LEONARDO FERREIRA LUCIANO
WENDEL CAIO MORO

PROJEÇÃO DAS VIOLAÇÕES DE DENIAL CONSTRAINTS DETECTADAS PELO FACET

(*versão pré-defesa, compilada em 8 de julho de 2023*)

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Prof. Dr. Eduardo Cunha de Almeida.

CURITIBA PR

2023

RESUMO

A visualização de violações de regras de qualidade de dados tem grande utilidade no processo de limpeza de dados. Uma das operações mais utilizadas para a visualização de violações de dados é a projeção de pares de tuplas que violam regras de dados. Esta operação é muito custosa em termos de tempo de processamento, visto que no pior caso, onde é preciso iterar sobre todas as combinações de pares de n tuplas, a complexidade é $O(n^2)$. Este trabalho apresenta e compara diferentes implementações para a projeção de violações de regras de qualidade de dados modeladas como *denial constraints* detectadas pelo algoritmo FACET.

Palavras-chave: Denial Constraints. Projeção. FACET.

ABSTRACT

Visualizing data quality rules is very useful in the data cleaning process. One of the most commonly used operations for visualizing data violations is the projection of pairs of tuples that violate data rules. This operation is very costly in terms of processing time, since in the worst case, where is necessary iterate over all combinations of pairs of n tuples, the complexity is $O(n^2)$. This work presents and compares different implementations for projecting violations of data quality rules modeled as *denial constraints* detected by the FACET algorithm.

Keywords: Denial Constraints. Projection. FACET.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Um banco de dados, como definido por Elmasri e Navathe (2016), é uma coleção de dados relacionados e tem três propriedades implícitas: representa algum aspecto do mundo real, é um conjunto logicamente coerente de dados com algum sentido inerente e, por fim, é projetado, construído e populado com dados tendo em vista um propósito específico. Este trabalho se concentra na primeira dessas propriedades.

Como define Fan (2015), a consistência dos dados que representam entidades do mundo real está relacionada à sua validade e integridade. Inconsistências nos dados, por sua vez, podem ser identificadas a partir de violações a restrições de integridade definidas para tais dados (Fan, 2015). Além disso, como apresentado por Chu et al. (2013), restrições de integridade vem sendo utilizadas para melhorar a qualidade dos dados, inclusive sendo utilizadas no processo de limpeza de dados.

Dito isso, têm-se que as regras de integridade de dados, sendo utilizadas como regras de qualidade de dados, têm um papel importante em garantir que um banco de dados, e seus dados mais especificamente, representem aspectos do mundo real, bem como em aumentar a qualidade de tais dados, de forma que se torna relevante a descoberta de tais regras, bem como a detecção de violações a elas e a posterior correção de tais violações. Este trabalho foca na segunda destas etapas, ou seja, na detecção de violações a regras de integridade de dados, apresentando uma ferramenta para melhorar a visualização das violações detectadas pelo algorítimo FACET.

Serão utilizados exemplos baseados nos apresentados por Pena et al. (2022) para ilustrar os conceitos, definições e implementações expostos neste trabalho. Dada a Tabela 1.1, pode-se definir, por exemplo, as seguintes regras de qualidade:

- 1. Cada funcionário tem um valor único de ID
- 2. Funcionários não podem supervisionar seus próprios supervisores
- 3. Dados dois funcionários de um mesmo departamento, o funcionário mais antigo não pode receber o menor salário

	ID	Name	Dept	StartDate	Salary	SID
t_1	100	C. Gardner	Sales	2012	3000	100
t_2	101	R. Geller	Research	2014	8000	102
t_3	102	D. Brown	Research	2014	6000	101
t_4	103	H. McCoy	Research	2015	8000	101

Tabela 1.1: Tabela Funcionários

Apesar de existirem diversos formalismos sendo estudados para representar regras de qualidade de dados (Fan, 2015), este trabalho foca nas *Denial Constraints* (DCs), que além de

serem expressivas a ponto de excederem os limites de outros formalismos (Chu et al., 2013), vem sendo, como apontado por Pena et al. (2022), utilizadas em trabalhos recentes na área de limpeza de dados. Vários trabalhos nessa área utilizam *queries* SQL para fazer a detecção de violações de regras de qualidade (Fan et al., 2008) (Fan et al., 2021) (Geerts, 2020) (Rekatsinas et al., 2017), que podem ser modeladas como DCs. Porém, é comum que para fazer essa detecção seja necessário avaliar predicados de desigualdade durante a execução das *queries*, predicados esses que acabam resultando em *non-equi joins*, cujo processamento é caro em termos de tempo (Pena et al., 2022). Há também trabalhos que apresentam soluções específicas para o problema da detecção eficiente de violações à DCs, como o VioFinder (Pena et al., 2020) e o FACET (Pena et al., 2022). Este trabalho será focado neste último, que é hoje a solução mais eficiente para a detecção de violações em DCs, segundo os experimentos apresentados no artigo onde foi proposto.

O FACET se vale de uma nova heurística para avaliar o custo de processamento de cada um dos predicados que compõem as DCs, de maneira que monta seu plano de avaliação de forma a adiar o processamento de predicados que são mais custosos. Além disso, o FACET também utiliza diferentes algoritmos e estruturas de dados para processar cada uma das classes de predicados (igualdades, diferenças e desigualdades), selecionando-os de forma a otimizar, para cada classe, o tempo de processamento. Dessa maneira, o FACET alcança resultados consideravelmente superiores em relação às alternativas desenvolvidadas até então para o problema, reduzindo o tempo de detecção das violações às regras de qualidade de dados.

Uma limitação do FACET, entretanto, está no formato de sua saída, que se restringe a informar o número de violações encontradas, sem apresentar em quais pares de tuplas tais violações ocorrem. A informação de quais são os pares de tuplas que violam uma determinada DC pode ser utilizada por engenheiros no processo de correção dos dados, seja de forma manual ou via ferramentas de limpeza da dados como, por exemplo, o HoloClean (Rekatsinas et al., 2017). Além disso, tal infomação pode ajudar na avaliação da importância de cada uma das DCs proposta para um determinado conjunto de dados. Assim sendo, o objetivo deste trabalho é estender o FACET com a implementação da projeção dos pares de tuplas que violam uma dada DC. Para realizar tal projeção, é necessário iterar por todos os pares de tuplas que violam a DC especificada, ou seja, a complexidade de tempo do problema é $O(n^2)$, de forma que o desafio consiste em encontrar uma forma de implementá-la de maneira eficiente em relação ao tempo de processamento. Para tanto, aqui propõe-se e avalia-se diferentes implementações para a projeção, cada uma com a premissa de reduzir o uso de memória ou o processamento por iteração. Além disso, aproveitando as estruturas de dados utilizadas para a projeção dos pares de tuplas, foi adicionado ainda outro método de visualização dos resultados, uma projeção somente das tuplas que estão envolvidas em alguma violação à DC especificada.

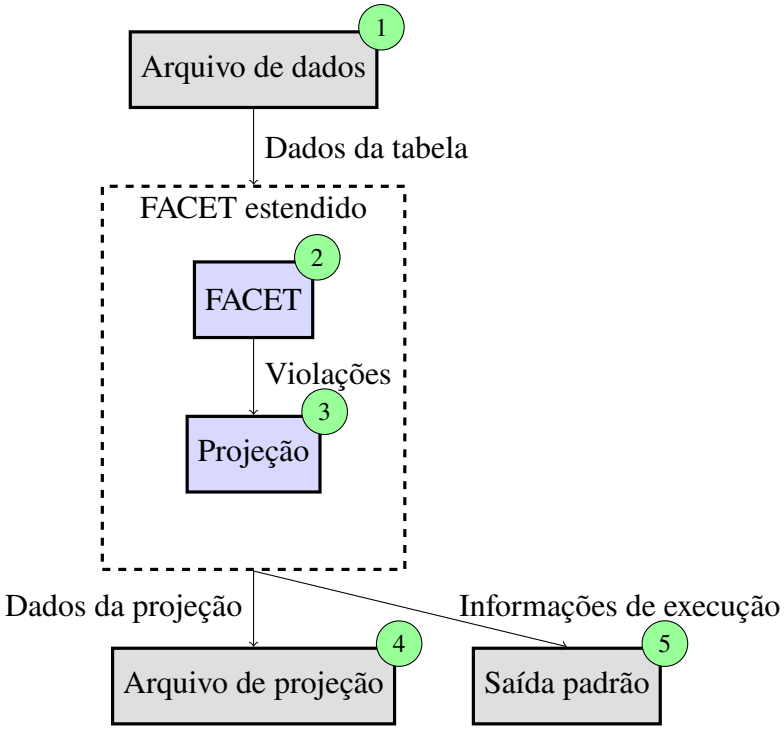
Uma visão geral da arquitetura da solução proposta neste trabalho será dada no capítulo 2; algumas definições, utilizadas neste trabalho como um todo, serão feitas no capítulo 3; o FACET e seu design serão descritos no capítulo 4; a proposta para os operadores de projeção no

FACET, com suas definições e o design da solução proposta, serão apresentados no capítulo 5; o capítulo 6 descreverá os experimentos realizados para avaliar a solução proposta; por fim, o capítulo 7 apresentará algumas melhorias que podem ser exploradas futuramente.

2 VISÃO GERAL

Neste capítulo, será apresentada uma visão geral do *design* e do funcionamento da solução, incluindo em como ela se integra e estende o FACET. Além disso, será mostrado o funcionamento básico da projeção e os principais componentes utilizados na solução, bem como comentado rapidamente sobre as variações de implementações e abordagens utilizadas em alguns desses componentes.

Figura 2.1: Design da solução



A Figura 2.1 mostra o diagrama com o design comum da solução proposta. De maneira geral, é adicionado um novo módulo, representado pelo componente "Projeção" no diagrama, a ser executado após o processamento do FACET, e que será reponsável por escrever no arquivo de projeção indicado pelo usuário na entrada os dados das violações detectadas pelo FACET. Voltando ao diagrama, uma execução da versão estendida do FACET começa em 1, com o arquivo de dados fornecido na entrada sendo lido pelo FACET estendido. Em 2, o FACET irá realizar seu processamento, detectando as violações à DC informada contidas nos dados lidos e enviando-as para o módulo de projeção proposto neste trabalho. Em 3, após todas as violações terem sido detectadas, o novo módulo vai iniciar a projeção de acordo com o método escolhido na entrada. Em 4, os valores das tuplas projetadas são escritos no arquivo de projeção, também informado na entrada. Em 5, o FACET estendido finaliza a escrita de seus *logs* na saída padrão e encerra a execução.

O componente “Projeção”, por sua vez, é composto por um laço principal, que irá iterar sobre os pares de identificadores de tuplas que o FACET detectou serem violações à DC analisada, e por uma estrutra de dados auxiliar, chamada tabela de projeção, que será utilizada para armazenar os dados de cada uma das tuplas envolvidas em alguma violação. Optou-se por criar uma estrutura de dados separada da tabela já construída pelo FACET para seu processamento, a fim de utilizar de maneira mais eficiente a memória, dado que a tabela do FACET armazena os dados por colunas e não por tuplas, como é o mais adequado para o uso que será feito dos dados.

Serão apresentadas e avaliadas diferentes implementações da tabela de projeção, sendo elas: uma implementação básica, onde cada tupla é armazenada como um *array* de *strings*, com cada string correspondendo ao valor de uma coluna que deve ser projetada; uma implementação onde cada uma das tuplas é armazenada como um *array* de *integers*, com a utilização de um dicionário para mapear valores do tipo *string* para *integers*; e, por fim, duas implementação onde cada uma das tuplas é armazenada como um único *array* de *bytes* contendo todos os dados que devem ser projetados, com uma pequena diferença na construção dos *arrays* de *bytes* entre as duas versões.

Além disso, serão apresentadas duas formas distintas de construção da tabela de projeção. A primeira delas a constrói junto com a tabela utilizada pelo FACET para a detecção de violações, o que implica ambas as tabelas compartilharem a memória durante a execução. Já a outra implementação constrói a tabela de projeção somente ao fim da execução do FACET, o que implica a releitura do arquivo de dados, mas evita a concorrência por memória durante a deteção de violações.

3 DENIAL CONSTRAINTS

3.1 DEFINIÇÃO

Uma *Denial Constraint* (DC) expressa um conjunto de predicados que não podem ser verdadeiros juntos para qualquer combinação de tuplas distintas. Elas são particularmente úteis pois, como apontado por Chu et al. (2013), este formalismo excede os limites do poder expressivo de outros formalismos existentes, além de permitirem aplicações eficientes em vários cenários, incluindo a descoberta de regras de qualidade de dados. O objetivo geral de DCs é de identificar relacionamentos conflituosos de combinações de valores de colunas com um conjunto de predicados.

Será utilizada a definição de DC apresentada em Pena et al. (2022). Sejam os predicados da forma $p : t.A\ o\ t'.B$, em que A, B são colunas de uma tabela r com esquema R e n tuplas; (t, t') é um par distinto de tuplas de r; e $o \in \{=, \neq, <, >, \leq, \geq\}$ é um conjunto de operadores de comparação. Dessa maneira, uma DC pode ser formulada da seguinte forma:

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

Cada DC expressa um conjunto de predicados que não podem ser verdadeiros juntos para qualquer par de tuplas. Um par de tuplas (t, t') satisfaz uma DC ϕ se avaliar para falso em pelo menos um dos predicados de ϕ, ou, caso contrário, o par de tuplas violará a DC ϕ, o que implica que a tabela r é inconsistente em relação à ϕ.

Na Tabela 3.1 há exemplos de algumas DCs. Neste caso, as DCs apresentadas representam, respectivamente, as três regras de qualidade de dados que foram definidas anteriormente para a Tabela 1.1.

$\phi_1 : t.ID = t'.ID$
$\phi_2 : t.ID = t'.SID \wedge t.SID = t'.ID$
$\phi_3 : t.Dept = t'.Dept \wedge t.StartDate < t'.StarDate \wedge t.Salary < t'.Salary$

Tabela 3.1: Algumas DCs da tabela Funcionários

3.1.1 DCs aproximadas

Uma DC sobre um determinado conjunto de dados é válida quando não há nenhum par de tuplas que a viole, mas esta restrição pode ser relaxada, de forma que será definido, como feito em Pena et al. (2020), uma *Denial Constraint* aproximada sobre uma relação r como sendo uma DC que é violada por um número limitado de pares de tuplas. Com base na proporção de

pares de tuplas violadas por uma DC aproximada ϕ sobre r, é calculado seu grau de aproximação, definido como:

$$g1(\phi, r) = \frac{|(t, t') \in r | (t, t') \text{ não satisfaz } \phi|}{|r| \cdot (|r| - 1)}$$

Uma vez definido $g1$, diz-se que, para ϵ tal que $0 \leq \epsilon < 1$, a DC ϕ é $\epsilon - aproximada$ em r se, e somente se, $g1(\phi, r) < \epsilon$.

Dada uma DC aproximada, seria possível utilizar o FACET para detectar todos os pares de tuplas que a violam. O objetivo deste trabalho é implementar a projeção de tais pares justamente para facilitar a correção dessas violações.

3.2 DCFINDER

O foco deste trabalho é na detecção de violações à DCs, mas, a fim de construir uma visão mais completa do processo de limpeza de dados, será descrito à seguir, de maneira sucinta, o algoritmo DCFinder, apresentado em Pena et al. (2020), e que é uma ferramenta capaz de identificar as DCs relacionadas a um determinado conjunto de dados.

3.2.1 Visão geral

Primeiramente, é importante notar que a descoberta de DCs é um processo computacionalmente caro devido ao tamanho do espaço de predicados P, que corresponde ao conjunto de todos os possíveis predicados p para um conjunto de dados. Note que qualquer subconjunto de predicados pertencente a P é um candidato a ser uma DC, de forma que apresentar uma solução que execute em tempo e espaço factíveis é um desafio. A solução apresentada por Pena et al. (2020), cujos resultados, segundo os experimentos apresentados no artigo onde o DCFinder foi proposto, superam de maneira significativa os de soluções existentes até então. Seu funcionamento pode ser dividido nas seguintes etapas:

- Inicialização de evidências
- Reconstrução de evidências
- Busca de cobertura mínima

As duas primeiras etapas têm por objetivo serem uma alternativa mais eficiente, em comparação com a iteração sobre todos os pares de tuplas, para a construção do conjunto de evidências. Já a terceira etapa se vale de uma heurística a fim de encontrar uma cobertura mínima para o conjunto de evidências previamente calculado.

Para facilitar a compreensão de cada uma das etapas, serão utilizados exemplos baseados na Tabela 3.2, primeiramente apresentada no artigo original e que é utilizada somente nesta seção para ilustrar o funcionameto do DCFinder.

