



UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE SISTEMAS DE INFORMAÇÃO

Rafael Barbaresco

Extensão de compatibilidade para tipos de dados *range* e operadores de conjuntos do banco de dados PostgreSQL para H2

Florianópolis
2024

Rafael Barbaresco

Extensão de compatibilidade para tipos de dados *range* e operadores de conjuntos do banco de dados PostgreSQL para H2

Trabalho de Conclusão de Curso submetido ao curso de Sistemas de Informação do Centro Tecnológico da Universidade Federal de Santa Catarina como requisito parcial para a obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Ronaldo dos Santos Mello, Dr.
Coorientadora: Profa. Patrícia Vilain, Dra.

Florianópolis

2024

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Barbaresco, Rafael

Extensão de compatibilidade para tipos de dados range e operadores de conjuntos do banco de dados PostgreSQL para H2 / Rafael Barbaresco ; orientador, Ronaldo dos Santos Mello, coorientadora, Patrícia Vilain, 2024.
169 p.

Trabalho de Conclusão de Curso (graduação) -
Universidade Federal de Santa Catarina, Centro Tecnológico,
Graduação em Sistemas de Informação, Florianópolis, 2024.

Inclui referências.

1. Sistemas de Informação. 2. banco de dados. 3. teste de software. 4. H2 database. 5. PostgreSQL. I. Mello, Ronaldo dos Santos. II. Vilain, Patrícia. III. Universidade Federal de Santa Catarina. Graduação em Sistemas de Informação. IV. Título.

Rafael Barbaresco

Extensão de compatibilidade para tipos de dados *range* e operadores de conjuntos do banco de dados PostgreSQL para H2

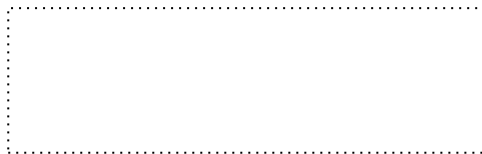
Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do título de Bacharel e aprovado em sua forma final pelo Curso de Sistemas de Informação.

Florianópolis, 27 de junho de 2024.




Coordenação do Curso

Banca examinadora



Prof. Ronaldo dos Santos Mello, Dr.

Orientador



Profa. Patrícia Vilain, Dra.

Coorientadora



Prof. Raul Sidnei Wazlawick, Dr.

Universidade Federal de Santa Catarina

Florianópolis, 2024.

RESUMO

A prática de testes de software tem crescido consideravelmente nos últimos anos. O processo de integração contínua exige executar os testes muitas vezes durante o desenvolvimento de software. Esse processo, em alguns casos, é contraído devido ao grande tempo de execução dos testes ou incompatibilidades entre os ambientes de testes e de produção. Testes que envolvem banco de dados (BD) costumam ser demorados devido ao tempo de acesso à memória secundária. Para mitigar o tempo de acesso ao BD durante os testes, é comum utilizar um BD de alto desempenho diferente do BD utilizado em produção, uma vez que ele possui menor tempo de acesso aos dados. Porém, isso pode causar incompatibilidades entre ambientes de teste e produção. O H2 é um Sistema de Gerência de BD (SGBD) em memória, de código aberto, que oferece um modo de compatibilidade para simular o funcionamento de outros SGBDs populares, entre eles o PostgreSQL. O H2 não oferece compatibilidade completa de todas as funções de outros SGBDs, limitando-se ao padrão SQL. A proposta deste trabalho é implementar, no H2, compatibilidade do tipo de dado "*range*", exclusivo do PostgreSQL e fora do padrão SQL, bem como as funções e operações relacionadas. A avaliação da implementação é feita explorando todos os cenários identificados que envolvem o uso do novo tipo de dado, bem como a aplicação da extensão em um caso de uso real. Espera-se com este trabalho estabelecer uma base para futuras extensões do modo de compatibilidade do H2, além da compatibilidade do tipo "*range*".

Palavras-chave: banco de dados; analisador de código; teste de software; integração contínua; H2 Database; PostgreSQL.

ABSTRACT

The practice of software testing has grown considerably in recent years. The continuous integration process requires running tests many times during software development. This process, in some cases, is slowed down due to long test execution times or incompatibilities between test and production environments. Tests involving databases (DB) tend to take a long time due to the access time to secondary memory. To mitigate DB access time during tests, it is common to use a high-performance DB different from the DB used in production, as it has shorter data access time. However, this can cause incompatibilities between test and production environments. H2 is an open source in-memory DB Management System (DBMS) that offers a compatibility mode to simulate the functionalities of other popular DBMSs, including PostgreSQL. H2 does not offer complete compatibility of all functions of other DBMSs, being limited to the SQL standard. The purpose of this work is to implement, in H2, compatibility with the "*range*" data type, exclusive to PostgreSQL and outside the SQL standard, as well as related functions and operations. The implementation validation is done exploring all identified scenarios that involve the use of the new data type, as well as the application of the extension in a real use case. This work is expected to establish a basis for future extensions of the H2 compatibility mode, beyond "*range*" compatibility.

Keywords: databases; code analyzer; software testing; continuous integration; H2 Database; PostgreSQL.

SUMÁRIO

1 INTRODUÇÃO	9
1.1 VISÃO GERAL	9
1.2 OBJETIVOS	11
1.2.1 Objetivo Geral	11
1.2.2 Objetivos Específicos	11
1.2.3 Delimitação do Escopo do Trabalho	11
1.3 MÉTODO	12
1.4 CONTEÚDO DA MONOGRAFIA.....	12
2 FUNDAMENTAÇÃO TEÓRICA	14
2.1 PADRÃO SQL	14
2.2 POSTGRESQL.....	14
2.2.1 Tipo de Dado <i>Range</i>	14
2.2.2 Operadores e Funções com <i>Range</i>	18
2.3 H2	20
2.3.1 Modo de Compatibilidade	20
2.3.2 Processamento de Consultas	21
2.3.3 Representação de Dados e Tipos de Dados	24
3 UMA EXTENSÃO AO SGBD H2 PARA O SUPORTE AO TIPO DE DADO <i>RANGE - H2 EXTENDED</i>	26
3.1 INTERPRETAÇÃO DE NOVOS TIPOS DE DADOS	26
3.2 INTERPRETAÇÃO DE FUNÇÕES E OPERADORES.....	29
4 PROCESSO DE DESENVOLVIMENTO	33
4.1 FRAMEWORK H2	33
4.2 REQUISITOS FUNCIONAIS	34
4.3 REQUISITOS NÃO-FUNCIONAIS.....	35
4.4 DESENVOLVIMENTO DA EXTENSÃO	36
4.4.1 Criação e Seleção do Parser Secundário	37
4.4.2 Criação da Representação do Novo Tipo de Dado <i>Range</i>	42
4.4.3 Interpretação das Funções	45
4.4.4 Interpretação e Resolução dos Operadores	48
5 CENÁRIOS COBERTOS E RESULTADOS	52
5.1 CONSTRUÇÃO E DECLARAÇÃO DE VALORES.....	54
5.2 OPERAÇÕES COM TABELA.....	56
5.3 OPERADORES ESPECIAIS	58

5.3.1 União, Diferença e Intersecção	58
5.3.2 Operações Condicionais	61
5.3.3 Enfileiramento e Prioridade de Operadores.....	65
5.4 FUNÇÕES ESPECIAIS	66
5.5 APLICAÇÃO EM UM CASO DE USO	68
6 CONCLUSÕES E TRABALHOS FUTUROS.....	73
REFERÊNCIAS	76
APÊNDICE A – Código Principal.....	77
APÊNDICE B – Código de Testes	113
APÊNDICE C – Artigo	160

1 INTRODUÇÃO

1.1 VISÃO GERAL

Testes de software é uma área promissora que cresceu muito nos últimos anos e estima-se que vai continuar crescendo a uma taxa de 6% ao ano até 2026 (Global Market Insights, 2019). Mudanças no processo de desenvolvimento de software, aumento de testes em equipes ágeis, aumento do consumo de aplicações móveis e crescimento de inteligência artificial e *machine learning* são fatores que contribuem para o aumento de testes de software (Global Market Insights, 2019).

O processo de integração contínua, aplicado aos modelos atuais de desenvolvimento de software, implica executar diversas suítes de teste a cada integração de novas funcionalidades no software (Sommerville, 2011, p. 65). Isso exige subir a aplicação e o Banco de Dados (BD) a cada integração, sendo o BD descartado ao final da execução dos testes. Apesar da integração contínua ser uma boa ideia, ela pode ser inviável devido ao tempo de execução dos testes ou incompatibilidades entre o ambiente de teste e o ambiente de produção (Sommerville, 2011, p. 698).

Uma prática comum para testes automatizados que envolvem BD é a utilização de um sistema gerenciador de BD (SGBD) com persistência em memória principal (RAM) para o ambiente de testes, enquanto a aplicação em produção ainda usa um SGBD de persistência em memória secundária (HD ou SSD). Essa prática é realizada devido à natureza rápida e volátil dos BDs em memória (Paris Technologies, 2020). Um BD em memória armazena todos os dados na memória principal, tornando-o mais rápido que BDs tradicionais de persistência em memória secundária, visto que o acesso à memória principal é muito mais rápido que o acesso à memória secundária. A desvantagem de armazenar os dados na memória principal ao invés da secundária é que os dados se tornam voláteis. Ao encerrar a aplicação todos os dados armazenados são perdidos. Casos de uso onde o conteúdo armazenado no BD só é relevante durante a execução da aplicação, como a execução de testes automatizados, favorecem o uso de um BD em memória principal.

Porém, a prática de utilizar, durante os testes, um SGBD diferente do SGBD utilizado em produção gera controvérsias (Hauer, 2017). Diversos problemas são

derivados da incompatibilidade e singularidades dos SGBDs, como diferentes formas de tratar uma mesma consulta ou a ausência de certos tipos de dados.

Comparando com outras formas de testar o BD durante testes automatizados, o problema de utilizar BD em memória principal está na confiabilidade dos testes. Testes de aplicações que usam um SGBD com persistência estritamente em memória secundária requerem um SGBD alternativo para que o teste seja executado apenas em memória principal. As diferenças no comportamento entre os SGBDs potencialmente comprometem o resultado dos testes. Pesquisas em meios de publicação de literatura revelam que existe uma carência de trabalhos que visam aprimorar a compatibilidade entre SGBDs de persistência em memória secundária e principal.

Um SGBD é um software que facilita o processo de definição, construção, manipulação e compartilhamento de BDs entre diversos usuários e aplicações (Elmasri; Navathe, 2015, p. 3-4). Existem diversos SGBDs disponíveis no mercado e, mesmo adotando o padrão SQL (ISO 9075), existem diferenças entre eles, derivados das ferramentas base para a construção do sistema ou decisões de projeto. Além das diferenças dentro do padrão SQL, muitos SGBDs oferecem funções exclusivas, fora do padrão SQL, que se destacam diante de outros SGBDs.

O SGBD Hypersonic 2 (H2) é um SGBD de persistência em memória principal, com código aberto, que tenta mitigar as incompatibilidades com outros SGBDs oferecendo um modo de compatibilidade. Porém, é muito trabalhoso manter o projeto atualizado com tudo o que é desenvolvido nos outros SGBDs. Por esse motivo, o modo de compatibilidade do H2 limita-se a somente alguns SGBDs populares e funcionalidades especificadas no padrão SQL.

