WED-SQL (Rascunho)

Bruno Padilha, João E. Ferreira
Departamento de Ciência da Computação
IME-USP
Rua do Matão 1010, 05508-090
São Paulo, SP, Brasil
brunopadilha@usp.br, jef@ime.usp.br

Calton Pu
CERCS, Georgia Institute of Technology
266 Ferst Drive, 30332-0765
Atlanta, GA, USA
calton@cc.gatech.edu

***Resumo*—Desenvolver e implementar um modelo de processo de negócios utilizando os conceitos do WED-flow resulta em um sistema de informação dinâmico, simplificando o desenvolvimento incremental e adaptativo, além de reduzir significativamente a complexidade no tratamento de exceções quando comparado aos arcabouços tradicionais (BPEL, álgebra de processos, Redes de Petri e etc). No entanto, uma aplicação baseada no WED-flow precisa implementar uma camada de software em um nível abaixo para fazer o controle transacional baseado em estados de dados (WED-states), assim como fornecer suporte ao tratamento de exceções ou utilizar uma ferramenta como a WED-tool gerenciar esses mecanismos. A proposta da WED-SQL é fornecer um arcabouço WED-flow distribuído e de alto desempenho para simplificar tanto o desenvolvimento quanto a implementação de tais aplicações. Uma caracteristica peculiar da WED-SQL é que seu mecanismo de controle está integrado a um SGBD relacional, que além de usufruir de todo um aparato transacional utiliza a linguagem SQL para a especificação das definições do WED-flow (WED-triggers, WED-transitions, WED-conditions, etc).**

Keywords*—*WED-flow, PostgreSQL, transações longas, ...

I. Introducão

Sistemas computacionais contemporaneos para o gerenciamento de processos de negócio estão sujeitos a constantes modificaçoes estruturais, ocasionadas tanto por execessões não previstas na fase de modelagem quanto para atender a novos requisitos. Ao longo do tempo, essas modificaçoes tendem a deteriorar a qualidade do código de tais sistemas, aumentando exponencialmente seu custo de manutenção além de, eventualmente, comprometer seu desempenho.

Os modelos clássicos para especificação de processos de negócios, dentre eles o WS-BPEL, algebra de processos e redes de petri, procuram capturar interações, relações e o comportamento entre processos, muitas vezes negligenciando a importância dos dados no projeto. Nenhuma dessas ferramentas, no entanto, é exatamente adequada para modelar processos que necessitam ser modificados frequentemente, o que implica em aumento exponencial no custo e na complexidade do projeto.

O modelo WED-flow[1], como uma alternativa aos modelos clássicos, captura não só a dinâmica da interação entre processos como também, com igual importância, os dados gerados e os eventos que implicam em alterações desses dados. Com isso, o WED-flow é capaz de construir um workflow de estados de dados com suas respectivas condições de transição entre os mesmos. E assim, agregando propriedades de recuperação transacional a esse workflow, prover mecanismos para o tratamento de exceções em tempo real.

Ao utilizar a abordagem WED-flow, modificações nos processos de negócio são traduzidas em novos estados de dados. Novas regras, ou novos processos, se traduzem em condições para se atingir um determinado estado de dados. Essa versatilidade permite capturar a natureza de processos verdadeiramente dinâmicos, simplificando a evolução incremental tanto do modelo de negócio quanto do software que o implemente.

Para que um modelo de processo de negócio baseado no WED-flow seja implementado em software, é preciso antes implementar suas definições e operações fundamentais [2], que são a base para o tratamento de exceções e para o controle transacional. Ao invés de incorporar essas definições a cada software que implemente um modelo WED-flow, uma maneira mais eficiente de o fazer é construir um arcabouço para padronizar a implementação do WED-flow em software. Nesse contexto, esse trabalho apresenta o arcabouço WED-SQL.

O WED-SQL foi construido dentro de um SGBD relacional e com isso utiliza a linguagem SQL para definir propriedades e controles de fluxo de um modelo WED-flow, além de contar com um robusto aparato transacional e ser altamente tolerante a falhas. Por utilizar como base tecnologias consolidadas e amplamente difundidas no universo da computação, o WED-SQL visa disseminar a adoçao do paradigma WED-flow na modelagem de processos de negócio , fornecendo uma ferramenta confiável, facilmente escalável, simples de ser utilizada e que permita a flexibilidade exigida na especificação de controle de fluxo dos processos de negócio modernos.

