Projeção Eficiente de Violações de Restrições de Negação

Leonardo F. Luciano[1], Wendel C. Moro[1], Eduardo C. de Almeida[1], Eduardo H. M. Pena[2]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)
Curitiba – PR – Brasil

[2]Departamento de Informática – Universidade Tecnológica Federal do Paraná (UTFPR)
Campo Mourão – PR – Brasil

{lfl18,wcm18,eduardo}@inf.ufpr.br, eduardopena@utfpr.edu.br

Abstract. *The visualization of data quality rule violations is highly useful in data cleaning. A widely used operation for this visualization is the projection of tuple combinations that violate the rules. However, this operation is costly when considering the state-of-the-art formalisms in data cleaning, such as denial constraints. In the worst case, all combinations of tuple pairs in the table violate the rule, resulting in quadratic complexity regarding the number of records. This paper presents and experimentally evaluates various techniques for efficiently implementing projection for denial constraint violations.*

Resumo. *A visualização de violações de regras de qualidade de dados possui grande utilidade na limpeza de dados. Uma operação amplamente utilizada para essa visualização é a projeção das combinações de tuplas que violam as regras. No entanto, essa operação é custosa quando consideramos os formalismos estado-da-arte em limpeza de dados, como as restrições de negação. No pior caso, todas as combinações de pares de tuplas da tabela violam a regra, resultando em uma complexidade quadrática em relação ao número de registros. Este artigo apresenta e avalia experimentalmente diversas técnicas para a implementação eficiente da projeção de violações de restrições de negação.*

1. Introdução

A detecção de violações a regras de dados é uma etapa fundamental do processo de limpeza de dados. Uma vez detectadas as violações, é de grande utilidade, tanto para engenheiros e cientistas de dados quanto para ferramentas de limpeza de dados, como, por exemplo, o HoloClean [Rekatsinas et al. 2017], poder visualizar os dados das tuplas envolvidas nas violações. O FACET [Pena et al. 2022] é um algoritmo que detecta as violações de um conjunto de dados a regras de dados modeladas como restrições de negação. Ele é capaz de detectar violações de maneira mais eficiente em termos de tempo que outras soluções, como o uso de consultas SQL em SGBDs ou o algoritmo VioFinder [Pena et al. 2020]. Uma limitação do FACET, entretanto, é que ele indica o número de violações encontradas, mas não exibe informações das tuplas envolvidas. O que propomos neste trabalho é estender o FACET de maneira a implementar a projeção das violações detectadas, permitindo a visualização dessas violações e ajudando engenheiros e cientistas a conhecer melhor seus dados.

O presente trabalho foi realizado com apoio do projeto Laboratório de Dados Educacionais (LDE).

2. Definições

Regras de qualidade de dados podem ser definidas de diferentes maneiras, mas aqui iremos representá-las utilizando restrições de negação, que é o formalismo mais utilizado no estado da arte em limpeza de dados [Rekatsinas et al. 2017] e excede os limites do poder expressivo de outros formalismos existentes, além de permitir aplicações eficientes em vários cenários, incluindo a descoberta de regras de dados [Chu et al. 2013].

Definição 1 (Restrição de negação) *Sejam $p_1, \ldots, p_m$, com $m \in \mathbb{Z}$ e $m \geq 1$, predicados da forma $t.A$ o $t'.B$, onde A, B são colunas de uma tabela r com esquema R e n tuplas, (t, t') é um par de tuplas distintas de r e $o \in \{=, \neq, <, >, \leq, \geq\}$ é um operador de comparação. Definimos uma restrição de negação ϕ como:*

$$\phi : \forall t, t' \in r, \neg(p_1 \wedge \ldots \wedge p_m),$$

determinando que os predicados $p_1, \ldots, p_m$ não devem ser simultaneamente satisfeitos.

Para a definição dos operadores implementados neste artigo, adaptamos notações de [Grefen and de By 1994] semelhantes a álgebra relacional tradicional, mas que suportam multiconjuntos. Primeiramente, apresentamos a definição do operador de projeção de tuplas, que permite visualizar os dados de tuplas envolvidas em violações.