O SGBD PostgreSQL, por sua vez, é um SGBD relacional tradicional, também com código aberto, e com armazenamento em memória secundária. O PostgreSQL é um SGBD popular, e por isso o H2 possui o modo de compatibilidade com ele. Uma singularidade do PostgreSQL é o tipo de dado "*range*", que define um conjunto de valores contínuos ou discretos (intervalo), bem como funções e operações com esse tipo de dado. Por ser uma singularidade fora do padrão SQL, o SGBD H2 não suporta operações com esse tipo de dado, mesmo em seu modo de compatibilidade.

Assim sendo, este trabalho visa criar uma extensão de compatibilidade do SGBD H2 para que ele reconheça o tipo de dado "*range*" e as funções e operações relacionadas. Considerando que o H2 possui suporte limitado para adaptações do

analisador de código, e que não existem APIs públicas que expandem seu modo de compatibilidade, este trabalho também oferece uma base para futuras expansões do modo de compatibilidade do H2.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

O objetivo geral deste trabalho é desenvolver uma extensão para o SGBD H2 para suporte de compatibilidade do tipo de dado "*range*", incluindo operadores de conjunto e demais funções relacionadas a esse tipo de dado presente no SGBD PostgreSQL.

1.2.2 Objetivos Específicos

Os objetivos específicos do presente trabalho são:

Objetivo 1: Síntese do conhecimento de SGBDs e analisadores de código;

Objetivo 2: Análise do estado da arte em relação aos SGBDs PostgreSQL e H2, bem como a compatibilidade entre eles;

Objetivo 3: Desenvolvimento de uma extensão do H2 para compatibilidade de tipos, funções e operações "*range*" do PostgreSQL;

Objetivo 4: Avaliação dos resultados.

1.2.3 Delimitação do Escopo do Trabalho

O escopo do módulo proposto limita-se à compatibilidade das consultas SQL e CRUD, envolvendo o tipo nativo de "*range*" do PostgreSQL, bem como as funções e operações relacionadas, para o SGBD H2. Além disso, funções que exigem um pré-requisito que o H2 ainda não oferece, como a criação dinâmica de novos tipos de dado (UDT - User Defined Type), não são consideradas neste trabalho.

Os UDTs são estruturas de dados personalizadas criadas pelos usuários para atender a requisitos específicos de modelagem de dados (Elmasri; Navathe, 2015, p. 247). No contexto dos BDs, os UDTs permitem a definição de novos tipos de dados além dos tipos primitivos fornecidos pelo SGBD. No entanto, neste projeto, o escopo

foi restringido às funcionalidades existentes no H2, excluindo considerações relacionadas à criação dinâmica de UDTs, que não estão atualmente suportadas pelo SGBD em questão.

1.3 MÉTODO

O método de pesquisa utilizado neste trabalho inicia-se com a identificação do problema. Detalha-se os objetos de estudo através de uma revisão da literatura e seleciona-se as áreas de conhecimento envolvidas no trabalho. Também se verifica as ferramentas e o ambiente necessário para a elaboração do trabalho.

Para o desenvolvimento do trabalho foram realizados testes iniciais no ambiente. A implementação da extensão segue um processo iterativo e incremental (Larman; Basili, 2003), incluindo a análise de requisitos, modelagem, construção de software e testes.

Por fim é feita a análise dos resultados e a validação do projeto, aplicando a extensão em um caso de uso.

1.4 CONTEÚDO DA MONOGRAFIA

Este trabalho é dividido da seguinte forma:

- Capítulo 1: Introdução - Apresentação geral, contextualizando a utilização de BD em memória e a deficiência de BDs em memória compatíveis com as exclusividades do PostgreSQL;
- Capítulo 2: Fundamentação Teórica - Explicação sobre o padrão SQL, as exclusividades do PostgreSQL relacionadas ao tipo de dado *range*, e o funcionamento do H2 no que se refere à compatibilidade com outros SGBDs e interpretação de consultas;
- Capítulo 3: Uma Extensão ao SGBD H2 para o Suporte ao Tipo de Dado Range - H2 Extended - Implementação proposta de uma extensão para compatibilidade dos tipos de dado range para o H2;
- Capítulo 4: Processo de Desenvolvimento - Detalhamento de todo o desenvolvimento do projeto, incluindo formalização dos requisitos, as decisões de projeto, e limitações da abordagem selecionada;

- Capítulo 5: Cenários Cobertos e Resultados - Apresenta as consultas exclusivas do PostgreSQL que a extensão é capaz de reproduzir;
- Capítulo 6: Conclusões e Trabalhos Futuros - Fechamento da monografia, apresentando os objetivos alcançados e pontos de melhoria, bem como sugestões de futuras implementações.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 PADRÃO SQL

A Linguagem de Consulta Estruturada (SQL - Structured Query Language) foi desenvolvida pela IBM nos anos 70 com o propósito de ser uma linguagem compreensível para pessoas se comunicarem com BDs relacionais. Ao longo dos anos foram surgindo variações da linguagem, e foi necessário formalizar um padrão para a língua. Atualmente o padrão utilizado internacionalmente é ISO/IEC 9075, "Tecnologia da informação - linguagens de banco de dados - SQL", e sua última atualização foi em 2016.

A maioria dos SGBDs populares estão em conformidade com o padrão SQL, incluindo o PostgreSQL e H2. Porém, o PostgreSQL não se limita ao padrão SQL.

2.2 POSTGRESQL

O PostgreSQL é um SGBD relacional, de código aberto, escrito em linguagem C. Foi criado em 1986 e, com mais de 30 anos de desenvolvimento ativo, é um SGBD robusto, consolidado e com grande reputação (PostgreSQL Global Development Group, 2022). A versão mais recente do PostgreSQL (versão 14, lançada em setembro de 2021) está em conformidade com 170 das 179 características mandatórias do padrão SQL. O PostgreSQL também oferece muitos recursos que vão além do padrão SQL, que podem ser tipos de dados, integridade de dados, otimização de performance e concorrência, confiabilidade, recuperação de desastres, segurança, extensibilidade, internacionalização e busca por texto. Um desses recursos, relativamente simples comparado aos demais recursos adicionais, é o tipo de dado "*range*", utilizado para representar intervalos de valores.

2.2.1 Tipo de Dado *Range*

Range é um tipo de dado exclusivo do PostgreSQL que representa um intervalo de valores de outro tipo conhecido (um "subtipo"). Esse tipo de dado também é acompanhado de diversas operações sobre intervalos. É um tipo de dado útil para casos onde operações são realizadas sob o intervalo entre os valores limites, ao invés

dos limites em si. Um exemplo simples de visualizar sua utilização é a representação de horários de uma agenda de um profissional qualquer, onde pode-se utilizar intervalos de datas para definir agendamentos. Com uma simples operação de sobreposição de intervalos, pode-se determinar conflitos com demais horários agendados. O Quadro 1 demonstra o exemplo com as consultas SQL.

Quadro 1 – Exemplo de utilização de tipo de dado *range* para reserva de salas.

```
-- Exemplo de reserva de sala, dia 01/01/2020 das 14:30 até 15:30.
CREATE TABLE reserva (sala int, horario tsrange);
INSERT INTO reserva VALUES (1108, '[2020-01-01 14:30, 2020-01-01 15:30)');

-- Exemplo de cálculo de conflito de horário, verificando sobreposição da data e hora
01/01/2020, das 15:00 até 16:00, com o horário registrado na tabela.
SELECT '[2020-01-01 15:00, 2020-01-01 16:00)>::tsrange && (
SELECT r.horario FROM reserva r
);
```

Fonte: Elaborado pelo autor.

Um tipo de dado *range* tem uma estrutura semelhante a uma String ou Varchar. Ele começa com um delimitador inferior (caractere '[' para limite fechado ou '(' para limite aberto), o valor do limite inferior conforme seu subtipo, um separador de limites (caractere vírgula ','), o valor do limite superior conforme seu subtipo, e por fim um delimitador superior (caractere ']' para limite fechado ou ')' para limite aberto). Caso um intervalo de valores não tenha limite superior ou inferior, o seu respectivo valor limite na representação do dado não é preenchido, sendo definido como vazio. Em casos excepcionais, onde um intervalo não possui valores representáveis, o dado é representado apenas com uma String "empty". Outra característica importante é que um tipo de dado *range* não pode representar lacunas. Essa característica afeta as operações de união e subtração de intervalos.

Por padrão, o PostgreSQL oferece os seguintes tipos de dado *range*:

- *int4range* - intervalo de números inteiros do tipo "integer";
- *int8range* - intervalo de números inteiros do tipo "bigint";
- *numrange* - intervalo de números racionais do tipo "numeric";
- *tsrange* - intervalo de data e hora do tipo "timestamp" sem fuso horário;
- *tstzrange* - intervalo de data e hora do tipo "timestamp" com fuso horário;
- *daterange* - intervalo de datas do tipo "date".

Para declarar uma constante de um tipo *range*, pode-se usar a conversão explícita de String para o tipo desejado, ou a função equivalente. A função pode receber 2 ou 3 argumentos, sendo o primeiro o limite inferior, o segundo o limite superior e o terceiro uma String com 2 caracteres, explicitando os tipos de delimitadores desejados. Os únicos valores aceitos para o terceiro argumento são '[]', '[]', '[)' e '()'. Caso não seja enviado o terceiro argumento, a função assume o padrão '[]'. O Quadro 2 demonstra as três formas de declaração de valores *range*.

Quadro 2 – Demonstração das formas de declaração de valores *range*.

```
-- Declaração de intervalos de valores do tipo "integer" utilizando conversão explícita e
função.      As      3      formas      abaixo      são      equivalentes:

SELECT                                             '[2,6)::int4range;
SELECT                                             int4range(2,6);
SELECT int4range(2,6,'[ ]');
```

Fonte: Elaborado pelo autor.

O tipo de dado *range* também possui uma função de canonização para normalizar a representação do dado. Dentre os tipos padrões oferecidos, apenas os tipos *int4range*, *int8range* e *daterange* possuem uma forma canônica '[]', ou seja, limite inferior fechado e limite superior aberto. Sempre que um dado desses tipos é criado fora da forma canônica, uma função de canonização é chamada para normalizar o dado, alterando os delimitadores e somando ou subtraindo uma unidade inteira para manter a equivalência do intervalo representado. Além dos tipos padrões oferecidos, o PostgreSQL também oferece suporte à criação de novos tipos de dado *range*, com uma sintaxe semelhante à UDT. O quadro 3 demonstra a função de canonização em ação, apresentando os valores criados ao declarar explicitamente os delimitadores do intervalo.

Quadro 3 – Demonstração da função de canonização.

```
-- Declaração de intervalo de valores do tipo "integer" fora da forma canônica.

SELECT                                             int4range(2,6,'[ ]');
--                                             resultado                                     '[2,7)'

-- Declaração de intervalo de valores do tipo "numeric", que não possui função de
canonização.

SELECT                                             numrange(2.0,6.0,'[ ]');
-- resultado '[2.0,6.0]'
```

Fonte: Elaborado pelo autor.

O PostgreSQL reconhece algumas palavras-chaves que podem ser utilizadas na construção dos intervalos. Um exemplo é a palavra-chave “today”, que retorna a data atual do sistema. Declarações de valores *range* utilizando palavras-chave dessa forma são permitidas apenas com conversão explícita de tipo. O Quadro 4 demonstra a utilização dessa palavra-chave na declaração de valores *daterange* e *tsrange*.