O restante desse artigo segue a seguinte estrutura: Na sessão II são apresentados aspectos da arquitetura de software e das estruturas de dados internas. A sessão III descreve sucintamente o funcionamento do sistema. A sessão IV descreve os algoritmos envolvidos. Na sessão V são discutidos em detalhes o gerenciamento de transações e o protocolo de comunicação. Finalmente, na sessão VI, são apresentados os próximos desafios para expandir as funcionalidades dessa ferramenta.

II. WED-SQL: Visão geral e arquitetura

/* Incluir um resumo das definições do WED-flow ? */
Com o objetivo de permitir sua utilização em um ambiente distribuido, por exemplo por meio de web-services, e também para dar amplo suporte ao tratamento de transações longas [**?**], o WED-SQL foi construido utilizando a arquitetura de software *Cliente-Servidor*.

A componente servidor, que recebe o nome de WED-server, é responsável por fazer o controle transacional de acordo com as WED-conditions definidas para um determinado WED-flow, verificando os WED-states e disparando as WED-triggers conforme necessário. Já a componente cliente, denominada WED-worker (quem sabe até WED-service ?), é quem efetivamente faz a computação necessárias para executar as WED-transitions.

A. WED-worker e WED-server

O WED-server é, basicamente, uma extensão para o SGBD (Sistema Gerenciador de Banco de Dados) PostgreSQL [5] composta de *triggers*, tabelas de controle e *stored procedures* [4]. A escolha do Postgresql como base do WED-server se deu por diversos motivos, dentre eles: ser de código aberto, implemetação modular baseada em catálogs (*system catalog driven*), facilidade de extender suas funcionalidades por meio de modulos escritos nas linguagens C, Python e SQL.

Por ser de código aberto, a licensa do Postgresql garante que o mesmo possa ser modificado e redistribuido, o que permite que o WED-SQL possa ser distribuido sem grandes empecilhos. Além disso, o acesso ao código fonte possibilita uma melhor compreensão de seus mecanismos internos, garantido uma implementação mais robusta e eficiente do paradigma WED-flow.

O postgresql armazena seus dados de controle em tabelas especiais denominadas catálogos de sistema. Esses catálogos, que são facilmente acessados com as devidas permissões, expõe aos usúarios informações vitais para extender suas funcionalidades por meio da criação de modulos que podem ser carregados tanto de modo dinâmico, ou seja, com o SGBD executando ou de modo estático, durante a inicialização. Dados de transações podem ser facilmente obtidos por meio desses catálogos de sistema, o que é partircularmente útil no caso do WED-server.

Devido à natureza dinâmica do paradigma WED-flow, a utilização de uma linguagem de programação de alto nível, como Python, é fundamental para que toda a sua expressividade seja implementada de forma plena. Do ponto de vista de implementação, o WED-server é composto de módulos escritos em Python, SQL e C, além de aproveitar o arcabouço transacional clássico (propriedades ACID, controle de concorrência e etc) fornecido pelo PostgreSQL.

Já o WED-worker tem a função de conectar-se ao WED-server para efetuar as WED-transitions. Cada WED-transition é associada a um ou mais WED-workers, dependendo da demanda de trabalho gerada pelo WED-server, e cada WED-worker é especializado em realizar uma WED-transition específica. A quantidade máxima de WED-workers trabalhando simultaneamente é limitada apenas pela quantidade de conecções que um dado WED-server é capaz de manter abertas.

B. Estrutura de dados

O diagrama Entidade-Relacionamento na figura 1 representa o modelo de dados gerenciado internamente pelo WED-server. Vale notar que, devido a flexibilidade de modificação da estrutura de dados exigida para representar os WED-states, não é possível capturar todos os aspectos do paradigma WED-flow por meio de um diagrama Entidade-Relacionamento clássico.

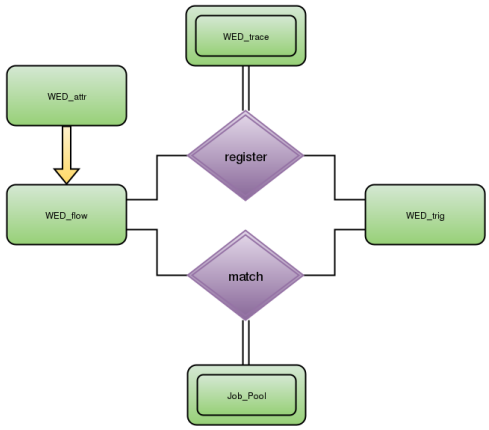

Figura 1. Diagrama ER do WED-server

Por exemplo, o relacionamento entre as entidades WED_attr e WED_flow é gerenciado pelo WED-server, ou seja, não há chaves indicando a relação entre essas duas entidades. A principal razão disso é pertimir adicionar ou remover WED-attributes em tempo de execução, o que demanda modificações em tempo de execução na estrutura de dados.

