools alone.

Though, most current visual analytics tools have some conceptual drawbacks. Indeed, they rarely rely on persistent databases (with the exception of [7]). Instead, the data is loaded from files or databases and is manipulated directly in memory because smooth visual interaction requires redisplaying the manipulated data 10-25 times per second. Standard database technologies do not support continuous queries at this rate; at the same time, ad-hoc in-memory handling of classical database tasks (e.g., querying, sorting) has obvious limitations. Based on our long-standing experience developing information visualisation tools [9] [1], we argue connecting a visual analysis tool to a persistent database management system (DBMS) has many benefits:

- Scalability: larger data volumes can be handled based on a persistent DBMS
- Persistence and distribution: several users (possibly on remote sites) can interact with a persistent database, whereas this is not easily achieved with memory-resident data structures. Observe that users may need to share not only raw data, but also visualizations built on top of this data. A visualization can be seen as an assignment of visual attributes (e.g., X and Y coordinates, color, size) to a given set of data items. Computing the value of the visual attributes may be expensive, and/or the choice of the visualized items may encapsulate human expertise. Therefore, visualizations have *high added value* and it must be easy to store and share them, e.g., allowing one user to modify a visualization that another user has produced.
- Data management capabilities provided by the database: complex data processing tasks can be coded in SQL and/or some imperative scripting language. Observe that such data processing tasks can also include user-defined functions (UDFs) for computations implemented outside the database server. These functions are not stored procedures managed by the database (e.g., Java Stored Procedure). These are executable programs external to the database.

The integration of a DBMS in a visualisation platform must take into account the following prevalent aspects in today's visual analytics applications:

- Convergence of visual analytics and workflow: current visual analytics tools are not based on workflow (process) models. This fits some applications where datasets and

tasks are always exploratory and different from one session to the next. Several visual analytics applications however, require a recurring process, well supported by a workflow system. The data processing tasks need to be organized in a sequence or in a loop; users with different roles may need to collaborate in some application before continuing the analysis. It also may be necessary to log and allow inspecting the advancement of each execution of the application. (Scientific) workflows platforms allow such automation of data processing tasks. They typically combine database-style processing (e.g., queries and updates) with the invocation of external functions, implementing complex domain-dependent computations. Well-known scientific workflow platforms include Kepler [3], or Trident [18]. These systems build on the experience of the data and workflow management communities; they could also benefit from a principled way of integrating powerful visualisation techniques.

- Handling dynamic data and change propagation: an important class of visual analytics applications has to deal with dynamic data, which is continuously updated (e.g., by receiving new additions) while the analysis process is running; conversely, processes (or visualisation) may update the data. The possible interactions between all these updates must be carefully thought out, in order to support efficient and flexible applications.

Our work addresses the questions raised by the integration of a DBMS in a visual analytics platform. Our contributions are the following:
1. We present a generic architecture for integrating a visual analytics tool and a DBMS. The integration is based on a core data model, providing support for (i) visualisations, (ii) declaratively-specified, automatically-deployed workflows, and (iii) incremental propagation of data updates through complex processes, based on a high-level specification. This model draws from the existing experience in managing data-intensive workflows [2, 6, 16].
2. We present a simple yet efficient protocol for swiftly propagating changes between the DBMS and the visual analytics application. This protocol is crucial for the architecture to be feasible. Indeed, the high latency of a "vanilla" DBMS connection is why today's visual analytics platforms do not already use DBMSs.
3. We have fully implemented our approach in a bare-bones prototype called EdiFlow, and de facto ported the InfoVis visual analytics toolkit [9] on top of a standard Oracle server. We validate the interest of our approach by means of three applications.

This article is organized as follows. Section 2 compares our approach with related works. Section 3 describes three applications encountered in different contexts, illustrating the problems addressed in this work. Section 4 presents our proposed data model, while the process model is described in Section 5. We describe our integration architecture in Section 6, discuss some aspects of its implementation in our EdiFlow platform, we then conclude in Section 8.

2 Related work

Significant research and development efforts have resulted in models and platforms for workflow specification and deployment. Recently, *scientific workflow* platforms have received significant attention. Different from regular (business-oriented) workflows, scientific workflows notably incorporate data analysis programs (or scientific computations more generally) as a native ingredient.

One of the first integration of scientific workflows with DBMSs was supported by [2]. Among the most recent and well-developed scientific workflow projects, Kepler [3] is designed to help scientists, analysts, and computer programmers to create, execute, and share models and analyses across a broad range of scientific and engineering disciplines. Kepler provides a GUI which helps users to select and then connect analytical components and data sources to create a scientific workflow. In this graphical representation, the nodes in the graph represent actors and the vertices are links between the actors.

SciRun [11] is a Problem Solving Environment, for modeling, simulation and visualization of scientific problems. It is designed to allow scientists to interactively control scientific simulations while the computation is running. SCIRun was originally targeted at computational medicine but has, later, been expanded to support other scientific domains. The SCIRun environment provides a visual interface for dataflow network's construction. As the system will allow parameters to be changed at runtime, experimentation is a key concept in SCIRun. As soon as a parameter is updated, at runtime, changes will propagated through the system and a re-evaluation induced.

GPFlow [15] is a workflow platform providing an intuitive web based environment for scientists. The workflow model is inspired by spreadsheets. The workflow environment ensures interactivity and isolation between the calculation components and the user interface. This enables workflows to be browsed, interacted with, left and returned to, as well as started and stopped.