Quadro 4 – Demonstração da declaração de valores *range* com a palavra-chave “today”.

```
-- Declaração de intervalos do tipo “date” e “timestamp” utilizando a palavra-chave “today”.
SELECT                                     '[2020-01-01,today)::daterange;
-- resultado      '[2020-01-01,X)',      onde X é a data atual.

SELECT                                     '[2020-01-01,today)::tsrange;
-- resultado      '['2020-01-01 00:00:00",X)', onde X é a data atual com horário 00:00:00.

SELECT '[2020-01-01 14:30,today)::tsrange;
-- resultado      '['2020-01-01 14:30:00",X)', onde X é a data atual com horário 00:00:00.
```

Fonte: Elaborado pelo autor.

Os tipos *range* também abrangem o conceito de infinito. Se o limite inferior for omitido ou se for utilizado a palavra-chave “infinity”, aquele limite significa matematicamente infinito negativo. Analogamente, se o limite superior for omitido ou se for utilizado a palavra-chave “infinity”, aquele limite será infinito positivo. A declaração de limites infinitos por omissão do valor só é possível com a conversão explícita, considerando que a sintaxe dos métodos não permite argumentos ausentes. O Quadro 5 demonstra a declaração de limites infinitos.

Quadro 5 – Demonstração de declaração de intervalos com limites infinitos.

```
-- Declaração de intervalos do tipo “integer” com limites infinitos.
SELECT                                     '[3,)::int4range;
-- intervalo que contém todos os valores inteiros iguais ou maiores do que 3.

SELECT                                     '(,5)::int4range;
-- intervalo que contém todos os valores inteiros iguais ou menores do que 5.

SELECT                                     '(,)::int4range;
-- intervalo que contém todos os valores inteiros.
```

Fonte: Elaborado pelo autor.

2.2.2 Operadores e Funções com *Range*

O PostgreSQL dispõe de operadores de intervalos. Esses operadores, em sua maioria, resultam na operação matemática equivalente: contém (@>), está contido (<@), sobreposição (&&), estritamente à esquerda (<<), estritamente à direita (<<), não se estende à direita (&<), não se estende à esquerda (&>), adjacentes (-|-), união (+), intersecção (*) e subtração (-). As exceções à operação matemática real são os operadores de união e subtração. Um tipo de dado *range* não é capaz de representar lacunas. Caso a operação resulte em um intervalo com lacunas, uma exceção é lançada. O quadro 6 demonstra a utilização de cada um dos operadores.

Quadro 6 – Demonstração das operações exclusivas para valores do tipo *range*.

Exemplos de utilização de cada um dos operadores.				
				Contenção
SELECT	int4range(1,10)		@>	5;
--	resultado			true
SELECT	5		<@	int4range(1,10);
--	resultado			true
SELECT	int4range(1,10)		@>	int4range(4,6);
--	resultado			true
SELECT	int4range(1,10)		<@	int4range(4,6);
--	resultado			false
SELECT	int4range(1,10)		@>	int4range(5,15);
--	resultado			false
SELECT	int4range(1,10)		<@	int4range(5,15);
--	resultado			false
				Sobreposição
SELECT	int4range(1,10)		&&	int4range(5,15);
--	resultado			true
SELECT	int4range(1,10)		&&	int4range(10,15);
--	resultado			false
SELECT	int4range(1,10,[''])		&&	int4range(10,15);
--	resultado			true
				Adjacência
SELECT	int4range(1,10)		- -	int4range(5,15);
--	resultado			false
SELECT	int4range(1,10)		- -	int4range(10,15);
--	resultado			true
--	Estritamente	à	esquerda	ou
				direita

SELECT	int4range(1,10)	<<	int4range(100,110);
--	resultado		true
SELECT	int4range(1,10)	<<	int4range(5,15);
--	resultado		false
SELECT	int4range(100,110)	>>	int4range(5,15);
--	resultado		true
SELECT	int4range(1,10)	>>	int4range(5,15);
--	resultado		false
--	Não se estende à		esquerda ou direita
SELECT	int4range(1,10)	<&	int4range(5,15);
--	resultado		true
SELECT	int4range(1,10)	&>	int4range(5,15);
--	resultado		false
SELECT	int4range(5,15)	<&	int4range(1,10);
--	resultado		false
SELECT	int4range(5,15)	&>	int4range(1,10);
--	resultado		true
--			União
SELECT	int4range(1,10)	+	int4range(5,15);
--	resultado		'[1,15]'
--			Intersecção
SELECT	int4range(1,10)	*	int4range(5,15);
--	resultado		'[5,10]'
--			Subtração
SELECT	int4range(1,10)	-	int4range(5,15);
--	resultado '[1,5]'		

Fonte: Elaborado pelo autor.

Além dos operadores de intervalos, o PostgreSQL também oferece funções dedicadas para o uso de intervalos. Felizmente, essas funções estão dentro do padrão SQL, salvo os argumentos de tipo *range*.

As funções oferecidas, além das funções para construção de intervalos já apresentadas, são descritas abaixo:

- *lower(intervalo)* - retorna o limite inferior do intervalo;
- *upper(intervalo)* - retorna o limite superior do intervalo;
- *isempty(intervalo)* - retorna true se o intervalo não contém elementos (ou seja, não é “empty”, conforme detalhado na seção 3.2.1), senão retorna false;

- *lower_inc(intervalo)* - retorna true se o delimitador inferior é inclusivo, senão retorna false;
- *upper_inc(intervalo)* - retorna true se o delimitador superior é inclusivo, senão retorna false;
- *lower_inf(intervalo)* - retorna true se o limite inferior é infinito, senão retorna false;
- *upper_inf(intervalo)* - retorna true se o limite superior é infinito, senão retorna false;
- *range_merge(intervalo1,intervalo2)* - retorna um intervalo que representa a união dos intervalos originais, preenchendo a lacuna entre eles caso não sejam adjacentes.

2.3 H2

O H2, assim como o PostgreSQL, é um SGBD relacional, de código aberto, porém escrito em linguagem Java (H2 Database Group, 2022). Ele foi lançado em 2005, muito mais recente do que o PostgreSQL. O H2 foi desenvolvido com a proposta de ser uma alternativa de alto desempenho para aplicações com baixo volume de dados. Sua configuração é simples e rápida, sendo primariamente configurado para persistência em memória principal. Um BD em memória principal não é confiável para uma aplicação final de produção, sendo sua utilização recomendada apenas para desenvolvimento, testes e operações que envolvem apenas o contexto durante a execução de uma aplicação.

2.3.1 Modo de Compatibilidade

Visando a cooperação do BD em memória principal durante o desenvolvimento, com o produto final usando outro BD de persistência em memória secundária, o H2 oferece um modo de compatibilidade com demais SGBDs populares. São eles IBM DB2, Apache Derby, HSQLDB, MS SQL Server, MySQL, Oracle e PostgreSQL. O propósito do modo de compatibilidade é replicar o comportamento de outros SGBDs, tornando os resultados de consultas equivalentes. Sendo um SGBD relativamente novo comparado aos demais SGBDs tradicionais, e considerando que o H2 possui modo de compatibilidade com diversos outros SGBDs, cada um com suas

exclusividades fora do padrão SQL, o H2 se compromete apenas a manter a compatibilidade dos recursos dentro do padrão SQL. Dessa forma, o tipo de dado *range*, bem como os demais recursos exclusivos do PostgreSQL, não são reconhecidos pelo H2. Além disso, o H2 não tem uma cobertura tão ampla do padrão SQL como o PostgreSQL. Alguns recursos presentes no PostgreSQL, como UDT, por exemplo, não estão presentes no H2.

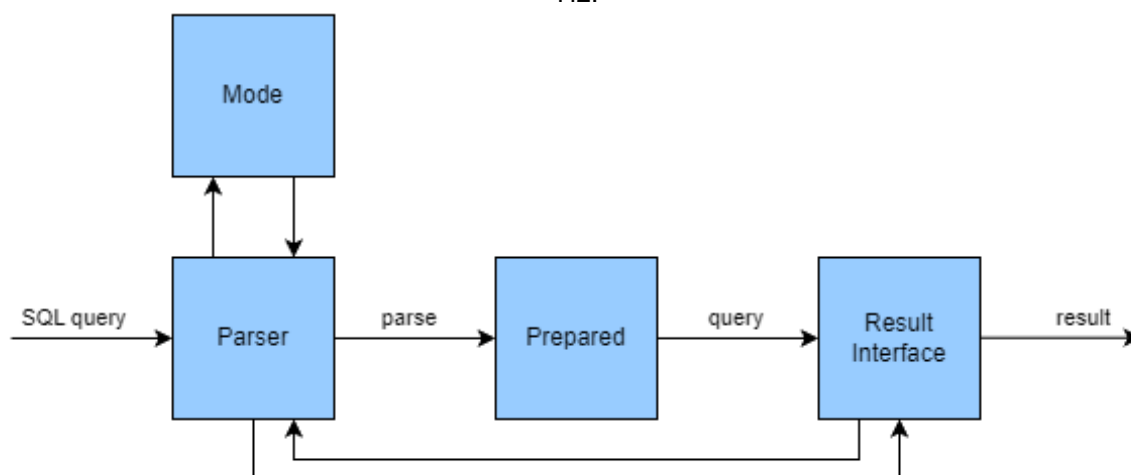
O H2 possui uma classe *Mode* onde são definidas as características de compatibilidade. Para cada SGBD com modo de compatibilidade disponível, é criada uma instância dessa com modificadores que replicam o comportamento daquele SGBD. Esses modificadores, em sua grande maioria, são *flags* ou *enums* que são verificados durante o processamento das consultas. Além das *flags* e *enums*, a classe também possui mapas que expandem ou restringem os tipos de dados reconhecidos pelo H2.

Para o modo de compatibilidade com o PostgreSQL são excluídos os tipos "number", "identity", "tinyint", "blob" e "varchar_ignorecase". Em contrapartida, são adicionados os tipos "jsonb", "money" e "oid". Os tipos adicionados são tratados como *alias* de tipos já conhecidos no H2. No caso dos tipos adicionados no PostgreSQL, respectivamente, são definidos como *alias* dos tipos "json", "numeric" com precisão 19 e escala 2, e "integer".

2.3.2 Processamento de Consultas

O processamento de consultas no H2 pode ser dividido em 3 etapas: preparação, otimização e resolução. Na preparação é feita a compilação do código, transformando a consulta em formato String em uma árvore de objetos. Na otimização são identificados problemas de sintaxe e semântica, bem como a simplificação de elementos na consulta que não dependem de contexto, ou seja, são constantes. Já a etapa de resolução calcula efetivamente o resultado da consulta. A Figura 1 ilustra o relacionamento entre os componentes envolvidos e o processo de interpretação e resolução de consultas.

Figura 1 – Componentes envolvidos no processo de interpretação e resolução de consultas no SGBD H2.



Fonte: Elaborado pelo autor.

Toda a etapa de preparação acontece dentro do objeto *Parser*. Esse objeto recebe a consulta em formato String e a transcreve em uma árvore de objetos *Prepared* e *Expression*, sendo a raiz sempre uma instância de *Prepared* e o restante *Expression*. Classes que estendem a superclasse *Prepared* são implementações de comandos reconhecidos na linguagem SQL (por exemplo, “select”), enquanto classes que estendem a superclasse *Expression* são implementações de parâmetros consumidos por uma instância de *Prepared*. Todos os comandos e expressões estendem diretamente ou indiretamente as classes *Prepared* e *Expression*. Vale ressaltar que a leitura da consulta SQL, nessa etapa de preparação, é feita da esquerda para a direita. Isso significa que algumas variáveis podem ter seu tipo de dado desconhecido até que seja lida a consulta inteira. Essas lacunas na tipagem dos dados devem ser resolvidas na etapa de otimização.