Do relacionamento entre as entidades WED_flow e WED_trig resultam a entidade fraca Job_Pool, que contém as WED-transitions a serem executadas, e a entidade fraca WED_trace, que contém os registros de execução de cada instância dos WED-flows.

C. Tabelas de controle

O WED-server utiliza cinco tabelas para definir regras, propriedades e controlar o fluxo de execução dos WED-flows: WED_attr, WED_flow, WED_trace, WED_trig e Job_pool.

Os WED-attributes são definidos na tabela WED_attr por um nome e, opcionalmente, por um valor padrão representados pelas colunas "aname"e "adv"respectivamente. Cada WED-attribute é identificado univocamente por seu nome, que também é a chave primária da tabela. Ao criar-se um novo WED-attribute, ou seja, ao inserir-se uma nova linha na tabela WED_attr, uma coluna de mesmo nome será automaticamente adicionada na tabela WED_flow.

A tabela WED_trig contém as WED-triggers, ou seja, cada entrada representa a associação de uma WED-condition com uma WED-transition. Possui os seguintes atributos:

- $tgid$: o identificador único de uma WED-trigger;
- $tgname$: atributo opcional utilizado para dar um nome à WED-trigger;
- $enabled$: permite que a WED-trigger seja desativada;
- $trname$: identificador único da WED-transition associada.
- $cname$: atributo opcional que pode ser utilizado para dar um nome à WED-condition associada;
- $cpred$: predicado da WED-condition associada. Aceita qualquer predicado valido na claúsula *WHERE* em SQL;
- $cfinal$: utilizado para indicar qual é a condição final. Embora apenas uma WED-condition possa ser

marcada como final, o operado lógico *OU* pode ser utilizado em seu predicado para definir-se multiplos WED-states finais.

- $timeout$: tempo limite para que um WED-worker finalize a WED-transition representada por "trname".

O histórico de execução das WED-transition fica armazenado na tabela WED_trace, que possui os seguintes atributos:

- wid: referência ao identificador da instância do WED-flow;
- $state$: WED-state que disparou as WED-transitions listadas em "trf";
- trf: lista de WED-transitions disparadas pelo WED-state em "state";
- trw: WED-transition que gravou o WED-state representado por "state". Um valor nulo indica que esse registro representa o WED-state inicial;
- $status$: indica o tipo do WED-state representado em "state". Os possíveis valores são "F","E"ou "R"que indicam que o WED-state é final, excessão ou regular, respectivamente.
- $tstmp$: indica o momento em que ocorreu o registro. Pode se recuperar a história de execução de uma instância ordenando-se as entradas nessa tabela por essa coluna.

A tabela Job_Pool contém entradas referentes as WED-transitions pendentes que precisam ser executadas pelos WED-workers. Seus atributos são:

- wid: referência ao identificador da instancia do WED-flow;
- $tgid$: referência à WED-trigger que disparou a WED-transition "trname";
- $trname$: nome da WED-transition a ser executada;
- $lckid$: parâmetro opcional que pode ser utilizado pelos WED-workers para se identificarem. Futuramente, poderá ser utilizado para fins de autenticação e validação dos WED-workers;
- $timeout$: tempo limite para a execução da WED-transition (tempo limite da transação). É uma cópia da coluna de mesmo nome da tabela WED_trig;
- $payload$: WED-state que disparou a referida WED-transition. Pode ser utilizado, por exemplo, por WED-workers que executam WED-transitions associadas a WED-conditions que possuem o operador lógico "OR"em seu predicado.

Finalmente, a tabela WED_flow é o ponto de entrada para inicializar-se uma nova instância. Essa tabela é criada dinâmicamente de acordo com os WED-attributes definidos na tabela WED_attr. Cada entrada em WED_attr corresponde a uma coluna em WED_flow. Suas entradas são o WED-state atual de cada instância, ou seja, são tuplas formadas por um identificador, representado na coluna "wid", e os WED-attributes.

```
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;
```

Figura 2. Exemplo de definição de um novo WED-flow

III. FUNCIONAMENTO

Para se criar um novo WED-flow é preciso definir um conjunto de WED-atributes e um conjunto de WED-triggers associando WED-transitions à WED-conditions. Essa definição, por ora, é expressa em SQL. Veja um exemplo na figura 2.