VisTrails [4] combines features of both workflow systems and visualization fields. Its main feature is to efficiently manage exploratory activities. The user interaction in VisTrails is performed by iteratively refining computational tasks and formulating test hypotheses. VisTrails maintains detailed provenance of the exploration process. Users are able to return to previous versions of a dataflow and compare their results. However, VisTrails is not meant to manage dynamic data. In VisTrails, dynamicity is performed by allowing users to change some attributes in order to compare visualization results. It does not include any model to handle data changes. Indeed, when the user starts its workflow process, VisTrails does not take into account the updated data in activities that have already started: there is no guarantee that the model for updates is correct.

Trident [18] is a scientific workflow workbench built on top of a commercial workflow system. It is developed by Microsoft corporation to facilitate scientific workflows management. Provenance in Trident is ensured using a publication/subscription mechanism called the Blackboard. This mechanism allows also for reporting and visualizing intermediate data resulting from a running workflow. One of the salient features of Trident is to allow users to dynamically select where to store results (on SQL Server for example)

issued by a given workflow. However, it does not support dynamic data sources nor does it integrate mechanisms to handle such data.

Orchestra [13] addresses the challenge of mapping databases which have potentially different schemas and interfaces. Orchestra is specially focusing on bioinformatics applications. In this domain, one find many databases containing overlapping informations with different level of quality, accuracy and confidence. Database owners want to store a relevant (”alive”)version of relevant data. Biologists would like to download and maintain local ”live snapshots” of data to run their experiments. The Orchestra system focus on reconciliation across schemas. It is a fully peer-to-peer architecture in which each participant site specifies which data it trusts in. The system allows all the sites to be continuously updated, and on demand, it will propagate these updates across sites. User interaction in Orchestra is only defined at the first level using trust conditions. Moreover, the deployed mechanism is not reactive. Indeed, there is no restorative functions called after each insert/update operation.

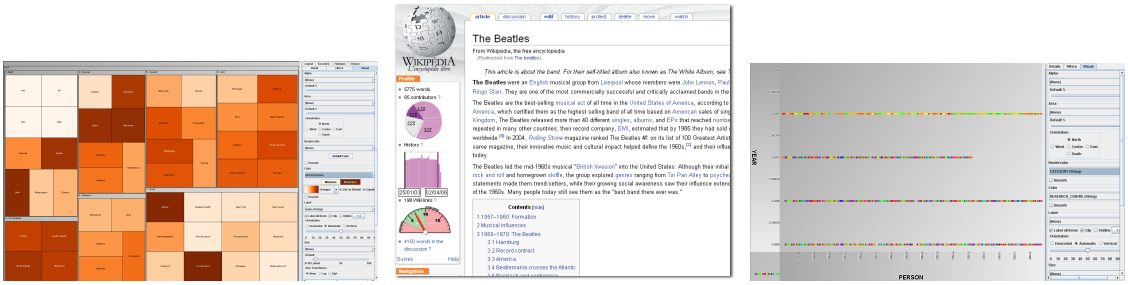
Several systems were conceived to create scientific workflows using a graphical interface and enabling data mining tasks (e.g., Knime [5], Weka [10]). However, none of these systems includes a repair mechanism to support the change in data sources during a task or process execution.

To summarize, all these platforms share some important features, which we also base our work on. Workflows are *declaratively specified, data-intensive* and (generally) *multi-user*. They include *querying and updating* data residing in some form of a database (or in less structured sources). Crucial for their role is the ability to invoke *external procedures*, viewed as black boxes from the workflow engine perspective. The procedures are implemented in languages such as C, C++, Matlab, Fortran. They perform important domain-dependent tasks; *procedures may take as input and/or produce as output large collections of data*. Finally, current scientific workflow platforms do provide, or can be coupled with, some *visualisation* tools, e.g., basic spreadsheet-based graphics, map tools.

With respect to these platforms, our work makes two contributions: (i) we show how a generic data visualisation toolkit can be integrated as a *first-class citizen*; (ii) we present a *principled way of managing updates* to the underlying sources, throughout the enactment of complex processes. This problem is raised by the high data dynamicity intrinsic to visual analytics applications. However, the scope of its potential applications is more general, as long-running scientific processes may have to handle data updates, too. *None of these platforms are currently able to propagate data changes to a running process*. The process model we propose could be integrated, with some modest programming effort, in such platforms, hence offering complementary functionalities to their existing ones.

Most of existing interactive platforms for data visualization [9] [1] focus on the interaction between the human expert and a data set consisting of a completely known set of values. They do not ease the inclusion of data analysis programs on the data. Moreover, as previously explained, they do not support the definition of structured processes, nor (by absence of an underlying DBMS) do they support persistence and sharing.

Unlike current data visualisation platforms, our work provides a useful coupling to DBMSs, providing persistent storage, scalability, and process support. Our goal is to



(a) US Elections (b) Wikipedia (c) INRIA scientists

Figure 1: Data visualizations examples.

drastically reduce the programming effort actually required by each new visual analytics application, while enabling them to scale up to very large data volumes. In this work, we present an architecture implementing a repair mechanism, to propagate data source changes to an executing process.

3 Use cases

The following applications illustrate the data processing and analysis tasks which this work seeks to simplify.

US Elections This application aims at providing a dynamic visualisation of elections outcome, varying as new election results become available. The database contains, for each state, information such as the party which won the State during the last three elections. On the voting day, the database gradually fills with new data. This very simple example uses a process of two activities: computing some aggregates over the votes, and visualizing the results. Upon starting, a TreeMap visualisation is computed over the database (distinguishing the areas where not enough data is available yet), as shown on the left up corner in Figure 1. The user can choose a party, then the 51 states are shown with varying color shades. The more the states vote for the respective party, the darker the color. When new vote results arrive, the corresponding aggregated values are recomputed, and the visualisation is automatically updated.