	Name	Phone	Position	Salary	Hired
t_0	W. Jones	202-222	Developer	\$2.000	2012
t_1	B. Jones	202-222	Developer	\$3.000	2010
t_2	J. Miller	202-333	Developer	\$4.000	2010
t_3	D. Miller	202-333	DBA	\$8.000	2010
t_4	W. Jones	202-555	DBA	\$7.000	2010
t_5	W. Jones	202-222	Developer	\$1.000	2012

Tabela 3.2: Tabela FuncionárioDCFinder

A Tabela 3.3 exibe o espaço de predicados considerado pelo DCFinder para a Tabela 3.2. Note que nem todos os predicados estão presentes, isto porque o DCFinder aplica, a fim de reduzir o espaço de busca, as seguintes restrições:

- Predicados com atributos diferentes só são considerados quando ambos os atributos tem o mesmo tipo e compartilham ao menos 30% dos valores
- Predicados com atributos não numéricos só são considerados quando o operador que utilizam está no conjunto $\{=, \neq\}$

$p_1 : t_x.Name = t_y.Name$	$p_{10} : t_x.Salary \leq t_y.Salary$
$p_2 : t_x.Name \neq t_y.Name$	$p_{11} : t_x.Salary > t_y.Salary$
$p_3 : t_x.Phone = t_y.Phone$	$p_{12} : t_x.Salary \geq t_y.Salary$
$p_4 : t_x.Phone \neq t_y.Phone$	$p_{13} : t_x.Hired = t_y.Hired$
$p_5 : t_x.Position = t_y.Position$	$p_{14} : t_x.Hired \neq t_y.Hired$
$p_6 : t_x.Position \neq t_y.Position$	$p_{15} : t_x.Hired < t_y.Hired$
$p_7 : t_x.Salary = t_y.Salary$	$p_{16} : t_x.Hired \leq t_y.Hired$
$p_8 : t_x.Salary \neq t_y.Salary$	$p_{17} : t_x.Hired > t_y.Hired$
$p_9 : t_x.Salary < t_y.Salary$	$p_{18} : t_x.Hired \geq t_y.Hired$

Tabela 3.3: Espaço de predicados para a tabela FuncionáriosDCFinder

Pode-se ver, na Tabela 3.4, um exemplo de DCs $\epsilon - aproximadas$ descobertas pelo DCFinder para a Tabela 3.2, com $\epsilon = 0,01$. Note que a DC ϕ_1 representa uma restrição de unicidade de valor no atributo *Salary*, enquanto outras DCs representam dependências funcionais, como definidas em Elmasri e Navathe (2016), que podem ser vistas na Tabela 3.5. Além disso, ϕ_9 explicita uma possível regra de negócio mais complexa, pois define que, para dois funcionários com a mesma posição, o funcionário mais antigo não pode receber menos que o mais novo.

Por fim, para prosseguir, serão necessárias algumas definições, todas apresentadas em Pena et al. (2020). Dada uma relação r, o espaço de predicados P de r e as tuplas t, t' de r, uma evidência é definida como o conjunto $e_{t,t'} = \{p | p \in P, (t, t') \text{ satisfaz } p\}$. A partir daí, é definido o conjunto de evidências E_r como $E_r = \{e_{t,t'} | \forall t, t' \in r\}$. A definição de cobertura mínima para um conjunto de evidências E_r está fora do escopo deste trabalho, mas é relevante saber que uma cobertura mínima de DCs pode ser obtida a partir do conjuto de todas as coberturas mínimas de E_r.

$\phi_1 : \neg(p_7)$
$\phi_2 : \neg(p_{10} \wedge p_{15})$
$\phi_3 : \neg(p_1 \wedge p_3 \wedge p_6)$
$\phi_4 : \neg(p_1 \wedge p_3 \wedge p_{14})$
$\phi_5 : \neg(p_1 \wedge p_4 \wedge p_5)$
$\phi_6 : \neg(p_1 \wedge p_4 \wedge p_{13})$
$\phi_7 : \neg(p_1 \wedge p_5 \wedge p_{14})$
$\phi_8 : \neg(p_1 \wedge p_6 \wedge p_{13})$
$\phi_9 : \neg(p_1 \wedge p_{10} \wedge p_{16})$
$\phi_{10} : \neg(p_3 \wedge p_6 \wedge p_{14})$
$\phi_{11} : \neg(p_2 \wedge p_3 \wedge p_5 \wedge p_{13})$
$\phi_{12} : \neg(p_3 \wedge p_5 \wedge p_{10} \wedge p_{16})$

Tabela 3.4: DCs aproximadas com grau $0,01$ da tabela FuncionáriosDCFinder

ϕ_3 : Name, Phone $\rightarrow$ Position
ϕ_4 : Name, Phone $\rightarrow$ Hired
ϕ_5 : Name, Position $\rightarrow$ Phone
ϕ_6 : Name, Hired $\rightarrow$ Phone
ϕ_7 : Name, Position $\rightarrow$ Hired
ϕ_8 : Name, Hired $\rightarrow$ Position
ϕ_{11} : Phone, Position, Hired $\rightarrow$ Name

Tabela 3.5: Mapeamento de DCs da tabela FuncionáriosDCFinder para DFs

3.2.2 Inicialização de evidências

Para construir o conjunto de evidências, o DCFinder se baseia na seletividade dos predicados p para criar uma evidência inicial comum a todos os pares de tuplas. Os predicados p tais que $p.o \in \{\neq, <, >, \leq, \geq\}$ tem menor seletividade do que predicados p' tais que $p'.o \in \{=\}$, ou seja, $\forall e \in E_r$, é mais provável que $p \in e$ do que $p' \in e$.

Sendo assim, as evidências são todas inicializadas contendo todos os predicados de menor seletividade, ou seja, com os predicados cuja operação está no conjunto $O = \{\neq, <, \leq\}$. Note que as operações $\{>, \geq\}$ tem a mesma seletividade de $\{<, \leq\}$, mas não foram incluídas em O pois isto só aumentaria o número de reconstruções a serem feitas na próxima etapa.

Ao fim desta etapa é construído um mapeamento B indo de cada par de tuplas para sua evidência, que, neste momento, é a evidência comum gerada a partir das regras citadas.

3.2.3 Reconstrução de evidências

As evidências relacionadas a cada par de tuplas na etapa anterior não necessariamente estão corretas, mas sim foram construídas de forma a tentar minimizar o número de correções necessárias. Esta etapa tem por objetivo justamente fazer as correções necessárias nas evidências.

Para poder corrigir as evidências incorretas, o DCFinder cria um mapeamento T de cada predicado p tal que $p.o \in \{=, >\}$ para os pares de tuplas que satisfazem p. Para a criação

desse mapeamento são utilizadas estruturas de dados auxiliares chamadas PLIs, que agrupam identificadores de tuplas de acordo com seu valor em relação a um atributo e, dessa forma, tornam mais eficiente a construção de T.

Por fim o algoritmo vai, para cada par de tuplas em $T[p]$, corrigir a evidência correspondente ao par no mapeamento B, de forma a deixá-la correta. Uma vez que todas as evidências estiverem corretas, as mesma são adicionadas ao conjunto de evidências junto com sua respectiva multiplicidade, de forma que, para qualquer evidência, será possível saber o número de pares de tuplas aos quais ela corresponde.

3.2.4 Busca de cobertura mínima

A última etapa definirá o conjunto MC, que conterá todas as coberturas mínimas aproximadas Q de E_r. Cada Q será computado de maneira incremental com o uso de uma DFS (Cormen et al., 2009), onde a cada nó um novo predicado $p \in P$ é então adicionado a Q, de forma que a recursão só se encerra quando todos os predicados foram utilizados ou quando Q já é uma cobertura aproximada de E_r. Neste último caso, o DCFinder verifica se Q é uma cobertura mínima e, em caso afirmativo, adiciona-o ao conjunto MC.

Para tentar reduzir o tempo de execução, cada predicado p que será adicionado a Q é escolhido de acordo com uma heurística. Tal heurística seleciona o predicado p que está presente no maior número de evidências de R_r para ser adicionado a Q, na esperança de encontar o mais rápido possível uma cobertura para E_r.

3.3 CLASSIFICANDO DCS

Dado que um conjunto de dados pode contar com uma grande quantidade de DCs, é relevante a questão de encontrar métricas capazes de estimar a importância de cada uma dessas DCs. Dito isso, são apresentadas aqui duas métricas, *coverage* e *succinctness*, que se propõem a classificar DCs de acordo com sua relevância. Cabe ressaltar, entretanto, que a classificação por tais métricas, bem como por outras existêntes, não exclui o fator humano do processo, uma vez que as métricas hoje conhecidas são incapazes de determinar com precisão a importância de uma DC.

Vale citar também que o DCFinder implementa um classificador de DCs de acordo com as duas métricas que serão apresentadas aqui, além de por uma terceira que não será abordada.

3.3.1 Coverage

Segundo Li et al. (2022), *Coverage* mede a significância estatística de uma DC. Têm-se que, quanto maior o número de predicados de uma DC satisfeitos pelos pares de tuplas analizados, mais suporte a DC em questão possui.

Sendo ϕ uma DC de uma relação r, $\phi.Pred$ é definido como o conjunto de predicados de ϕ. Além disso, *k-evidence* (kE) para ϕ é o conjunto de pares (t, t') de tuplas distintas

de r que satisfazem k predicados de $\phi.Pred$. Dado isso, pela definição de DC, $0 \leq k \leq \phi.Pred - 1$, de forma que é possível introduzir um parâmetro de peso $w(k)$ para kE tal que $w(k) = (k+1)/|\phi.Pred|$. Com isso, $Coverage(\phi)$ é definido como:

$$Coverage(\phi) = \frac{\sum_{k=0}^{|\phi.Pred|-1} |kE| * w(k)}{\sum_{k=0}^{|\phi.Pred|-1} |kE|}$$

Como o parâmetro de peso $w(k)$ varia entre 0 e 1, o mesmo ocorrerá com o valor de $Coverage(\phi)$, sendo que valores mais próximos de 1 indicam que ϕ tem uma maior significância estatística.

Para compreender melhor o cálculo de *coverage*, será utilizada novamente a Tabela 3.2 e a DC ϕ_2, que pode ser consultada na Tabela 3.4. Primeiramente, será calculado kE para cada k entre 0 e 1, ou seja, serão encontrados os pares de tuplas que satisfazem zero predicados ($0E$) e um predicado ($1E$) de ϕ_2:

$$0E = \{(0,5), (2,1), (3,1), (3,2), (3,4), (4,1), (4,2)\}$$
$$1E = \{(0,1), (0,2), (0,3), (0,4), (1,0), (1,2), (1,3), (1,4), (1,5), (2,0),$$
$$(2,3), (2,4), (2,5), (3,0), (3,5), (4,0), (4,3), (4,5), (5,0), (5,1),$$
$$(5,2), (5,3), (5,4)\}$$

A partir deste resultado, têm-se que $|0E| = 7$ e $|1E| = 23$. Daí, $w(0) = 0.5$ e $w(1) = 1$:

$$Coverage(dc_2) \approx 0.883$$

3.3.2 Succinctness

De acordo com Li et al. (2022), *Succinctness*, denotado $Succ(\phi)$, representa a menor largura possível de uma DC ϕ dividida por sua própria largura $Len(\phi)$. Além disso, segundo Chu et al. (2013), é possível garantir que a escala da Succintness esteja entre 0 e 1.

Pode-se formular a Succintness da seguinte maneira:

$$Succ(\phi) = Min(\{Len(\phi)|\forall\phi\})/Len(\phi)$$

Será utilizado o número de predicados de uma DC ϕ como heurística. Considere o exemplo da Tabela 3.1, onde há 3 DCs. Neste caso, $Len(\phi_1) = 1 < Len(\phi_2) = 2 < Len(\phi_3) = 3$, e $Succ(\phi_1) = 1$, $Succ(\phi_2) = 0.5$, e $Succ(\phi_3) \approx 0.333$.

4 FACET

O FAst Constraint-based Error DeTector (FACET), proposto em Pena et al. (2022), é, como dito, um algoritmo que permite a detecção eficiente de violações de DCs. Os principais pontos que permitem ao FACET superar as soluções anteriores em performance são os seguintes:

- Uso de estruturas de dados adaptadas ao algoritmo de refinamento utilizado;
- Uso de um novo algoritmo de refinamento de desigualdades;
- Uso de uma nova heurística para planejar a ordem de avaliação dos predicados.

A seguir, será apresentada uma visão geral de seu *design*, mas antes seu funcionamento será ilustrado utilizando alguns exemplos, todos eles considerando a Tabela 1.1 como parte da entrada para a execução do FACET.

Primeiramente, considere uma execução utilizando a DC ϕ_1 como entrada. Neste caso, o FACET retornaria que não há nenhuma violação nos dados. De fato, todos os ID da tabela são únicos, ou seja, não há nenhum par de tuplas (t, t') para o qual $t.ID = t'.ID$.

Agora será verificada a resposta do FACET ao avaliar a DC ϕ_2. Desta vez, a execução do algoritmo detectaria a existência de duas violações da referida DC. Note que os empregados das tuplas t_2 e t_3 se supervisionam mutuamente, de forma que, para os pares $(t, t') \in \{(t_2, t_3), (t_3, t_2)\}$, $t.ID = t'.SID$ e $t.SID = t'.ID$, ou seja, a expressão $\neg(t.ID = t'.SID \wedge t.SID = t'.ID)$ é falsa e, portanto, ϕ_2 é violada pelos pares de tuplas (t_2, t_3) e (t_3, t_2). Além disso, veja que $\neg(t.ID = t'.SID \wedge t.SID = t'.ID)$ também é falsa para $(t, t') = (t_1, t_1)$, mas esse par não se caracteriza uma violação de ϕ_2, uma vez que, neste caso, $t = t'$.

Por fim, será analisado a execução do FACET ao avaliar a DC ϕ_3. Agora a saída apontaria a existência de uma violação da DC. Consultando a tabela, pode-se conferir que somente o par de tuplas (t_3, t_4) viola ϕ_3, uma vez que t_3 e t_4 trabalham no mesmo departamento, com t_3 tendo uma data de início anterior a t_4 e recebendo um salário menor.

4.1 REFINAMENTOS

A seguir, será apresentada a definição de alguns componentes utilizados pelo FACET em sua implementação, como definido por Pena et al. (2022).

- **Representação de intermediários:** o FACET processa representações compactas, em vez de processar tuplas individualmente. Sendo $tids$ a representação de um conjunto de identificadores, em que as tuplas de uma tabela r são representadas por $tids_r = \{t_1, ..., t_n\}$, o par ordenado $(tids_1, tids_2)$ representa o conjunto de pares de tuplas (t, t') tais que $t \in tids_1$, $t' \in tids_2$ e $t \neq t'$. Como exemplo, o par $(\{t_1, t_2\}, \{t_2, t_3\})$ representa os pares de tuplas (t_1, t_2), (t_1, t_3), (t_2, t_3).

- **Operador de refinamento:** o operador de refinamento toma como entrada pares $(tids_1, tids_2)$ e um predicado p, e retorna o conjunto de pares $(tids'_1, tids'_2)$ que representam todos os subconjuntos de pares de tuplas de $(tids_1, tids_2)$ que satisfazem p.

- ***Pipeline* de refinamento:** Seja uma tabela r e o refinamento de um predicado p_1, seguido de um predicado p_2. Para montar suas estruturas de dados auxiliares, p_1 consumirá o par $(tids_l, tids_r)$, visto que é o primeiro do *pipeline*. A partir dessas estruturas são encontrados os pares de tuplas que satisfazem p_1, e então, de forma incremental, são criados os pares de identificadores de tuplas que servirão de entrada para a próxima etapa. Na etapa do predicado p_2, os dois lados de cada par de identificadores de tuplas, a partir do refinamento anterior, são consumidos para a construção de uma nova estrutura auxiliar. Encontrar pares de tuplas qualificados segue o mesmo processo anterior. Os pares de tuplas resultantes dessa etapa representam as tuplas de pares satisfazendo p_1 e p_2, ou seja, representam as violações da DC $\phi : \neg(p_1 \wedge p_2)$. Um exemplo pode ser apresentado com a Tabela 1.1 e o *pipeline* com os predicados $p_1 : t.Dept = t'.Dept$ e $p_2 : t.Salary < t'.Salary$, respectivamente. O refinamento do predicado p_1 produz o par de tids $(\{t_2, t_3, t_4\}, \{t_2, t_3, t_4\})$. O refinamento do predicado p_2, por sua vez, consome este par e produz um novo par de tids $(\{t_3\}, \{t_2, t_4\})$.

4.2 ALGORITMOS

De forma semelhante ao que fazem outras soluções especializadas na detecção de violações de DCs, o FACET divide os predicados em classes a depender da operação realizada em cada um deles. As classes consideradas são as seguintes:

- **Igualdades:** predicados que utilizam o operador =
- **Diferenças:** predicados que utilizam o operador $\neq$
- **Desigualdades:** predicados que utilizam algum dos operadores em $\{<, >, \leq, \geq\}$

Cada uma das classes está relacionada a um algoritmo ou conjunto de algoritmos que será utilizado para realizar os refinamentos associados a elas. Tais algoritmos serão apresentados de maneira sucinta neste trabalho.

Além disso, o FACET representará internamente os intermediários usando diferentes estruturas de dados, a depender do algoritmo que processará tais intermediários. Para igualdades, serão utilizados *arrays* de *integers*, que atendem bem as necessidades do algoritmo utilizado sem acrescentar *overheads* de memória e processamento. As diferenças e desigualdades, por outro lado, devido à necessidade de processar diversas uniões e diferenças de conjuntos, se beneficiam do uso de *bitmaps* comprimidos, estruturas de dados mais adequadas para operações *bit*-a-*bit*.

4.2.1 Igualdades

A implementação do processamento de refinamentos com predicados de igualdade utiliza uma abordagem baseada no *hash-join* tradicional, consistindo em duas etapas, uma de construção, onde é criada uma estrutura de dados auxiliar, e outra de sondagem, onde a estrutura de dados construída na etapa anterior é consultada e alterada.