Note que os valores dos predicados para as WED-conditions, representados por meio da coluna "cpred", tem a mesma sintaxe utilizada para expressar as restrições de uma cláusula *WHERE* em SQL. De fato, a expressão em "cpred"será utilizada *as-is* pelo WED-server para disparar as WED-transitions. Ao utilizar-se duplo $ para delimitar essa expressão, elimina-se a necessidade de "escapar"as aspas simples ou outros caracteres especiais.

Note também que a condição final de um WED-flow é declarada nessa mesma tabela WED_trig, embora de modo simplificado. São necessários apenas o predicado em "cpred"e o valor Verdade em "cfinal". Caso não haja uma condição final na tabela WED_trig, todas as instâncias desse WED-flow terminarao em um WED-state de excessão.

É recomendado encapsular a definição de um WED-flow em uma única transação, uma vez que, no caso de um erro de sintaxe na definição de uma WED-trigger, por exemplo, seria necessário remover manualmente do WED-server as definições executadas até o momento do erro.

Embora seja possível definir multiplos WED-flows em uma única base de dados, o WED-server permite que cada um deles esteja localizado em uma base de dados distinta, de acordo com um determinado significado semântico. Com isso também é possível isolar atributos que não devem ser compartilhados entre diferentes WED-flows.

A. Definindo os WED-workers

Como mencionado anteriormente, quem executa as WED-transitions disparadas pelo WED-server são os WED-workers. Sendo assim, é necessário criar esses WED-workers e associá-los às respectivas WED-transitions. É recomendado ter ao menos um WED-worker associado a cada WED-transition.

Um WED-worker é uma aplicação cliente do WED-server e, por esse motivo, pode ser escrito em qualquer linguagem de progra- mação que tenha suporte para conectar-se ao sgbd PostgreSQL e que implemente o protocolo de comunicação do WED-server (descrito em detalhes nas próximas sessões). No escopo deste trabalho os WED-workers são escritos em Python utilizando-se o pacote *BaseWorker*, que acompanha o WED-SQL. A figura 3 ilustra a definição de um novo WED-worker.

O primeiro passo da implementação é importar a classe abstrata *BaseClass* do pacote *BaseWorker*. Essa classe implementa a lógica do protocolo de comunicaçãoce também

```
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)
```

Figura 3. Exemplo de definição de um novo WED-worker

gerencia as conecções com o WED-server.

Em seguida é preciso definir uma classe concreta que extenda a BaseClass e implemente o método wed_trans(), definindo os seguintes atributos utilizados para inicialização:

- *trname*: nome da WED-transition que será executada por esse WED-worker. Precisa ser o mesmo nome utilizado na definicão do WED-flow;
- *dbs*: parametros da conecção com o WED-server no formato aceito pelo driver *psycopg2*. No mínimo devem ser especificados o usuário, o nome da base de dados onde o WED-flow foi carregado e o nome do WED-worker por meio do campo *"application_name"*;
- *wakeup_interval*: é o intervalo de tempo no qual o WED-worker fica suspenso esperando por uma notificação do WED-server;

O método wed_trans() recebe como parâmetro o WED-state da instância do WED-flow que disparou a transação, e retorna uma string com os novos valores dos WED-attributes dessa mesma instância. A sintaxe utilizada para esse valor de retorno deve ser a mesma utilizada em uma cláusula *SET* da operação *UPDATE* em SQL. O valor "None"pode ser retornado para abortar a transação. Essa classe concreta que implementa o WED-worker deve então ser instanciada e executada, invocando-se seu método *run()*.

IV. Algoritmos

Após definir e carregar um WED-flow em uma base de dados do WED-server e inicializar seus respectivos WED-workers, uma nova instância é inicializada inserindo-se um WED-state na tabela WED_flow. Essa nova instância receberá um identificador único que será utilizado tanto para o registro de sua história de execução, na tabela WED_trace, quanto para o controle transacional das WED-transitions. Se os valores padrão definidos para os WED-attributes, por meio da coluna "adv", na tabela WED_attr forem os valores de um WED-state inicial, basta executar o comando abaixo para se inicializar uma nova instância:

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

Para cada nova instância inserida na tabela wed_flow, o WED-server irá comparar os predicados das WED-conditions definidos na tabela wed_trig com o WED-state representado na nova instância. Para cada condição satisfeita a respectiva WED-transition será disparada na forma de uma entrada na tabela job_pool e uma notificação será enviada ao WED-worker responsável por executá-la. Além disso, serão adicionadas novas entradas na tabela wed_trace registrando os eventos ocorridos. O WED-server ficará então aguardando até que uma nova instância ou uma WED-transition seja iniciada. Vale notar que WED-states iniciais que não sejam finais e que não disparem ao menos uma WED-transition serão rejeitados pelo WED-server.