Wikipedia The goal of the application is to propose to Wikipedia readers and contributors some measures related to the history of an article. e.g., how many authors contributed to an article? How did a page evolve over time? The metrics are produced and visualized by the application, whereas the (current) Wikipedia page is displayed directly from the original site, as shown at the center in Figure 1. This application can be decomposed in four elementary tasks: (i) compute the differences between successive versions of each article; (ii) compute a contribution table, storing at each character index, the identifier of the user who entered it; (iii) for each article, compute the number of distinct effec-

tive contributors; and (iv) compute the total contribution (over all contribution tables) of each user. All these computations' results must be continuously updated to reflect the continuous changes in the underlying data. A total recomputation of the aggregation is out of reach, because change frequency is too high (10 edits per second on average for the French Wikipedia, containing about 1 million pages). Moreover, updates received at a given moment only affect a tiny part of the database. Thus, the Wikipedia application requires: a DBMS for storing huge amounts of data; a well-defined process model including ad-hoc procedures for computing the metrics of interest; incremental re-computations; and appropriate visualisations.

INRIA activity reports We have been involved in the development of an application seeking to compute a global view of INRIA researchers by analysing some statistics. The data are collected from Raweb (INRIA's legacy collection of activity reports available at http://ralyx.inria.fr). These data include informations about INRIA teams, scientists, publications and research centres. Our goal was to build a self-maintained application which, once deployed, would automatically and incrementally re-compute statistics, as needed. To that end, we first created a database out of all the reports for the years 2005 to 2008. Simple statistics were then be computed by means of SQL queries: age, team, research center distribution of INRIA's employees. They were further visualised as shown in Figure 1. Other aggregates were computed relying on external code such as the similarity between two people referenced in the reports in order to determine whether an employee is already present in the database or needs to be added.

All these applications feature data- and computation-centric processes which must react to data changes while they are running and need visual data exploration. The Wikipedia application is the most challenging, by the size of the database, the complexity of its metrics, and the high frequency of updates requiring recomputations.

4 Data model

In this Section, we describe our conceptual data model in Section 4.1, and its concrete implementation in a relational database in Section 4.2.

4.1 Conceptual data model

The conceptual data model of visual analytics application is depicted in Figure 2. For readability, the entities and relationships are organized in several groups.

The first group contains a set of entities capturing *process definitions*. A process consists of some activities. An activity must be performed by a different group of users (one can also see a group as a role to be played within the process). The process' control flow is not reflected in the data model, but in the separated process model (see Section 5). An activity instance has a start date and an end date, as well as a *status* flag which can take the values: *not_started* (the activity instance is created, e.g. by a user who assigns it to

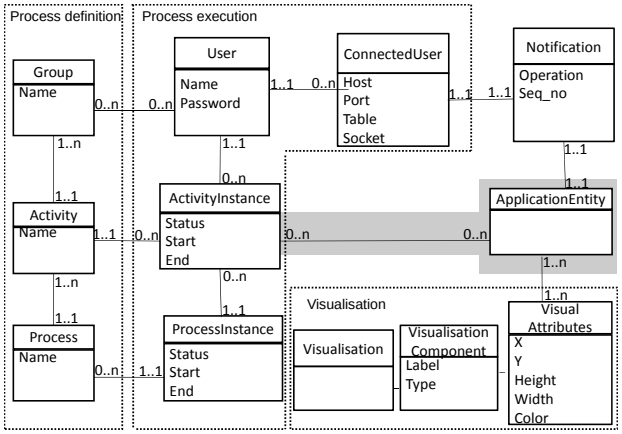


Figure 2: Entity-relationship data model for EdiFlow.

another for completion, but work on the activity has not started yet), *running* (the activity instance has started but it has not finished) and *completed* (once the activity instance has finished executing. The status of a process instance can take similar values.

Entities in the second group allow recording *process execution*. Individual users may belong to one or several groups. A user may perform some activity instances, and thus participate to specific process instances. A *ConnectedUser* pairs a user with the host and port from which the user connects at a given time. This information is needed to propagate to a potentially remote visualisation component (running on the remote user's desktop) updates received while the process is running, as will be discussed in Section 6.

The gray area can be seen as a meta-model, which has to be instantiated in any concrete application with one or several entities and relationships reflecting the application. For instance, in the Wikipedia application, one would use the entities *Article*, *User*, and *Version*, with relationships stating that each version of an article is produced by one update of the article by one user etc. Black-box functions, such as e.g. Wikipedia user clustering functions, must also be captured by this application-dependent part of the data model. In the most general case, one may wish to be able to trace the history of a given application data instance, e.g., identify the activity instance which created it, updated it etc. To that purpose, specific customized relationships of the form *createdBy*, *validatedBy* etc. can be defined in the conceptual model. They are represented in Figure 2 by the relationship on a gray background between *ApplicationEntity* and *ActivityInstance*.

The third group of entities is used to model visualization. A *Visualization* consists of one or more *VisualisationComponent*s. Each component offers an individual perspective over a set of entity instances. For example, in Figure 1(b), three visualisation components are shown in the bar at the left of the article, making up a given visualization associated to the article's edit history. Components of a same visualisation correspond to different ways of rendering the same objects. In each visualisation component, a specific set of *VisualAttributes* specify how each object should be rendered. Common visual attributes include (x, y) coordinates, width, height, color, label (a string), whether the data instance

is currently *selected* by a given visualization component (which typically triggers the recomputation of the other components to reflect the selection, etc).

Finally, the *Notification* entity is used to speedily propagate updates to the application entities in various places within a running process. A notification is associated to one or more instances of a particular application entity. It refers to an update performed at a specific moment indicated by the *seq_no* timestamp, and indicates the kind of the update (insert/delete/modify).