Dado o predicado $p : t.A = t'.B$ e assumindo, sem perda de generalidade, que a coluna A será utilizada na etapa de construção, enquanto que a coluna B será usada na etapa de sondagem, vamos descrever o funcionamento do algoritmo. Primeiramente o algoritmo criará, a partir das tuplas do lado esquerdo da entrada, uma tabela *hash* tal que cada entrada é do formato $\langle k, (tids_1, tids_2)\rangle$, onde $tids_1$ contém as tuplas do lado esquerdo da entrada cujo valor em A é igual a k e estando $tids_2$ vazio. Seguindo para a etapa de sondagem, o algoritmo vai iterar pelas tuplas do lado direito da entrada, adicionando cada tupla tal que o valor em B é k ao conjunto $tids_2$. Por fim, a partir de uma iteração pelas entradas da tabela *hash* criada na construção, serão enviados para a próxima etapa da *pipeline* os pares $(tids_1, tids_2)$ que contenham ao menos um par de tuplas distintas.

4.2.2 Diferenças

O refinamento de predicados de diferenças é similar ao de igualdades, seguindo uma abordagem baseada em *hash-join*, porém com a diferença de como a saída é construída.

Da mesma forma que nas igualdadades, a coluna A será utilizada na etapa de construção. Seguindo o mesmo processo realizado para os refinamentos de igualdades, será criada uma tabela *hash* tal que cada entrada da tabela possui uma chave k associada à um par de $tids$ $(tids'_1, tids'_2)$. Note que um $tids'_2$ vazio indica que nenhuma das tuplas do lado direito possui o valor da chave k de B associado ao mesmo valor de k de A das tuplas do lado esquerdo. Em suma, para toda tupla do lado esquerdo, existe um valor de A que é diferente de todo valor de B das tuplas do lado direito, dado isso, o par $(tids'_1, tids_2)$ é enviado para a próxima etapa da *pipeline*.

Em caso de um $tids'_2$ não vazio, temos uma situação distinta, onde, para toda tupla do lado esquerdo, existe um valor k de A que é igual ao valor k de B das tuplas do lado direito. Para esse caso, o que será feito é criar o conjunto $tids''_2 = tids_2 \setminus tids'_2$, enviando o par $(tids'_1, tids''_2)$ para a próxima etapa do refinamento.

4.2.3 Desigualdades

Diferente do que ocorre com igualdades e diferenças, o FACET conta com a implementação de mais de um algoritmo para o processamento de desigualdades. Tal decisão se justifica pelo fato das desigualdades serem frequentemente os predicados cujo processamento é mais custoso em termos de tempo, de forma que, contando com diferentes opções de algoritmos, o FACET pode escolher utilizar aquele cujos benefícios e limitações oferecem um balanço mais positivo em termos de performance.

Para as explicações de cada um dos algoritmos, considere o predicado $p : t.A \; o \; t'B$, onde $o \in \{<, >, \leq, \geq\}$.

4.2.3.1 IEJoin

O algoritmo IEJoin é proposto por Zuhair Khayya (2015), recebendo como entrada uma *query* e duas condições de $join$ de desigualdade. Três vetores, um de permutação e dois de deslocamento, são utilizados na execução do algoritmo, que também utiliza um *bitmap* para determinar a satisfação de predicados. Durante a execução, primeiramente os valores de atributo são ordenados para a junção, e então é computado o vetor de permutação e os vetores de deslocamento. Utilizando os vetores, os valores da tupla à direita são acessados, e para cada tupla acessada que satisfaz o segundo predicado, um *bitmap* é marcado. Após esssa etapa, o algoritmo itera pelo vetor de deslocamento, com base na entrada mais à esquerda, checando os bits marcados para determinar se as tuplas também satisfazem o primeiro predicado.

Na etapa de geração de pares, o FACET utiliza uma estratégia proposta por Tobias Bleifuß (2017). Essa técnica propõe que durante a checagem do *bitmap*, os pares de tuplas são inseridos em pares de saídas. Quando tuplas produzirem resultados diferentes, independente da entrada, o par corrente pode ser inserido na saída e seguir para a iteração de um novo par de saída.

O custo do algoritmo é baseado principalmente na etapa de ordenação. Com entradas de resultados qualificativos, no entanto, a iteração pelos vetores e checagem do bitmap se torna a parte mais dominante em relação ao custo, o que pode acarretar em baixo desempenho em predicados de baixa seletividade.

4.2.3.2 Hash-Sort-Merge (HSM)

Este algoritmo foi inicialmente proposto em Pena et al. (2020) e, como o nome indica, utiliza uma abordagem baseada no *sort-merge* tradicional. Também aqui, como no algoritmo de processamento de igualdades, a abordagem consiste em duas etapas, sendo uma de construção e outra de *merge*.

A etapa de construção terminará com duas tabelas *hash* criadas, uma tabela para a coluna A, construída a partir da entrada do lado esquerdo, e outra tabela para a coluna B, construída a partir da entrada do lado direito. Na tabela da coluna A, cada entrada será do formato $\langle k, tids_A \rangle$, onde $tids_A$ contém as tuplas do lado esquerdo da entrada tais que seus valores em A são iguais a k. As entradas da tabela da coluna B são semelhantes às entradas da tabela de A, mas considerando a coluna B e as tuplas do lado direito da entrada. Ambas as tabelas estão ordenadas em relação aos valores das chaves, ordenações essas que dependem da operação de p. Além disso, foram removidas de ambas as tabelas as entradas localizadas nas extremidades e que garantidamente não poderiam gerar nenhum par.

Na etapa de *merge*, as tabelas serão percorridas de maneira intercalada a fim de formar os pares $(tids_1, tids_2)$ que satisfaçam o predicado p. Tais pares serão formados incrementalmente,

de forma que a cada iteração serão utilizados os valores de iterações anteriores e operações de união lógica (OR), tentando sempre manter os pares com o maior tamanho possível a fim de reduzir o número de pares enviados para a próxima etapa da *pipeline* de refinamentos e também para aproveitar melhor a compressão dos *bitmaps*.

Este algoritmo, por reutilizar resultados de iterações anteriores para formar novos pares, consegue reduzir os efeitos negativos de predicados de baixa seletividade, contornando uma das limitações do IEJoin. Entretanto, para colunas de alta cardinalidade, o HSM utiliza um número muito grande de operações de união lógica, o que acaba por reduzir a performance deste algoritmo.

4.2.3.3 Binning-Hash-Sort-Merge (BHSM)

Este algoritmo, proposto junto com o FACET em Pena et al. (2022), é uma extensão do HSM, com a diferença de que nele, ao invés das chaves das tabelas *hash* serem valores, elas são intervalos de valores das colunas, ou seja, para a tabela da coluna A, cada uma das entradas estará no formato $\langle K, tids_A \rangle$, onde $tids_A$ contém as tuplas do lado esquerdo da entrada tais que seus valores em A estão dentro do intervalo K. As entradas da tabela de B são similares, mas considerando a coluna B e as tuplas do lado direito da entrada. Dadas as alterações apresentadas, a construção das tabelas e a execução do *merge* seguem o exposto para o HSM, com somente um passo adicional ao fim do *merge*, que consiste em aplicar o HSM para as tuplas de $tids_A$ e $tids_B$ em cada uma das entradas das tabelas hash.

Utilizando a técnica de separação em intervalos descrita acima, chamada *binning*, o BHSM consegue reduzir o número de operações de união lógica necessárias, melhorando consideravelmente a performance em predicados de alta cardinalidade em comparação com o HSM.

4.3 PLANO DE AVALIAÇÃO DE REFINAMENTOS

Definir um plano de avaliação que permita realizar primeiramente refinamentos com maior seletividade, ao mesmo tempo que adia o processamento de predicados computacionalmente muito custosos em termos de tempo de processamento, é uma tarefa importante para melhorar a performance da detecção de violações. Para isso, o FACET irá primeiramente definir uma ordenação para processar os predicados de classes distintas e, depois, ordenará os predicados dentro de cada uma das classes.

4.3.1 Predicados de classes diferentes

A estratégia de refinamento de classes diferentes é a de avaliar primeiro predicados de maior seletividade, nesse caso seriam avaliadas igualdades, seguidas de diferenças e inegualdades, respectivamente.

Em Pena et al. (2022), a seguinte intuição é utilizada para justificar a maior seletividade de predicados de igualdade em comparação com predicados de inegualdade: considere um predicado $p : t.A = t'.A$ e $f(k)$ como a frequência de uma coluna de valor k no domínio A, denotado $dom(A)$. Assuma que $1 < f(k) < n$, isto é, todo valor k aparece mais de uma vez e A não é uma coluna de valor único. Para o refinamento deste predicado, uma tabela *hash* é construída para a coluna A e então é a realizada a iteração por um total de $|dom(A)|$ entradas, o que resulta em $\sum_k f(k)^2$ pares de tuplas. Agora considere que o operador de p foi alterado para formar um novo predicado $p' : t.a \neq t'.A$. Pode-se notar intuitivamente o aumento das computações, pois para cada valor de k, é necessário realizar uma operação de diferença de conjuntos, e consequentemente há $\sum_k f(k).(n - f(k))$ pares de tuplas.

4.3.2 Predicados de mesma classe

Trabalhos anteriores definem a ordem de avaliação de predicados de mesma classe baseando-se na seletividade estimada, obtida através de amostragem, de cada um desses predicados, avaliando primeiramente aqueles com maior seletividade a fim de reduzir o número de intermediários a serem processados nas etapas seguintes. O FACET, por sua vez, introduz uma nova heurística para decidir qual predicado deve ser processado primeiro, priorizando a avaliação de predicados com base em informações estimadas de cardinalidade das colunas usadas. Utilizar essa nova ordenação pode levar o FACET a processar primeiro predicados com menor seletividade, mas a economia de processamento que esta abordagem proporciona ao não realizar a estimativa por amostragem acaba por reduzir o tempo geral de detecção.

Uma exceção a esta regra geral utilizada na ordenação dos predicados dentro de uma mesma classe é o caso em que um predicado $p : t.A \; o \; t'.B$ esteja operando sobre um par de $tids$ reflexivo e que $A = B$. Nestas condições, o processamento deste predicado pode receber diversas otimizações, de forma que o mesmo passa a ter maior prioridade em relação aos demais predicados de mesma classe. Detalhes destas otimizações foram omitidos neste trabalho, mas quase todos os algoritmos de avaliação de refinamentos podem ser otimizados na situação descrita.

4.3.3 Escolha do algoritmo para desigualdades

Por questões de simplicidade, as vantagens e desvantagens de cada algoritmo de desigualdade não serão detalhadas neste trabalho, de maneira que os maiores detalhes, assim como gráficos de performance, podem ser encontrados em Pena et al. (2022).

Para a escolha do algoritmo em ocasiões de desigualdades, o FACET analisa os seguintes casos: se a DC contém apenas um par de inegualdades, é verificada a viabilidade do IEJoin. Se o predicado tiver uma cardinalidade de 2^{13} ou maior, o IEJoin é utilizado. Em outros casos, pode ser utilizado o HSM ou BHSM, também dependendo da cardinalidade. A escolha entre HSM e BHSM se dá por uma sequência de execuções dos dois algoritmos, e então após uma execução

inicial dos algoritmos, o que é rápido, visto a estrutura de baixa cardinalidade, o algoritmo escolhido pelo FACET se dá pela percepção de qual dos dois está se beneficiando mais pelo uso de *cache*.

5 OPERADOR DE PROJEÇÃO

Uma vez apresentado o FACET, será definido precisamente o operador de projeção proposto neste trabalho para a projeção de pares de tuplas. Esta definição especifica como a projeção dos pares de tuplas que violam uma determinada DC será feita. Além do operador de projeção de pares de tuplas, será definido um operador de projeção para tuplas, o qual foi implementado e utilizado para projetar as tuplas que fazem parte de alguma violação à DC analizada.

Uma vez feitas as definições, será apresentado o *design* da solução, com a descrição de cada uma das implementações da tabela de projeção realizadas e cujos resultados serão avaliados neste trabalho. Por fim, serão apresentadas ainda duas formas de construção da tabela de projeção.

5.1 DEFININDO OS OPERADORES

A seguir, serão apresentados os operadores de projeção que serão adicionados ao FACET. A cada execução da versão estendida do FACET, será possível escolher se alguma projeção deve ou não ser feita. Caso se decida realizar alguma projeção, deve ser informado qual método deve ser utilizado, se o método de projeção de pares de tuplas ou somente de tuplas. Sempre que uma execução com projeção for ser realizada, será necessário também informar o arquivo de saída onde os resultados devem ser escritos. Além disso, podem ser especificadas quais colunas devem ser projetadas (atributo $colunas$, como definido nos operadores apresentados a seguir). Caso não sejam especificadas as colunas que devem ser projetadas, todas as colunas da tabela serão projetadas.

Tanto a definição da projeção para o método de projeção de pares de tuplas quanto para o método de projeção de tuplas envolvidas em violações partem da definição da projeção da Álgebra Relacional, como aparece em Elmasri e Navathe (2016). Ambos os operadores apresentados neste trabalho permitem, assim como o operador da Álgebra, que somente os valores de algumas colunas sejam projetadas, com o descarte dos valores das demais. Uma diferença que vale para ambos os operadores que serão definidos, entretanto, é que não é realizada a remoção de duplicatas, seguindo uma abordagem semelhante ao que ocorre nos banco de dados relacionais tradicionais.

5.1.1 Operador de projeção de pares de tuplas

Primeiramente, será definido o operador de projeção que será utilizado para o método de projeção de pares de tuplas que violam uma determinada DC, apresentada na Definição 5.1.1.

Definição 5.1.1 (Operador de projeção de pares de tuplas)**.**

$$\Pi^{tp}_{colunas}(tids_1, tids_2)$$

Este operador recebe como entrada dois conjuntos de identificadores de tuplas ($tids_1$ e $tids_2$) junto com uma lista de colunas da tabela processada ($colunas$), retornando uma nova tabela que contém os valores das $colunas$ para os pares de tuplas (t, t') tais que $t \in tids_1$ e $t' \in tids_2$, com $t \neq t'$. A ordem dos valores das colunas será a mesma de $colunas$, primeiro para a tupla t e depois para a tupla t'.

Retomando a Tabela 1.1, considere cada uma das DCs para ela definidas, que podem ser consultadas na Tabela 3.1. Os resultados da execução, para cada uma das DCs, da versão estendida do FACET com a projeção dos resultados utilizando o método de projeção de pares de tuplas serão verficados a seguir.

Tomando a DC ϕ_1, vê-se que o conjunto de suas violações é vazio, de forma que seria exibida uma projeção sem tuplas. Neste caso, a única coluna para o qual foi requisitada a projeção é o coluna ID e o resultado da execução pode ser visto na Tabela 5.1.

t.ID	t'.ID

Tabela 5.1: $\Pi^{tp}_{ID}(\{\}, \{\})$

Passando para a DC ϕ_2, o FACET agora projetará as colunas para os pares de tuplas em $\{(t_2, t_3), (t_3, t_2)\}$. Aqui, foram requisitados para a projeção as colunas ID e SID e o resultado pode ser consultado na Tabela 5.2.

	t.ID	t.SID	t'.ID	t'.SID
(t_2, t_3)	101	102	102	101
(t_3, t_2)	102	101	101	102

Tabela 5.2: $\Pi^{tp}_{ID,SID}(\{t_2, t_3\}, \{t_2, t_3\})$

Agora, analizando a DC ϕ_3 e sabendo que sua única violação é o par de tuplas em $\{(t_3, t_4)\}$, o resultado da projeção pode ser visto na Tabela 5.3. As colunas selecionados para a projeção aqui foram Dept, StartDate e Salary.

	t.Dept	t.StartDate	t.Salary	t'.Dept	t'.StartDate	t'.Salary
(t_3, t_4)	Research	2014	\$6000	Research	2015	\$8000

Tabela 5.3: $\Pi^{tp}_{Dept,StartDate,Salary}(\{t_3\}, \{t_4\})$

Por fim, vale ressaltar que não necessariamente as colunas projetadas devem fazer parte dos predicados das DCs avaliadas. Qualquer coluna dos dados pode ser requisitada para a projeção, de maneira que, por exemplo, um usuário poderia executar o FACET para detectar as violações da DC ϕ_2, mas somente projetando a coluna Name, o que resultaria na projeção exibida na Tabela 5.4.

	t.Name	t'.Name
(t_2, t_3)	R. Geller	D. Brown
(t_3, t_2)	D. Brown	R. Geller

Tabela 5.4: $\Pi^{tp}_{Name}(\{t_2, t_3\}, \{t_2, t_3\})$

5.1.2 Operador de projeção de tuplas

A Definição 5.1.2 apresenta o operador que será utilizado para o método de projeção tuplas envolvidadas em violações à DC analizada.

Definição 5.1.2 (Operador de projeção de tuplas)**.**

$$\Pi^{t}_{colunas}(tids_1, tids_2)$$

Este operador recebe como entrada dois conjuntos de identificadores de tuplas ($tids_1$ e $tids_2$) junto com uma lista de colunas da tabela processada ($colunas$), retornando uma nova tabela que contém os valores das $colunas$ para as tuplas $t \in tids$, onde $tids = tids_1 \cup tids_2$. A ordem dos valores das colunas será a mesma de $colunas$.

Retomando a Tabela 1.1 e considerando cada uma das DCs para ela definidas, que podem ser consultadas na Tabela 3.1, serão verificados a seguir os resultados da execução, para cada uma das DCs, da versão estendida do FACET com a projeção dos resultados utilizando o método de tuplas.

