WED-SQL: A SQL Extension for Implementation of Business ProcessesBusiness Process Implementation through an SQL Extension: WED-SQL

Bruno Padilha[1], André Luis Schwerz[2], Rafael Liberato Roberto[2], João Eduardo Ferreira[1] and 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 Net require an additional software layer or some external toolkits to be able to enforce a data-state based transaction control and deal 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 WED-SQL, a distributed framework that provides a reliable and efficient way to design and implement business processes. Our main contribution is the integration of WED-flow concepts into the PostgresSQL RDBMS. This integration enables the WED-SQL to take full advantage of transactional properties and also benefit from the SQL language to specify the WED-flow definitions.*

1. Introduction

Most modern business process management software systems are prone to recurrent structural modifications during its life cycle. These changes are necessary to incorporate new requisites to the business processes, once it is often impossible to come up with all requirements upfront in the modeling phase of project - some of them only appear after the software is running in production. Furthermore, it is too expensive and time consuming to try to predict all the possibles paths and its outcomes, which may arise in form of software exceptions that must be dealt with. Over time, these modifications usually lead to code deterioration - compromising the system overall performance - and exponential increase of maintenance costs.

Classic business process specification models, for example WS-BPEL, Process Algebra and Petri Net, focus primarily on behavioral interactions and relations between processes, 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 to the classical models, captures both the inter-process relations and the data generated by their interactions with the same relevance. 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 can be often translated to creation of new processes, is a simple matter of defining new data states and conditions for their transitions.This captures the true nature of dynamic processes and simplifies an incremental evolution of the whole project. Exceptions are also easier to be handled and integrated to the project, once this approach is based on transactional properties and provides several transactional recovery mechanisms.

In order to provide a standard implementation as well as simplifying its software development, in this work we present the WED-SQL, a 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 allows developers to put more effort on business logic, which results in better WED-flow projects and software that makes better use of reusable code.

Once the WED-SQL framework has been built inside the PostgreSQL [7] Relational Database Management System (PostgreSQL RDBMS), it is highly fault tolerant, has full transactional support while WED-flow properties and definitions are specified using the SQL language. By employing well consolidate technologies - widespread in the computer science field - this framework targets to disseminate the adoption of the WED-flow approach on business process management software design. It is a reliable and scalable tool, capable to provide the flexibility demanded by modern transaction control systems.

The remaining of this paper is structured as follows: Section 2 presents WED-SQL architecture and its internal data structures. Section 3 briefly describes how the system works. A discussion about the algorithms involved is presented in Section 4. Following, a detailed description of transactional control and the communication protocol used for external communication is given in section 5. Finally, we discuss the next challenges to improve and expand this framework in Section 6.

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 an 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.

The WED-SQL has been designed to manage business process in distributed environments (e.g., web-services composition) and support execution of long-lived transactions. Its architecture is based on the client-server model and its two main components are described as follows. The *WED-server* is responsible for both transactional and flow controls. This includes transitions coordination, time-out management for transactions to complete, matching between WED-states and WED-conditions, and WED-triggers activation. On the other hand, each *WED-workers* is responsible for the execution of a specific WED-transition.

2.1. WED-worker and WED-server

Fundamentally, the WED-server is a PostgreSQL extension package that includes Database Triggers, control tables and Stored Procedures [5]. We opted for PostgreSQL as the WED-SQL foundation mainly due to its following aspects: open source project, catalog-driven operation, and support for user-written code using low and high level programming languages (e.g., C, Python, Perl, TCL and SQL itself).

Standard RDBMS need to store some metadata about internal structures of tables, columns, indexes, etc. This data is kept in what is know as system catalogs. PostgreSQL not only stores much of its control data in these catalogs but also bases its operations on then, in what is called a catalog-driven operation. Since they are readily available to users as ordinary tables, 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 straightforward of them is a SQL dialect called pgSQL, which is basically SQL with variables and flow control structures. These modules can also be written in C language and loaded as dynamic libraries, providing the developer with ultimate logic control and higher performance. Despite these two language having native support in PostgreSQL, sometimes pgSQL is not expressive enough for complex logic, while C is too low level and requires an overload of technical details for implementation of most tasks. The PostgreSQL documentation recommends using a higher level language other than C unless it is explicitly needed. Fortunately, the PostgreSQL also offers support for Python, a much more versatile and expressive programming language than the two former. Due to 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 statements are everywhere 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.