4.2 Concrete data model

We assume a simple *relational* enactment of this conceptual model. We have considered XML but settled for relations since performant visualisation algorithms are already based on a tabular model [9]. Thus, a relation is created for each entity, endowed with a primary key; relationships are captured by means of association tables with the usual foreign key mechanism. By issuing a query to the database, one can determine "which are the completed activity instances in process P", or "which is the R tuple currently selected by the user from the visualization component VC_1".

We distinguish two kinds of relations. DBMS-hosted relations are by definition persistent inside a database server and their content is still available after the completion of all processes. Such relations can be used in different instances, possibly of different processes. In contrast, temporary relations are memory-resident, local to a give process instance (their data is not visible and cannot be shared across process instances), and their lifespan is restricted to that of the process instance which uses them. If temporary relation data is to persist, it can be explicitly copied into persistent DBMS tables, as we explain shortly below.

5 Process model

We consider a process model inspired by the basic Workflow Management Coalition model. Figure 3 outlines (in a regular expression notation) the syntax of our processes. We use a set of variables, constants and attribute names N, a set of atomic values V, and a set of atomic data types T; terminal symbols used in the process structure are shown in boldface. The main innovative ingredient here is the treatment of *data dynamics*, i.e., the possibility to control which changes in the data are propagated to which part(s) of which process instances. We now describe the process model in detail.

Relations and queries. A process is built on top of a set of relations implementing the data model. Relations are denoted by capital letters such as R, S, T, possibly with subscripts. A query is a relational algebraic expression over the relations. We consider as operators: selection, projection, and cartesian product. Queries are typically designated by the letter Q possibly with subscripts.

Variables. A variable is a pair composed of a name, and of an (atomic) value. Variables come in handy for modelling useful constants, such as, for example, a numerical

Process	::=	Configuration Constant* Variable+ Relation+ Function* StructProcess
Configuration	::=	DBdriver DBuri DBuser
Constant	::=	name value name $\in N$, value $\in V$
Variable	::=	name type name $\in N$, type $\in T$
Relation	::=	name primaryKey RelType
RelationType	::=	(attName attType)*, attName $\in N$, attType $\in T$
Function	:=	name classPath
StructuredProcess	:=	Activity | Sequence | AndSplitJoin | OrSplitJoin | ConditionalProcess
Sequence	::=	Activity **,** StructuredProcess
AndSplitJoin	::=	**AND-split** (StructuredProcess)+ **AND-join**
OrSplitJoin	::=	**OR-split** (StructuredProcess)+ **OR-join**
ConditionalProcess	::=	**IF** Condition StructuredProcess
Activity	::=	activityName Expression
Expression	::=	askUser | callFunction | runQuery

Figure 3: XML schema for the process model.

threshold for a clustering algorithm. Variables will be denoted by lower-case letters such as v, x, y.

Procedures. A procedure is a computation unit implemented by some external, black-box software. A typical example is the code computing values of the visual attributes to be used in a visualisation component. Other examples include e.g., clustering algorithms, statistical analysis tools.

A procedure takes as input l relations $R_1, R_2, \ldots, R_l$ which are read but not changed and m relations $T_1^w, T_2^w, \ldots, T_m^w$ which the procedure may read *and* change, and outputs data in n relations:

$$p : R_1, R_2, \ldots, R_l, T_1^w, T_2^w, \ldots, T_m^w \rightarrow S_1, S_2, \ldots, S_n$$

We consider p as a black box, corresponding to software developed outside the database engine, and outside of EdiFlow by means of some program expressed e.g., in C++, Java, MatLab. Functions are processes with no side effects ($m = 0$).

Delta handlers. Associated to a procedure may be *procedure delta handlers*. Given some update (or delta) to a procedure input relation, the delta handler associated to the procedure may be invoked to propagate the update to a process. Two cases can be envisioned:

1. Update propagation is needed while the procedure is being executed. Such is the case for instance of procedures which compute point coordinates on a screen, and must update the display to reflect the new data.

2. Updates must be propagated after the procedure has finished executing. This is the case for instance when the procedure performs some quantitative analysis of which only the final result matters, and such that it can be adjusted subsequently to take into account the deltas.

The designer can specify one or both of these handlers. Formally, each handler is a procedure in itself, with a table signature identical to the main procedure. The convention is that if there are deltas only for some of p's inputs, the handler will be invoked providing

empty relations for the other inputs. With respect to notations, $p_{h,r}$ is the handler of p to be used while p is *r*unning, and $p_{h,f}$ is the handler to be used after p *f*inished. Just like other procedures, the implementation of handlers is opaque to the process execution framework. This framework, however, allows one to recuperate the result of a handler invocation and inject it further into the process, as we shall see.

Distributive procedures. An interesting family of procedures are those which distribute over union in all their inputs. More formally, let X be one of the R_i inputs of p, and let ΔX be the set of tuples added to X. If p is distributive then:

$p(R_1, \ldots, X \cup \Delta X, \ldots, T_m^w)$=$p(R_1, \ldots, X, \ldots, T_m^w) \cup p(R_1, \ldots, \Delta X, \ldots, T_m^w)$

There is no need to specify delta handlers for procedures which distribute over the union, since the procedure itself can serve as handler.

Expressions. We use a simple language for expressions, based on queries and procedures. More formally:

$$e ::= Q \mid p(e_1, e_2, \ldots, e_n, T_1^w, T_2^w, \ldots, T_p^w).t_j, 1 \leq j \leq m$$

The simplest expressions are queries. More complex expressions can be obtained by calling a procedure p, and retaining only its j-th output table. If p changes some of its input table, evaluating the expression may have side effects. If the side effects are not desired, p can be invoked by giving it some new empty tables, which can be memory-resident, and will be silently discarded at the end of the process. Observe that the first n invocation parameters are expressions themselves. This allows nesting complex expressions.