Tomando a DC ϕ_1, percebe-se que o conjunto de suas violações é vazio, de forma que seria exibida uma projeção sem tuplas. Neste caso, a única coluna para o qual foi requisitada a projeção é o coluna ID e o resultado da execução pode ser visto na Tabela 5.5.

t.ID

Tabela 5.5: $\Pi^{t}_{ID}(\{\}, \{\})$

Passando para a DC ϕ_2, o FACET agora projetará as colunas para as tuplas presentes em $\{(t_2, t_3), (t_3, t_2)\}$, isto é, para as tuplas em $\{t_2, t_3\}$. Aqui foram requisitados para a projeção as colunas ID e SID e o resultado pode ser consultado na Tabela 5.6.

	t.ID	t.SID
t_2	101	102
t_3	102	101

Tabela 5.6: $\Pi^{t}_{ID,SID}(\{t_2, t_3\}, \{t_2, t_3\})$

Agora, analizando a DC ϕ_3 e sabendo que sua única violação é o par de tuplas em $\{(t_3, t_4)\}$, ou seja, que as únicas tuplas envolvidas em violações são $\{t_3, t_4\}$, o resultado da projeção pode ser visto na Tabela 5.7. As colunas selecionados para a projeção aqui foram *Dept*, *StartDate* e *Salary*.

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

Tabela 5.7: $\Pi^t_{Dept,StartDate,Salary}(\{t_3\},\{t_4\})$

5.2 DESIGN DA SOLUÇÃO

Agora será apresentado o design da solução que foi implementada e que está sendo avaliada neste trabalho. Primeiramente, serão descritas as quatro diferentes implementações que podem ser utilizadas para a tabela de projeção e, a seguir, serão detalhadas as duas formas pelas quais tal tabela pode ser construída.

Vale notar que, independente de como a tabela de projeção é construída ou qual implementação é utilizada para ela, a projeção em si é sempre feita da mesma forma. Na execução do método de projeção de pares de tuplas, o que será feito é, para cada par de conjuntos de identificadores de tuplas $(tids_1, tids_2)$ que chegarem até o estágio de *output* do FACET, iterar por todos os pares de identificadores de tuplas (t, t') tais que $t \in tids_1$ e $t' \in tids_2$, com $t \neq t'$, buscando para os identificadores das tuplas de cada par os valores de suas colunas a serem projetadas na tabela de projeção e enviando-os para o *buffer* de saída.

Para o método de projeção somente de tuplas que participam de alguma violação, será mantido, junto com a tabela de projeção, um conjunto (sem repetição) de identificadores de tuplas que participam de alguma violação, de maneira que, para realizar a projeção, simplesmente será feita uma iteração sobre os identificadores em tal conjunto, buscando na tabela de projeção, para cada identificador, os valores das colunas que devem ser projetadas e enviando-os para o *buffer* de saída.

5.2.1 Implementações da tabela de projeção

Agora serão descritas cada uma das abordagens da implementação interna da tabela de projeção. Vale notar que, para todas essas implementações, a interface externa da tabela de projeção, e portanto o resto do algoritmo, permanecem os mesmos. Além disso, para qualquer uma das versões, foi utilizada uma tabela *hash* para mapear cada um dos identificadores de tuplas para seus dados, de maneira que as diferenças entre as versões residem nas estruturas de dados que cada uma delas utiliza para armazenar as suas tuplas.

5.2.2 Versão *baseline*

Como sugere o nome, esta versão foi tomada como ponto de partida para a comparação com as demais versões. Nela, a tabela de projeção é construída de maneira que cada tupla é armazenada como um *array* de *strings*, onde cada *string* corresponde a um valor lido diretamente do arquivo de dados.

A vantagem desta abordagem é, além de sua simplicidade, a ausência da necessidade de transformações de dados. Da maneira que os dados são lidos do arquivo de dados, eles são armazenados na tabela de projeção, sem nenhuma manipulação. Há dois possíveis problemas com esta abordagem, sendo o primeiro deles relacionado ao fato de que, mesmo que dois valores do arquivo de dados sejam *strings* idênticas, elas serão armazenadas em dois locais de memória distintos, o que pode levar a uma grande quantidade de *strings* redundantes em memória. Além disso, cada vez que uma tupla é projetada, seja individualmente ou como parte de um par, é preciso iterar pelo *array* de *strings* que contém os dados da tupla, adicionando suas *strings* uma a uma ao *buffer* de saída.

Para exemplificar como funciona a tabela de projeção para esta versão, será analizada a projeção exibida na Tabela 5.3. Uma representação da tabela de projeção para este caso pode ser vista na Figura 5.1, onde a linha de índice 0 contém as *strings* correspondentes aos valores das colunas da tupla t_3, enquanto que os valores das colunas da tupla t_4 são armazenados na linha de índice 1. Note que a *string* "Research" está duplicada nas duas tuplas, mesmo sendo idêntica em ambas as ocorrências.

Figura 5.1: Tabela de projeção da versão *baseline* para a projeção da Tabela 5.3

0	Research	2014	$6000
1	Research	2015	$8000

5.2.3 Versão *dictionary*

Para tentar resolver o problema da redundância de valores na memória descrito na versão *baseline*, foi implementado uma espécie de dicionário, que associa *strings* à *integers* e permite duas operações: a obtenção da *string* associada a um *integer* chave e obtenção do *integer* chave associado a uma *string*. Para permitir tanto o mapeamento quanto as operações, o dicionário é composto por um *array*, que possibilita descobrir, em tempo $O(1)$, a qual *string* um *integer* está associado, e por uma tabela *hash*, que permite descobrir, também em tempo $O(1)$, qual o *integer* associado a uma *string*.

Na operação de obtenção da *string* associada a um *integer* chave, o dicionário simplesmente recebe um *integer*, consulta suas estruturas de dados internas e retorna a *string* a qual ele está associado, se houver. Na operação de obtenção do *integer* chave associado a uma *string*, o dicionário recebe uma *string* e verifica se a *string* informada já possui um *integer* associado a ela. Em caso negativo, a *string* é armazenada no dicionário e um novo *integer* é associado a ela e retornado. Em caso positivo, entretanto, o *integer* associado a *string* informada é simplesmente retornado, de forma que não é alocada uma nova posição de memória para tal *string*, de maneira que, com todas as tuplas da tabela de projeção utilizando o mesmo dicionário, não há mais redundância de *strings* em memória.

Para poder utilizar este dicionário, que será compartilhado por todas as tuplas da tabela de projeção, o que será feito é armazenar cada tupla como um *array* de *integers*. Para valores de colunas do tipo *integer* do arquivo de dados, a conversão é direta da *string* para o valor *integer* que ela representa. Quando tratam-se de colunas do tipo *string*, a conversão se dá por meio do mapeamento realizado pelo dicionário. Por fim, para as colunas do tipo *float*, o valor é convertido de *string* para seu valor correspondente em *float* e, depois, tal valor é convertido novamente, agora para o *integer* resultante da representação binária do *float*.

Esta versão é capaz de reduzir o número de *strings* redundantes em memória, o que pode ser útil para tabelas com uma grande quantidade de colunas do tipo *string* e onde tais colunas tenham uma cardinalidade consideravelmente menor do que o número de linhas total da tabela. É possível, entretanto, que ela tenha uma performance pior do que as demais versões nos casos mais gerais devido ao grande *overhead* de processamento que foi adicionado pelo uso do dicionário e das conversões de dados para *integers*.

A Figura 5.2 mostra a tabela de projeção e o dicionário utilizados para a projeção exibida na Tabela 5.3. Note que as *strings* “Research”, “$6000” e “$8000” foram substituídas, respectivamente, pelas chaves 0, 1 e 2, de maneira que tal mapeamento é mantido pelo dicionário. Veja que a duplicação da *string* “Research” não existe mais, o que, no caso de *strings* grandes e com muitas repetições, pode reduzir o uso de memória.

Figura 5.2: Tabela de projeção e dicionário da versão *dictionary* para a projeção da Tabela 5.3

0	**0**	2014	**1**
1	**0**	2015	**2**

Chave	Valor
0	Research
1	$6000
2	$8000

5.2.4 Versão *bytes-all-columns*

Antes da apresentação desta versão, é preciso avisar que ela, diferente de todas as demais, não permite a seleção de colunas a serem projetadas, mas sempre projeta todas as colunas do arquivo de dados, na ordem em que lá aparecem. Ela foi mantida aqui pois pode vir a ser utilizada em melhorias futuras da projeção no FACET, para casos específicos. A versão *bytes*,

que será apresentada a seguir, se baseia nesta versão, mas volta a permitir a escolha das colunas a serem projetadas.

Nela, cada uma das tuplas da tabela de projeção será armazenada simplesmente como um *array* de *bytes*. Tal *array* será obtido diretamente da conversão da *string* que corresponde a toda uma tupla no arquivo de dados para sua representação em *bytes*. Nesta versão, quando a projeção estiver sendo executado efetivamente, o que será enviado para o *buffer* de saída são *arrays* de *bytes* ao invés de *strings*, que era o que ocorria nas duas versões anteriores.

São duas as principais vantagens desta abordagem: a primeira delas é enviar para o *buffer* de saída uma representação já em *bytes* dos valores das colunas, de maneira que tais valores podem ser mais eficientemente escritos no arquivo de projeção. A segunda é que agora não é mais necessário, nem na construção da tabela de projeção e nem na execução da projeção, iterar sobre as colunas que devem ser projetadas, o que é necessário em todas as versões apresentadas até aqui.

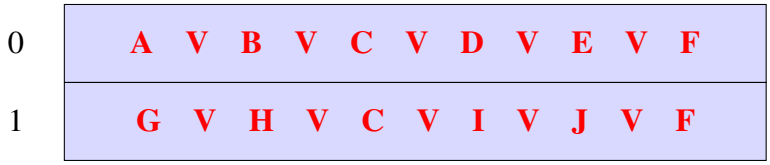
Para exemplificar o uso da tabela de projeção aqui, não será possível utilizar a projeção da Tabela 5.3, pois nela nem todas as colunas da Tabela 1.1 estão sendo projetadas e, como dito, esta versão está limitada a projetar sempre todas as colunas na ordem em que aparecem no arquivo de dados. Para prosseguir com o exemplo, será, portanto, utilizada uma projeção diferente, exibida na Tabela 5.8, e que é o resultado da projeção de todas as colunas da Tabela 1.1 utilizando o método de projeção de tuplas para as violações detectadas ao avaliar a DC ϕ_3, que pode ser consultada na Tabela 3.1.

	t.ID	t.Name	t.Dept	t.StartDate	t.Salary	t.SID
t_3	100	D. Brown	Research	2014	$6000	101
t_4	103	H. McCoy	Research	2015	$8000	101

Tabela 5.8: $\Pi_{ID,Name,Dept,StartDate,Salary,SID}(\{t_3\},\{t_4\})$

Um vez definida a projeção, a Figura 5.3 apresenta a representação da tabela de projeção utilizada neste caso. Note que agora cada linha da tabela é uma concatenação de binários, onde cada binário representa ou o valor de uma coluna da tupla ou o caracter ",". O binário "A", por exemplo, representa a *string* "100", o binário "B" representa a *string* "D. Brown" e assim por diante, como mostrado na figura.

Figura 5.3: Tabela de projeção e representações binárias da versão *bytes-all-columns* para a projeção da Tabela 5.8



Binário	*String*
A	100
B	D. Brown
C	Research
D	2014
E	$6000
F	101
G	103
H	H. McCoy
I	2015
J	$8000
V	,

5.2.5 Versão bytes

Nesta última versão, como na versão *bytes-all-columns*, cada uma das tuplas da tabela de projeção será armazenada como um *array* de *bytes*. A diferença aqui é que, ao invés de converter a *string* que corresponde a toda a tupla do arquivo de entrada, esta versão vai iterar, durante a construção da tabela de projeção, sobre cada uma das colunas que devem ser projetadas, convertendo individualmente as *strings* que correspondem a cada um dos valores das colunas para sua representação em *bytes* e concatenando os resultados de tais conversões em um *array* de *bytes*. Além dissso, entre dois binários representando valores de colunas da tupla, será necessário concatenar um binário representando o caracter “,”, isto para permitir que, no momento da projeção, o *array* de *bytes* possa ser enviado inteiro para o *buffer* de saída, sem que exista a necessidade de iterar por cada um dos valores da tupla.

Desta forma é possível permitir a seleção de colunas a projetar mantendo as principais vantagens da versão *bytes-all-columns*, somente com a adição, na construção da tabela de projeção, da iteração sobre as colunas a serem projetadas. Essa iteração, entretanto, tem custo baixo, pois será realizada $O(n)$ vezes, enquanto que o tempo de projeção será dominado pelas $O(n^2)$ projeções de pares de tuplas.

Agora, voltando a projeção da Tabela 5.3, pode-se verificar o estado da tabela de projeção para este caso na Figura 5.4. Note que a tabela de projeção desta versão é bastante semelhante a da versão *bytes-all-columns*, com a diferença que somente estão presentes nela os binários que representam valores de colunas que devem ser projetadas.

Figura 5.4: Tabela de projeção e representações binárias da versão *bytes* para a projeção da Tabela 5.3

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

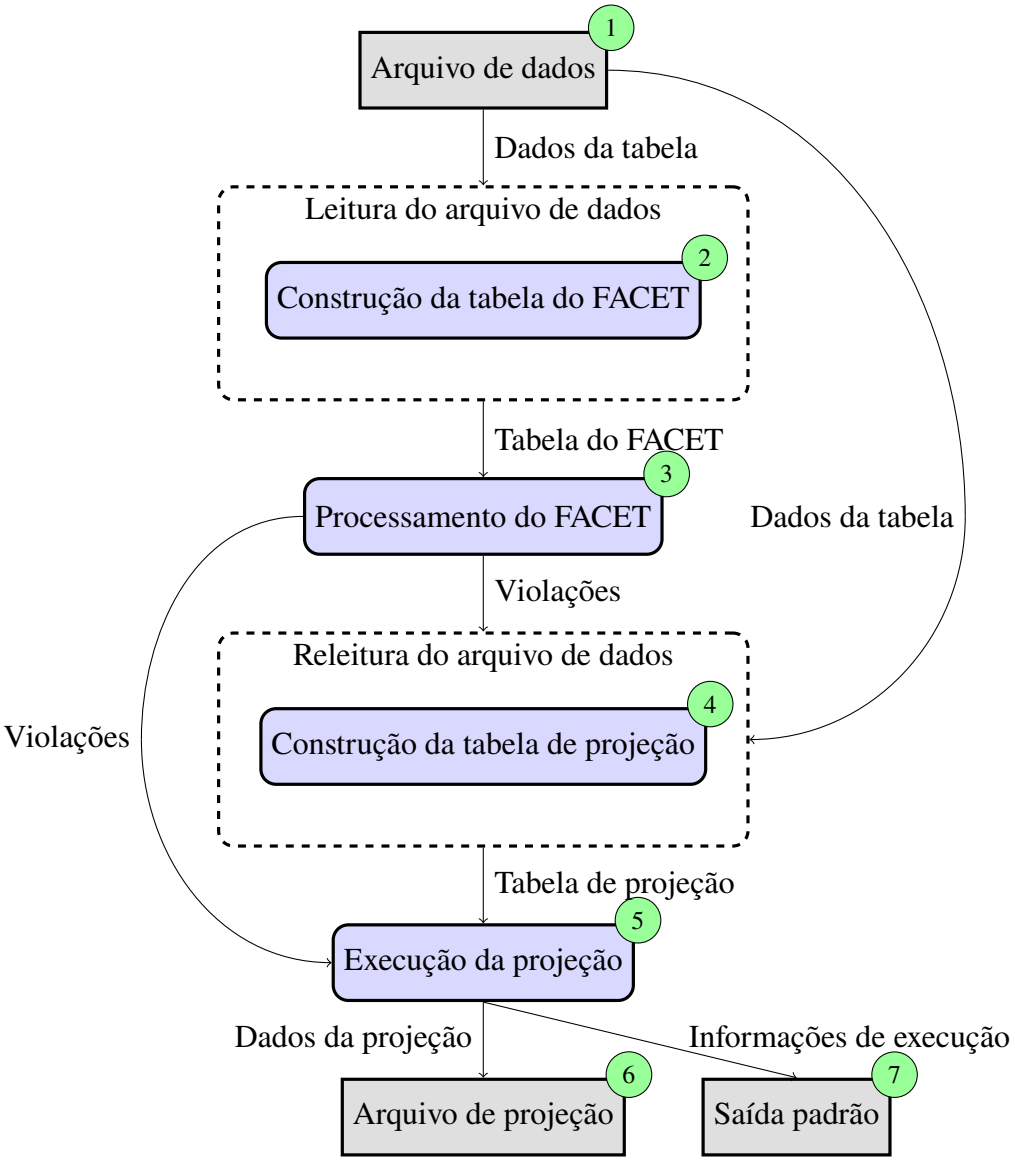
Binário	*String*
A	Research
B	2014
C	$6000
D	2015
E	$8000
V	,

5.3 ABORDAGENS PARA A CONSTRUÇÃO DA TABELA DE PROJEÇÃO

Além das diferentes versões implementadas para a tabela de projeção, serão utilizadas e analisadas também duas abordagens distintas para sua construção. Ambas as abordagens serão detalhadas a seguir.