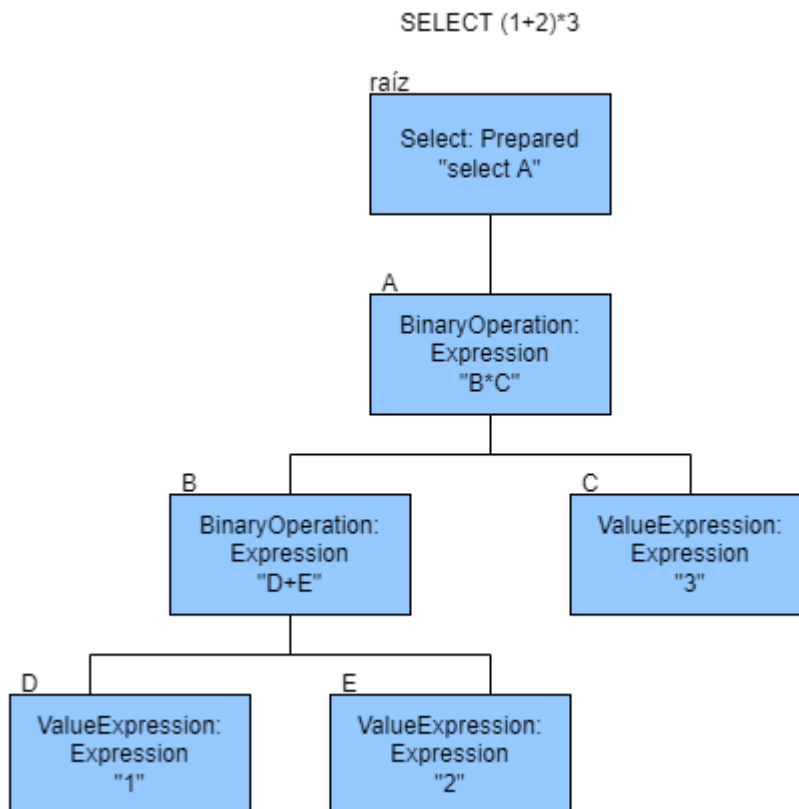
Na etapa de otimização é feita uma análise de todos os tipos de dados envolvidos na consulta. Cada elemento na árvore deve ser capaz de identificar qual será o tipo do seu retorno. Quando um elemento possui outros elementos filhos, o elemento atual consulta recursivamente o tipo de retorno dos seus elementos filhos a fim de identificar com quais tipos de dado ele deve trabalhar. Caso um elemento identifique que seus filhos possuem algum tipo de dado retornado incompatível para sua operação, uma exceção é lançada. Ao final da etapa de otimização, a "consulta", que foi transformada em uma árvore de objetos na etapa anterior, está pronta para ser processada.

Durante o processo de resolução, é executada de fato a consulta SQL. Nesta etapa são recuperados os valores de tabelas e calculadas as operações instruídas na consulta.

Em alguns casos excepcionais, uma nova instância de *Parser* é inicializada. Essa nova instância é utilizada apenas para processar subexpressões da consulta em andamento e não depende do contexto original. Isso significa que a consulta interior não depende da exterior, ou seja, nenhum valor da consulta exterior é passado para a consulta interior. A consulta externa, por sua vez, recebe o resultado da consulta interna como um todo, sem informações de como a consulta interna chegou naquele resultado.

Tomamos como exemplo a consulta `SELECT (1+2)*3`. A árvore criada na etapa de preparação é representada na Figura 2.

Figura 2 – Árvore com o mapeamento da consulta “SELECT (1+2)*3”.



Fonte: Elaborado pelo autor.

É importante ressaltar que, durante as etapas de preparação e otimização, o *Parser* reconhece apenas o tipo de expressão e os tipos dos dados envolvidos. Isso

significa que o *Parser* não reconhece os valores quantitativos 1, 2 e 3 nas folhas do exemplo. Ele reconhece apenas que esses valores são do tipo “integer”.

Após concluída a construção da árvore na etapa de preparação, inicia-se a etapa de otimização. Um método na raiz da árvore é chamado para iniciar o processo de análise, que aciona recursivamente os métodos de otimização dos elementos filhos.

Por fim, após concluída a análise sem erros na otimização, é iniciada a etapa de resolução. Semelhante à etapa anterior, um método na raiz é chamado para buscar o resultado da consulta, que aciona recursivamente os métodos de resolução de todos os filhos na árvore. Dessa forma, as folhas da árvore terão seus valores calculados primeiro e o fluxo de execução vai subindo na árvore até voltar para a raiz.

2.3.3 Representação de Dados e Tipos de Dados

O reconhecimento de tipos de dados no H2 é feito de forma inteiramente estática. Ao iniciar uma instância do H2, este carrega todos os tipos reconhecidos e não é possível adicionar novos tipos em tempo de execução. A representação de dados e seus respectivos tipos é definida, principalmente, por 3 classes: *Value*, *TypeInfo* e *DataType*. Além dessas classes, tipos exclusivos de outros BDs são definidos pela classe *Mode*, como mencionado anteriormente.

A classe *Value* define os valores armazenados em tabelas ou produzidos em tempo de execução. Cada instância de *Value* representa um valor específico. A classe *Value*, em si, é abstrata e possui atributos estáticos com uma numeração de todos os tipos de dados possíveis, semelhante a um *enum*. Para cada tipo de dado, é definida uma classe concreta que estende a classe *Value*. Todos os valores envolvidos numa consulta SQL são instâncias independentes de implementações de *Value*, com exceção de valores nulos. Ao executar uma consulta, o H2 nunca cria uma instância *Value* que representa um valor nulo. Para cada tipo de dado, o H2 possui uma instância única de *Value* para representar o valor nulo daquele tipo. Sempre que uma consulta for produzir um valor nulo, essa instância é resgatada.

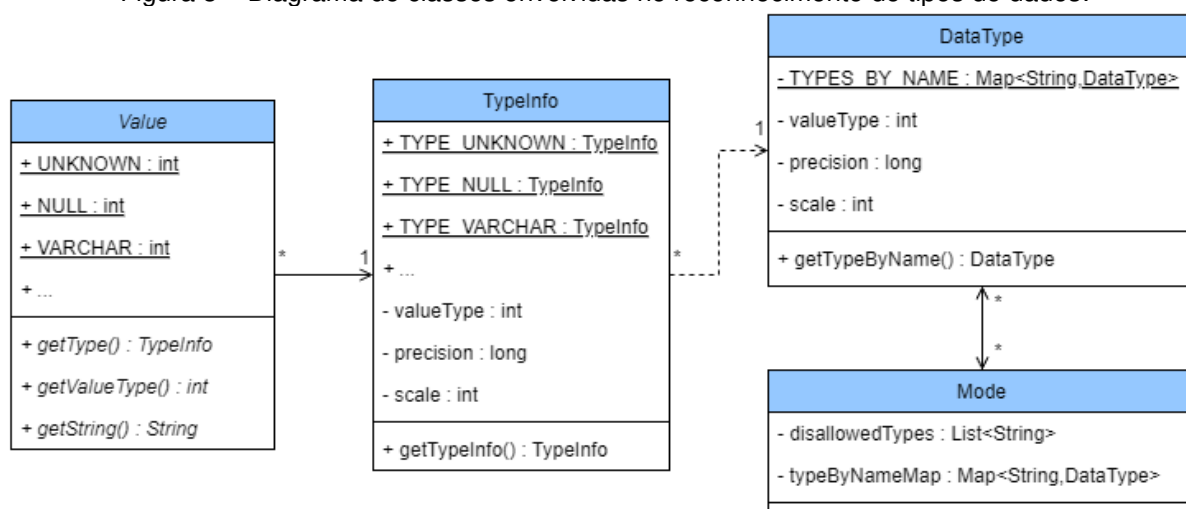
A classe *TypeInfo* possui definições de tipos de dados. Instâncias dessa classe são usadas para determinar os tipos de valores envolvidos numa consulta ou armazenados em determinada coluna de uma tabela. Cada instância dessa classe possui uma referência para o valor estático de tipo correspondente em *Value*, e

demais informações necessárias, como tamanho e precisão. Semelhante à classe *Value*, essa classe também armazena de forma estática instâncias de *TypeInfo* padrão para cada tipo de dado reconhecido. Instâncias dessa classe são obtidas utilizando sua função de busca e construção de *TypeInfo* para que não sejam criadas instâncias redundantes.

A classe *DataType* funciona como um dicionário de tipos de dados em formato String para seus respectivos tipos em *Value* e *TypeInfo*. Cada instância possui o valor de tipo definido em *Value* e informações adicionais para que a classe *TypeInfo* identifique sua instância correspondente. A classe *DataType* também é capaz de traduzir tipos exclusivos de outros bancos, consultando a instância de *Mode* onde são definidas as exclusões e inclusões de tipos.

Por fim, a classe *Mode* possui uma lista com todos os nomes de tipos proibidos em determinado BD, e um mapa de tradução de nomes de tipos permitidos para sua instância de *DataType*.

Figura 3 – Diagrama de classes envolvidas no reconhecimento de tipos de dados.

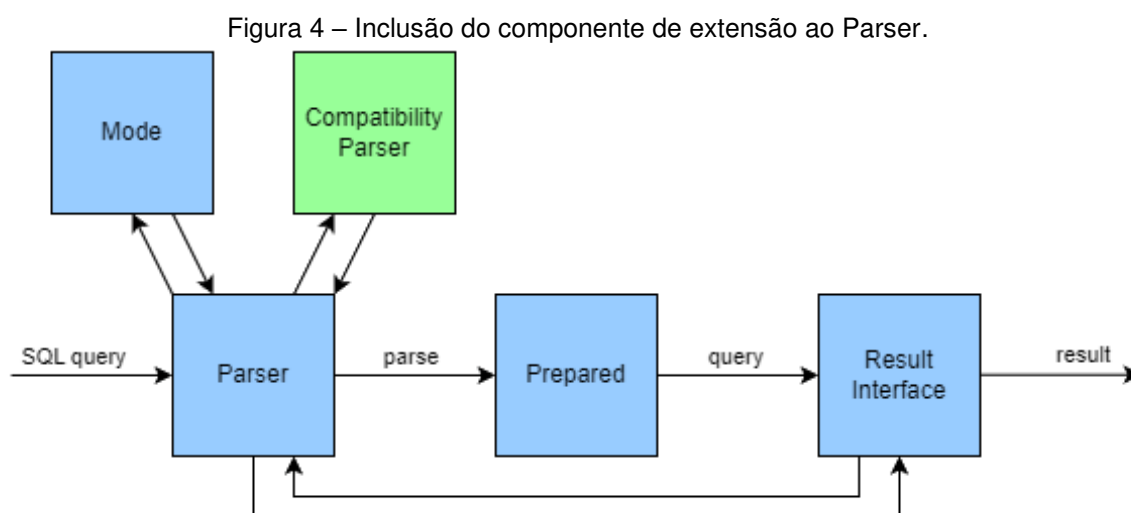


Fonte: Elaborado pelo autor.