Activities. We are now ready to explain the building blocks of our processes, namely activities.

$$a ::= v \leftarrow \alpha \mid upd(R) \mid \quad (S_1, S_2, \ldots, S_n) \leftarrow p(e_1, e_2, \ldots, e_n, T_1^w, T_2^w, \ldots, T_n^w)$$

Among the simplest activities are *variable assignments* of the form $v \leftarrow \alpha$. Another simple activity is a declarative update of a table R, denoted $upd(R)$. Unlike the table modifications that an opaque procedure may apply, these updates are specified by a declarative SQL statement. Finally, an activity may consist of invoking a procedure p by providing appropriate input parameters, and retaining the outputs in a set of tables.

Visualisation activities must be modelled as procedures, given that their code cannot be expressed by queries.

Processes. A process description can be modelled by the following grammar:

$$P ::= \epsilon \mid a, P \mid P \| P \mid P \vee P \mid e?P$$

In the above, a stands for an activity. A process is either the empty process (ϵ), or a *sequence* of an activity followed by a process (,), or a *parallel (and) split-join* of two processes ($\|$), or an *or split-join* of two processes (with the semantics that once a branch is triggered, the other is invalidated and can no longer be triggered). Finally, a process can consist of a *conditional block* where an expression e (details below) is evaluated and if this yields true, the corresponding process is executed.

Reactive processes. A *reactive process* can now be defined as a 5-tuple consisting of a set of relations, a set of variables, a set of procedures, a process and a set of *update propagations*. More formally:

$$RP ::= R^*, v^*, p^*, P, UP^*$$

An *update propagation* UP specifies what should be done when a set of tuples, denoted ΔR, are added to an application-dependent relation R, say, at $t_{\Delta R}$. Several options are possible. We discuss them in turn, and illustrate with examples.

1. Ignore ΔR for the execution of all processes which had started executing before $t_{\Delta R}$. The data will be added to R, but will only be visible for *process instances having started after* $t_{\Delta R}$. This recalls locking at process instance granularity, where each process operates on exactly the data which was available when the process started. We consider this to be the default behaviour for all updates to the relations part of the application data model.

Use case: A social scientist applies a sequence of semi-automated partitioning and clustering steps to a set of Wikipedia pages. Then, the scientist visualises the outcome. In this case, propagating new items to the visualisation would be disruptive to the user, which would have to interrupt her current work to help apply the previous steps to the new data.

2. Ignore ΔR for the execution of all *activities* which had started executing (whether they are finished or not) before $t_{\Delta R}$. However, *for a process already started, instances of a specific activity which start after* $t_{\Delta R}$ may also use this data.

Use case: The social scientist working on a Wikipedia fragment first has to confirm personal information, give some search criteria for the pages to be used in this process. Then, she must interact with a visualisation of the chosen pages. For this activity, it is desirable to provide the user with the freshest possible snapshot, therefore additions between the beginning of the process instance, and the moment when the user starts the last activity, should be propagated.

3. As a macro over the previous option and the process structure, one could wish for ΔR to be propagated to *instances of all activities that are yet to be started in a running process*.

Use case: Intuitively, data should not "disappear" during the execution of a process instance (unless explicitly deleted). In the previous use case, if we add an extra activity at the end of the process, that activity would typically expect to see the whole result of the previous one.

4. Propagate the update ΔR to *all the terminated instances of a given activity*. We can moreover specialize the behavior on whether we consider only activity instances whose *process instances have terminated*, only activity instances whose *process instances are still running*, or both.

Use case: We consider a process whose first activities are automatic processing steps, e.g., computing *diff*s between the old and the new version of a Wikipedia page, updating a user's contribution, the page history etc. The last activity is a visualisation one where the scientist should be shown fresh data. Typically, the visualisation activity will last

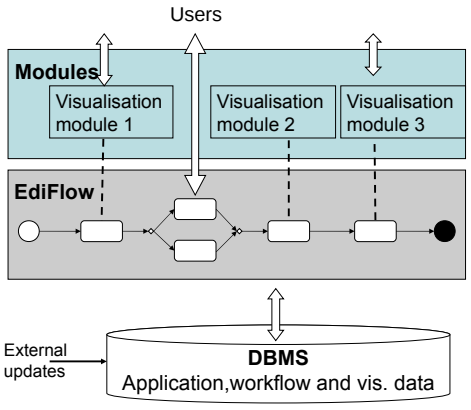


Figure 4: EdiFlow architecture.

for a while, and it may refresh itself at intervals, to reflect the new data. In this case, it makes sense to apply the automated processing activities to the new pages received while running the process instance, even after the respective activities have finished.

5. Propagate the update ΔR to all the running instances of a given activity, whether they had started before $t_{\Delta R}$ or not.

Use case: This may be used to propagate newly arrived tuples to all running instances of a visualisation activity, to keep them up-to-date.

Formally then, an update propagation action can be described as:

$$UP ::= R, a, (('\text{ta}', ('\text{rp}'|'\text{tp}')) \mid '\text{ra}' \mid ('\text{fa}', '\text{rp}'))$$

where R is a relation and a is an activity. An update propagation action describes a set of instances of activity a, to which the update ΔR must be propagated. The possible combinations of terminal symbols designate:

[ta rp:] *t*erminated *a*ctivity instances part of *r*unning processes;
[ta tp:] *t*erminated *a*ctivity instances part of *t*erminated processes;
[ra:] *r*unning *a*ctivity instances (obviously, part of running processes);
[fa rp:] *f*uture *a*ctivity instances part of *r*unning *p*rocesses.

It is possible to specify more than one compensation action for a given R and a given activity a. For instance, one may write: $(R, a,$ 'ra'), $(R, a,$ 'fa', 'rp').

For simplicity, the syntax above does not model the macro possibility numbered 3 in our list of options. One can easily imagine a syntax which will then be compiled into UP's as above, based on the structure of P.