Para iniciar uma WED-transition, os WED-workers podem atuar de dois modos distintos: imediatamente ao receber uma notificação do WED-server, ou quando o limite de tempo de espera por notificações termina (atributo wakeup_interval) , nesse caso será feita uma consulta à tabela job_pool em busca de WED-transitions pendentes. Independentemente do modo de atuação e antes de inicializar a transação, cada WED-worker precisa notificar o WED-server qual WED-transition e em qual instância do WED-flow será executada a transação. Essa notificação é feita requisitando-se uma trava especial chamada *Advisory Lock*, que será discutida em detalhes na sesão V. Essa trava é necessária tanto para avisar a outros WED-workers que estejam trabalhando na mesma WED-transition que não a executem para a mesma instância, quanto para que o WED-server possa controlar o tempo limite da execução de cada WED-transition. Não é possivel executar uma WED-transition sem que o respectivo WED-worker consiga obter essa trava previamente.

De posse da referida trava, um WED-worker terá que finalizar a transação dentro do tempo limite de execução da WED-transition. Essa transação é finalizada atualizando-se o WED-state atual da instância em que está sendo executada a WED-transition por meio de um *UPDATE* na tabela wed_flow com o valor retornado pelo método wed_trans().

Quando uma instância é atualizada, o WED-server novamente irá verificar se esse novo WED-state dispara novas WED-transitions e, em caso afirmativo, irá registrá-las na tabela job_pool, além de atualizar o histórico de execução na tabela wed_trace. Caso esse novo WED-state satifaça a condição final e não haja nenhuma WED-transition pendente, essa instancia é marcada como finalizada e não poderá ser modificada, a menos que sejam inseridos novos WED-attributes ou que sejam modificadas ou adicionadas novas WED-triggers. Em caso de não haver WED-transitions pendentes e esse WED-state não for final, a instância será marcada como excessão e uma entrada especial será adicionada a tabela job_pool para indicar que algum mecanismo de recuperação deve ser iniciado.

V. Gerenciamento transacional

Cada instância de um WED-flow pode ser vista como uma Transação Longa (LLT) e, de acordo com [3], suas WED-transitions podem ser vistas como passos Saga. Sendo assim, é função do WED-server garantir a integridade dos estados de dados, manter a consistência transacional e garantir que

Algoritmo 1 WED-server: disparo de WED-transitions

Input: Instância i de um WED-flow
Output: **true** se a transação foi bem sucedida,
false em caso contrário

1: $L \leftarrow$ lista vazia de WED-transitions
2: $s \leftarrow$ WED-state atual de i
3: **for** tg in WED-triggers **do**
4: $c, tr \leftarrow tg$(WED-condition),tg(WED-transition)
5: **if** s satisfaz c **then**
6: **if** c é a condição final **then**
7: insira $\_FINAL$ em L
8: **else**
9: insira tr em L
10: **end if**
11: **end if**
12: **end for**
13: **if** L está vazia **then**
14: **if** i é uma nova instância **then**
15: Rejeite i e aborte a transação
16: **return false**
17: **else if** i **NÃO** possui WED-transitions pendentes **then**
18: Maque i como WED-state de exceção
19: Insira _EXCPT em Job_Pool
20: **end if**
21: Atualize i em WED_trace
22: **return true**
23: **else if** L contém $\_FINAL$ **then**
24: **if** i possui WED-transitions pendentes **then**
25: Rejeite i e aborte a transação
26: **return false**
27: **else**
28: Marque i como WED-state final
29: Atualize i em WED_trace
30: **return true**
31: **end if**
32: **else**
33: **for** tr em L **do**
34: Insira tr na tabela Job_Pool
35: Envie notificação para o WED-worker que executa tr
36: **return true**
37: **end for**
38: **end if**

Algoritmo 2 WED-worker

Input: Nome da WED-transition: *trname*
Output: **true** se a transação foi bem sucedida,
false em caso contrário

1: **loop**
2: $n \leftarrow$ Valor nulo
3: **while** n é nula **do**
4: $n \leftarrow espera\_notificacao(trname, wkup)$
5: **if** n é nula **then** {nenhuma notificação recebida após esperar $wkup$ segundos}
6: $n \leftarrow$ busque por WED-transitions $trname$ pendentes no WED-server
7: **end if**
8: **end while**
9: Inicia a transação
10: **if** obter_trava(n) **then**
11: Execute a WED-transition $trname$ para uma determinada instância i do WED-flow
12: Finalize a transação
13: **return true**
14: **else**
15: Aborte a transação
16: **return false**
17: **end if**
18: **end loop**

todo WED-flow termine ou em um estado final ou em um estado de excessão. Nesse caso, deve permitir que um passo compensatório possa ser executado.