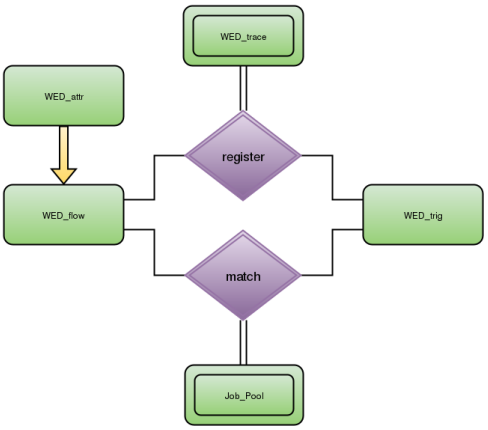

Figure 1. Approximate WED-server's Entity-Relationship model

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 in Figure 1 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 this 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 application. Besides the identifying attribute (*wid*), it has a extra one for each WED-attribute defined for a given application, in other words, for each row inserted in *WED_attr* a column representing the new WED-attribute $aname$ is added to *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, i.e., the WED-transitions that are 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. This attribute will be used for authentication purposes in the future work;
- $timeout$: Reference to the attribute of same name in entity *WED_trig*. It is the transaction time limit defined for this WED-transition.
- $payload$: It represents the WED-state that satisfied the WED-condition associated with this WED-transition.

Identified by the ternary relationship *Register*, *WED_trace* represents the execution history 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" indicates a final WED-state, "E" indicates that an exception has occurred and "R" means a regular WED-state;
- $tstmp$: The exact moment when this record was created. It 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, i. e. , 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;
- $enabled$: Allows a WED-trigger to be disabled;
- $trname$: Unique name of the associated WED-transition;
- $cname$: Optional parameter that may be used to name the associated WED-condition;
- $cpred$: WED-condition's predicate . Accepts the same syntax and tokens allowed in a SQL's *WHERE* clause;
- $cfinal$: Marks a WED-condition as final. Although only one is allowed, multiple stop conditions can be defined by combining them 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. However, the WED_flow table is 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 identified by the same name of this new WED-attribute, will be added to WED_flow. Each row in WED_flow represents a distinct application instance.

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. For the time being, this is accomplished, by writing these definition in SQL as illustrated in Fig. 2.

```
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

In ongoing work, we are proposing a SQL language extension to express the WED-flow basic elements.

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. To declare a WED-condition as final, the designer only needs 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 the WED-SQL enables multiple WED-flow applications to be defined in a single database, they can also be isolated from each other. Therefore, applications can be separated by some semantic meaning or even to restrict access to some specific WED-attributes that must not be shared.

3.1. Creating WED-workers

Since WED-workers are client applications of the WED-server, they can be written in any programming language that is able to connect to the PostgreSQL database by simply implementing the WED-SQL's communication protocol (described in Section **??**). The WED-SQL provides one WED-worker implementation through 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 communication 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:

```
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. WED-worker implementation in Python

- *trname*: WED-transition name that will be carried out by this WED-worker. It must match the name used on the WED-flow definition;
- *dbs*: WED-server connection parameters complying with the *psycopg2* driver format. At least the username, database name, and the WED-worker name are required;
- *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 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 that represents 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 and 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 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. In addition, this events will be added to the instance's tracing record into the WED_trace table. If a WED-state satisfies the final condition for a given instance and if there is no pending WED-transitions, either running or awaiting in the Job_Pool table, this instance is then marked as final and cannot be further modified. A instance will be marked as an exception when it doesn't have any pending WED-transitions and its current WED-state is not final. In this case, 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 no 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 initiating a WED-transition, the WED-worker needs to "inform" the WED-server about what WED-transition and in which instance it is going to be performed. This is done by requesting the WED-server a special lock. If this lock is granted, then the WED-worker can proceed as described in Algorithm 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 on 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 a WED-transition cannot be committed by a WED-worker that is not in possession of 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 by updating the current WED-state for this instance in the WED_flow table.

5. Transactional Management

Each instance of a WED-flow application is treated as a Long-Lived Transaction (LLT). Thereby WED-transitions may be regarded as *SAGA* steps [3]. Therefore, the WED-server is responsible for ensuring the integrity of data states by enforcing the transactional consistency and that each instance will always end in a final WED-state. , When an exception is raised, the WED-server must provide support for an external recovery mechanism, eventually leading this instance to a final WED-state.

In the following sections, 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.

5.1. Notifications

Event-driven computer systems can be 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

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**

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 *Continual Query* [4], mainly consists of continuously run a hand tailored query for each targeted data event.

When employing the Continual 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 Continual 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 Continual Query model, it is named *Asynchronous Notification*. Via the command *NOTIFY*, the DBMS can asynchronously notify a client listening that is listening to a com-

munication 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 Continual 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.

The ISO/ANSI SQL standard defines four levels for transaction isolation. In increasing order of restrictiveness are they: *Read Uncommitted, Read Committed, Repeatable Read e Serializable*

In the less strict level, *Read Uncommitted*, a transaction can see modifications performed by others ongoing concurrent transactions. This phenomena, know as *dirty reads*, is nonetheless not allowed in PostgreSQL due to the MVCC architecture, and transactions that request this isolation level are effectively performed in the *Read Committed* level. Anyway, the SQL standard permits a DBMS to to run a transaction in a stricter level than requested.

Transactions running in the *Read Committed* level will view data committed by concurrent transaction, in the sense that two consecutive runs of a select statement may return a different set as a result, even though it is performed inside a single transaction. This phenomena is know as *nonrepeatable read*.