6 An architecture for reactive processes

Our proposed architecture is depicted in Figure 4. The workflow management logic runs on top of the DBMS; visualisation software is integrated under the form of modules. Pro-

cesses are specified in a high-level syntax following the structure described in Section 5. The enactment of a process thus specified consists of adding the necessary tuples to the Process and Activity relations. During process executions, the necessary data manipulation statements are issued to (i) record in the database the advancement of process and activity instances, (ii) evaluate on the database queries and updates, allow external procedures to read and update the application-driven entities and (iii) record the connections between users and application instances, and application data.

In the sequel, Section 6.1 shows how to implement various degrees of isolation between concurrent processes operating on top of the same database. Section 6.2 outlines update propagation. Section 6.3 considers an important performance issue: efficient synchronization between memory-resident tables, that visualisation uses, and disk-resident tables.

6.1 Isolation

Applications may require different levels of sharing (or, conversely, of isolation) among concurrent activities and processes.

Process- and activity-based isolation Let a_1 be an instance of activity a, such that a_1 is part of a process instance p_1. By default, queries evaluated during the execution of p carry over the whole relations implementing the application-specific data model. Let R be such a relation.

It may be the case that a_1 should only see the R tuples created as part of executing p_1. For instance, when uploading an experimental data set, a scientist only sees the data concerned by that upload, not the data previously uploaded by her and others. Such isolation is easily enforced using relationships between the application relations and the *ActivityInstance* table (recall Figure 2 in Section 4). A query fetching data from R for a_1 should select only the R tuples created by p_1, the process to which a_1 belongs, etc. These mechanisms are fairly standard.

Time-based isolation As discussed in Section 5, the data visible to a given activity or process instance may depend on the starting time of that instance. To enable such comparisons, we associate to each application table R a *creation timestamp*, which is the moment when each R tuple entered the database (due to some process or external update). R tuples can then be filtered by their creation date.

Isolating process instances from tuple deletions requires a different solution. If the process instance p_3 erases some tuples from R, one may want to prevent the deleted tuples from suddenly disappearing from the view of another running process instance, say p_4. To prevent this, tuples are not actually deleted from R until the end of p_3's execution. We denote that moment by $p_3.end$. Rather, the tuples are added to a *deletion table* $R_{\_}$. This table holds tuples of the form $(tid, t_{del}, pid, \perp)$, where tid is the deleted R tuple identifier, t_{del} the deletion timestamp, pid the identifier of the process deleting the tuple. The fourth attribute will take the value $p_3.end$ at the end of p_3. To allow p_3 to run as if the deletion occurs, EdiFlow rewrites queries of the form select * from R implementing activities of p_3 with:

select * from R where tid not in (select tid from R$_-$ where pid=p_3)

When p_3 terminates, if no other running process instance uses table R[1], then we delete from R and R_- the tuples $\sigma_{pid=p_3}(R_-)$. Otherwise, R and R_- are left unchanged, waiting for the other R users to finish. However, a process instance started after $t_0 > p_3.end$ should not see tuples in R_- deleted by p_3, nor by any other process whose end timestamp is smaller than t_0. In such a recently-started process, a query of the form select * from R is rewritten by EdiFlow as:

select * from R where tid not in (select tid from R$_-$ where processend$< t_0$)

We still have to ensure that deleted tuples are indeed eventually deleted. After the check performed at the end of p_3, Ediflow knows that some deletions are waiting, in R_-, for the end of a process instances started before $p_3.end$. We denote these process instances by $wait_{R,p_3}$. After $p_3.end$, whenever a process in $wait_{R,p_3}$ terminates, we eliminate it from $wait_{R,p_3}$. When the set is empty, the tuples $\sigma_{pid=p_3}(R_-)$ are deleted from R and R_-.

6.2 Update propagation

We now discuss the implementation of the update propagation actions described in Section 5. EdiFlow compiles the UP (update propagation) statements into statement-level triggers which it installs in the underlying DBMS. The trigger calls EdiFlow routines implementing the desired behavior, depending on the type of the activity (Section 5), as follows. Variable assignments are unaffected by updates. Propagating an update ΔR_i to relation R_i to a query expression leads to incrementally updating the query, using well-known incremental view maintenance algorithms [12]. Propagating an update to an activity involving a procedure call requires first, updating the input expressions, and then, calling the corresponding delta handler.

6.3 Synchronizing disk-resident and in-memory tables

The mechanisms described above propagate changes to (queries or expression over) tables residing in the SQL DBMS. However, the visualisation software running within an instance of a visualisation activity needs to maintain portions of a table in memory, to refresh the visualisation fast. A protocol is then needed to efficiently propagate updates made to a disk-resident table, call it R_D, to its possibly partial memory image, call it R_M. Conversely, when the visualisation allows the user to modify R_M, these changes must be propagated back to R_D. Observe that R_M exists on the client side and therefore may be on a different host than R_D.

To that end, we install CREATE, UPDATE and DELETE triggers monitoring changes to the persistent table R_D. Whenever one such change happens, the corresponding trigger adds to the Notification table stored in the database (recall the data model in Figure 2) one

[1]The definition of a process explicitly lists the tables it uses, and from the process, one may retrieve the process instances and check their status (Figure 2).

tuple of the form (seq_no, ts, tn, op), where seq_no is a sequential number, ts is the update timestamp, tn is the table name and op is the operation performed. Then, a notification is sent to R_M that "there is an update". Smooth interaction with a visualization component requires that notifications be processed very fast, therefore we keep them very compact and transmit no more information than the above. A notification is sent via a socket connected to the process instance holding R_M. Information about the host and port where this process runs can be found in the Client table (Figure 2). When the visualisation software decides to process the updates, it reads them from the Notification table, starting from its last read seq_no value.