Definição 2 (Operador de projeção de tuplas) *Seja r uma tabela com esquema R e n tuplas, α uma lista de colunas de r, tid a coluna de r que representa os identificadores das tuplas de r e $tids_1, tids_2$ dois conjuntos de identificadores de tuplas de r. Definimos o operador de projeção de tuplas $\pi^{vt}_{\alpha}(r)$ como:*

$$\pi^{vt}_{\alpha}(r) : \pi_{\alpha}(\sigma_{tid \in (tids_1 \cup tids_2)}(r))$$

A seguir, vamos definir o operador de projeção de pares de tuplas, que permite visualizar os dados dos pares de tuplas que violam uma restrição de negação.

Definição 3 (Operador de projeção de pares de tuplas) *Seja r uma tabela com esquema R e n tuplas, α uma lista de colunas de r, tid a coluna de r que representa os identificadores das tuplas de r e $tids_1, tids_2$ dois conjuntos de identificadores de tuplas de r. Definimos o operador de projeção de pares de tuplas $\pi^{vp}_{\alpha}(r)$ como:*

$$\pi^{vp}_{\alpha}(r) : \pi_{\alpha}(\sigma_{tid \in tids_1}(r)) \bowtie_{t.tid \neq t'.tid} \pi_{\alpha}(\sigma_{tid \in tids_2}(r))$$

A implementação atual do FACET já nos fornece os conjuntos $tids_1$ e $tids_2$, bem como temos acesso ao arquivo de dados que define r. Sendo assim, o que exploramos e apresentamos a seguir são formas eficientes de armazenar e acessar os dados de r.

2.1. Implementações da tabela de projeção

Em ambas as implementações, cada linha do arquivo de dados é lida como um *array* de *strings* e, para construir cada tupla da tabela de projeção, é preciso iterar pelo *array* de acordo com a ordem em que as colunas devem ser projetadas. O que diferencia as versões é a forma como representamos as tuplas na tabela de projeção.

Na implementação básica, que corresponde às versões *baseline*, cada tupla da tabela é um *array* de *strings*. Uma vantagem desta implementação é que não são necessárias conversões de tipo para armazenar as *strings* lidas, o que reduz o processamento

realizado. Por outro lado, ao executar a projeção, é preciso iterar pelas colunas, enviando para cada uma delas a *string* correspondente para o *buffer* de saída, o que aumenta o processamento necessário. A Figura 1 exibe uma representação da tabela de projeção desta implementação para a projeção da Tabela 1.

Na tentativa de reduzir o tempo total de execução, o que fazemos na implementação eficiente, que corresponde às versões *bytes*, é armazenar as tuplas da tabela de projeção como um *array* de *bytes*. Esta implementação tem como vantagem, ao executar a projeção, enviar as tuplas inteiras para o *buffer* de saída, sem necessidade de iterar pelas colunas, o que reduz o processamento realizado. Por outro lado, para construir a tabela de projeção, é necessário converter cada uma das *strings* lidas para sua representação em *bytes*, concatenando tais representações no *array* de cada tupla, aumentando o processamento necessário. A Figura 2 exibe uma representação da tabela de projeção desta implementação para a projeção da Tabela 1.

$t.tid$	t.Dept	t.StartDate	t.Salary
t_3	Research	2014	6000
t_4	Research	2015	8000

Tabela 1.
$\pi^{vt}_{Dept,StartDate,Salary}(funcionarios)$, **com** $tids_1 = \{t_3\}$ **e** $tids_2 = \{t_4\}$