De um modo geral, a classe *DataType* é utilizada como um tradutor para *TypeInfo* na etapa de preparação do processamento de consultas, onde o *Parser* traduz uma consulta recebida em formato String e constroi a árvore de objetos. Instâncias de *TypeInfo* são utilizadas nas três etapas para validação da consulta e tomadas de decisão. Ainda instâncias de *Value* são criadas (ou referenciadas, no caso de valores nulos) na etapa de resolução para produzir o resultado final em forma de valores.

3 UMA EXTENSÃO AO SGBD H2 PARA O SUPORTE AO TIPO DE DADO *RANGE* - *H2 EXTENDED*

O *H2 Extended* é uma proposta, definida neste trabalho, de extensão ao SGBD H2 visando expandir o seu modo de compatibilidade. A extensão proposta é a inclusão de um compilador secundário (*CompatibilityParser* ou *Parser secundário*) paralelo ao compilador nativo do H2 (*Parser* ou *Parser primário*). Esse compilador secundário é capaz de interpretar tipos de dados e expressões exclusivas dos outros SGBDs disponíveis no modo de compatibilidade. A Figura 4 complementa o diagrama apresentado na Figura 1, incluindo o novo componente desenvolvido neste trabalho, *CompatibilityParser*, destacado em verde.



Fonte: Elaborado pelo autor.

A intenção deste trabalho é implementar somente o reconhecimento de tipos de dado *range* e o comportamento das funções e operadores relacionados. Posteriormente, o trabalho pode ser expandido para suportar novas funções, operadores e tipos de dados tanto do PostgreSQL quanto dos demais SGBDs suportados.

3.1 INTERPRETAÇÃO DE NOVOS TIPOS DE DADOS

Para permitir ao H2 reconhecer um novo tipo de dado existem duas possíveis abordagens: "profunda" e "superficial". Essas abordagens foram sugeridas pelos moderadores da equipe do H2, em seu canal de suporte (Grandin, 2021). A

abordagem profunda consiste em fazer o H2 reconhecer verdadeiramente um novo tipo de dado, ou seja, o tipo do dado existir nas classes *Value*, *TypeInfo* e *DataType*, enquanto a versão superficial consiste em traduzir o novo tipo de dado para outro tipo já conhecido, utilizando o mapa de *DataType* auxiliar em *Mode*.

A versão profunda pode ser alcançada de duas formas. A primeira forma de implementação é adicionar o novo tipo de dado de forma estática, juntamente com os demais tipos de dados reconhecidos, nas classes *Value*, *TypeInfo* e *DataType* (Figura 3). Essa opção é refutada porque o novo tipo de dado ficaria disponível para todos os modos de compatibilidade. Uma das premissas deste trabalho é que novos tipos de dados devem ser disponíveis apenas ativando o modo de compatibilidade com o SGBD ao qual pertencem.

A segunda forma de implementação da abordagem profunda é adicionar o novo tipo de dado dinamicamente ao ativar um modo de compatibilidade. Essa abordagem é mais complexa que a anterior porque implica em mudanças na estrutura de reconhecimento de tipos de dados, que atualmente é feita de forma completamente estática. Essencialmente, abrangeria a implementação da base de UDT no H2. Essa abordagem foi desencorajada pelos moderadores do H2 por ser uma abordagem muito invasiva em partes delicadas do sistema (Grandin, 2021). Independente da forma que for implementada a abordagem profunda, diversos componentes que consomem um tipo de dado precisam ser adaptados para atender o novo tipo. Um exemplo é a necessidade de criar um algoritmo de ordenação para o novo tipo de dado quando é criado um índice em uma coluna do novo tipo.

A solução proposta por este trabalho adota a abordagem superficial. O modo de compatibilidade do H2 já possui suporte para traduzir um novo tipo de dado para outro tipo conhecido, evitando assim a necessidade de alterações nas partes do sistema que possuam dependência com o tipo de dado em questão. Entretanto, a complexidade inerente ao tipo de dado *range* vai além do que é atualmente oferecido pelo H2.

Para isso, a proposta de solução consiste em expandir o modo de compatibilidade do H2, implementando a compatibilidade do tipo de dado *range*, específico do PostgreSQL e fora do padrão SQL. Esta abordagem também permitirá a inclusão de novos tipos complexos para futuras extensões do modo de compatibilidade do H2.

Todos os tipos de dado derivados do *range* serão adicionados, no mapeamento de tipos do modo de compatibilidade, como *alias* de “varchar” sem precisão, ou seja, sem definição de limite de caracteres. Por padrão, quando não há definição de limite, o H2 define como limite a quantidade máxima de caracteres permitidas para aquele tipo de dado. Para BDs convencionais, um limite de caracteres alto pode ser um problema, mas para BDs em memória esse limite é irrelevante. A Tabela 1 apresenta a lista completa de *alias* de tipos de dados no PostgreSQL após a inclusão dos tipos de dado *range*.

Tabela 1 – Lista dos tipos de dados no PostgreSQL com seus respectivos *alias* no H2.

Tipo de dado no PostgreSQL	Alias no H2
jsonb	json
money	numeric(19,2)
oid	integer
int4range	varchar
int8range	varchar
numrange	varchar
tsrange	varchar
tstzrange	varchar
daterange	varchar

Fonte: Elaborado pelo autor.

Além da inclusão de um *alias*, uma classe que representa um valor *range* precisa ser criada. Essa classe possui uma assinatura de tipo de dado “varchar” para integração com o resto do sistema, agrega limites inferior e superior, de um mesmo subtipo reconhecido pelo H2, e define delimitadores do intervalo inclusivos ou não-inclusivos. A integridade do dado é mantida pelo parser secundário ao instanciar um valor *range*.

A desvantagem dessa abordagem é que, para todos os fins, um valor de tipo *range* é reconhecido como um valor válido de tipo *varchar*. Consultas no PostgreSQL que normalmente resultam em erro de sintaxe ao introduzir um valor *range* onde deveria receber *varchar* passam a ser reconhecidas e executadas sem erros na adaptação do H2. Essa flexibilidade adicional, porém, é tratada como caso de

irrelevância considerando que o programador escreve as consultas no dialeto do PostgreSQL.

3.2 INTERPRETAÇÃO DE FUNÇÕES E OPERADORES

Para lidar com os operadores e funções, é necessário criar o parser secundário que atua junto com o parser primário. O parser secundário é selecionado de acordo com o modo de compatibilidade ativo. Em pontos críticos da compilação da consulta, onde o parser primário normalmente retornaria um erro, o parser secundário é consultado. Este avalia se deve assumir o processamento dos tokens atuais e a continuação da compilação. Se essa decisão for afirmativa, o parser primário transfere o controle, enviando os tokens relevantes, e aguarda a resposta transformada para retomar a compilação.

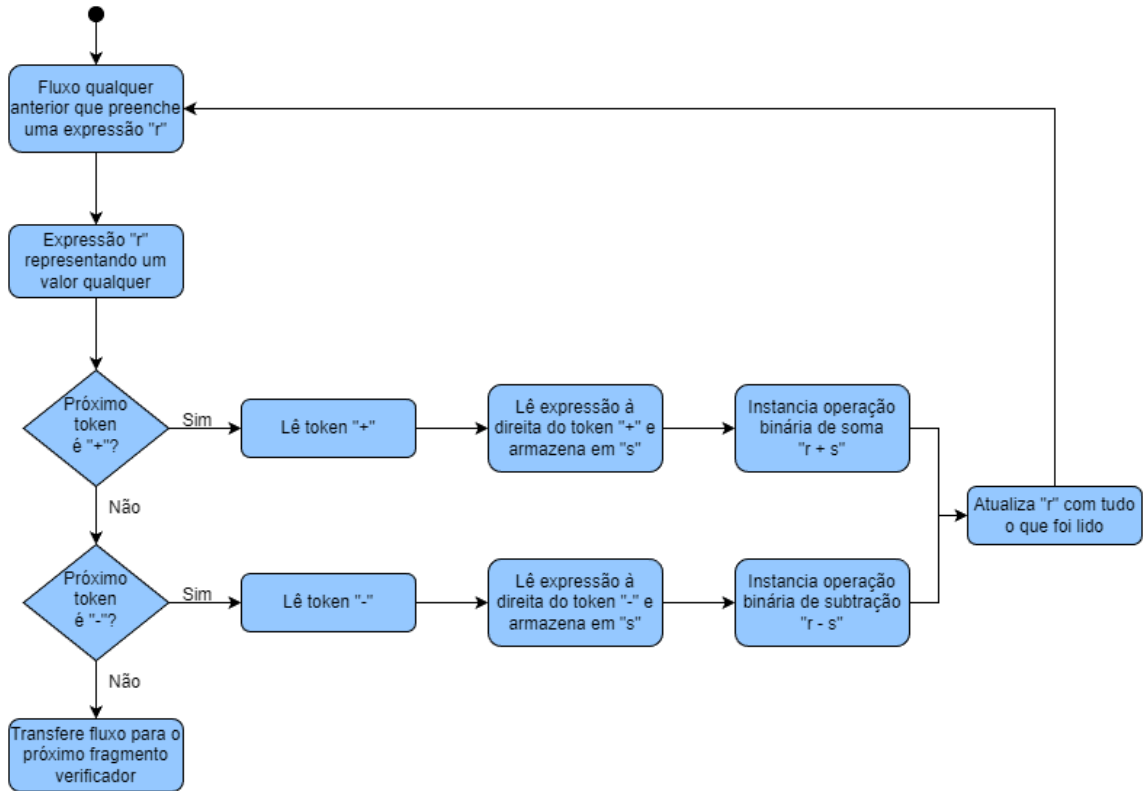
Para ilustrar, considere o fluxo de consultas no H2 que tradicionalmente segue um caminho específico para as operações de soma e subtração (Figura 5). Quando o próximo token é “+” ou “-”, o parser primário cria imediatamente uma operação binária de soma ou subtração. O que foi lido até o momento se torna o primeiro argumento, e a leitura da próxima expressão é solicitada para ser usada como segundo argumento. No entanto, devido à sobreposição dos operadores de soma e subtração com os operadores de união e subtração de valores de *range*, foi necessário adicionar uma chamada ao parser secundário neste ponto. Esta chamada permite ao parser secundário decidir se o processamento da operação deve ser realizado pelo parser primário ou secundário. A Figura 6 exibe o fluxograma do parser primário depois da integração das chamadas ao parser secundário, destacadas em verde, evidenciando as modificações realizadas.

Para determinar se o parser secundário deve ou não interpretar uma operação, a lógica de seleção é baseada na natureza dos dados envolvidos. Um exemplo dessa dinâmica é observado na situação em que o H2 não é intrinsecamente capaz de lidar com a operação de adição entre duas variáveis do tipo *varchar*. Nesse cenário, o parser secundário entra em ação e influencia a interpretação da operação.