Nas sessoes seguintes serao apresentados em detalhes os mecanismos de controle transacional utilizados e o protocolo de comunicação entre multiplos WED-workers e um WED-server.

A. Notificações

Ao modelar um sistema cujas operações principais são realizadas com base em eventos de dados, é preciso também conceber-se algum tipo de mecanismo capaz de capturar tais eventos. Quando esses dados estão armazenados em um SGBD, muitas vezes o único mecanismo de detecção disponível é continuamente realizar uma consulta (query) a cada intervalo de tempo. Esse modelo, conhecido por *Continuous Query*(ref...), possui alguns inconvenientes embora, muitas vezes, é a única solução disponível.

O primeiro problema com o *Continous Query* é que é preciso manter uma conecção continuamente aberta com o SGBD durante toda a execução do sistema, mesmo que ela fique ociosa em boa parte do tempo. Alternatvamente pode-se abrir e fechar essa conecção inumeras vezes em um curto periodo de tempo, nesse caso o custo da consulta é significativamente menor do que o custo de se realizar esse procedimento. Em ambos os casos há um desperdício de recursos computacionais, assim como de consumo de energia, uma vez que esse consulta será executada em *spin lock*.

Outro detalhe desse modelo é que, por se tratar de uma consulta que é realizada a cada intervalo de tempo, há sempre um atraso na detecção de mudanças nos estados de dados, que por sua vez pode acabar comprometendo a capacidade do sistema de responder a eventos em tempo real.

O SGBD PostgreSQL oferece um mecanismo que pode ser utilizado como uma alternativa ao modelo anterior: Notificações Assíncronas. Por meio do comando "NOTIFY <canal>,<payload>", o SGDB pode notificar de modo assíncrono um cliente que tenha se registrado, via comando "LISTEN <canal>", em um canal de notificação específico. Além disso, também é possivel enviar uma mensagem ao cliente, por meio de «payload>". Essa mensagem pode ser, por exemplo, uma cadeia de characteres no formato *JSON* que represente o estado de dados de uma instância de um WED-flow.

A grande vantagem do modelo de Notificações Assíncronas sobre o modelo de Continuous Query é que o cliente recebe a notificação em tempo real da ocorrência do evento de dados, contanto que o mesmo esteja escutando no canal de notificação nesse momento. Além disso, devido ao fato de o cliente não

precisar verificar a cada certo intervalo de tempo se há alguma nova mensagem, ele pode esperar pela notifição "dormindo", ou seja, sua execução pode ser suspensa enquanto não há uma nova mensagem, liberando os recursos da máquina para realizar outras tarefas assim como economizando energia.

No WED-SQL são utilizados os dois modelos de detecção de eventos de dados. Durante uma execução normal, o WED-server notifica os WED-workers sempre que uma nova WED-trigger é disparada, ou seja, sempre que surge uma nova tarefa na tabela Job_Pool. Além disso, a mensagem enviada aos WED-workers é exatamente a nova tarefa que foi gerada, eliminando a necessidade de se fazer uma busca na tabela Job_Pool. Cada WED-transition possui um canal exclusivo de notificações identificado por seu nome definido na tabela WED_trig. Os WED-workers registram-se em seu respectivo canal e suspendem sua operação até que sejam acordados com uma notificação. Caso uma notificação seja enviada e não haja ninguem escutando em um determinado canal, a mesma será descartada.

Afim de garantir que todas as WED-transitions sejam eventualmente executadas, e de preferência sejam iniciadas no menor intervalo de tempo possível, os WED-workers não dependem exclusivamente das notificações recebidas para funcionar. De fato, ao inicializar uma nova conecção com o WED-server, o primeiro passo é procurar por WED-transitions pendentes de execução na tabela Job_Pool, e só após entrar em estado de suspensão aguardando por notificações. Os WED-workers podem ainda ser configurados para "acordar"após um certo intervalo de tempo sem receber novas notificações e verificar por tarefas pendentes, uma vez que, em se tratando de um sistema potencialmente distribuido, é possível que mensagens possam se perder por falhas na rede de comunicaçao ou por outros eventos nao detectáveis tanto pelo WED-server quanto pelos WED-workers.