Figura 1. Armazenamento *baseline*

0	Research	2014	6000
1	Research	2015	8000

Figura 2. Armazenamento em *array* de *bytes*

0	A V B V C
1	A V D V E

Binário	*String*
A	Research
B	2014
C	6000
D	2015
E	8000
V	,

2.2. Estratégias de construção da tabela de projeção

Foram avaliadas também duas estratégias distintas para a construção da tabela de projeção. A primeira delas consiste em construí-la junto com o FACET, aproveitando a leitura do arquivo de dados realizada por ele para construir a tabela de projeção. A vantagem desta estratégia é permitir que o arquivo de dados seja lido somente uma vez. Entretanto, utilizá-la implica manter a tabela de projeção em memória durante todo o processamento do FACET, mesmo sem utilizá-la. Usar esta estratégia pode levar a concorrência por memória entre a tabela de projeção e as estruturas de dados utilizadas pelo FACET, bem como é capaz de aumentar o número de *cache misses* durante a detecção de violações. A Figura 3 apresenta um diagrama da execução usando esta estratégia.

Para evitar a concorrência por recursos com o processamento do FACET, implementamos uma versão que somente constrói a tabela de projeção ao fim da detecção das violações. Esta estratégia implica a releitura completa do arquivo de dados, mas permite que o FACET execute com todos os recursos disponíveis exclusivamente para ele. Além disso, com esta implementação podemos fazer ainda uma outra otimização, que consiste em não construir a tabela de projeção com todas as tuplas do arquivo de dados,

mas somente com aquelas que efetivamente participam de alguma violação. A Figura 4 apresenta um diagrama da execução usando esta estratégia.

Figura 3. Estratégia de leitura única

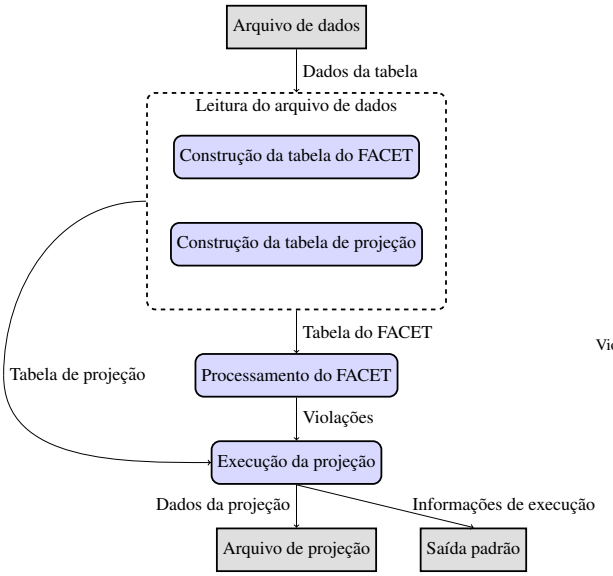
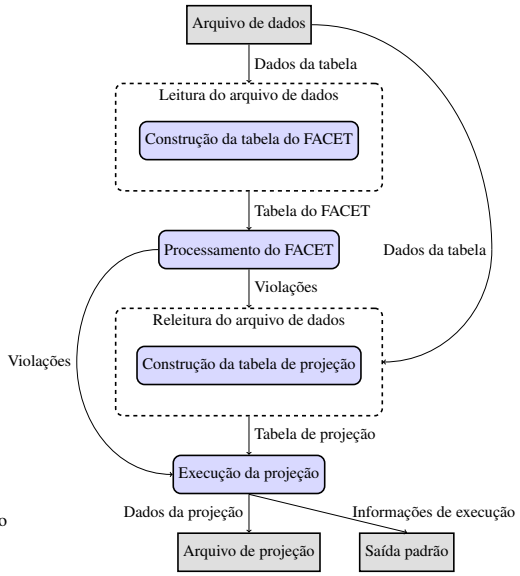


Figura 4. Estratégia de releitura



3. Experimentos

A seguir, apresentaremos nossa avaliação experimental, detalhando nossa bancada de testes, os *datasets* e restrições de negação utilizados no processo, os resultados obtidos e comentários sobre eles.

3.1. Bancada de testes

As configurações utilizadas para a bancada de testes foram: processador AMD Ryzen 5 1600AE, Sistema Operacional Fedora 37, versão do Kernel 6.3.4, 16GB de memória RAM e OpenJDK versão 11.0.19. O armazenamento utilizado foi um SSD Adata XPG SX6000 (Armazenamento de 256GB, Leitura 1800MB/s, Gravação 900MB/s) com a partição formatada em Btrfs. Utilizamos também a ferramenta tlp[1] para fixar o *clock* do processador em 3.2GHz.