The synchronization protocol between R_M and R_D can be summarized as:

1. A memory object is created in the memory of the Java process (R_M);
2. It asks a connection manager to create a connection with the database;
3. The connection manager creates a network port on the local machine and associates locally a quadruplet to R_M: $(db, R_D, ip, port)$;
4. The quadruplet is sent to the DB to create an entry in its ConnectedUser table;
5. The DB connects to the client using the IP/PORT and expects a $hello$ message to check that it is the right protocol;
6. The connection manager accepts the connection, sends the $hello$ message and expects a REPLY message to check that it is the expected protocol too;
7. When the R_D is modified, the database trigger sends a $notify$ message with the table name as parameter to the R_M (through the $ip/port$ address);
8. When it receives the notification, the visualisation software holding R_M connects to the SQL server and queries the created/updated/deleted list of rows. It can update them at any time it wants;
9. When R_M is modified, it propagates its changes to the R_D and processes the triggered notifications in a smart way to avoid redundant work;
10. When R_M is deleted, it sends a disconnect message to the database that closes the socket connection and removes the entry in the ConnectedUser table;
11. The Notification table can be purged of entries having seq_no lower than the lowest value in the Client table.

6.4 EdiFlow tool implementation

EdiFlow is implemented in Java, and currently we have deployed it on top of both Oracle 11g and MySQL 5. EdiFlow processes are specified in a simple XML syntax, closely resembling the XML WfMC syntax XPDL.

Procedures are implemented as Java modules using the Equinox implementation of the OSGi Service Platform. A procedure instance is a concrete class implementing the EdiflowProcess interface. This interface requires four methods: initialize(), run(ProcessEnv env), update(ProcessEnv env) and String getName(). The class ProcessEnv represents a procedure environment, including all useful information about the environment in which the processes are executed. An instance of ProcessEnv is passed as a parameter to a newly created instance of a procedure. Integrating a new processing algorithm into the platform requires only implementing one procedure class, and serving the calls to the methods. All the dependencies in term of libraries (JAR files) are managed by the OSGi Platform.

The implementation is very robust, well documented, efficient in term of memory footprint and lightweight for programming modules and for deploying them, which is important for our goal of sharing modules. We have implemented and ran the sample applications described in Section 3.

7 Experimental validation

In this Section, we report on the performance of the EdiFlow platform in real applications.
Hardware. Our measures used a client PC with Intel 2.66GHz Dual Core CPUs and 4GB memory running. Java heap size was set to 850MB. The Oracle database is mounted on a workstation with 8 CPUs equipped with 8GB RAM. The PC is connected to the database through the local area network.
Dataset. We used a dataset of co-publications between INRIA researchers. We analyse this data set to produce visual results which have interesting insight for the INRIA scientific managers, and has to proceed while new publications are added to the database. This dataset includes about 4500 nodes and 35400 edges. The goal is to compute the attributes of each node and edge, display the graph over one or several screens and update it as the underlying data changes.

7.1 Layout procedure handlers

Our first goal was to validate the interest of procedure handlers in the context of data visualization. In our INRIA co-publication scenario, the procedure of interest is the one computing the positions of nodes in a network, commonly known as *layout*. We use the Edge LinLog algorithm of Noack [14] which is among the very best for social networks, and provides aesthetically good results. What makes EdgeLinLog even more interesting in our context is that it allows for effective delta handlers (introduced as part of our process model in Section 5).

In our implementation, the initial computation assigns a random position to each node and runs the algorithm iteratively until it converges to a minimum energy and stabilizes. This computation can take several minutes to converge but, since the positions are computed continuously, we can store the positions in the database at any rate until the algorithm stops. Saving the positions every second or at every iteration if it takes more than

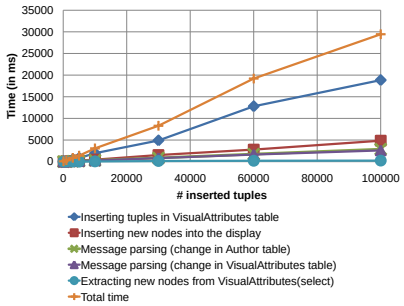


Figure 5: Time to perform insert operations.

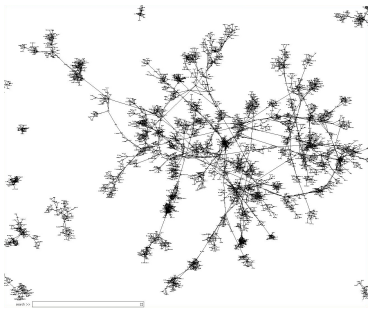


Figure 6: Part of the graph of INRIA co-publications.

one second allows the system to appear reactive instead of waiting for minutes before showing anything.

If the network database changes, for example when new publications are added to/removed from the database, the handler proceeds in a slightly different manner. First, it updates the in-memory co-publication graph, discards the nodes that have been removed and adds new nodes. To each new node it assigns a position that is close to their neighbours that have already been laid-out. This is to improve the time and quality of the final layout. If disconnected nodes are added, they are assigned a random position. Then, the algorithm is run iteratively like for the initial computation, but it terminates much faster since most of the nodes will only move slightly: the convergence of the iterations will be much faster. Like before, we store in the DBMS the results of some of the iterations to allow the visualization views to show them.

Using this strategy, we have obtained an incremental layout computation, remarkably stable and fast.

7.2 Robustness evaluation

Our second experimental goal was to study how the EdiFlow event processing chain scales when confronted with changes in the data. For this experiment, the DBMS is connected via a 100 MHz Ethernet connection to two EdiFlow instances running on two machines. The first EdiFlow machine computes visual attributes (runs the layout procedure), while the second extracts nodes from VisualAttributes table and displays the graph. This second EdiFlow machine is a laptop.

