Business Process Implementation through an SQL Extension: WED-SQL

Bruno Padilha[1], **André Luis Schwerz**[2], **Rafael Liberato Roberto**[2], **Calton Pu**[3]

[1]IME – University of São Paulo (USP)
São Paulo, SP, Brazil

[2]Universidade Tecnológica Federal do Paraná (UTFPR)
Campo Mourão, PR, Brazil

[3]CERCS, Georgia Institute of Technology
Atlanta, GA, USA

brunopadilha@usp.br,{andreluis,liberato}@utfpr.edu.br, calton@cc.gatech.edu

Abstract. *Despite the significant evolution of the design and implementation of business process models, a transactional approach that evolves an incremental and adaptive strategy remains an important challenge to be overcome for database management systems. Traditional frameworks such as BPEL, Process Algebra and Petri Networks provide an extra software layer or use some external toolkit, or even a framework, to be able to enforce a data-state based transaction control, what includes dealing with semantic exceptions. However, the complexity of implementation based on these traditional frameworks especially to treat exceptions is too high. In this paper, we present the WEDSQL, a distributed framework that provides a reliable e efficient way to write and implement business process applications. Our main contribution is the integration of WED-flow concepts with the Relational Database Management System (PostgreSQL). This integration enables the WED-SQL to take full advantage of all the transactional properties provided by PostgreSQL and also benefit from the SQL language to specify the WED-flow definitions (e.g. WED-conditions, WED-transitions, and so on).*

1. Introduction

Most of modern business process management software systems are subject to recurrent structural modifications during its life cycle. This changes are necessary to incorporate new requisites to the software, once it is often impossible to come up with all requirements upfront in the modeling phase of project - some of then only appear after the software is running in production. Furthermore, it is too expensive and time consuming to try to predict all the possibles situations and its outcomes, which ultimately may raise exceptions that must be dealt with. Over time, these modifications leads to code deterioration, exponentially increasing the maintenance costs and compromising the quality of the system as well negatively impacting its overall performance.

Classic business process specification models, for example WS-BPEL, Process Algebra and Petri Networks, focus primarily on behavioral interactions and relations between process, pushing data analysis into the background and overlooking its relevance for the project. Therefore, none of these models exactly fits the needs of design dynamic

business process management systems, which may result in costly system adaptations and maintenance.

The WED-flow approach [2], in contrast with the classical models, capture both the inter-process relations and, with the very same relevance, the data generated by their interactions. With this in mind, the WED-flow allows systems architects to think in terms of data states and transitions, similarly to a finite automata designing. Modifying or adding new business rules to the project, which is often translated in creation of new processes, is a simple matter of defining new data states and conditions for transitions, thus capturing the true nature of dynamic processes and simplifying an incremental evolution of the whole project. Real time exceptions are also easier to be handled and integrated to the project, once this approach is based on transactional properties, thus providing several recovery mechanisms.

In order to provide a standard implementation as well as simplifying its software development, in this work we present the WED-SQL, an WED-flow framework. The main purpose of WED-SQL is to implement all WED-flow definitions [1] and encapsulate the control structure, what includes the transactional control, exceptions handling, data states detection, state transitioning and ensure data integrity. This framework effectively is an abstraction layer that allow developers to put more effort on business logic and come up with better WED-flow projects, also making better use of reusable code.

The WED-SQL framework has been built inside the PostgreSQL [7] relational database management system (PostgreSQL RDBMS) , thereby uses SQL language to specify WED-flow elements, has full transactional support and it is highly fault tolerant. Once it employs well consolidate technologies, widespread in the computer science field, this framework targets to disseminate the adoption of WED-flow on business process management software design, by means of an easy to use, reliable and scalable tool that is capable to provide the flexibility demanded by modern transaction control systems.

The remaining of this paper is structured as follows: Section II presents WED-SQL general software architecture aspects and internal data structures. Section III briefly describes how the system works. A discussion about the algorithms involved is presented in section IV. Following, a detailed description of transactional control and the communication protocol used for external communication is given in section V. Finally, on section VI, we discuss the next challenges to improve and expand this framework capabilities.

2. WED-SQL: general view and architecture

To better understand what the WED-SQL does, first one needs to understand the basic definitions of the WED-flow paradigm. A WED-flow application is composed of a set of WED-attributes $\mathcal{A} = \{a_1, a_2, a_3, \ldots, a_n\}$, a set of WED-states $\mathcal{S} = \{s_1, s_2, s_3, \ldots, s_m\}$ where $\forall s_i \in \mathcal{S}, s_i = (v_1, v_2, v_3, \ldots, v_n)$ is a n-tuple where each $i \in [1, n], v_i$ is a value for $a_i \in \mathcal{A}$ and the size of $s_i = |\mathcal{A}|$, a set of WED-conditions $\mathcal{C}$, a set of WED-transitions $\mathcal{T}$ and a set of WED-triggers $\mathcal{G}$. A WED-condition $c \in \mathcal{C}$ is a predicate over $\mathcal{S}$, that is, we say that $s \in \mathcal{S}$ satisfies c if the values of s makes c true. A WED-transition is a function $t : \mathcal{S} \rightarrow \mathcal{S} \in \mathcal{T}$ that receives as input a WED-state s and returns another WED-state s'. Finally, a WED-trigger $g = (c, t)$ where $g \in \mathcal{G}, c \in \mathcal{C}, t \in \mathcal{T}$ is a 2-tuple that associates a WED-condition c with a WED-transition t. Therefore, an instance of a WED-flow application stars with a initial WED-state s that should match some WED-condition c to