3.2. Resultados

Os *datasets* utilizados como referência também foram usados nos experimentos com o FACET apresentados em [Pena et al. 2022]. Nos gráficos, as versões com nomenclatura *single* representam as implementações sem releitura da entrada.

***Dataset Flights* -** $\phi_2 : \neg(t_1.Origin = t_2.Destination \wedge t_1.Destination = t_2.Origin \wedge t_1.Distance \neq t_2.Distance)$**:** o *dataset* contém 10000000 de linhas e 11 colunas. O FACET detecta 7106922 violações, sendo que 253644 tuplas estão envolvidas nessas violações. O resultado pode ser visto na Figura 5.

Na projeção de pares de tuplas, as versões *bytes* tem um tempo de projeção menor do que as versões *baseline*. Isto ocorre devido a quantidade considerável de dados a

[1]https://linrunner.de/tlp/index.html

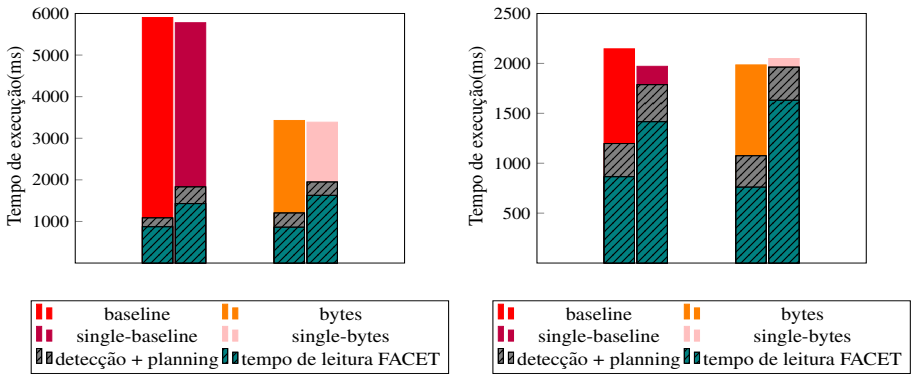


Figura 5. Flights(ϕ_2) usando o método para exibir pares de tuplas, à esquerda, e o método para exibir tuplas, à direita

serem projetados, que permite um melhor aproveitamento das otimizações. As versões com leitura única se saem melhor no tempo de execução total na maioria dos casos.

***Dataset TPC-H* - $\phi_4 : \neg(t_1.receiptdate \geq t_2.shipdate \wedge t_1.shipdate \leq t_2.receiptdate)$:** para este *dataset* utilizamos apenas o método de projeção de tuplas, pois a quantidade de violações encontradas pelo FACET é de aproximadamente 13 bilhões, o que torna inviável a projeção de pares de tuplas com os recursos disponíveis, pois essa quantidade de dados não caberia em disco. A quantidade de linhas neste *dataset* é 1000000, com 6 colunas. Todas as tuplas participam de alguma violação de ϕ_4. O resultado pode ser visto na Figura 6.

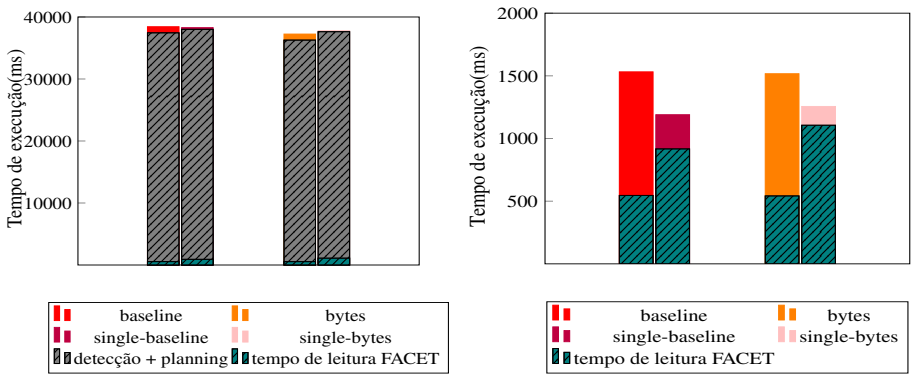


Figura 6. TPC-H(ϕ_4) usando o método para exibir tuplas, com tempos de detecção + *planning*, à esquerda, e usando método para exibir tuplas, ignorando os tempos de detecção + *planning*, à direita