Conversely, a transaction performing in the *Repetable Read* level cannot view any results produced by concurrent transactions. In case of writing conflicts, only one transaction will be allowed to proceed and the others will be aborted. Nevertheless, the SQL standard predicts that a phenomena called *phantom read*, in which a result set that matches some condition differs on two successive runs of the same statement, may occur in this level. This cannot happen in PostgreSQL tough.

The last and more strict level is the *Serializable*. According to the standard, a set o concurrent transaction running in this level must yield the same result as running then serially in some order. Due to this requirement, none of the previously mentioned phenomena can occur. In PostgreSQL, this level behaves just like the *Repeatable Read* apart of the introduction of a mechanism to detect non serializable executions orders.

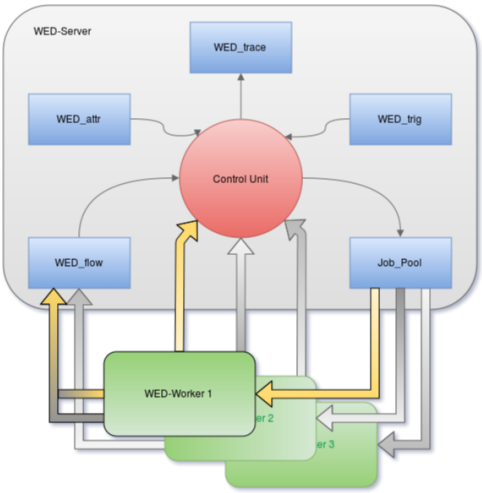

Figure 4. WED-workers one and two working on the same instance may need to synchronize their update at the WED_flow table

Regarding the WED-server, the default transaction isolation level is the *Read Committed*. Due to how its WED-state detection mechanism works, and also to the MVCC model, concurrent WED-transitions don't need to fully isolated from each other, in addition, the order of their execution is never critical for a correctly designed WED-flow project. For instance, if a WED-transition aborts after setting up a new WED-state, the possible fired new WED-transitions will never be seen by the WED-workers, and the database returns to the previous consistent state.

Concerning the WED-workers, they are able to work in parallel on a same WED-flow instance until the final moment, when they need to update the WED-state. At this point, each transaction will be granted a lock to update this instance at a time, while the others must either wait or abort. During a normal operation of the WED-server,this last step of performing a WED-transition should be fast, therefore worthwhile for concurrent transactions to wait at least for some small amount of time before they eventually decide to abort. This waiting time to commit is bounded by the number of concurrent transactions waiting for the lock multiplied by how long the WED-server takes to perform the update. Distinct WED-flow instances can be worked fully in parallel (See Figure 4).

6. Future Work

The next phases of this work will be focused in two main tasks: Create a declarative language capable of natively express all the WED-flow's definitions and operations; Develop a connection manager module to reduce computer resources utilization by the WED-server when too many WED-workers are simultaneously in use.

Regarding the language, the idea is to extend the SQL language, defining new tokens and grammatical rules, and integrate it to the PostgreSQL parser. By approximating the syntax used on the WED-server to the mathematical definitions of the theoretical model, we aim to make the implementation of a WED-flow project as smooth as possible.

PostgreSQL forks itself every time a new connection is established with a client application, creating a new system process. A analysis of the system resources utilization has show that each of these new process uses, on average, 9MB of RAM memory,

what imposes a considerable limitation on the amount of simultaneous connections that the WED-server is capable of dealing with. Although a connection pool, in which connections are shared among non overlapping transactions, would suffice to circumvent this limitation, it cannot be employed with the *Asynchronous Notifications* pattern, once a connection must be kept alive to ensure that messages don't get lost.

Since the asynchronous notifications are based on a *Publish-Subscribe* approach, in which message senders are not aware of receivers, it is possible to implement some kind of a dedicated WED-worker to receive all notifications and wake the others when is appropriate. This could easily double the number of simultaneous WED-transitions that the WED-server can handle using the same amount of RAM memory, once each WED-worker currently needs two connections - one to receive notifications and another one to perform the WED-transition. There is also some third-party solutions like the RabbitMQ software.

7. Final Conclusions

The WED-flow approach has been evolving since its first proposal back in 2010 [1]. Previous and ongoing studies have raised new demands for a framework capable of support more advanced features present in modern WED-flow models, which are no longer meet by the only prototype tool available until now, the WED-tool [6].

The WED-SQL framework has meticulously been built to supply these demands. Its integration to a RDBMS ensures that a application's flux control is always performed under transactional rules and governed by ACID properties. By adopting a client-server architecture it was designed to run in a distributed environment and be scaled to support thousands of parallel transactions. In summary, WED-SQL is not only the most advanced WED-flow framework but it also aims spread the adoption of the WED-flow approach to design and implement business process models by providing an easy to use and reliable tool.

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, Kelly Rosa Braghetto, Calton Pu e João Eduardo Ferreira. *An Implementation of a Transaction Model for Business Process Systems*, Journal of Information and Data Management - SBBD, 2012

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