fire the WED-trigger g that, in turn, will initiate a WED-transition t, which must complete by some predefined timeout,thus generating the next WED-state. This keeps going until a final WED-state is reached, when this instance is then finalized. Roughly speaking, a WED-flow application is a transactional data state transitioning machine.

Aiming to be capable of running in a distributed environment, for instance in a web-services composition way, as well to broadly support Long Lived Transaction handling, the WED-SQL is based on a Client-Server system architecture and has two main component thereafter: a server component called WED-server and a client component called WED-worker. The sever side is responsible for both transactional and flow control, what means that besides coordinating the WED-transitions, enforcing individually defined completion time limits, it is also in charge of matching WED-states against WED-conditions and WED-trigger firing. On the other hand, the client side is responsible for doing the computation needed to perform each WED-transition.

2.1. WED-worker and WED-server

Fundamentally, the WED-server is a PostgreSQL extension package consisting of *database triggers*, control tables and stored procedures [5]. Choosing PostgreSQL as the WED-server foundation was a decision based on several of its aspects, such as be an open source project, catalog-driven operation, support for user-written code using low and high level programming languages (e.g., C, Python, Perl, TCL and SQL itself).

WED-SQL directly benefits from the fact that, being an Open-source software, PostgreSQL's license states that it can be used, modified, copied and distributed without any kind of payment or agreement. Besides that, having access to the source code makes development less complex and error prone once it is easier to analyze and, consequently, better comprehend its internal mechanisms.

Relational database systems needs to store metadata about internal structure of tables, columns, indexes, etc. This data is kept on what is know as system catalogs. PostgreSQL not only stores much of its control data in these catalogs but also base its procedures on then, in what is called a catalog-driven operation. Since they are readily available to users as ordinary tables, thus can be modified, it is fairly easy to modify the PostgreSQL behavior. This property is specially useful to allow the WED-server to manage timed transactions related to WED-transitions and exception handling.

As previously mentioned, users can write PostgreSQL modules using a few different languages. The most simple and straight forward of then is a SQL dialect called pgSQL, which is basically SQL with variables and flow control structures. Also, modules can be written in C language and loaded as a dynamic library, providing the user with ultimate logic control and performance. Despite these two language having native support in PostgreSQL, sometimes pgSQL is not expressive enough when the logic involved is too complex, while C, albeit fast, is too low level and overloaded with technical implementation details too perform most of the tasks. The PostgreSQL documentation recommends using a higher leve language other than C unless it is explicit needed. Fortunately, it also offers support for Python, a much more versatile and expressive programming language than the two former. Given the dynamic nature of WED-flow models, Python has been proven the perfect tool to do most of the job on the WED-server side, only recurring to C when critical or where an extra performance is absolutely necessary. And,of course, SQL

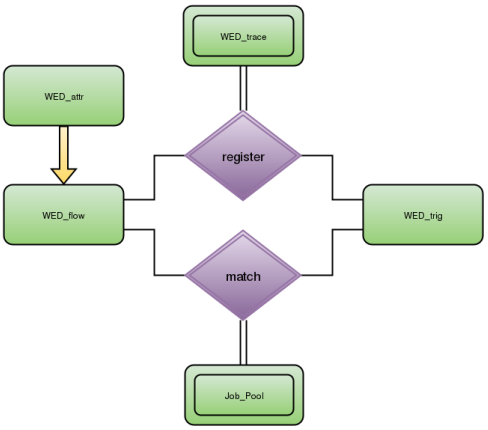

Figure 1. Approximate WED-server's Entity-Relationship model

statements are every where in both WED-server and WED-workers.

The role of the WED-workers is to perform the WED-transitions. They must connect to the WED-server and "ask" for a pending job before opening a new transaction. All WED-transitions are encapsulated in a single transaction with a time limit to complete, otherwise they are automatically aborted by the WED-server. Each WED-transition must have at least one WED-worker associated and, depending on the workload on the server side, can have many more. On the other hand, each WED-worker is specialized in execute a specific WED-transition. The maximum number of WED-workers simultaneously transacting depends on the number of concurrent connections that the WED-server is able to keep alive.

2.2. Data Model

The diagram depicted on figure* represents the data model used by the WED-server. It is not a true Entity-Relationship model once the relation between the *WED_attr* and *WED_flow* (represented by the arrow in the diagram) is managed by a stored procedure rather than by the RDBMS. This is necessary to comply with the dynamic nature of the WED-flow approach, which allows the system designer to evolve the system on-the-fly, for instance, including new WED-attributes and WED-conditions. In that case, the system's data structure may need to be modified accordingly.