Vemos aqui que, apesar da soma dos tempos de leitura e projeção das versões com leitura única serem menores, o tempo total de execução é muito semelhante ao das versões com releitura, o que indica um aumento, nas versões de leitura única, no tempo de detecção + *planning*. Suspeitamos que isto ocorra devido ao fato da tabela de projeção competir por memória com o processamento do FACET nas versões de leitura única.

***Dataset Tax* - $\phi_7 : \neg(t_1.AreaCode = t_2.AreaCode \wedge t_1.Phone = t_2.Phone)$:** este *dataset* contém 1000000 de linhas e 9 colunas. Neste caso, não há violações em relação à ϕ_7. O tempo de execução total pode ser verificado na figura 7.

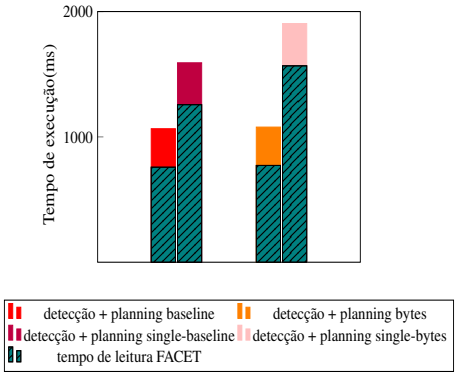


Figura 7. Diferença no tempo de execução entre as versões com leitura única e com releitura usando o *dataset* Tax(ϕ_7) e o método de projeção de tuplas

Este gráfico nos permite ver o resultado bastante positivo de uma otimização possível somente na versão com releitura. Nela, quando não há violações a serem projetadas, não construímos a tabela de projeção, mas somente escrevemos o cabeçalho da projeção no arquivo e encerramos a execução. Note que na versão com leitura única isso não é possível, uma vez que a tabela de projeção é construída antes da detecção.

4. Conclusão

As otimizações implementadas nas versões *bytes* são capazes de reduzir o tempo de execução em comparação com as versões *baseline*, principalmente naqueles casos onde há uma quantidade maior de dados a serem projetados e, especialmente, um número maior de colunas selecionadas. Em relação a estratégia de construção da tabela de projeção, acreditamos que a versão com leitura única seja, em geral, levemente superior, mas tenha sua performance reduzida quando há uma exigência maior de memória no processamento do FACET ou quando há poucas violações a serem projetadas. Outros experimentos devem ser realizados para obtermos mais evidências das conclusões obtidas, bem como para melhor delimitar os casos em que nossas otimizações são mais relevantes.

Referências

Chu, X., Ilyas, I. F., and Papotti, P. (2013). Discovering denial constraints. *Proc. VLDB Endow.*, 6(13):1498–1509.

Grefen, P. and de By, R. (1994). A multi-set extended relational algebra: a formal approach to a practical issue. In *Proceedings of 1994 IEEE 10th International Conference on Data Engineering*, pages 80–88.

Pena, E. H. M., de Almeida, E. C., and Naumann, F. (2022). Fast detection of denial constraint violations. *Proc. VLDB Endow.*, 15(4):859–871.

Pena, E. H. M., Lucas Filho, E. R., de Almeida, E. C., and Naumann, F. (2020). Efficient detection of data dependency violations. In *29th ACM CIKM*, page 1235–1244.

Rekatsinas, T., Chu, X., Ilyas, I. F., and Ré, C. (2017). Holoclean: Holistic data repairs with probabilistic inference. *Proc. VLDB Endow.*, 10(11):1190–1201.