5.3.1 Releitura do arquivo de dados

Figura 5.5: Construção da tabela de projeção com releitura do arquivo de dados



A primeira das abordagens constrói a tabela de projeção a partir de uma releitura do arquivo de dados realizada após o FACET concluir a detecção de violações. A Figura 5.5 representa de maneira simplificada esse processo. Em 1, o arquivo de dados é lido pelo FACET. Em 2, o FACET constrói a tabela que ele utilizará para detectar as violações. Em 3, o FACET realiza seu processamento, detectando as violações e enviando-as para o módulo de projeção. Em 4, já no módulo de projeção e uma vez encerrada a etapa de detecção, o arquivo de dados é lido novamente, agora com a construção da tabela de projeção. Em 5, a tabela de projeção é utilizada para a projeção dos pares de tuplas ou das tuplas. Em 6, os valores das colunas das tuplas são escritos no arquivo de projeção. Em 7, a escrita dos *logs* é finalizada e a execução, encerrada.

Além disso, durante a execução do FACET, é construído e atualizado um conjunto com todos os identificadores de tuplas envolvidas em alguma violação da DC analizada, de forma

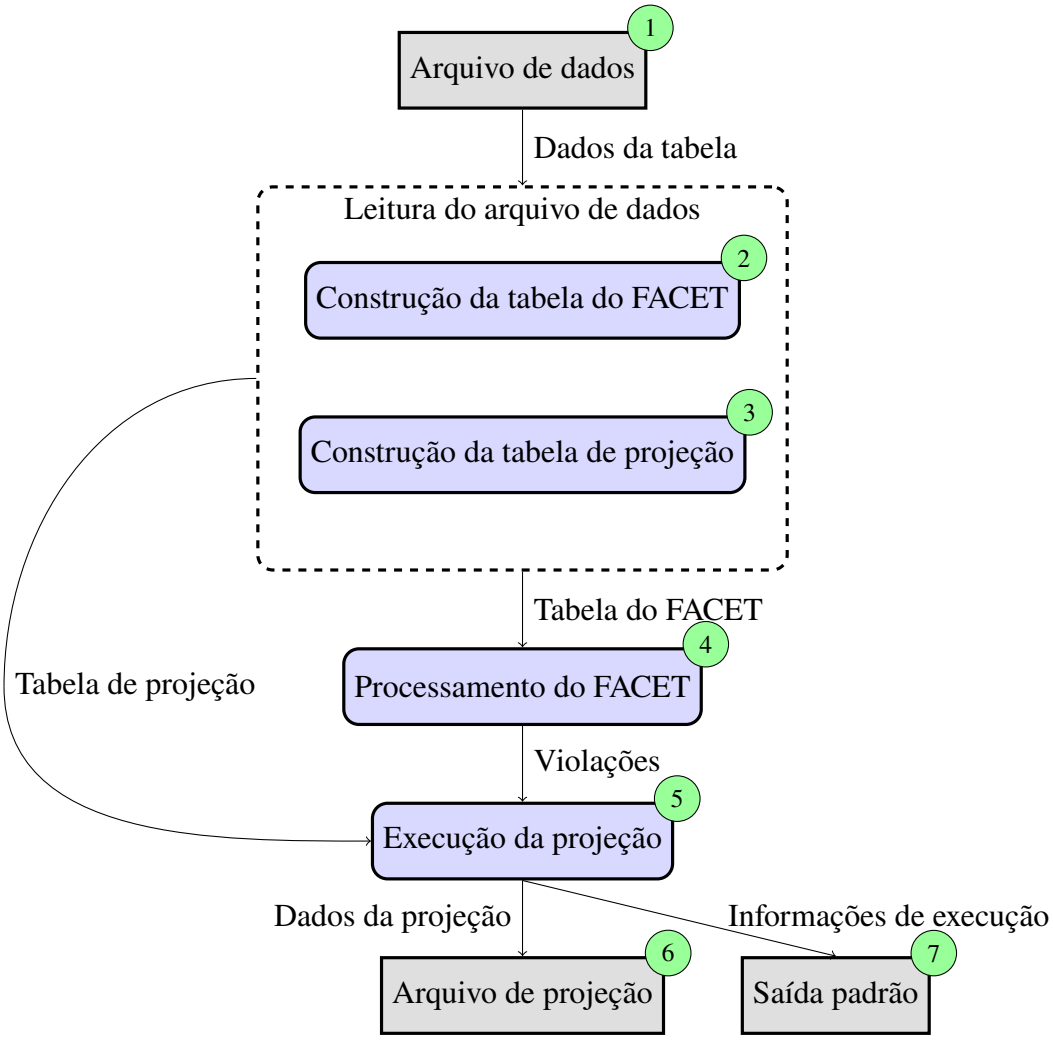
que, com a conclusão da execução, o arquivo de dados é novamente lido e a tabela de projeção é construída somente com os dados das tuplas cujos identificadores estão contidos no referido conjunto, ou seja, a tabela de projeção contém somente as tuplas que de fato vão ser projetadas, seja de maneira individual ou como parte de um par.

Há duas vantagens nesta abordagem. A primeira delas é construir a tabela de projeção somente com as tuplas que realmente serão utilizadas na projeção, não desperdiçando nem memória e nem processamento, especialmente naqueles casos em que o número de tuplas que participam de alguma violação é consideravelmente menor do que o número total de tuplas da tabela de entrada. A segunda é que não há concorrência por memória com o FACET durante sua execução. Note que esta última vantagem só fará diferença significativa em casos onde a execução em memória seja um gargalo para a execução do FACET.

As versões utilizando esta abordagem de construção da tabela de projeção receberam os nomes das versões como foram apresentados até aqui, ou seja, são as versões *baseline*, *dictionary*, *bytes-all-columns* e *bytes*.

5.3.2 Leitura única do arquivo de dados

Figura 5.6: Construção da tabela de projeção com leitura única do arquivo de dados



Esta abordagem, por sua vez, tem por objetivo justamente evitar a releitura do arquivo de entrada e, para isso, constrói a tabela de projeção em conjunto com a construção da tabela utilizada pelo FACET para a detecção das violações à DC analizada. A Figura 5.6 exibe uma representação simplificada desse processo. Em 1, o arquivo de dados é lido pelo FACET. Em 2, o FACET constrói a tabela que ele utilizará para detectar as violações. Em 3, juntamente com a construção da tabela do FACET e se aproveitando da mesma leitura do arquivo de dados, a tabela de projeção é construída no módulo de projeção. Em 4, o FACET realiza seu processamento, detectando as violações e enviando-as para o módulo de projeção. Em 5, a tabela de projeção é utilizada para a projeção dos pares de tuplas ou das tuplas. Em 6, os valores das colunas das tuplas são escritos no arquivo de projeção. Em 7, a escrita dos *logs* é finalizada e a execução, encerrada.

Note que, nesta abordagem, como ainda não foram detectadas as violações no momento em que é construída a tabela de projeção, não é possível saber quais tuplas devem ser projetadas,

de forma é necessário sempre construir a tabela de projeção com os dados de todas as tuplas do arquivo de dados.

A maior vantagem desta abordagem é justamente realizar somente uma leitura do arquivo de dados, o que reduz o processamento necessário. Entretanto, em execuções onde memória sejam um limitante, é possível que esta abordagem acabe por ter uma performance pior do que aquela que executa a releitura do arquivo, uma vez que pode ocorrer a concorrência por memória com o processamento do FACET.

As versões utilizando esta abordagem de construção da tabela de projeção estão prefixadas com "*single*", ou seja, são as versões *single-baseline*, *single-dictionary*, *single-bytes-all-columns* e *single-bytes*.

Por fim, é importante fazer um apontamento relevante sobre esta abordagem de construção da tabela de projeção na versão *single-bytes-all-columns*. Nas demais versões, os valores vindos do arquivo de dados são enviados para o construtor das tuplas da tabela de projeção em valores separados por coluna, o que não ocorre na versão *byte-all-columns*. Nesta versão, o construtor recebe as *strings* que representam a tupla como um todo, de forma que, ao construir as tuplas junto com a leitura do FACET, é preciso realizar uma concatenação dos valores de cada uma das colunas e enviar a *string* obtida com a concatenação para o construtor, o que gera um aumento no tempo de processamento, que pode ser notado nos resultados dos experimentos.

6 AVALIAÇÃO EXPERIMENTAL

A seguir, será apresentada a avaliação experimental, detalhando a bancada de testes, os *datasets* utilizados no processo, os resultados obtidos e discussões sobre eles.

6.1 BANCADA DE TESTES

As configurações utilizadas para a bancada de testes são descritas na Tabela 6.1. Vale destacar que a avaliação experimental foi executada em um SSD Adata XPG SX6000 (Armazenamento de 256GB, Leitura 1800MB/s, Gravação 900MB/s) com a partição formatada em Btrfs, visto que a proposta é focada na leitura e escrita em disco.

Processador	AMD Ryzen 5 1600AE
Sistema Operacional	Fedora 37
Versão do Kernel	6.3.4
Quantidade de RAM	16GB
OpenJDK	11.0.19

Tabela 6.1: Configuração do Sistema

Para mitigar ainda mais as variações entre as execuções dos testes e manter uma maior consistência nos resultados, foi utilizado a ferramenta tlp[1] para fixar o *clock* do processador em 3.2GHz.

6.2 *DATASETS*

Os *datasets* utilizados são apresentados na Tabela 6.2, enquanto as *denial constraints* utilizadas para cada um dos *datasets*, com a quantidade de violações a cada uma delas, são apresentada na Tabela 6.3. Os *datasets* e as DCs, descritas na tabela 6.3, são as mesmas que Pena et al. (2022) utiliza para a validação experimental do FACET. A Tabela 6.4 apresenta praticamente as mesmas combinações de *datasets* e DCs da Tabela 6.3, mas agora indicando o número de tuplas envolvidas em alguma violação.

Dataset	N° de colunas	N° de linhas
Actorkey	4	1000000
Flights	11	1000000
TPC-H	6	1000000
Tax	9	1000000

Tabela 6.2: *Datasets* listando a quantidade de colunas e linhas

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

Dataset	Índice	*Denial constraint*	N° de Violações
Actorkey	ϕ_1	$\neg(t_1.title = t_2.title \wedge t_1.role = t_2.role \wedge t_1.name = t_2.name \wedge t_1.charname = t_2.charname)$	33082
Flights	ϕ_2	$\neg(t_1.Origin = t_2.Destination \wedge t_1.Destination = t_2.Origin \wedge t_1.Distance \neq t_2.Distance)$	7106922
Flights	ϕ_3	$\neg(t_1.Origin = t_2.Origin \wedge t_1.Destination = t_2.Destination \wedge t_1.Flights > t_2.Flights \wedge t_1.Passengers < t_2.Passengers)$	59703344
TPC-H	ϕ_4	$\neg(t_1.receiptdate \geq t_2.shipdate \wedge t_1.shipdate \leq t_2.receiptdate)$	13066546833
TPC-H	ϕ_5	$\neg(t_1.discount < t_2.discount \wedge t_1.price > t_2.price)$	227537598372
TPC-H	ϕ_6	$\neg(t_1.quantity = t_2.quantity \wedge t_1.tax = t_2.tax \wedge t_1.discount < t_2.discount \wedge t_1.price > t_2.price)$	504669483
Tax	ϕ_7	$\neg(t_1.AreaCode = t_2.AreaCode \wedge t_1.Phone = t_2.Phone)$	0

Tabela 6.3: *Datasets* juntamente das restrições e quantidade de violações que o FACET retorna ao final da fase de detecção e *planning*

Dataset	*Denial constraint*	N° de tuplas com alguma violação
Actorkey	ϕ_1	7443
Flights	ϕ_2	253644
Flights	ϕ_3	908812
TPC-H	ϕ_4	1000000
TPC-H	ϕ_5	1000000
TPC-H	ϕ_6	999912

Tabela 6.4: *Datasets* listando a quantidade de tuplas com alguma violação

Dado que o número pares de tuplas a serem projetados é igual ao número total de violações encontradas para a DC analizada, as DCs ϕ_4, ϕ_5 e ϕ_6, descritas na tabela 6.3, são casos

onde é inviável utilizar o método de projeção de pares de tuplas. A DC ϕ_5, por exemplo, teria aproximadamente 227 bilhões de pares de tuplas projetadas. Para casos com uma quantidade muito grande de violações, foi utilizado apenas o método de projetar as tuplas que estejam envolvidas em alguma violação.

6.3 RESULTADOS

Nesta seção, serão apresentados os gráficos dos resultados relacionados às execuções da versão estendida do FACET para cada um dos *datasets* e DCs descritos na Tabela 6.3, com comentários sobre esses resultados. Para os testes executados com a especificação das colunas a serem projetadas, foram utilizadas as colunas que aparecem nas DCs descritas na Tabela 6.3.

Em todos os gráficos de resultados de execuções onde foram especificadas as colunas a serem projetadas, as versões *bytes-all-columns* e *single-bytes-all-columns* não são exibidas, pois, conforme comentado na seção 5.2.4, estas versões não consideram os argumentos de especificação de colunas e portanto não há o porque dos resultados para as mesmas serem incluídos.

Ao executar os testes para as versões do *dataset* TPC-H (DCs ϕ_4, ϕ_5 e ϕ_6), optou-se por desabilitar o *Simultaneous Multithreading* (SMT). Nos primeiros testes foi percebido que, para esse *dataset* específico, nos tempos de detecção e *planning* de DCs, estavam ocorrendo variações consideráveis entre as versões, o que foi mitigado ao desabilitar esse recurso do processador.

Nos gráficos, cada fase está indicada com uma cor diferente, fixando a cor apenas para os tempos da fase de leitura do FACET e o de detecção e *planning*. Dado isso, a fase de leitura do FACET é indicada pela cor verde tracejada, o tempo de detecção e *planning*, pela área cinza tracejada, enquanto as áreas sem tracejos indicam o tempo de projeção. Nas versões que não são de leitura única, o tempo de releitura e construção da estrutura de dados auxiliar é incluído no tempo de projeção. As cores fixas só possuem uma exceção para o gráfico indicado na figura 6.23, pois é um teste específico para a análise do caso onde não há dados a projetar.

Por fim, antes de prosseguir, é válido ressaltar que por questões de instabilidade e por falta de prazo para investigar os motivos, optou-se por não adicionar os experimentos relacionados ao método de pares de tuplas para a amostra *Flights* e DC ϕ_3. Nos testes realizados, percebeu-se que os tempos de execuções para a escrita em disco estavam com bastante variação, levantando a suspeita em princípio de fragmentação de disco ou invalidação de *cache*, mas, como não foi averiguado, não foi concluído nada sobre essas suspeitas.

6.3.1 *Actorkey(ϕ_1)*

Para o *dataset* de *Actorkey*, os resultados usando o método de projeção de pares de tuplas são exibidos nas Figuras 6.1 e 6.2, e para o método de projeção de tuplas, nas Figuras 6.3 e 6.4.

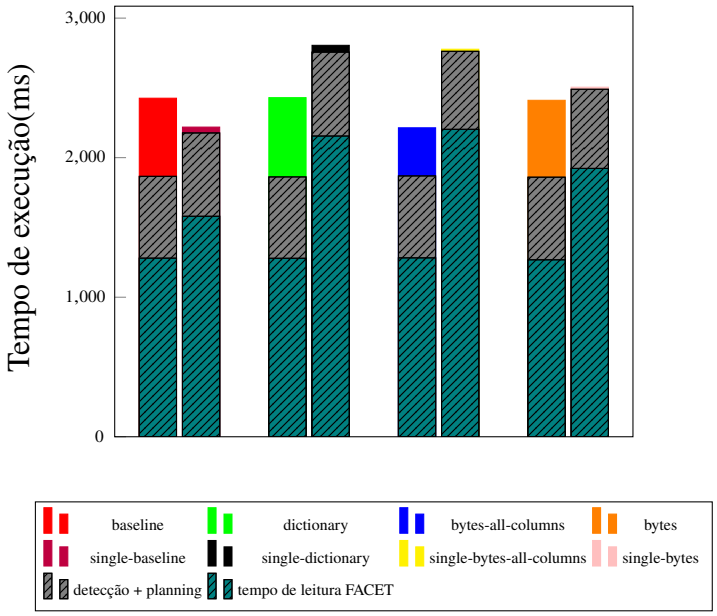


Figura 6.1: Actorkey(ϕ_1) usando o método para exibir pares de tuplas

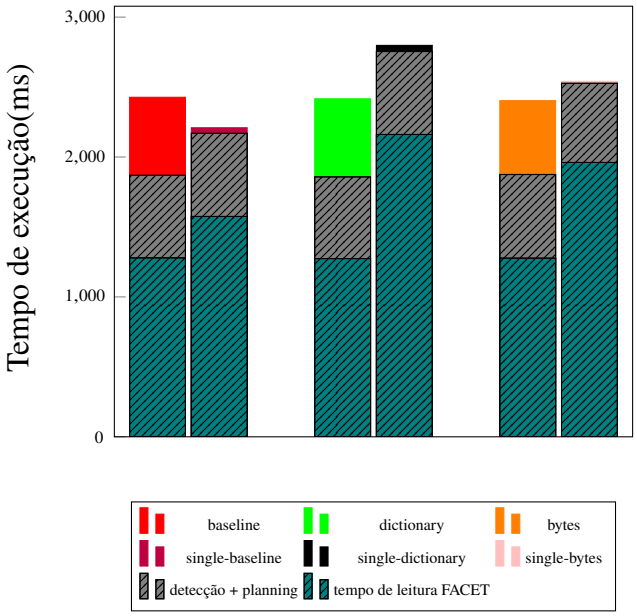


Figura 6.2: Actorkey(ϕ_1) usando o método para exibir pares de tuplas especificando colunas