WED-attributes are represented by the entity *WED_attr* using the following attributes:

- $aname$: Name of the WED-attribute and also the primary key;
- adv: Default value for this WED-attribute.

The *WED_flow* entity represents the instances of a WED-flow applications. Besides the identifying attribute (*wid*), it has a extra one for each WED-attribute defined for a given application, in other words, for each element of *WED_attr* there is an attribute $aname$ in *WED_flow*.

The weak entities *Job_Pool* and *WED_trace* results from the relation between the entities *WED_flow* and *WED_trig*. Identified by the ternary relationship *Match*, *Job_Pool* represents pending WED-transitions, that is, the ones awaiting to be processed by a WED-worker, using the following attributes:

- wid: Foreign key that identifies a WED-flow instance;
- $tgid$: Foreign key that identifies which WED-trigger fired this WED-transition;
- $trname$: Name of the WED-transition to be performed;
- $lckid$: Optional parameter used by WED-workers to be able to identify themselves. May be used for authentication purposes in the future
- $timeout$: References the same name attribute of entity *WED_trig*. It is the transaction time limit defined for this WED-transition.
- $payload$: Represents the WED-state that matched the WED-condition associated with this WED-transition.

Identified by the ternary relationship *Register*, *WED_trace* represents the history of execution for all WED-flow instances in a WED-server. Its attributes are :

- wid: Foreign key that identifies a WED-flow instance;
- $state$: WED-state that fired the WED-transitions represented by the multivalued attribute *trf*;
- trf: WED-transitions fired by the WED-state in *state* attribute;
- trw: WED-transition that committed the respective WED-state in *state*. A null value indicates a initial WED-state for the given WED-flow instance;
- $status$: Labels *state* accordingly to its status: "F" for a final WED-state, "E" to indicate that an exception has occurred and "R" meaning a regular WED-state;
- $tstmp$: The exact moment when this record was created. Can be used to retrieve the execution history of a WED-flow instance in chronological order.

Finally, the *WED_trig* entity represents the WED-triggers, that is, the associations of WED-conditions with WED-transitions, using the following attributes:

- $tgid$: Primary key;
- $tgname$: Optional parameter that indicates the name of the WED-trigger;
- $enabled$: Allow a WED-trigger to be disabled;
- $trname$: Unique name of the associated WED-transition;
- $cname$: Optional, can be used to name the associated WED-condition;
- $cpred$: WED-condition's predicated. Accepts the same syntax and tokens allowed in a SQL's *WHERE* clause;
- $cfinal$: Mark a WED-condition as final. Although only one is allowed, multiple stop conditions can be defined combining then with the logical operator *OR*;
- $timeout$: Time limit to perform the WED-transition

This conceptual data model is mapped to five database tables inside the WED-sever, one for each entity previously explained, using the same name. The WED_flow table is, however, special. It will be dynamically modified every time a new record is added, modified or removed from the WED_attr table. For example, after inserting a new record into WED_attr, a new column, having the same name of this new WED-attribute, will be added to WED_flow. Each row in WED_flow represents a distinct application instance.

```
BEGIN;

INSERT INTO wed_attr (aname, adv) VALUES ('a1','ready');
INSERT INTO wed_attr (aname) VALUES ('a2'),('a3');

INSERT INTO wed_trig (tgname,trname,cname,cpred,timeout)
  VALUES ('t1','tr_a2','c1', $$a1='ready' and (a2 is null)$$, '3d18h');
INSERT INTO wed_trig (tgname,trname,cname,cpred,timeout)
  VALUES ('t2','tr_a3','c2', $$a1='ready' and (a3 is null)$$, '00:00:30');
INSERT INTO wed_trig (tgname,trname,cname,cpred,timeout)
  VALUES ('tf','tr_final','cf', $$a1='ready'
          and (a2 is not null)
          and (a3 is not null)$$, '00:00:10');
INSERT INTO wed_trig (cpred,cfinal) VALUES ($$a1 <> 'ready'$$, True);

COMMIT;
```

Figure 2. Defining a new WED-flow application

3. How it works

A new WED-flow application is created by defining a set of WED-attributes and a set of WED-triggers that associates a WED-condition to a WED-transition. This is accomplished, for the time being, writing these definition in SQL (see an example on Fig. 2. In the foreseen future we plan to propose a SQL language extension to express at least the basic WED-flow operations.

Note that the expression of a WED-condition predicate (*cpred*) uses the same syntax of the SQL clause *WHERE*. In fact, this expression is employed *as-is* for the WED-server while matching it against a WED-state. When surrounded by double $ marks, special symbols like quotes don't need to be escaped in the predicate.

Also note that, although the application's final condition statement is declared in the same WED_trig table it does not have a WED-transition associated, otherwise it would be ignored by the WED-server anyway. To declare a WED-condition as final, one only need to specify its predicate and set the value of the *cfinal* attribute to true. If the final condition is missing, all instances of a given application will end up in an exception state.

We encourage developers to encapsulate all definitions for a new WED-flow application into a single transaction. By doing so one can prevent a partially defined application in case of syntax errors, which may force the developer to manually truncate the system tables before running the script again.