We study the robustness of our architecture when adding increasing numbers of tuples to the database. Inserting tuples requires performing the sequence of steps below, out of which steps 1, 2 are performed on the first EdiFlow machine, while steps 3, 4 and 5 are performed on all machines displaying the graph.

1. Parsing the message involved after insertion in nodes table. It refers to step 7 in the protocol described in section 6.3.
2. Inserting the resulting tuples in the VisualAttributes table managed by EdiFlow in the

DBMS.
3. Parsing the message involved after insertion in VisualAttributes table. After inserting tuples, in VisualAttributes, a message is sent to all machines displaying the graph. The message is parsed to extract the new tuple information. It refers to step 9 in the protocol described in section 6.3.
4. Extracting the visual attributes of the new nodes, from the VisualAttributes table, in order to know how to display them at the client.
5. Inserting new nodes into the display screen of the second machine.
The times we measured for these five steps are shown in Figure 5 for different numbers of inserted data tuples. The Figure demonstrates that the times are compatible with the requirements of interaction, and grow linearly with the size of the inserted data. The dominating time is required to write in the VisualAttributes table. This is the price to pay for having these attributes stored in a place from where one can share them or distribute them across several displays.

8 Conclusion

In this article, we have described the design and implementation of EdiFlow, the first workflow platform aimed at capturing changes in data sources and launching a repair mechanism. EdiFlow unifies the data model used by all of its components: application data, process structure, process instance information and visualization data. It relies on a standard DBMS to realize that model in a sound and predictive way. EdiFlow supports standard data manipulations through procedures and introduces the management of changes in the data through *update propagation*. Each workflow process can express its behaviour w.r.t data change in one of its input relations. Several options are offered to react to such a change in a flexible way.

EdiFlow reactivity to changes is necessary when a human is in the loop and needs to be informed of changes in the data in a timely fashion. Furthermore, when connected to an interactive application such as a monitoring visualization, the human can interactively perform a command that will change the database and trigger an update propagation in the workflow, thus realizing an interactively driven workflow.

We are currently using EdiFlow to drive our Wikipedia aggregation and analysis database as a testbed to provide real-time high-level monitoring information on Wikipedia, in the form of visualizations or textual data [8]. We are also designing a system for computing and maintaining a map of scientific collaborations and themes available on our institutions.

We still need to experiment with it to find out the limitations of EdiFlow in term of performances, typical and optimal reaction time and ability to scale with very large applications.

We strongly believe that formally specifying the services required for visual analytics in term of user requirements, data management and processing, and providing a robust implementation is the right path to develop the fields of visual analytics and scientific workflows together. For more details, examples, pictures and videos of the usage of

EdiFlow, see the EdiFlow website: `http://scidam.gforge.inria.fr/`.

References

[1] Protovis. http://vis.stanford.edu/protovis/.

[2] A. Ailamaki, Y. E. Ioannidis, and M. Livny. Scientific workflow management by database management. In *SSDBM*, 1998.

[3] I. Altintas, C. Berkley, E.Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: An extensible system for design and execution of scientific workflows. In *SSDBM*, 2004.

[4] E.W. Anderson, S.P. Callahan, D.A. Koop, E. Santos, C.E. Scheidegger, H.T. Vo, J. Freire, and C.T. Silva. VisTrails: Using provenance to streamline data exploration. In *DILS*, 2007.

[5] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Kilian Thiel, and Bernd Wiswedel. KNIME - the Konstanz information miner: version 2.0 and beyond. *SIGKDD Explorations*, 11(1), 2009.

[6] Marco Brambilla, Stefano Ceri, Piero Fraternali, and Ioana Manolescu. Process modeling in web applications. *ACM Trans. Softw. Eng. Methodol.*, 2006.

[7] S-M. Chan, L. Xiao, J. Gerth, and P. Hanrahan. Maintaining interactivity while exploring massive time series. In *VAST*, 2008.

[8] F. Chevalier, S. Huot, and J-D. Fekete. Wikipediaviz: Conveying article quality for casual wikipedia readers. In *PaciViz*, 2010.

[9] J-D. Fekete. The InfoVis toolkit. In *InfoVis*, 2004.

[10] Eibe Frank, Mark A. Hall, Geoffrey Holmes, Richard Kirkby, and Bernhard Pfahringer. Weka - a machine learning workbench for data mining. In *The Data Mining and Knowledge Discovery Handbook*. 2005.

[11] S. G.Parker and C. R.Johnson. SCIRun: a scientific programming environment for computational steering. In *ACM SC*, 1995.

[12] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *ACM SIGMOD*, 1993.

[13] Zachary G. Ives, Todd J. Green, Grigoris Karvounarakis, Nicholas E. Taylor, Val Tannen, Partha Pratim Talukdar, Marie Jacob, and Fernando Pereira. The ORCHESTRA collaborative data sharing system. *SIGMOD Record*, (3), 2008.

[14] Andreas Noack. An energy model for visual graph clustering. In *Proceedings of the 11th International Symposium on Graph Drawing*, volume 2912 of *LNCS*, 2004.

[15] A. Rygg, P. Roe, and O. Wong. GPFlow: An intuitive environment for web based scientific workflow. In *Proceedings of the 5th International Conference on Grid and Cooperative Computing Workshops*, 2006.

[16] S. Shankar, A. Kini, D.J. DeWitt, and J. Naughton. Integrating databases and workflow systems. *SIGMOD Record*, 2005.

[17] J. Thomas and K. Cook, editors. *Illuminating the Path: Research and Development Agenda for Visual Analytics*. IEEE Press, 2005.

[18] Project Trident: A scientific workflow workbench. http://research.microsoft.com/en-us/collaboration/tools/trident.aspx.