Ao deparar-se com a operação de adição entre dois valores *varchar*, o parser secundário identifica a incompatibilidade com as capacidades nativas do H2. Em resposta, o parser secundário assume o controle da operação, tratando a conversão necessária e executando a operação apropriada. Após a execução da operação, o

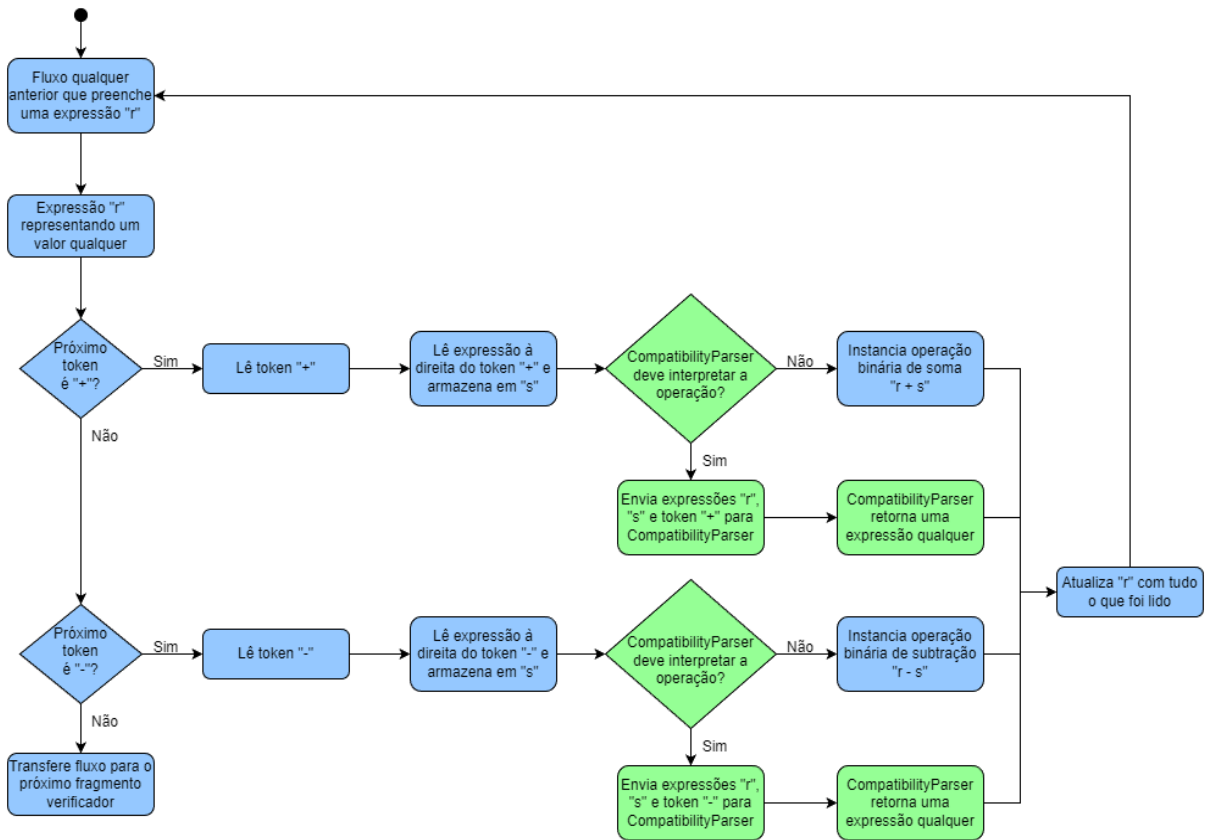
parser secundário devolve os resultados transformados, permitindo que o parser primário retome a compilação da consulta de maneira coerente.

Figura 5 – Fluxograma original do Parser durante leitura de tokens "+" e "-".



Fonte: Elaborado pelo autor.

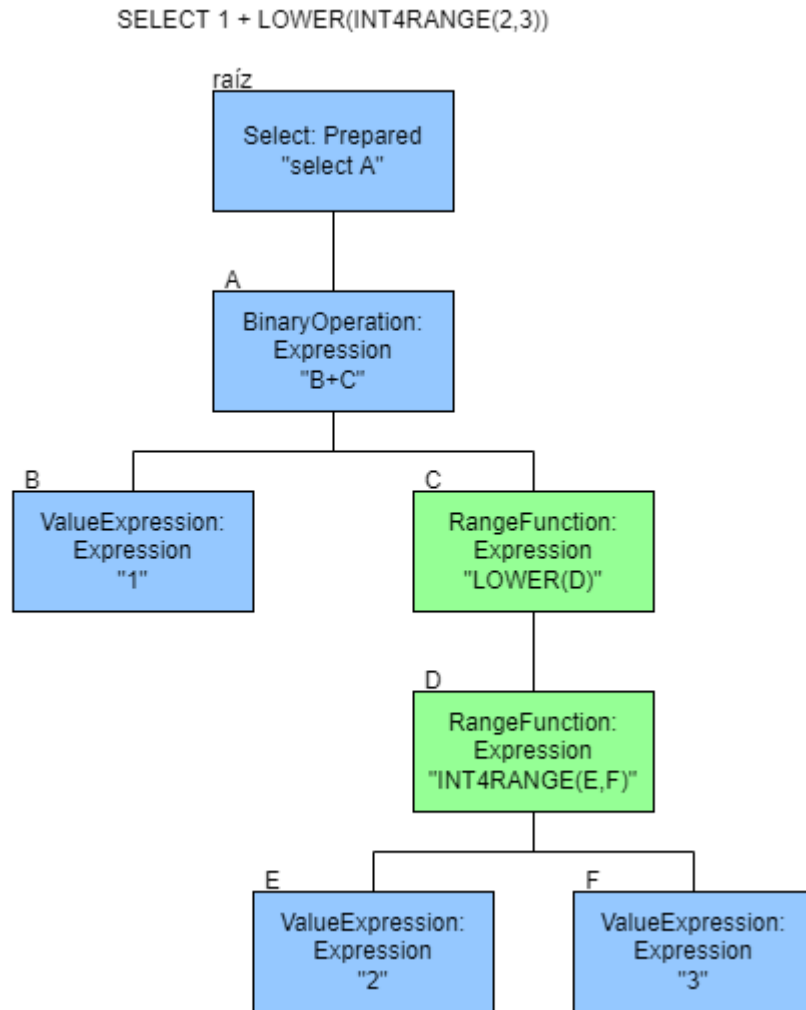
Figura 6 – Fluxograma do Parser com chamadas ao Parser secundário durante leitura de tokens “+” e “-”.



Fonte: Elaborado pelo autor.

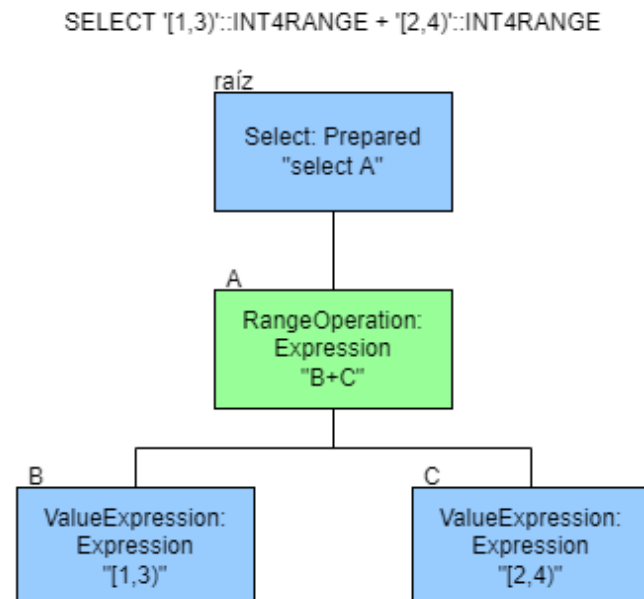
As classes dos objetos resposta transformados também precisam ser criadas. Esses objetos são instâncias de *Expression* para se encaixarem na árvore do objeto *Prepared*. Essas classes possuem a implementação do comportamento desejado das funções e operadores exclusivos do PostgreSQL. As Figuras 7 e 8 ilustram árvores criadas na etapa de preparação em conjunto com o parser secundário. Os objetos destacados em azul são criados pelo parser primário do H2, enquanto os objetos destacados em verde são criados pelo parser secundário do modo de compatibilidade PostgreSQL. A Figura 7 apresenta um exemplo com funções exclusivas do PostgreSQL, e a Figura 8 apresenta um exemplo com a operação de união, também exclusiva do PostgreSQL.

Figura 7 – Árvore com o mapeamento da consulta “SELECT 1+LOWER(INT4RANGE(2,3))”.



Fonte: Elaborado pelo autor.

Figura 8 – Árvore com o mapeamento da consulta “SELECT '[1,3)::INT4RANGE + '[2,4)::INT4RANGE”.



Fonte: Elaborado pelo autor.

4 PROCESSO DE DESENVOLVIMENTO

Nesta seção é descrito o processo de desenvolvimento da extensão do H2, abordando as restrições impostas pelo grupo de desenvolvimento do H2, levantamento de requisitos, elaboração do código e testes.

4.1 FRAMEWORK H2

O grupo de desenvolvimento do H2 aceita contribuições da comunidade e estabelece um conjunto de diretrizes para os contribuintes. Dentre as regras estabelecidas, destacam-se:

- A contribuição deve ser feita em formato de "*pull request*" no GitHub;
- Deve ser desenvolvido em Java 8;
- Deve ter cobertura de testes de pelo menos 90% do código produzido;
- Deve complementar a documentação com as novas funcionalidades introduzidas.

Considerando que esta proposta é uma versão experimental e que o processo de integração do código no H2 oficial está fora do escopo do trabalho, sendo realizada após a conclusão e revisão do trabalho, a complementação da documentação não foi elaborada neste trabalho. Para realizar um "pull request" no GitHub, porém, deve ser criado no início do desenvolvimento um "fork" a partir do branch base do H2 oficial. Este novo repositório deve ser público e respeitar os direitos autorais do grupo de desenvolvimento do H2.

A restrição da linguagem de programação é auto-explicativa, considerando que o H2 foi desenvolvido em Java 8 e não funciona com versões anteriores. A regra estabelecida pela equipe serve para proibir os contribuintes de utilizar versões superiores ao Java 8, tornando o projeto não compilável para os demais contribuintes trabalhando em paralelo.

Para os testes automatizados, o projeto possui sua própria estrutura de testes e exemplos de utilização. Existe um diretório específico onde as classes de teste devem ser escritas. Ao disparar a rotina de testes do projeto, a estrutura procura todos os métodos *main* no diretório de testes e os executa. O projeto possui dois arquétipos de classes de teste, uma para testes unitários diretamente do código Java e outro para testes de integração com o BD. Para os testes com o BD, o desenvolvedor deve se

responsabilizar por limpar os vestígios dos testes. Caso o teste utilize um BD compartilhado entre testes, deve-se limpar todos os dados ao final dos testes. Caso o teste crie um novo BD, deve-se descartá-lo ao final dos testes. O projeto também oferece sua própria biblioteca de asserções, facilitando a escrita dos testes e padronizando os testes entre os contribuintes.

Por fim, antes de iniciar o projeto, foi consultada a equipe do H2 para confirmar se eles aceitariam uma contribuição para o H2 que vai além do padrão SQL. A equipe foi bem receptiva quanto à adição de elementos fora do padrão SQL, mas demonstraram uma preocupação com a abordagem profunda explicada no capítulo anterior. Existem componentes que exigem um certo cuidado com alterações, principalmente de terceiros. Diante desta preocupação, foi decidido que a implementação deve ser o menos invasiva possível. Isso significa utilizar as ferramentas que o H2 já oferece e minimizar as alterações feitas no parser primário, sem aumentar sua responsabilidade.