Although it is possible to define multiple applications in a single WED-flow database, the WED-server allows each one of then to be isolated in a different database. Therefore, applications can be grouped together by some semantic meaning and WED-attributes that must not be shared between some of then can be isolated.

3.1. Creating WED-workers

Since WED-workers are client applications of the WED-server, they can be written in any programming language that, somehow, is able to connect to the PostgreSQL database by simply implementing the WED-SQL's communication protocol (described in details on section 5). The WED-SQL provides a standard WED-worker implementation by means of a Python package named *BaseWorker*. Figure 3 depicts an example of a WED-worker implemented with this package.

The first step to write a WED-worker using the BaseWorker package is to import the module *BaseClass*, which in fact is an abstract class that implements the communi-

```
from BaseWorker import BaseClass
import sys

class MyWorker(BaseClass):

    #  trname and dbs variables are static in order to conform
    #with the definition of wed_trans()

    trname = 'tr_aaa'
    dbs = 'user=aaa dbname=aaa application_name=ww-tr_aaa'
    wakeup_interval = 5

    def __init__(self):
        super().__init__(MyWorker.trname,MyWorker.dbs,MyWorker.wakeup_interval)

    # Compute the WED-transition and return a string as the new WED-state,
    #using the SQL SET clause syntax. Return None to abort transaction
    def wed_trans(self,payload):
        print (payload)

        return "a2='done', a3='ready', a4=(a4::integer+1)::text"
        #return None

w = MyWorker()

try:
    w.run()
except KeyboardInterrupt:
    print()
    sys.exit(0)
```

Figure 3. Standard WED-worker implementation in Python

cation protocol with the WED-server as well as manages all connections and transactions involved.

Next, a concrete class must implement the abstract method *wed_trans()* from BaseClass and initialize the following class attributes:

- *trname*: WED-transition name that will be carried out by this WED-worker. Must match the name used on the WED-flow definition;
- *dbs*: WED-server connection parameters conforming the *psycopg2* driver format. At least the username, database name and the WED-worker name must be provided;
- *wakeup_interval*: Time interval in which the WED-worker execution is suspended (sleeping) if there is no new WED-transitions to be performed (explained in details on section 5.1);

The wed_trans() method works on a specific WED-flow instance at a time. It receives, as a parameter, the WED-state that fired this WED-transition and returns a string representing a new WED-state. This return value must comply with the same syntax used to specify a *SET* clause of a *UPDATE* SQL statement. An empty value can be returned to abort the transaction.

Finally, the concrete class can be instantiated and executed by invoking its *run()* method.

4. Algorithms

After a WED-flow application is defined, loaded into the WED-server and the WED-workers are initialized and ready to run, new instances are created by inserting a WED-state into the WED_flow table. The WED-server will assign to a new instance a WED-flow unique identifier (wid), which in turn will be used to both track its execution history as well as for transaction management purposes. If the default values set for the WED-attributes, in the WED_attr table, happens to match the ones of a initial WED-state, then

a new instance can be created by simply running the following SQL statement on the WED-server:

```
INSERT INTO wed_flow DEFAULT VALUES;
```

When a new instance is initiated or when a WED-worker successfully commits a new WED-state, the WED-server henceforth runs its main algorithm (Alg. 1). It starts by evaluating all WED-conditions predicates against the new WED-state. Then, for each satisfied predicate, the WED-transition associated with this WED-condition will be fired and registered on the Job_Pool table ,a notification will also be sent to the respective WED-worker. Besides that, this events will be added to the instance's tracing record on the WED_trace table. If a WED-state satisfies the final condition and if there is no pending WED-transitions, either running or awaiting in the Job_Pool table, for a given instance, it is then marked as final and cannot be further modified. In the case when there is no pending WED-transitions and the new WED-state is not final, the instance will be marked as an exception and a special record will be inserted into Job_Pool to signal some external recovery mechanism [2]. It is worth mentioning that unless a initial WED-state is also final, it must fire at least one WED-transition, otherwise the WED-server will reject the new instance.

WED-workers spend most of their idle time suspended, waiting for a notification send by the WED-server when there is work to be done. After some time waiting, if not notification has arrived they wake up and ask the WED-server for work (this behavior will be explained in section 5.1). Either way, before initiate a WED-transition, the WED-worker need to "inform" the WED-server of in which instance and what WED-transition it is about to perform. This is done by requesting the WED-server a special lock. If this lock is granted, then the WED-worker can proceed (See alg. 2)

This lock system is necessary once more than one WED-worker can be simultaneously activated for the same WED-transition. Therefore, its main role is to avoid the same WED-transition from being performed for the same WED-flow instance at the same time. Furthermore, it also helps the WED-server to manage transactions. It is wise to assume that no WED-transition can be committed by a WED-worker not holding a lock.

Once in possession of a lock, the WED-worker must finish its job within the time limit defined for the WED-transition. The transaction is successfully finalized updating the current WED-state for this instance on the WED_flow table.