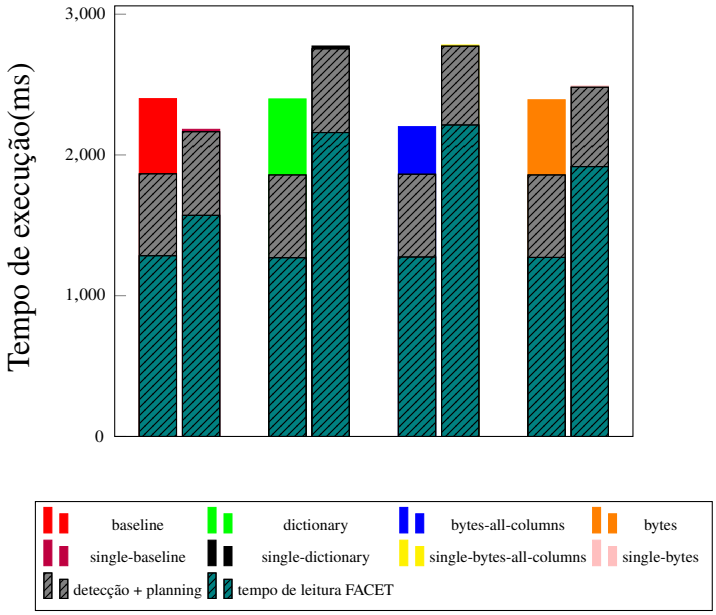


Figura 6.3: Actorkey(ϕ_1) usando o método para exibir tuplas

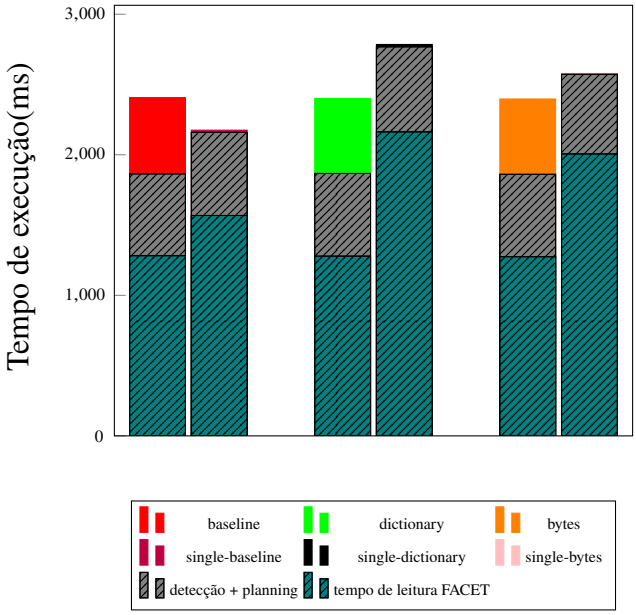


Figura 6.4: Actorkey(ϕ_1) usando o método para exibir tuplas especificando colunas

Neste *dataset*, os tempos especificando colunas e a projeção geral não variam, visto que a DC ϕ_1 contém todas as colunas do *dataset*. Pode-se também notar pelos tempos que a leitura única apenas leva vantagem na versão *baseline*, enquanto nas demais a leitura única impactou de forma negativa o tempo geral de execução.

Pode-se também presumir que, como a quantidade de tuplas com alguma violação indicadas pela tabela 6.4 não se distancia muito da quantidade de violações indicadas pela tabela 6.3, o tempo gasto para execução não se diferenciou praticamente entre os diferentes métodos e versões para este *dataset* especificamente. Isto é reforçado também pela pequena quantidade de colunas que o *dataset* possui.

6.3.2 *Flights(ϕ_2)*

Para o *dataset flights* utilizando a DC ϕ_2, pode-se ver uma maior diferença entre as execuções de versões distintas. Desta vez, a versão de leitura única foi levemente superior na maioria das implementações, com exceção da *bytes-all-columns*, provavelmente devido a especificidade da implementação da versão, o que foi explicado em 5.2.4. Os resultados podem ser vistos para o método de pares de tuplas nas figuras 6.5 e 6.6 e para o métodos de tuplas nas figuras 6.7 e 6.8.

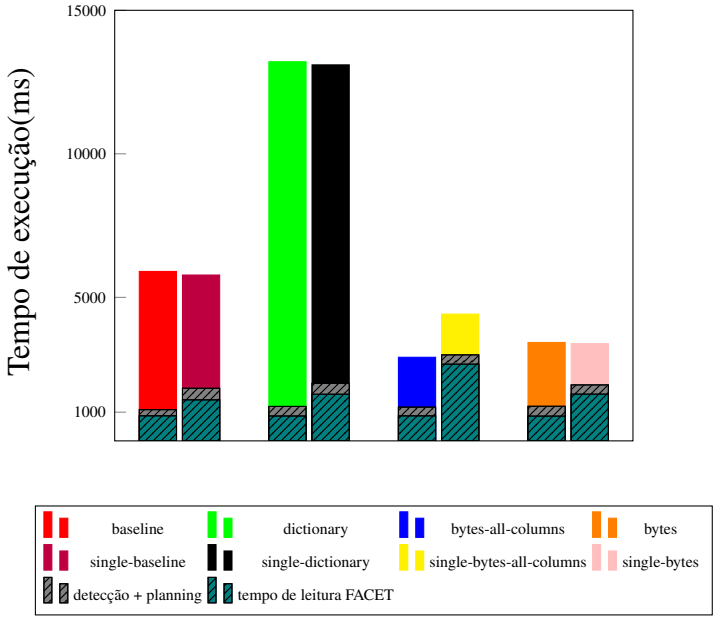


Figura 6.5: Flights(ϕ_2) usando o método para exibir pares de tuplas

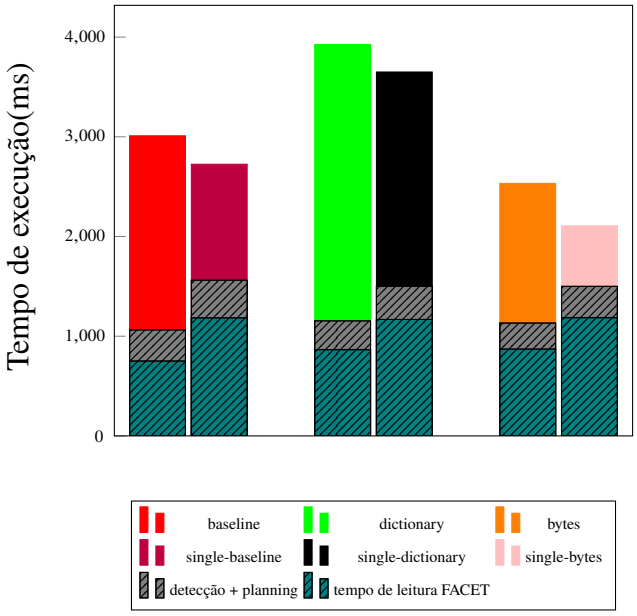


Figura 6.6: Flights(ϕ_2) usando o método para exibir pares de tuplas especificando colunas

Desta vez também é possível verificar que, ao especificar as colunas a serem projetadas, os tempos de execução ficaram menores, visto que a DC ϕ_2 utiliza somente um pequena parcela das colunas do *dataset*. Para reforçar este fato, as versões *dictionary* e *single-dictionary*, que apresentaram os piores tempos, reduziu de aproximadamente 13000 milisegundos de projeção para apenas aproxidamente 4000 milisegundos.

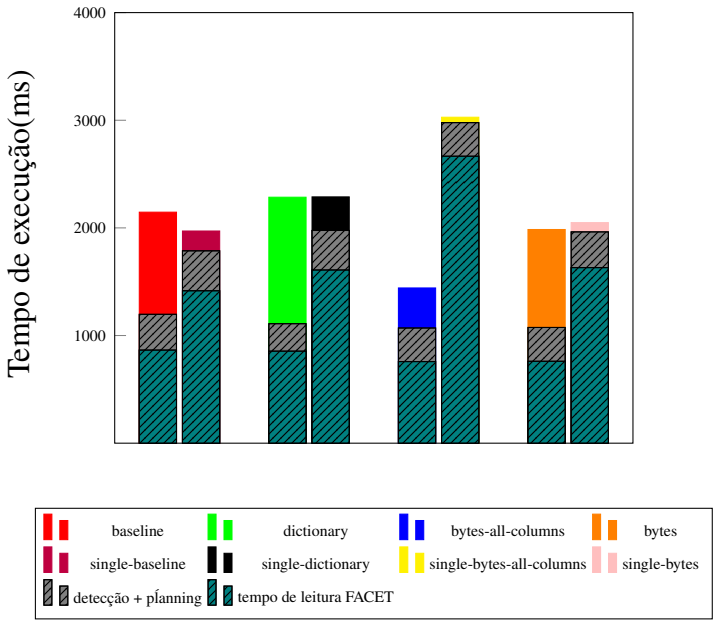


Figura 6.7: Flights(ϕ_2) usando o método para exibir tuplas

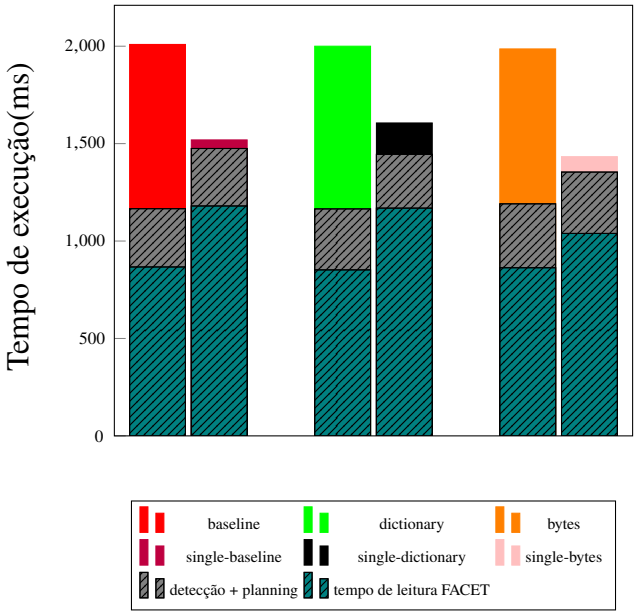


Figura 6.8: Flights(ϕ_2) usando o método para exibir tuplas especificando colunas

Também pode-se verificar a execução mais rápida do método de projeção de tuplas em relação ao de pares de tuplas, visto que nesta DC há aproximadamente 253 mil tuplas com

violações apontadas pela tabela 6.4 em comparação à aproximadamente 7 milhões de violações apontadas pela tabela 6.3.

6.3.3 *Flights(ϕ_3)*

Para o *dataset flights* utilizando a DC ϕ_3, onde somente foram executados os testes para o método de projeção de tuplas, novamente na maioria das implementações a versão de leitura única obtêm uma leve vantagem sobre a versão de leitura dupla. Os resultados são descritos nas figuras 6.9 e 6.10 para o método de projeção de tuplas.

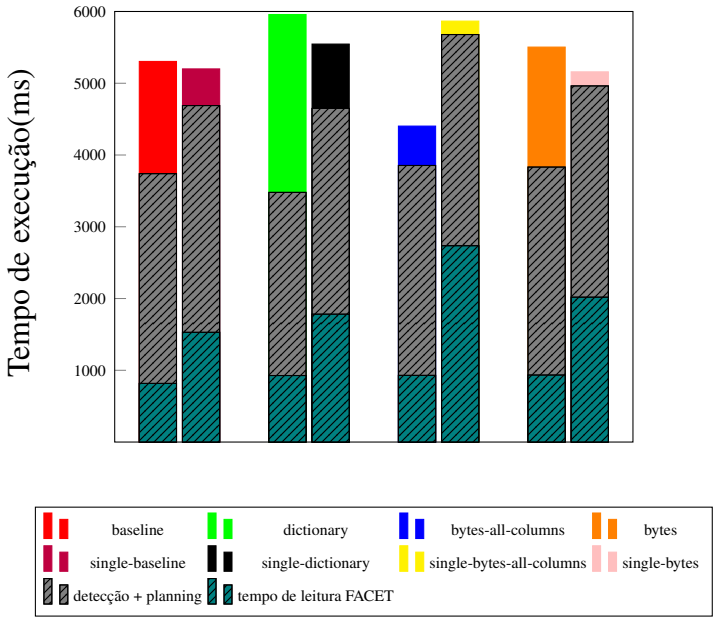


Figura 6.9: Flights(ϕ_3) usando o método para exibir tuplas

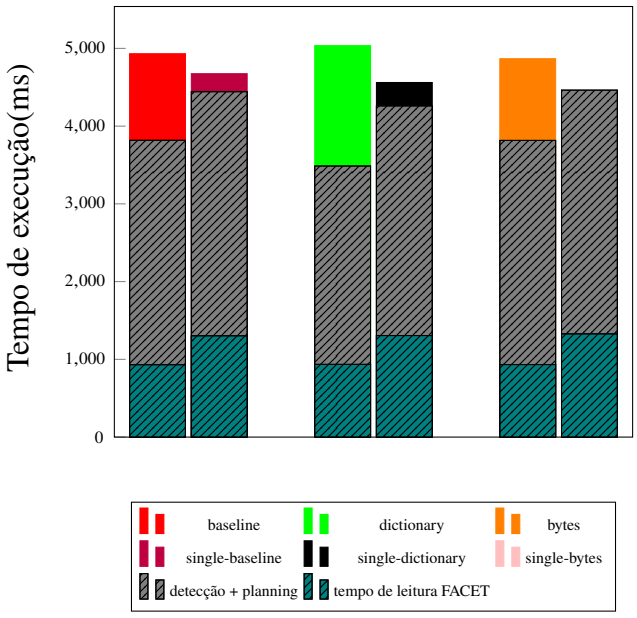


Figura 6.10: Flights(ϕ_3) usando o método para exibir tuplas especificando colunas

Também verifica-se, novamente, um tempo de execução menor ao especificar as colunas a serem projetadas, uma vez que a restrição ϕ_3 utiliza um pequeno conjunto das colunas do *dataset*.

6.3.4 TPC-H(ϕ_4)

Para o *dataset* TPH-C e DC ϕ4, foi utilizado também apenas o método de projeção de tuplas, visto que o número de violações descrito pela tabela 6.3 é de aproximadamente 13 bilhões, enquanto que, de acordo com a tabela 6.4, teria de ser projetado 1 milhão de tuplas envolvidas em violações. Os resultados de execução podem ser averiguados pelas figuras 6.11 e 6.13, enquanto as figuras 6.12 e 6.14 representam a mesma amostragem, porém sem exibir o tempo de detecção e *planning*.

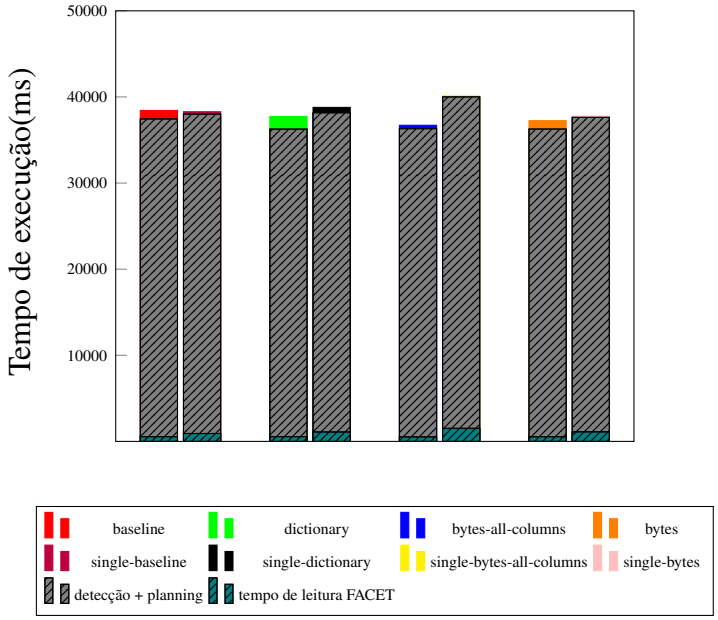


Figura 6.11: TPC-H(ϕ_4) usando o método para exibir tuplas

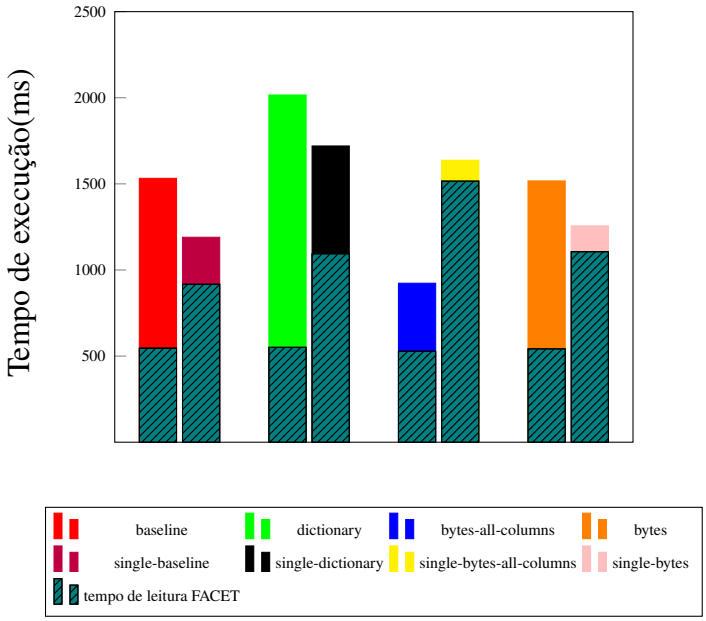


Figura 6.12: TPC-H(ϕ_4) usando o método para exibir tuplas, ignorando os tempos de detecção + *planning*