4.2 REQUISITOS FUNCIONAIS

De acordo com os comportamentos observados no PostgreSQL e o que é viável à implementação no H2, foram definidos os seguintes requisitos funcionais:

- RF-01: Deve-se reconhecer e interpretar a declaração literal dos tipos de dado *int4range*, *int8range*, *numrange*, *daterange*, *tsrange* e *tstzrange*:
 - RF-01.1: Deve-se aceitar string "empty" como um possível valor.
 - RF-01.2: Caso haja limites, deve-se validar os tipos de dados dos limites.
 - RF-01.3: Caso haja limites, deve-se validar limite superior maior ou igual ao limite inferior.
 - RF-01.4: Caso os limites superior e inferior sejam iguais e pelo menos um delimitador for aberto (parênteses), deve-se normalizar para a string "empty".
 - RF-01.5: Deve-se permitir omissão de limite, representando limite infinito positivo para limite superior e infinito negativo para limite inferior.
 - RF-01.6: Deve-se aplicar função de canonização para os tipos *int4range*, *int8range* e *daterange*.

- RF-02: Deve-se reconhecer e resolver as funções de criação dos tipos de dado `int4range`, `int8range`, `numrange`, `daterange`, `tsrange` e `tstzrange`:
 - RF-02.1: Deve-se validar os tipos de dados dos parâmetros.
 - RF-02.2: Deve-se validar limite contínuo.
 - RF-02.3: Deve-se validar limite superior maior ou igual ao limite inferior.
 - RF-02.4: Caso os limites superior e inferior sejam iguais e pelo menos um delimitador for aberto (parênteses), deve-se normalizar para a string "empty".
 - RF-02.5: Deve-se aplicar função de canonização para os tipos `int4range`, `int8range` e `daterange`.
- RF-03: Deve-se reconhecer e resolver as funções `lower`, `upper`, `isempty`, `lower_inc`, `upper_inc`, `lower_inf`, `upper_inf` e `range_merge`.
- RF-04: Deve-se reconhecer e resolver os operadores binários de contenção ("`<@`" e "`@>`"), sobreposição ("`&&`"), adjacência ("`-|`"), estritamente à esquerda/direita ("`<<`" e "`>>`"), não se estende à direita/esquerda ("`&<`" e "`&>`"), união ("`+`"), interseção ("`*`") e diferença ("`-`").
- RF-05: Deve-se permitir encadeamento de operações binárias quando o resultado da operação prioritária possuir tipo de dado compatível para a resolução da operação seguinte:
 - A prioridade de resolução do encadeamento de operações deve seguir:
 - 1) Interseção ("`*`").
 - 2) União ("`+`") e diferença ("`-`").
 - 3) Demais operadores.
 - Caso haja mais de uma operação de mesma prioridade, deve-se resolver da esquerda para a direita.

4.3 REQUISITOS NÃO-FUNCIONAIS

De acordo com as restrições e recomendações do grupo de desenvolvimento do H2, foram definidos os seguintes requisitos não-funcionais:

- RNF-01: Deve ser feito a partir do *branch* base do projeto H2 no GitHub, para posteriormente iniciar processo de "pull request" do GitHub.
- RNF-02: O repositório em desenvolvimento deve ser público, contendo informações dos direitos autorais do grupo de desenvolvimento do H2.
- RNF-03: Deve ser desenvolvido em Java 8.
- RNF-04: Deve ter cobertura de testes automatizados de pelo menos 90% do código produzido.
- RNF-05: As novas funcionalidades devem estar disponíveis apenas ao ativar o modo de compatibilidade com o PostgreSQL.
- RNF-06: Decisões de desenvolvimento devem ser tomadas visando o menor impacto possível no compilador nativo (parser primário) e reconhecimento de tipos:
 - RNF-06.1: Deve utilizar um compilador complementar (parser secundário), sem aumentar a responsabilidade do compilador nativo (parser primário).
 - RNF-06.2: Deve utilizar o modo de compatibilidade oferecido pelo H2, incluindo a interpretação de novos tipos de dados como *alias* de tipos conhecidos.

4.4 DESENVOLVIMENTO DA EXTENSÃO

O desenvolvimento do projeto foi dividido em 6 tarefas:

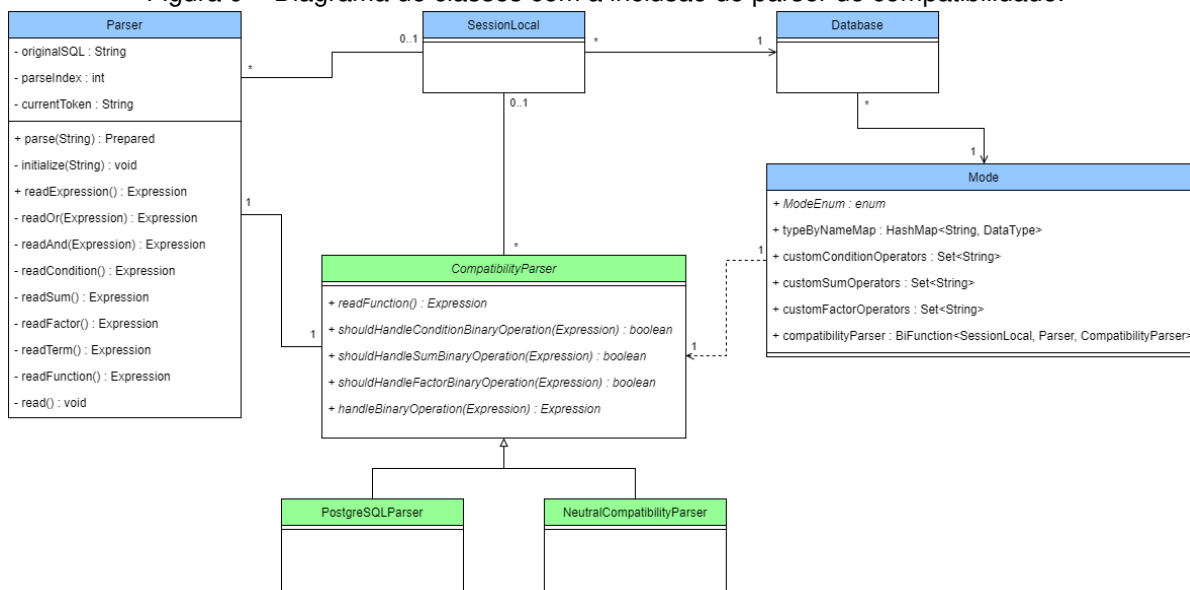
- Criação e seleção do parser secundário, escolhido a partir do modo de compatibilidade;
- Criação da representação do novo tipo de dado *range*;
- Implementação dos métodos de instanciação dos tipos de dado *range*;
- Reconhecimento e resolução dos operadores *range*;
- Implementação dos demais métodos relacionados ao tipo de dado *range*.
- Elaboração dos testes automatizados.

4.4.1 Criação e Seleção do Parser Secundário

O parser nativo do H2 foi construído com um anti-padrão de projeto conhecido como "classe superpoderosa". É um componente composto por apenas uma classe, com muito código e responsabilidades atribuído a ele. Esse padrão de projeto é desencorajado porque dificulta a leitura e compreensão do código, ainda mais para um projeto de código aberto que aceita contribuições externas. Outros problemas derivados desse padrão são a dificuldade de expansão e manutenção do código, bem como sua testabilidade.

O parser primário já possui alguns blocos verificadores para comportamentos diferenciados de acordo com o modo de compatibilidade ativo. Porém, tendo em vista a situação atual do parser primário, bem como a natureza sensível de um compilador de BD e a preocupação da equipe de desenvolvimento, foi definido nos requisitos que não deve ser aumentada a responsabilidade do parser primário. Por isso foi criado um parser secundário, selecionado na classe *Mode*, que é responsável pelas definições do modo de compatibilidade. A Figura 9 apresenta a inclusão do parser secundário, destacado em verde. Esse parser secundário é instanciado juntamente com o método construtor do próprio parser primário. Sempre que for necessária a interpretação de algo exclusivo da compatibilidade de algum BD, o parser primário transfere o fluxo de execução para o parser secundário, transferindo assim a responsabilidade do componente.

Figura 9 – Diagrama de classes com a inclusão do parser de compatibilidade.



Fonte: Elaborado pelo autor.

A seleção do parser secundário é feita de acordo com o modo de compatibilidade ativo. Visando expansões futuras, foi implementada uma estrutura para receber um parser específico para cada modo de compatibilidade. Porém, como o escopo deste trabalho abrange apenas o PostgreSQL, foi implementado apenas um parser para compatibilidade com o PostgreSQL. Para o modo de compatibilidade padrão e os demais modos de compatibilidade, foi definido um parser secundário neutro. Esse parser neutro recebe as chamadas do parser primário mas sempre retorna valores que não interferem no fluxo natural do parser primário.

O parser secundário é utilizado durante a construção da árvore de objetos Expression, na etapa de preparação e interpretação de consultas. Cada comportamento exclusivo do PostgreSQL (e demais SGBDs do modo de compatibilidade) deve possuir uma classe que estende Expression, com a implementação do comportamento. Em momentos na interpretação de consultas SQL onde um *token* potencialmente possui um comportamento diferenciado de acordo com o modo de compatibilidade, o parser primário consulta o parser secundário. Este responde se o fluxo de execução deve ser transferido ou não. Em caso positivo, o parser primário transfere as expressões construídas até o momento e o *token* atual para o parser secundário. O parser secundário instancia o objeto Expression e o conecta na árvore de objetos Expression. Dessa forma, o comportamento diferenciado do token em questão é completamente abstrato para o parser primário. Naturalmente, apenas nas etapas de otimização e de resolução esse objeto Expression terá seu comportamento diferenciado executado.

É importante ressaltar, conforme observado na Figura 9, que o parser primário não possui relação direta com o modo de compatibilidade. A classe *SessionLocal* serve como intermediário entre o parser primário e o modo de compatibilidade. Existem casos excepcionais onde um novo parser primário é instanciado sem sessão. Esses casos excepcionais não dependem do modo de compatibilidade. Porém, com a inclusão das chamadas ao parser secundário, criou-se a necessidade de selecionar um parser secundário mesmo quando não há sessão. Para esses casos, ao invés de consultar a sessão inexistente, o próprio parser primário instancia um parser secundário neutro.

Foram identificados 4 momentos na interpretação de consultas SQL onde o parser primário deve transferir o fluxo de execução para o parser secundário instanciar o objeto Expression:

- interpretação de função nomeada;
- interpretação de operadores lógicos;
- interpretação de operadores binários com prioridade de soma e subtração;
- interpretação de operadores binários com prioridade de multiplicação e divisão.

Os Algoritmos 1 a 8 apresentam o pseudocódigo original do parser primário, utilizado para interpretar uma consulta SQL. Os Algoritmos 4, 5 e 6 necessitam intervenção do parser secundário para interpretação dos operadores de nível de condição, soma e multiplicação, respectivamente. O Algoritmo 8 necessita da intervenção de um bloco verificador para funções criadas no parser secundário. Esse novo bloco deve ser prioritário em relação às funções nativas do padrão SQL, devido à possibilidade de sobrecarga de métodos (métodos com nomes iguais). Nos algoritmos 4, 5, 6 e 8 foi destacado onde é necessário a intervenção do parser secundário.