5. Transactional Management

Each instance of a WED-flow application can be treated as a Long Lived Transaction (LLT), thereby, according to [3], the WED-transitions may be regarded as *SAGA* steps. Therefore, it is a WED-server obligation to ensure the integrity of data states by means of enforcing the transactional consistence, and thus guarantee that every instance always end in a final WED-state. In the case where an exception is raised, it must provide support for an external recovery mechanism, eventually leading this instance to a final WED-state.

In the following topics we discuss in details how the WED-server manages the timed transactions. We also present how to tackle some concurrency control challenges as well as explain the WED-SQL communication protocol.

Algorithm 1 WED-server

Input: WED-flow instance i
Output: **true** for a successfully transaction,
false on the otherwise

1: $L \leftarrow$ empty WED-transitions list
2: $s \leftarrow$ current WED-state of i
3: **for** tg in WED-triggers **do**
4: $c, tr \leftarrow tg$(WED-condition),tg(WED-transition)
5: **if** s satisfies c **then**
6: **if** c is the final condition **then**
7: insert $\_FINAL$ in L
8: **else**
9: insert tr in L
10: **end if**
11: **end if**
12: **end for**
13: **if** L is empty **then**
14: **if** i is a new instance **then**
15: Reject i and abort the transaction
16: **return false**
17: **else if** i **DOES NOT** has any pending WED-transition **then**
18: Set i as an exception
19: Insert _EXCPT into Job_Pool
20: **end if**
21: Update i at WED_trace
22: **return true**
23: **else if** L contains $\_FINAL$ **then**
24: **if** i has pending WED-transitions **then**
25: Reject i and abort the transaction
26: **return false**
27: **else**
28: Set i as a final WED-state
29: Update i at WED_trace
30: **return true**
31: **end if**
32: **else**
33: **for** tr in L **do**
34: Insert tr into Job_Pool
35: Notify the WED-worker(s) responsible for tr
36: **return true**
37: **end for**
38: **end if**

Algorithm 2 WED-worker

Input: WED-transition's name: *trname*
Output: **true** for a successfully transaction,
false on the otherwise

1: **loop**
2: $n \leftarrow$ Null value
3: **while** n is Null **do**
4: $n \leftarrow wait\_for\_notification(trname, wkup)$
5: **if** n is Null **then** {No notification has arrived after $wkup$ seconds}
6: $n \leftarrow$ Ask WED-server for any $trname$ pending WED-transition
7: **end if**
8: **end while**
9: Initiate the transaction
10: **if** got_lock(n) **then**
11: Run the WED-transition $trname$ for a given i WED-flow instance
12: Commit the transaction
13: **return true**
14: **else**
15: Abort the transaction
16: **return false**
17: **end if**
18: **end loop**

5.1. Notifications

Event-driven computer systems can be roughly described as a system that reacts in the face of a new event. In order to accomplish that, event detection mechanisms are often integrated to such systems. Since a data event based system usually keeps its data stored in a database, data event detection is generally made by a polling operation, that is, by continually checking some data for changes. This database polling technique, known as *Continuous Query* [4], mainly consists of continuously run a hand tailored query for each targeted data event.

When employing the Continuous Query model in a system that is response time sensitive, one needs to keep in mind the computational cost of this approach, which in turn depends both on the number of distinct events to be detected and on how often they pop up. In order to implement this model efficiently, it is critical to have a good time estimation of events occurrence, especially in the presence of multiple distinct events, once there is an overhead cost involved in performing a query that cannot be disregarded. In the other hand, if this time estimation is unpredictable for some or all of the events, the suitable query must be executed as frequent as possible, otherwise threatening the system's real time response capability. Anyway, the Continuous Query model is not well suitable to every data event driven system, which ultimately may lead to waste of computational resources, increasing the overall power consumption and possible overloading the system.

Since the WED-SQL is also a real time data event detection system, it is indeed part of the WED-server main algorithm, isn't possible to take advantage of this and somehow inform the WED-workers of the occurrence of its event of interest ? The answer is

yes ! It can.

The PostgreSQL provides a mechanism that can be used as an alternative to the Continuous Query model, it is named *Asynchronous Notification*. Via the command *NOTIFY*, the DBMS can asynchronously notify a client listening that is listening to a communication channel. A clients, in turn, can registry itself via command *LISTEN* on a specific channel to receive notifications. Furthermore, the server can also send the client a payload message attached to each notification.

The biggest advantage of the Asynchronous Notification model over Continuous Query is that, providing that a client is listening to a given communication channel, messages can be delivered instantly after the detection of a event. Moreover, now the client doesn't need to poll the server, so it can be suspended until a new notification arrives, thus freeing computational resources as well as saving power. In the WED-SQL framework, these two mechanisms are employed in a complementary combination.

Under normal execution, WED-workers are notified whenever a new WED-trigger is fired, in other words, this happens every time that a new record is inserted into the Job_Pool table. In addition, the payload message sent together with the notification is exactly the new "job" generated, so the WED-workers do not need to go to the WED-server to fetch a task, saving time and the cost of an extra query. Every WED-transition, in turn, has a proper notification channel identified by its own name as recorded on the WED_trig table. WED-workers subscribe themselves to his WED-transition channel and go into suspended mode until they are awakened with a new notification. In case a notification is sent and there is nobody listening to that channel, it will be discarded.