O WED-state recebido pelos WED-workers, seja por meio de notificação ou consultando a tabela Job_Pool, não representa o estado atual da instância mas sim o estado de dados no momento em que a WED-transition foi disparada. Sendo assim, dado que a condição de disparo de uma WED-transition é conhecida, qual a razão de se enviar o estado de dados aos WED-workers ? Eficiência. No caso de WED-conditions definidas por uma conjunção de predicados, ou seja, utilizando o operador lógico *OR*, pode ser necessário verificar os valores dos WED-attributes que dispararam a WED-transtion associada. Enviar o estado de dados aos WED-workers elimina a necessidade de consultas à tabela WED-trace. É muito importante lembrar que, ao definir um WED-worker, não se deve considerar atributos que não foram definidos na WED-condition, uma vez que, além de ser um erro de projeto do WED-flow, outras transações executando em paralelo podem alterá-los a qualquer momento. Cabe ao projetista do WED-flow garantir que uma WED-transition não dependa de WED-atributes que não foram definidos na WED-condition associada.

B. Controle transacional e de concorrência

Antes que uma WED-transition possa ser executada, é necessário que a respectiva tarefa esteja registrada na tabela Job_Pool e que o WED-worker que irá executá-la garanta exclusividade de execução. Isso é efeito requisitando uma trava ao WED-server na tarefa especificada. Essa trava é necessária por dois motivos: permitir que o WED-server tenha controle sobre o tempo de vida da transação; gerenciar conflitos entre transações concorrentes.

O modelo WED-flow prevê que cada WED-transition tenha um tempo limite de execução. Para atender a esse requisito, o WED-server precisa identificar a transição que está sendo executada e monitorar o seu tempo de vida. Vale lembrar que a execução de uma WED-transition está encpsulada em uma unica transação. Um modo de forçar as transações a se identificarem é exigir que as mesmas solicitem uma trava exclusiva ao WED-server. Outra possível solução seria identificar a transação semanticamente de acordo com o novo WED-state gerado, e nesse caso teriamos dois problemas: o WED-server apenas seria capaz de identificar a transação no momento da atualização da instância, fazendo com que as transações pudessem ficar ativas muito além de seu tempo limite para só então serem finalizadas, potencialmente limitando o tempo máximo de execução para todas as transações; seria necessário manter um conjunto de possíveis estados de dados pré-definidos, limitando a capacidade do modelo em lidar com exceções. Além do mais, o WED-SQL utiliza essas travas de execução para resolver problemas de concorrência entre WED-workers e, com isso, é a solução adotada para a identificação das transações.

Cada WED-transition pode ter multiplos WED-workers identicos associados. Isso pode ser necessário para atender a demanda gerada por um WED-flow com muitas instâncias. Para evitar que mais de um WED-worker execute a mesma WED-transition na mesma instância, o mesmo deve registrar interesse em executá-la por meio da requisição ao WED-server de uma trava exclusiva na linha correspondente da tabela Job_Pool, que é identificada por seu WED-flow id (wid) e a WED-trigger id (tgid) que a disparou.

O SGDB PostgreSQL oferece um mecanismo de travas explicitas para aplicaçoes que necessitam exercer um controle mais refinado sobre os dados acessados. Essas travas podem ser utilizadas tanto a nível de tabelas quanto a nivel de tuplas, ou seja, linhas ou registros das tabelas. No contexto do WED-SQL, apenas as travas a nivel de tuplas são utilizadas explicitamente.

A trava a nivel de tupla mais comumente utilizada é chamada *FOR UPDATE*. Utilizada em conjunto com o comando SQL *SELECT*, essa trava, de acesso exclusivo, faz com que outras transações concorrentes tenham que esperar ou abortar ao tentarem acessar a tupla bloqueada. Outro tipo de trava que também pode ser utilizada a nível de tupla são as *Advisory Locks*, ou travas semnaticas. Para utilizar esse tipo de trava é preciso definir seu significado semântico na aplicação, uma vez que o PostgreSQL não obriga que sua regra de acesso seja cumprida, ou seja, os registros não são realmente bloqueados.