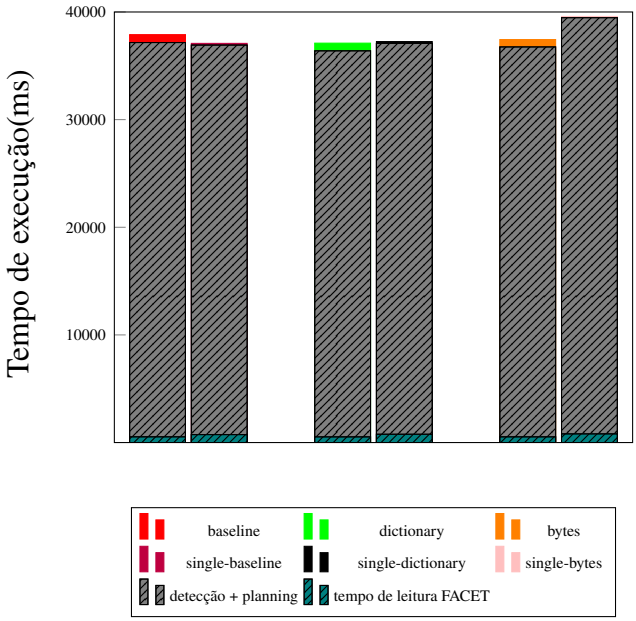


Figura 6.13: TPC-H(ϕ_4) usando o método para exibir tuplas especificando colunas

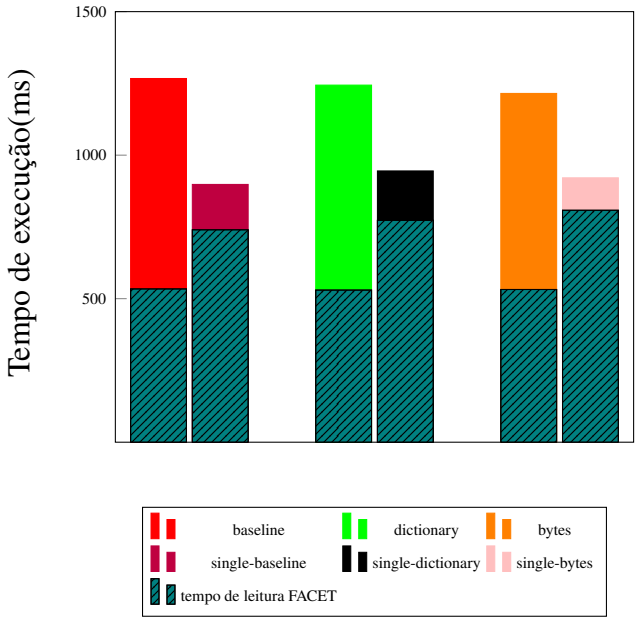


Figura 6.14: TPC-H(ϕ_4) usando o método para exibir tuplas especificando colunas, ignorando os tempos de detecção + *planning*

Aqui, ao contrário de outros casos, pode-se verificar que a versão de leitura única tem um tempo de execução um pouco maior do que a versão com releitura para a maioria dos casos, evidenciados pelas figuras 6.11 e 6.13. Este aumento se concentra nos tempos de detecção e *planning* para as todas versões, com exceção da *bytes* e *single-bytes*. A suspeita, a princípio, é que isto seja devido ao fato da tabela de projeção ser mantida em memória durante o processamento do FACET.

Porém, se não for considerado os tempos de detecção e planning exibidos nas figuras 6.12 e 6.14, então a leitura única levaria vantagem, mas não pode-se concluir que a versão de leitura única é melhor, visto que não houve investigação o suficiente para concluir que a variação da detecção e *planning* estejam sendo comprometidas pela estrutura extra em memória.

6.3.5 TPC-H(ϕ_5)

Para o *dataset* TPH-C e DC ϕ5, também foi utilizado somente o método de projeção de tuplas, também por conta do alto número de violações. Os resultados podem ser observados nas figuras 6.15 e 6.17, e a mesma amostragem sem tempos de detecção e *planning* pelas figuras 6.16 e 6.18.

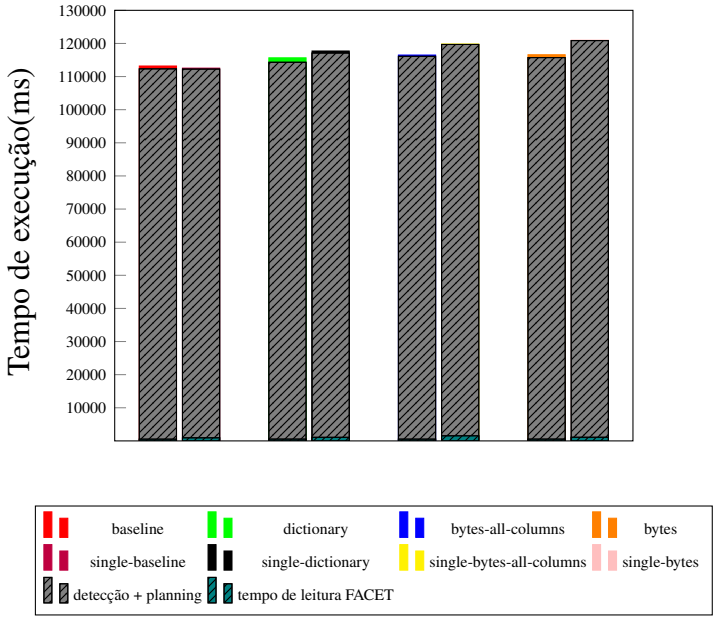


Figura 6.15: TPC-H(ϕ_5) usando o método para exibir tuplas

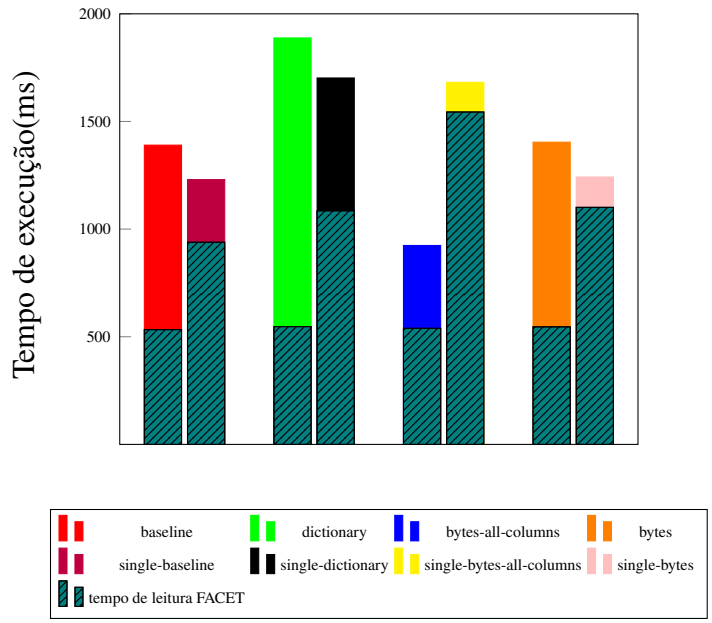


Figura 6.16: TPC-H(ϕ_5) usando o método para exibir tuplas, ignorando os tempos de detecção + *planning*

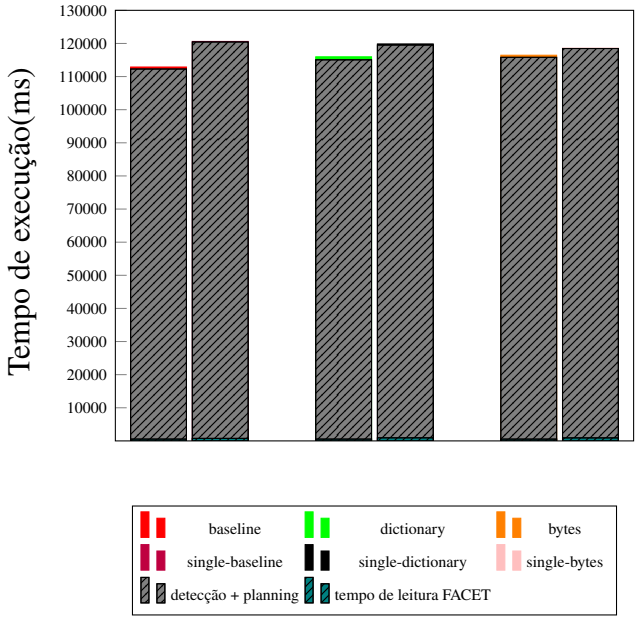


Figura 6.17: TPC-H(ϕ_5) usando o método para exibir tuplas especificando colunas

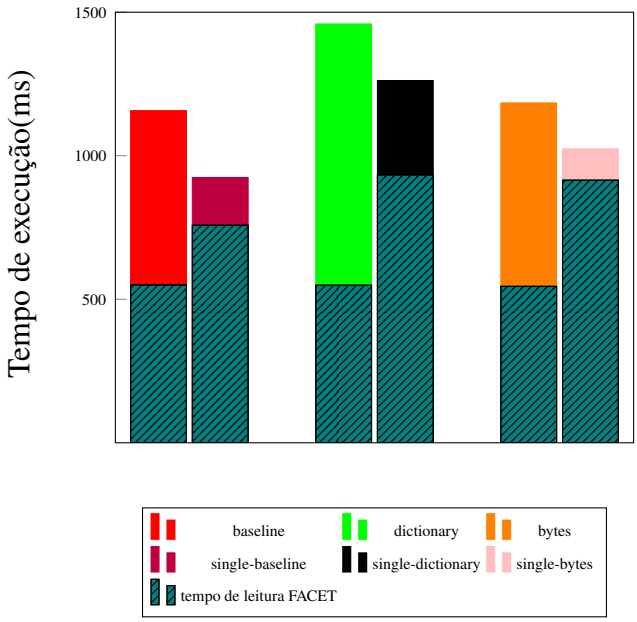


Figura 6.18: TPC-H(ϕ_5) usando o método para exibir tuplas especificando colunas, ignorando os tempos de detecção + *planning*

Novamente as execuções com uma leitura única do arquivo de dados apresentam tempos de execução mais altos em comparação com as versões com releitura. Este é mais um caso em que a fase de detecção e *planning* esta sendo afetada negativamente nas versões de leitura única. Mais uma vez, existe a suspeita de que a tabela de projeção em memória possa estar influenciando na *cache* do processador, o que poderia estar afetando o desempenho desse estágio.

Do mesmo modo que a DC ϕ_4, as execuções desconsiderando o tempo de detecção e *planning* mostram a vantagem de leitura única, mas da mesma forma não é possível concluir que ela seja uma implementação melhor sem que sejam realizados mais experimentos.

6.3.6 TPC-H(ϕ_6)

Para o *dataset* TPH-C e DC ϕ6, é utilizado novamente apenas método de projeção de tuplas. Os resultados são mostrados nas figuras 6.19 e 6.21, enquanto as figuras 6.20 e 6.22 exibem a mesma amostragem sem os tempos de detecção e *planning*.

Diferente do que ocorre com os experimentos com as DCs ϕ_4 e ϕ_5, desta vez o tempo de detecção e *planning* do FACET não foi significativamente afetado nas versões com leitura única. Suspeita-se que a diferença inexista aqui por conta da menor carga de processamento com a qual o FACET tem de lidar nessa etapa para este experimento, o que pode ser percebido pelo fato do tempo de execução ser consideravelmente menor do que nos experimentos com as DCs ϕ_4 e ϕ_5. Por fim, pode-se verificar que, mesmo com um tempo de leitura do FACET mais elevado nas versões de leitura única, elas foram, em geral, melhores do que a versão com releitura, especialmente ao projetar todas as colunas do *dataset*.

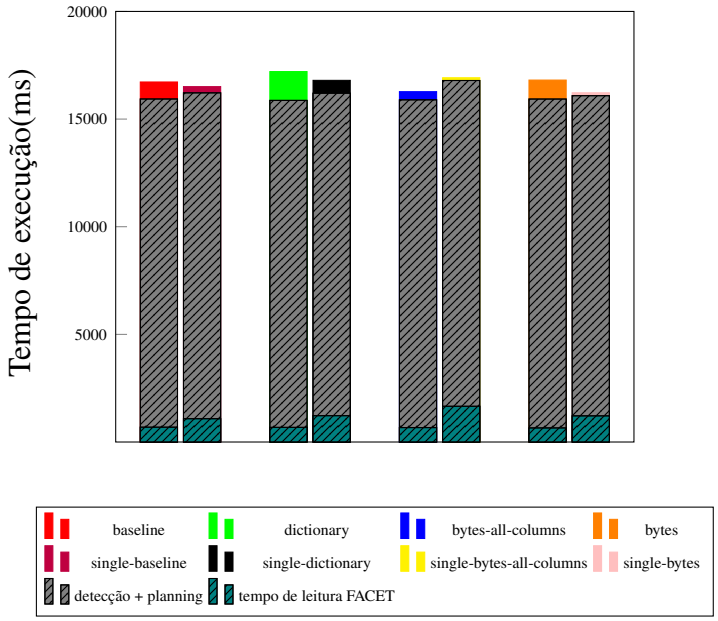


Figura 6.19: TPC-H(ϕ_6) usando o método para exibir tuplas

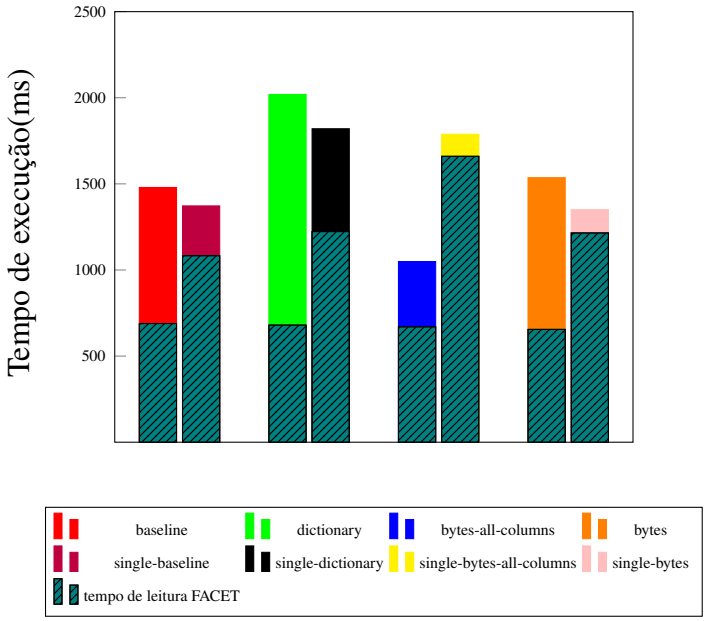


Figura 6.20: TPC-H(ϕ_6) usando o método para exibir tuplas, ignorando os tempos de detecção + *planning*

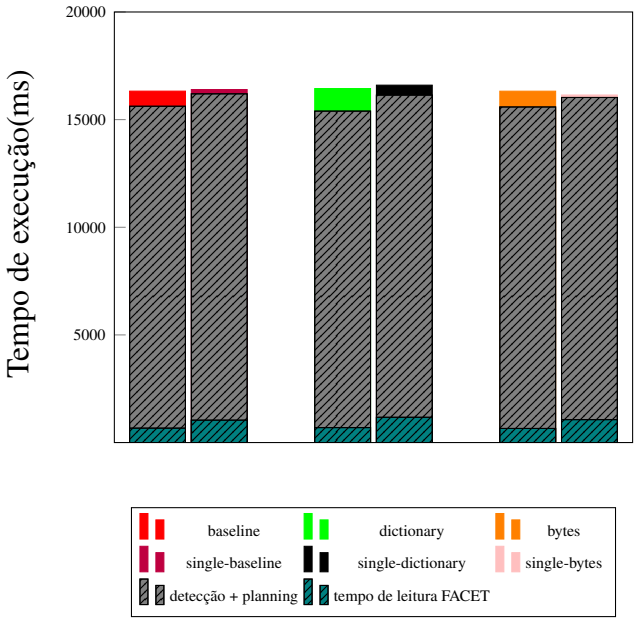


Figura 6.21: TPC-H(ϕ_6) usando o método para exibir tuplas especificando colunas

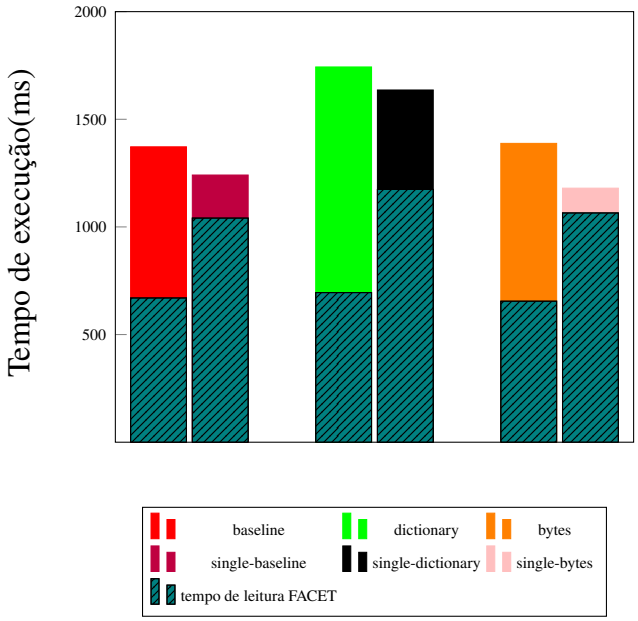


Figura 6.22: TPC-H(ϕ_6) usando o método para exibir tuplas especificando colunas, ignorando os tempos de detecção + *planning*

6.3.7 *Tax*(ϕ_7)

O *dataset Tax* com a DC ϕ_7 é um teste interessante, visto que, pela tabela 6.3, o número de violações neste caso é zero. É positivo incluir este resultado específico para mostrar que, em casos onde não há violações encontradas, a versão com releitura supera consideravelmente a versão de leitura única em termos de performance de tempo. Como na versão com releitura, ao construir a tabela de projeção já é sabido que não foram encontradas violações, não é necessário reler o arquivo de dados e nem construir a tabela, mas simplesmente escrever o cabeçalho da projeção no arquivo de projeção e encerrar a execução. Já na versão de leitura única, como a tabela de projeção é construída antes da detecção de violações, é necessário construí-la com todas as tuplas do arquivo de dados, desperdiçando tempo e memória construindo uma estrutura de dados que acaba não sendo utilizada. O resultado pode ser visto na Figura 6.23.