Once the WED-server and the WED-workers could be running on physically different machines, communicating over a network, messages can get lost. In order to ensure that all WED-transitions are eventually performed as well as to minimize their waiting to run time, that is, the time that a WED-transition sits into the Job_Pool table waiting to be performed, WED-workers don't rely solely on asynchronous notifications. In fact, before they even register on a notification channel, their first step is to scan the Job_Pool table for pending WED-transitions, and only then eventually goes into suspension state. With this in mind, WED-workers can be also configured to wake up after some time suspended and go check for available tasks on the WED-server.

As previously mentioned, the message sent out to WED-workers carries a WED-state that is the same one recorded on Job_Pool. By no means it should be regarded as the current state of a WED-flow instance. Remember that WED-transitions can be executed in parallel and therefore modifications to a instance may happen even before a pre-fired WED-transition starts. This WED-state reflects a instance snapshot by the time when the WED-transition was fired. So, if the firing conditions is already know by the WED-worker, what's the reason to send this data to then ? The answer is simple: efficiency. When a predicate of a WED-condition is as disjunction of predicates, the WED-worker may need to know which ones were satisfied by the WED-state. Having this data before hand eliminates the need to query the WED_trace table.

It is very important to note that a WED-worker should never rely in WED-attributes values that were not specified in the WED-condition's predicate, once, apart from being a project error, it may end up capturing spurious conditions or refusing to

perform a legitimate one.

5.2. Performing WED-transitions

WED-workers can only perform WED-transitions properly recorded on the Job_Pool table. Moreover, they must request the WED-server an exclusive execution lock on the task to be performed, being free to proceed once this lock is granted. The WED-server rely on this lock protocol to enforce each transaction to complete or abort within the time limits specified for each WED-transition. It also used to solve conflicts between concurrent WED-workers.

One characteristic of a well designed WED-flows project is the presence of reasonable execution time limits for its WED-transitions. Although it is up to the project designers to define these time limits, it is the role of the WED-server to enforce these restrictions. In order to accomplish that, it must first identify a WED-transition that is about to begin, and then keep track of its running time. It i worth to remember that each WED-transition is wrapped in a single transaction.

One way to force transactions to identify themselves is to require a exclusive lock when performing WED-transitions, effectively informing the WED-server of what task in Job_Pool is going to be worked on. Another possible solution would be semantically identify the WED-transition based on the new WED-state that is about to be committed, but not without a few caveats: First, it could allow a transaction to stay alive much longer than necessary in the case when a WED-transition would be aborted anyway due to time out. Second, the WED-server would need to know which new WED-states are valid for each WED-transition to set, potentially harming its exception handling capability. Moreover, the locking solution was chosen once it can also be used to solve concurrency conflicts.

PostgreSQL provides client applications with several explicit locking mechanisms, just in case a more refined control over the data is required. These locks can be used in either table or row level and are automatically acquired for the DBMS native operations. In the WED-SQL context though, only row level locks are explicitly employed.

The most common exclusive access row level lock is acquired using the SQL expression *FOR UPDATE* by the end of a *SELECT* command. When a transaction try to lock a row that was previously locked with that lock, it will be either blocked, and wait until the current transaction ends, or aborted if the expression *NOWAIT* was present after *FOR UPDATE*. There is however another type of lock that can be used to achieve this same behavior: they are named *Advisory Locks*.

Advisory Locks work on a semantic level, that is, they are only useful when they have some semantic meaning in the application. Put in other words, the DBMS do not enforce their use and nothing is really locked. It is up to the application to use then rightly. As an example, in the WED-server, pending WED-transitions siting in the Job_Pool table are uniquely identified by the attributes pair (*wid*,*tgid*). WED-workers requests Advisory Locks based on this composite key, which is only granted only once for each pair until the transaction is over. When a lock is not granted for a WED-work, it knows that some other "sibling" is currently working on that WED-transition, even tough the record is not

really blocked on the database, and then it is free to look for another task. Once again, the WED-server will blocked transactions not holding a valid lock.

Given that the *FOR UPDATE* approach seems to be the more straight forward way to manage WED-transitions, why did we decide to stick with Advisory Locks ? Well, one reason is purely technical and is based on how PostgreSQL works with different types of locks. Another one is due to how the WED-state transitions is carried out by our framework.

Regarding the technical aspects of the DBMS, information about *FOR UPDATE* locks are kept on disk instead of the RAM memory. Because of that, this data does not show up in system catalogs, and can only be obtained in real time using a third party module (pgrowlocks) that, besides being supplied by PostgreSQL itself, is rather inefficient. On the other hand, information about Advisory Locks can be easily, and efficiently, obtained directly from system catalogs.

On WED-SQL, a WED-transition begins on the Job_Pool table and ends on the WED_flow table. So, it would be somewhat awkward to lock a row on a table, using a FOR UPDATE lock, to update a record on another one. Not only that but it also could get messy. Therefore, the semantic locks based solution is simpler to implement, in this context is safer for usage and is also a more elegant approach.