Por exemplo, no WED-server as WED-transitions pendentes são identificadas pelo par (wid,tgid). Esse identificador é utilizado pelos WED-workers para requisitar uma *Advisory Lock*, que será concedida apenas à primeira transação que a solicitar para o dado registro. As demais transações que tentarem acessar o mesmo registro não conseguiram obter a trava, e estarão livres para tentar executar outras WED-

transitions pendentes, embora o registro não esteja de fato bloqueado. Vale lembrar que, obter essa trava para a instância específica que será atualizada, é mandatório para completar a transação com sucesso.

Por que são utilizadas *Advisory Locks* ao invés de *FOR UPDATE* no WED-SQL ? Por dois motivos: primeiramente pelo modo como o PostgreSQL funciona internamente e, em segundo lugar, pela natureza da operação de transição de estados no contexto do WED-SQL.

O principal motivo para não utilizar travas do tipo *FOR UPDATE* é que as informações sobre quais tuplas estão bloqueadas são armazenadas em disco, e não em memória RAM. Com isso, é necessário utilizar um plugin externo ao SGBD para recuperar tais informações, necessárias ao WED-server para realizar o controle do tempo de vida de cada transação. Por outro lado, as informações sobre *Advisory Locks* podem ser obtidas facilmente, por meio de uma simples consulta a um catálogo interno do PostgreSQL.

Outro motivo para não se utilizar *FOR UPDATE* no controle transacional, é que o registro que seria bloqueado não é o registro que será atualizado. As travas são obtidas na tabela Job_Pool, porém as atualizações são executadas na tabela WED_flow. Desse modo é mais simples, seguro e eficiente utilizar travas semanticas no WED-server.

C. Isolamento e paralelismo

O PostgreSQL utiliza o modelo MVCC para manter a integridade dos dados. Isso significa que cada transação enxerga uma "foto"do banco de dados no momento em que a mesma inicia, garantindo seu isolamento uma vez que não há compartilhamento de dados parciais entre transações ativas. Nesse modelo de controle de concorrencia, travas de leitura não conflitam com travas de escrita.

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 possivel 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

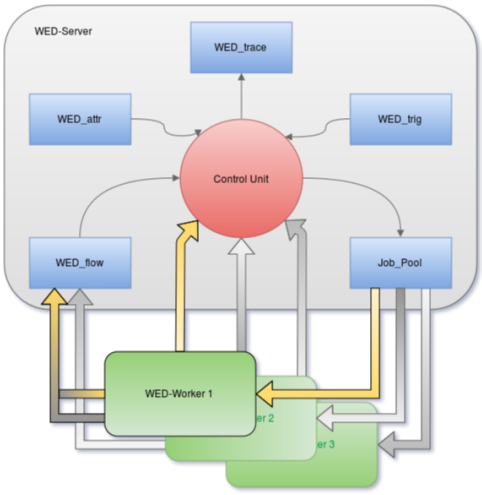

Figura 4. Os WED-workers 1 e 2 trabalham na mesma instância e precisam sincronizar a atualização do WED-state na tabela WED-flow

phantom read, no qual um conjunto de tuplas que satisfaça uma determinada condição difere em duas execuções da mesma consulta na 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 possives 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).

VI. Trabalhos Futuros

(linguagem, gerenciador de conecções, ...)

Na próxima fase desse projeto serão incorporadas duas funcionaliades 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 conecções com o WED-server.

Quanto à linguagem, a ideia é criar uma extensao 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 conecçao 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 conecções que podem ser estabelecidas simultaneamente. No caso do WED-SQL, cada WED-worker precisa manter uma conecção ativa com o WED-server para receber as notificações assíncronas em tempo real, além de estabelecer uma segunda conecção 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 conecção 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.

VII. Conclusão

Desde sua concepção em 2010 [**?**], 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 [**?**] implemente o WED-flow, na prática ela ja não atende aos novos requisitos dessas novas aplicações.

A WED-SQL está sendo estratégicamente 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 garatir a integridade dos dados com as propriedades ACID. A adoção do modelo cliente servidor tem por objetivo viabilizar a sua implementação em ambientes distribuidos, uma vez que os WED-workers tipicamente serão implementados como *Web Services* e multiplos 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 com sob o paradigma WED-flow.

Acknowledgment

The authors would like to thank...

Referências

[1] João E. Ferreira, Osvaldo K. Takai, Simon Malkowski e 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, páginas 150–167. Springer-Verlag, 2010.

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

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

[4] Ramez Elmasri, Shamkant B. Navathe. *Fundamentals of Database Systems*, 6th ed, ch 1.6.9, 2010

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