Deve-se apontar que para este gráfico específico, como não existe violações para projeção e para diferenciar as execuções das versões, optou-se por destacar os tempos de detecção e *planning* nas cores que foram utilizadas para os tempos de projeções nos gráficos anteriores.

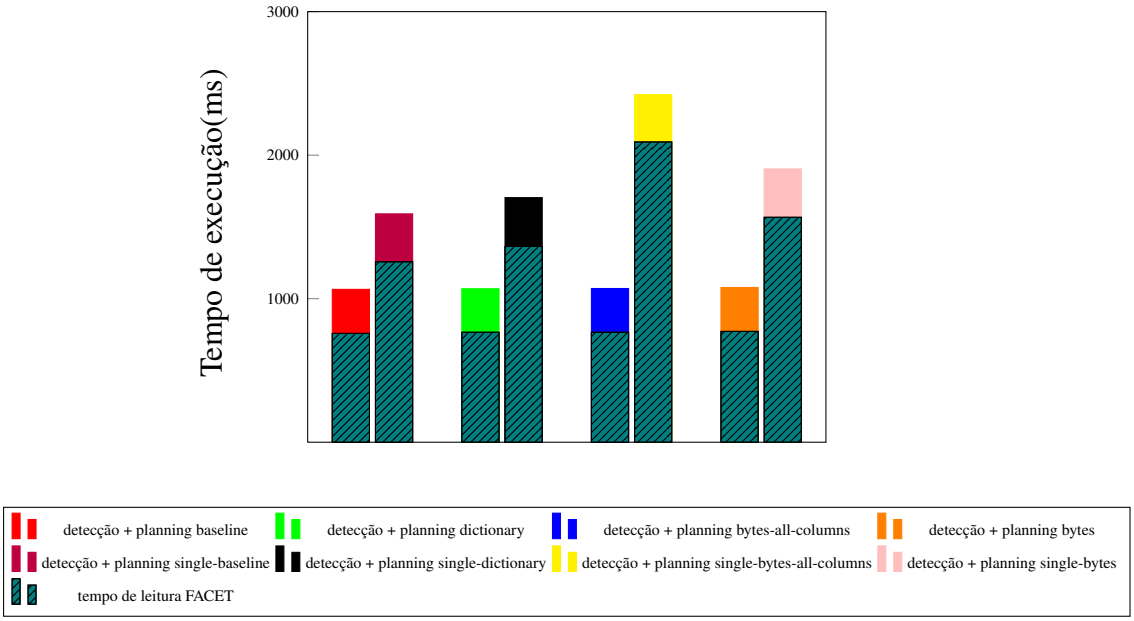


Figura 6.23: Impacto no tempo de execução do FACET entre versões single e leitura dupla usando a base Tax(ϕ_7) e o método de projeção de tuplas

6.3.8 Conclusão dos Resultados

De acordo com os resultados dos experimentos, não há uma versão ou abordagem que tenha se destacado como superior, mas eles permitem a visualização das situações em que cada uma das versões se destaca e ajudam a levantar hipóteses sobre as causas dessas diferenças. Entretanto, mais testes são necessários para aumentar a confiança na corretude de tais hipóteses.

Antes de expor as conclusões, é preciso deixar claro que quando o tempo de execução de cada uma das versões se refere ao tempo total de execução da versão estendida do FACET, e não somente do tempo de projeção. Essa postura foi adotada pois é esperado que a abordagem de leitura do arquivo de dados somente uma vez aumente o tempo de leitura do FACET e reduza o tempo da projeção quando comparada com a abordagem que inclui releitura. Além disso, supõe-se que a abordagem de leitura única pode também afetar o tempo de detecção e *planning* do FACET.

As versões *dictionary* e *single-dictionary* não apresentaram resultados satisfatórios. Mesmo sendo as implementações mais complexas, de maneira geral, os resultados obtidos por elas ficaram piores ou no mesmo nível das demais versões. Uma atenção especial deve ser dado aos resultados das Figuras 6.5 e 6.7, onde as versões aqui analizadas foram muito piores do que todas as demais, provavelmente devido ao elevado número de colunas do *dataset flights* e ao *overhead* esperado dessas versões por fazerem conversões complexas de dados, especialmente em colunas dos tipos *string* e *float*, dos quais o *dataset* em questão é composto.

Em relação a versão *single-bytes-all-columns*, pode-se verificar que, na maioria dos casos, ela foi inferior em performance em relação às demais versões, sendo em média superior somente às versões *dictionary* e *single-dictionary*. Isso se deve ao fato explicado em 5.2.4, isto é,

da necessidade de concatenar as *strings* lidas do arquivo de dados para o construtor das tuplas da tabela de projeção.

Agora, olhando para a versão *bytes-all-columns*, pode-se verificar que ela superou as demais versões em grande parcela dos experimentos, mesmo que não o tenha feito de maneira tão expressiva. Este resultado é esperado, dado que esta versão é a com menor processamento necessário para o envio para o *buffer* de saída dos valores das colunas das tuplas. Mas é preciso lembrar que esta versão tem uma limitação, que é a de não permitir que sejam selecionadas as colunas que devem ser projetadas, o que restringe seu uso. Mesmo assim, para aproveitar os seus bons resultados, ela poderia ser utilizada sempre que as colunas a serem projetadas não forem especificadas na entrada, caso onde sua limitação não teria impacto algum.

Comparando agora as versões *baseline* e *single-baseline* com as versões *bytes* e *single-bytes*, pode-se verificar que seus resultados são muito semelhantes, com uma leve vantagem para as versões de leitura única de ambas as implementações. Uma excessão à semelhança nos resultados está nos experimentos com o *dataset flights* e a DC ϕ_2, que pode ser visto na Figura 6.5, onde pode-se verificar uma clara vantagem da versão *bytes* na projeção de pares de tuplas de todas as colunas. A hipótese aqui é de que essa vantagem ocorra devido ao alto número de colunas do *dataset flights* e por conta do fato de, diferente das versões *baseline* e *sigle-baseline*, as versões *bytes* e *single-bytes* não precisarem iterar pelas colunas ao enviar os dados para o *buffer* de saída durante a iteração pelos pares de tuplas ao executar a projeção.

Por fim, pode-se verificar que, ao comparar as versões de leitura única e leitura dupla, há uma pequena vantagem para as versões de leitura única. Note, entretanto, que esta vantagem só ocorre naqueles experimentos que executaram em menos tempo (que são a maioria em nossos experimentos). Para os experimentos onde mais tempo foi necessário e grande parte do tempo foi ocupado pela detecção e *planning* do FACET, como nas Figuras 6.11, 6.13, 6.15 e 6.17, ocorreu uma pequena vantagem das versões com releitura do arquivo de dados. Supõe-se que essa diferença seja causada pelo fato de, nas versões de leitura única, o processamento do FACET ser influenciado pela presença em memória da tabela de projeção, entretanto mais experimentos seriam necessários para afirmar isso com uma maior certeza.

6.4 LIÇÕES APRENDIDAS

As principais lições aprendidas no desenvolvimento deste trabalho se referem à realização dos experimentos, de forma que decidimos relatá-las nesta seção. Vamos descrever alguns dos problemas ocorridos e, em seguida, o que deveríamos ter feito de diferente para evitá-los ou superá-los.

O principal erro cometido foi subestimar a complexidade da realização dos experimentos. Acabamos por deixar muito para o fim do prazo de entrega deste trabalho a execução dos testes e acabamos ficando sem tempo de refiná-los e analizá-los com mais atenção. O correto seria ter iniciado mais cedo a experimentação, de forma que poderíamos, para aqueles testes onde notamos

comportamentos interessantes ou inesperados, explorar em detalhes o que estava ocorrendo em sua execução, analizando, tanto na máquina física quanto na JVM, dados de memória, processamento e escrita em disco, a fim de encontrar evidências que suportassem ou não nossas hipóteses. Além disso, com mais tempo, poderíamos ter elaborado outros experimentos que serviriam para testar outras hipóteses que tinhamos sobre as implementações feitas. Tudo isso teria nos permitido encontrar mais evidências e tirar conclusões mais sólidas sobre os experimentos realizados.

Além disso, percebemos, já ao fim do prazo de desenvolvimento, que havíamos realizado uma quantidade insuficiente de testes e experimentos mais amplos durante o processo de implementação da solução. Enquanto projetávamos e implementávamos nossa solução, testamos somente um subconjunto limitado dos *datasets* e das DCs, inclusive utilizando alguns testes criados por nós mesmos que nos permitiam verificar aspectos críticos relacionados a corretude da implementação. A ausência de testes mais gerais, testando ao menos de tempos em tempos todos os *datasets* e suas DCs, não nos permitiu ver que as versões que estavamos implementando acabavam não diferindo tanto em performance quanto seria interessante e nos enviesaram a considerar suas contribuições maiores do que de fato são, uma vez que, em nosso subconjunto de testes, os resultados eram mais relevantes, o que não se sustentou quando ampliamos o escopo de testes. Deveríamos ter preparados *scripts*, semelhantes ao utilizados nos experimentos, mas simplificados, e os utilizado para avaliar cada uma das versões logo após sua implementação, a fim de ter dados preliminares já durante o desenvolvimento.

7 IMPLEMENTAÇÕES FUTURAS

Vamos listar e explicar rapidamente algumas propostas de implementações futuras que agregariam valor à projeção no FACET proposta neste trabalho. Elas não puderam ser implementadas nem mais exploradas neste trabalho por questões de tempo, mas acreditamos que seriam o caminho a ser seguido em trabalhos futuros relacionados a este.

- **Limitação do número de resultados projetados:** seria interessante oferecer uma opção de limitar o número de pares de tuplas ou tuplas a serem projetados, de maneira que, mesmo em casos com uma grande quantidade de resultados, a projeção ainda poderia ser realizada em tempo factível, permitindo alguma análise dos dados, mesmo que reduzida, inclusive em casos com muitas violações.

- **Intervalo de resultados projetados:** visto os problemas que observamos do método de pares de tuplas, principalmente em relação ao *dataset* TPC-H mostrado na seção 6.3 com aproximadamente 227 bilhões de violações, poderíamos acrescentar um parâmetro para a implementação do nosso trabalho com uma opção para o usuário poder especificar o intervalo de projeções que ele deseja ver.

- **Eliminação de duplicatas:** outra proposta seria implementar a opção de remover duplicatas na projeção tanto de pares de tuplas quanto de tuplas somente, adicionando dois novos operadores de projeção ainda mais semelhantes aos operados de projeção da Álgebra Relacional como estão definidos por Elmasri e Navathe (2016).

- **Decisão automatizada da abordagem de construção da tabela de projeção:** seria possível implementar um mecanismo de decisão automatizada de qual abordagem de construção da tabela de projeção utilizar em cada execução com base nos dados informados. Por exemplo, esperamos que em *datasets* pequenos em relação ao tamanho da memória disponível, utilizar o método de leitura única poderia ser vantajoso, de forma que seria possível estimar a memória disponível para execução, determinar o tamanho do *dataset* e, com isso, optar por uma ou outra abordagem.

- **Compressão:** não chegamos a averiguar profundamente o tema, mas seria possível analisar formas de compressão das tuplas, tanto para reduzir o tempo de projeção como o uso de memória.

- **Projeção de identificadores no lugar de tuplas:** ao invés de projetarmos valores de colunas selecionadas das tuplas, seria possível implementar uma projeção somente de identificadores de tuplas. Por exemplo, poderíamos projetar apenas seus identificadores em relação à linha em que se encontram no arquivo de dados ou seus campos chaves, o

que poderia reduzir consideravelmente o espaço necessário para armazenar o arquivo de saída, permitindo a projeção mesmo quando houvesse uma quantidade muito grande de violações. Acreditamos que essa abordagem poderia ser uma boa opção especialmente para ferramentas que realizam limpeza de dados.

- **Interface gráfica de acesso ao FACET estendido:** Para permitir o acesso à versão estendida do FACET, pretendíamos implementar também um servidor e um cliente HTTP, mas devido aos prazos, decidimos deixar tal implementação para o futuro. Um cliente HTTP seria disponibilizado para execução local na máquina dos usuários, enquanto que um servidor HTTP seria hospedado na nuvem, sendo acessível somente remotamente, ou também disponível para execução na máquina local do usuário, a depender de decisão futura. O servidor teria apenas uma rota, responsável por receber parâmetros que seriam utilizados para executar a versão estendida do FACET, enviando como resposta o arquivo de projeção gerado durante a execução. O cliente, seria responsável por fazer requisições parametrizadas pelo usuário para o servidor; armazenar na máquina do usuário os arquivos de projeção recebidos nas respostas do servidor; e exibir os resultados de execução a partir dos arquivos de projeção, sejam eles pares de tuplas que violam a DC analizada ou as tuplas envolvidas nessas violações.

8 CONCLUSÃO

A implementação dos operadores de projeção, tanto dos pares de tuplas que violam uma DC quanto das tuplas que fazem parte de alguma violação, foi bem-sucedida e é uma contribuição relevante para a área de limpeza de dados, na medida que potencializa a utilidade do FACET como uma ferramenta para garantir a qualidade de dados.

As alternativas de implementações e abordagens apresentadas permitiram explorar diversas possíveis otimizações, descobrindo, a partir dos experimentos, os casos onde, a primeira vista, cada uma delas desempenha melhor, de maneira que podem ser combinadas em uma implementação futura em que, a partir dos dados de entrada, tanto as implementações quanto as abordagens sejam dinamicamente definidas.

Os resultados que foram obtidos já são positivos, mas ainda é necessário outros experimentos para levantar mais evidências que corroborem as hipóteses apontadas, a fim de que seja possível compreendendo melhor cada uma das alternativas, encontrar novas formas de aprimorá-las. Além disso, o processo de desenvolvimento trouxe variadas ideias de como acrescentar novas funcionalidades que possam contribuir ainda mais com a utilização da projeção no processo de limpeza de dados.

REFERÊNCIAS

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

Cormen, T. H., Leiserson, C. E., Rivest, R. L. e Stein, C. (2009). *Introduction to algorithms*. MIT Press, Cambridge, Massachusetts London, England, 3rd ed edition.

Elmasri, R. e Navathe, S. (2016). *Fundamentals of database systems*. Pearson, Hoboken, NJ, seventh edition edition. OCLC: ocn913842106.

Fan, W. (2015). Data quality: From theory to practice. *SIGMOD Rec.*, 44(3):7–18.

Fan, W., Geerts, F., Jia, X. e Kementsietsidis, A. (2008). Conditional functional dependencies for capturing data inconsistencies. *ACM Trans. Database Syst.*, 33(2).

Fan, W., Tian, C., Wang, Y. e Yin, Q. (2021). Parallel discrepancy detection and incremental detection. *Proc. VLDB Endow.*, 14(8):1351–1364.

Geerts, F., M. G. P. P. e. a. (2020). Cleaning data with llunatic. *Proc. VLDB Endow.*, 29(4):867–892.

Li, Y., Nadal, S. e Romero, O. (2022). A data quality framework for graph-based virtual data integration systems. Em *Advances in Databases and Information Systems: 26th European Conference, ADBIS 2022, Turin, Italy, September 5–8, 2022, Proceedings*, página 104–117, Berlin, Heidelberg. Springer-Verlag.

Pena, E. H. M., de Almeida, E. C. e 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. e Naumann, F. (2020). Efficient detection of data dependency violations. Em *Proceedings of the 29th ACM International Conference on Information amp; Knowledge Management*, CIKM '20, página 1235–1244, New York, NY, USA. Association for Computing Machinery.

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

Tobias Bleifuß, Sebastian Kruse, F. N. (2017). Efficient denial constraint discovery with hydra. *Proc. VLDB Endow.*, 11(3):311–323.

Zuhair Khayya, William Lucia, M. S. M. O. P. P. J.-A. Q.-R. N. T. P. K. (2015). Lightning fast and space efficient inequality joins. *Proc. VLDB Endow.*, 8(13):2074–2085.