5.3. Transaction isolation and parallelism

PostgreSQL employs a multi version concurrency control model (MVCC) to maintain data integrity. This means that each transaction sees a snapshot of the database taken right before it begins to run, which keeps then isolated from each other since no uncommitted data is shared between concurrent transactions. Another property of this model is that data read locks never conflicts with data write locks, minimizing lock contention.

O padrão ISO/ANSI SQL define quatro níveis de isolamento transacional. Em ordem de restrição: *Read Uncommited, Read Commited, Repeatable Read e Serializable*

No nível menos restritivo,*Read Uncommited*, uma transação enxerga modificações realizadas por outras transações concorrentes que ainda não finalizaram. Esse fenômeno é conhecido como *dirty read*. No PostgreSQL, transações que executam nesse nível de isolamento são na verdade executadas no nível *Read Commited*, que é o nível mínimo de isolamento possível na arquitetura MVCC. De qualquer modo, o padrão permite que um SGBD execute suas transações em um nível mais restritivo de isolamento do que o requisitado.

Transações que executam no nível *Read Commited* podem enxergar modificações realizadas por transações concorrentes que finalizem durante sua execução. Por exemplo, a leitura de uma mesma tupla duas ou mais vezes, dentro de uma mesma transação, pode retornar valores diferentes. Esse fenômeno é conhecido por *nonrepeatable read*. Esse é o nível padrao do PostgreSQL.

Ao contrário do nível anterior, no nível *Repetable Read*, uma transação ativa não enxerga tuplas modificadas por outras transações que finalizem previamente. Em caso de conflito, apenas uma transação finaliza e as outras são abortadas. Ainda assim, é possível ocorrer um fenômeno chamado de *phantom read*, no qual um conjunto de tuplas que satisfaça uma determinada condição difere em duas execuções da mesma consulta na

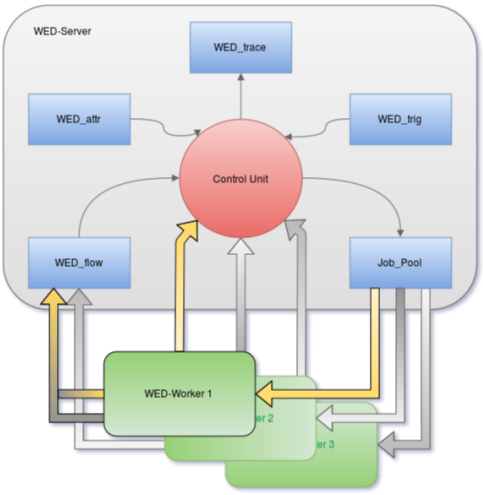

Figure 4. Os WED-workers 1 e 2 trabalham na mesma instância e precisam sincronizar a atualização do WED-state na tabela WED-flow

mesma transação. No PostgreSQL esse fenômeno não ocorre.

O último e mais restritivo nível de isolamento é o *Serializable*. De acordo com o padrão, um conjunto de transações concorrentes executando nesse nível deve produzir o mesmo resultado da execução sequencial em alguma ordem. Sendo assim, nenhum dos fenômenos mencionados anteriormente ocorre. No PostgreSQL, esse nível de isolamento funciona exatamente como o *Repeatable Read* a menos da introdução de um mecanismo para detectar anomalias que poderiam levar a um resultado não condizente com todas as possíveis execuções de um conjunto de transações em série.

No WED-server, assim como no PostgreSQL, o nível padrão de isolamento transacional é o *Read Commited*. Isso só é possível graças ao seu mecanismo de detecção de estados de dados que, valendo-se do modelo MVCC, gera uma nova versão das tabelas relevantes toda vez que uma transação modifica uma instância de um WED-flow. Sendo assim, no caso de duas transações concorrentes, por exemplo, não é preciso que uma delas seja abortada, o que ocorreria em níveis mais restritos de isolamento. Caso uma transação aborte, o WED-server descarta as alterações realizadas. Transações que operam em instância distintas são executadas totalmente em paralelo. Além disso, os WED-workers não exergarao as novas WED-transitions disparadas até que transação que escreveu o novo estado de dados finalize com sucesso.

WED-workers que trabalham simultaneamente na mesma instância executam em paralelo até o momento final, onde necessitam atualizar o estado de dados dessa instância. Nesse momento, a execução se dará de modo sequencial, uma vez que o PostreSQL irá conceder uma trava implicita para a tupla que será modificada. Vale notar que a atualização da instância deve ser a ultima operação executada pelo WED-worker, limitando o tempo de espera à quantidade de transações pendentes multiplicada pelo tempo de execução do comando SQL UPDATE. Caso contrário, a execução ocorre em paralelo (veja a figura 4).

6. Trabalhos Futuros

Na próxima fase desse projeto serão incorporadas duas funcionalidades essenciais para o sucesso do projeto WED-SQL: uma linguagem declarativa para expressar as operações nativas do WED-flow e um módulo específico para gerenciar as conexões com o WED-server.

Quanto à linguagem, a ideia é criar uma extensão da SQL (de codinome WSQL) que encapsule as operações nativas do WED-flow e integrá-la ao interpretador do PostgreSQL, o qual será então responsável por traduzir e executar código SQL nativo e outras operações que, nesse momento, são realizadas por meio de um conjunto de *shell scripts*. Por meio dessa abstração, será possível deixar a sintaxe declarativa do WED-server mais próxima do modelo teórico, simplificando sua utilização.

Em Linux, um processo pode criar um clone de si mesmo para executar uma porção específica de código de modo independente. O PostgreSQL, assim como muitos outros serviços que atendem à requisições externas, utiliza essa técnica sempre que for estabelecida uma nova conexão com uma aplicação cliente. Ao analisar a utilização de recursos do sistema operacional , é possível observar que cada novo sub-processo gerado ocupa cerca de 9MB de memória RAM. Como consequência, há um limite físico no número de conexões que podem ser estabelecidas simultaneamente. No caso do WED-SQL, cada WED-worker precisa manter uma conexão ativa com o WED-server para receber as notificações assíncronas em tempo real, além de estabelecer uma segunda conexão para efetivamente executar a WED-transition. Sendo assim, para cada WED-worker ativo são criados dois sub-processos do PostgreSQL, o que acaba por consumir em média 18MB de RAM.

As notificações assíncronas do PostgreSQL são baseadas em um modelo de comunicação conhecido por *Publish-Subscribe*, onde quem envia uma mensagem não esta ciente da existência de quem a recebe. Sendo assim, é possível utilizar uma especie de WED-worker específico que funcionaria com um detector de eventos em tempo real para receber e distribuir as mensagens para os demais WED-workers, o qual utilizaria uma única conexão com o WED-server, efetivamente dobrando a quantidade de WED-transitions simultâneas. Há diversas soluções prontas para implementar esse mecanismo, umas delas é o software RabbitMQ.

7. Conclusão

Desde sua concepção em 2010 [1], o paradigma WED-flow não parou de evoluir. Novos estudos, realizados antes e durante o desenvolvimento deste trabalho, mostram que embora a ferramenta WED-tool [6] implemente o WED-flow, na prática ela ja não atende aos novos requisitos dessas novas aplicações.

A WED-SQL está sendo estrategicamente elaborada para suprir essa demanda. Sua integração ao SGBD garante que o controle de fluxo proposto pelo modelo seja executado com propriedades transacionais nativas, além de dispor do mais avançado aparato para o controle de concorrência assim como garantir a integridade dos dados com as propriedades ACID. A adoção do modelo cliente servidor tem por objetivo viabilizar a sua implementação em ambientes distribuídos, uma vez que os WED-workers tipicamente serão implementados como *Web Services* e múltiplos WED-servers poderão ser utilizados simultaneamente, seja para manter a disponibilidade, quanto para balanceamento de

carga. Ao utilizar uma linguagem declarativa própria para a WED-SQL, é possível aproximar sua sintaxe da utilizada no modelo teórico, simplificando sua utilização, e eventualmente lançar uma versão especial do PostgreSQL otimizada para o WED-flow.

Resumindo, trata se de uma ferramenta moderna, altamente escalável, construída em um ambiente transacional cujo principal objetivo é simplificar a utilização e a implementação de modelos de processos de negócio desenvolvidos sob o paradigma WED-flow.

Acknowledgment

The authors would like to thank...

References

[1] João E. Ferreira, Osvaldo K. Takai, Simon Malkowski and Calton Pu. *Reducing exception handling complexity in business process modeling and implementation: the WED-flow approach.*, Em Proceedings of the 2010 international conference on On the move to meaningful internet systems - Volume Part I, OTM'10, pages 150–167. Springer-Verlag, 2010.

[2] João Eduardo Ferreira, Kelly Rosa Braghetto, Osvaldo Kotaro Takai and Calton Pu. *Transactional recovery support for robust exception handling in business process services.* Em Proceedings of the 19th International Conference on Web Services (ICWS), pages 303–310, 2012

[3] Hector Garcia Molina, Kenneth Salem. *Sagas*, Em Proceeding of the 1987 ACM SIGMOD International Conference on Management of Data, page 249-259, 1987

[4] Ling Liu, Calton Pu, Wei Tang. *Continual Queries for Internet Scale Event-Driven Information Delivery*, IEEE Trans. on Knowl. and Data Eng. 11, pages 610–628, 1999

[5] Ramez Elmasri, Shamkant B. Navathe. *Fundamentals of Database Systems*, 6th ed, chapters 1.6.9 and 22.3, 2010

[6] Marcela Ortega Garcia, Pedro Paulo de S. B. da Silva, Kelly Rosa Braghetto e João Eduardo Ferreira. *Wed-tool: uma ferramenta para o controle de execução de processos de negócio transacionais*, Proceedings of the 27th Brazillian Symposium on Databases - Demos and Applications Session, page 36, 2012

[7] PostgreSQL, v9.5.x
http://www.postgresql.org/about/