Algoritmo 1: Função principal da interpretação de consultas

Function *parse(sql)*:

Output: Objeto Prepared, contendo a raiz da árvore de objetos Expression.

```

1 begin
2   |   sql ← sql com normalização de espaços, caixa alta ou baixa e
      |   eliminação de comentários.
3   |   currentToken ← primeiro token contendo em sql.
4   |   prepared ← objeto da classe correspondente ao comando em
      |   currentToken.
5   |   currentToken ← próximo token.
6   |   expression ← readExpression(sql, currentToken)
7   |   prepared ← preenche objeto expression dentro do objeto prepared.
8   |   return prepared
9 end

```

Algoritmo 2: Leitura de expressão geral

Function *readExpression(sql, currentToken)*:

Output: Objeto Expression raiz da árvore de objetos Expression. O bloco presente nesta função, especificamente, agrupa expressões internas com operações OR.

```

1 begin
2   |   r ← readAnd(sql, currentToken)
3   |   while currentToken == "OR" do

```

```

4 |         |         | currentToken ← próximo token.
5 |         |         | s ← readAnd(sql, currentToken)
6 |         |         | r ← expressão equivalente à operação OR entre r e s.
7 |         | end
8 |         | return r
9 | end

```

Algoritmo 3: Leitura de expressões AND

Function *readAnd(sql, currentToken)*:

Output: Objeto Expression agrupando expressões internas com operações AND.

```

1 | begin
2 |     | r ← readCondition(sql, currentToken)
3 |     | while currentToken == "AND" do
4 |     |     | currentToken ← próximo token.
5 |     |     | s ← readCondition(sql, currentToken)
6 |     |     | r ← expressão equivalente à operação AND entre r e s.
7 |     |     | end
8 |     |     | return r
9 | end

```

Algoritmo 4: Leitura de expressão de condição

Function *readCondition(sql, currentToken)*:

Output: Objeto Expression que agrupa uma única operação de condição.

```

1 | begin
2 |     | r ← readSum(sql, currentToken)
3 |     | isCondition ← verificação se o conteúdo de currentToken
4 |     |     | corresponde a uma operação de condição reconhecida.
5 |     |     | If isCondition then
6 |     |     |     | operation ← currentToken
7 |     |     |     | currentToken ← próximo token.
8 |     |     |     | s ← readSum(sql, currentToken)
9 |     |     |     | r ← expressão equivalente à operação descrita em
10 |    |     |     |     | operation entre r e s. Intervenção do parser de
11 |    |     |     |     | compatibilidade necessária.
12 |    |     | end
13 |    |     | return r
14 | end

```

Algoritmo 5: Leitura de expressões de soma e subtração

Function *readSum(sql, currentToken)*:

Output: Objeto Expression agrupando expressões internas com operações de soma e subtração.

```

1 | begin
2 |     | r ← readFactor(sql, currentToken)
3 |     | isSum ← verificação se o conteúdo de currentToken corresponde a
4 |     |     | "+" ou "-".
5 |     |     | while isSum do
6 |     |     |     | operation ← currentToken
7 |     |     |     | currentToken ← próximo token.
8 |     |     |     | s ← readFactor(sql, currentToken)

```

```

8      |      |      |      r ← expressão equivalente à operação descrita em
      |      |      |      |      operation entre r e s. Intervenção do parser de
      |      |      |      |      compatibilidade necessária.
9      |      |      |      |      isSum ← verificação se o conteúdo de currentToken
      |      |      |      |      |      corresponde a "+" ou "-".
10     |      |      |      |      end
11     |      |      |      |      return r
12     |      |      |      |      end

```

Algoritmo 6: Leitura de expressões de multiplicação e divisão

Function *readFactor*(*sql*, *currentToken*):

Output: Objeto Expression agrupando expressões internas com operações de multiplicação, divisão e resto da divisão.

```

1  begin
2  |      r ← readTerm(sql, currentToken)
3  |      isFactor ← verificação se o conteúdo de currentToken corresponde
      |      |      a "*", "/" ou "%".
4  |      while isSum do
5  |      |      operation ← currentToken
6  |      |      currentToken ← próximo token.
7  |      |      s ← readTerm(sql, currentToken)
8  |      |      r ← expressão equivalente à operação descrita em
      |      |      |      operation entre r e s. Intervenção do parser de
      |      |      |      compatibilidade necessária.
9  |      |      isFactor ← verificação se o conteúdo de currentToken
      |      |      |      corresponde a "*", "/" ou "%".
10 |      |      end
11 |      return r
12 end

```

Algoritmo 7: Leitura de expressão de termo geral

Function *readTerm*(*sql*, *currentToken*):

Output: Objeto Expression que representa a operação de uma palavra-chave ou apenas um identificador.

```

1  begin
2  |      r ← valor nulo.
3  |      isKeyword ← verificação se o conteúdo de currentToken
      |      |      |      corresponde a alguma palavra-chave reconhecida, caso
      |      |      |      |      contrário é identificador.
4  |      if isKeyword then
5  |      |      keyword ← currentToken
6  |      |      currentToken ← próximo token.
7  |      |      r ← expressão equivalente à operação descrita em
      |      |      |      keyword.
8  |      else
9  |      |      identifier ← currentToken
10 |      |      currentToken ← próximo token.
11 |      |      if currentToken == "(" then
12 |      |      |      r ← readFunction(sql, identifier, currentToken)
13 |      |      |      else if currentToken == "." then
14 |      |      |      r ← expressão que encapsula 2 identificadores

```

15				separados por um token “.”.
16			else	r ← expressão que encapsula um identificador qualquer.
17			end	
18		end		
19		if <i>currentToken</i> == “:” then		
20			currentToken ← próximo token.	
21			s ← tipo de dado descrito em <i>currentToken</i> .	
22			r ← expressão equivalente à operação “CAST” do valor em r para o tipo de dado em s.	
23		end		
24		return r		
25	end			

Algoritmo 8: Leitura de expressão de função

Function *readFunction(sql, identifier, currentToken)*:

Output: Objeto Expression que representa a operação de uma palavra-chave ou apenas um identificador.

```

1 begin
2   r ← valor nulo.
3   isBuiltIn ← verificação se o conteúdo de identifier corresponde a
   alguma função dentro do padrão SQL.
4   If isBuiltIn then
5     args ← leitura dos argumentos entre tokens “(” e “)”,
   separados por token “,”.
6     r ← expressão com a implementação da função descrita
   em identifier com argumentos descritos em args.
7     currentToken ← próximo token.
8   end
9   isCompatibility ← verificação se o conteúdo de identifier
   corresponde a alguma função reconhecida com modo de
   compatibilidade.
10  If isCompatibility then
11    args ← leitura dos argumentos entre tokens “(” e “)”,
   separados por token “,”.
12    r ← expressão com a implementação da função descrita
   em identifier com argumentos descritos em args.
   Intervenção do parser de compatibilidade necessária.
13    currentToken ← próximo token.
14  end
15  return r
16 end

```

4.4.2 Criação da Representação do Novo Tipo de Dado *Range*

O reconhecimento de dados e tipos de dados no H2 é controlado principalmente pelas classes *Value*, *TypeInfo* e *DataType*. Sendo assim, é necessário vincular os novos tipos de dados a essas 3 classes.

Foi definido nos requisitos que seria utilizado o suporte já disponível no modo de compatibilidade para interpretação dos novos tipos de dado como *alias* de outro tipo conhecido, neste caso, *Varchar*. Isso é feito adicionando os nomes dos novos tipos na classe *Mode*, no objeto que se refere ao modo de compatibilidade PostgreSQL. Além da identificação dos novos tipos e o tipo nativo equivalente, também é definido precisão padrão, cujo valor é o máximo permitido para valores baseados em *string* (2^{20} caracteres) e sem escala.

A interpretação de todos os tipos *range* como *alias* de um mesmo tipo implica em algumas consequências de maior flexibilidade no uso do dado e outras consequências limitadoras. Considerando que na solução proposta todos os tipos *range* passam a ter a mesma assinatura, diferentes tipos de *range* passam a ser compatíveis e não resultam em erro de compilação. Também existem possíveis valores de *range* que não possuem limites, sendo eles conjunto vazio e infinito. No PostgreSQL, um conjunto vazio de um determinado tipo *range* (por exemplo, *int4range*) não é equivalente ao conjunto vazio de outro tipo *range* (por exemplo, *daterange*). Dessa forma, a operação `"empty"::int4range + 'empty'::daterange` resulta em erro de compilação. Entretanto, na solução proposta, a inferência dos valores limites resultam no valor universal nulo, compatível com todos os tipos de dados. Para permitir a operação com conjunto vazio, este é interpretado como um coringa universal para todos os tipos *range*, semelhante ao comportamento nativo do H2 com valores nulos. O mesmo comportamento se aplica a conjuntos infinitos, onde a inferência dos valores limites também resulta em nulo.

Outro problema da inferência do tipo original a partir dos limites é que, na maioria dos casos, não é possível distinguir um dado *integer* de um *bigint*. Dessa forma, os tipos *int4range* e *int8range* passam a ser compatíveis entre si.

Para viabilizar o controle do tipo do dado durante a execução das consultas, foi criada uma classe *RangeValue* que estende *Value*, representando um valor *range*. Essa classe serve para todos os tipos nativos de *range* no PostgreSQL. A decisão de criar uma classe única e genérica para todos os tipos de *range* foi tomada visando a possibilidade de, futuramente, utilizá-la para a criação de novos tipos *range*, de forma dinâmica, pelo usuário. Apesar do H2 ainda não oferecer suporte para UDT, uma característica necessária para todo SGBD, ela está na lista de tarefas da equipe de desenvolvimento, e é uma questão de tempo até que seja disponibilizada.

A classe *Value* possui uma subclasse para cada tipo primitivo de dado Java. Para valores *varchar*, por exemplo, a classe correspondente é *ValueVarchar*. Cada subclasse possui um atributo "value" do tipo correspondente, onde é armazenado o valor da instância. Para a classe *RangeValue*, foi estendida uma classe intermediária *ValueStringBase*, que possui o atributo "value" em formato *String*. Na classe *RangeValue* foram adicionados campos individuais para os limites inferior e superior, e delimitadores inferior e superior. O atributo "value" recebe o valor completo em formato *string*, legível para o usuário. Apesar de possuir sua própria classe para representação de valores, seu tipo de dado em *Value* ainda é definido como "varchar". Ainda como consequência da abordagem superficial, a persistência de um valor *range* em tabelas resulta no valor sendo armazenado em formato *ValueVarchar*.

Foi criado um conversor de dados (*DataConverter*) para tratar conversões de objetos *ValueVarchar* para *RangeValue*, ou diretamente um valor *String* para *RangeValue*. Semelhante ao parser secundário, cada modo de compatibilidade recebeu um conversor próprio. O conversor é acionado na recuperação de dados de tabelas ou em operações de conversão explícita ("cast"). Existem diferentes formas de converter um objeto *Value*, mas todas convergem para o método "convertTo" em *Value*. Portanto, foi complementado esse método para consultar o conversor do modo de compatibilidade antes de seguir o fluxo original.

Objetos *TypeInfo* implementam uma superclasse *ExtTypeInfo*. É possível criar outras implementações dessa classe e cada instância de *TypeInfo* pode agregar outras implementações de extensão. Essa estrutura foi explorada para criar novas instâncias de *TypeInfo*, todas elas associadas ao tipo de dado em *Value* referente a *varchar*, mas também agregando uma implementação adicional exclusiva para os tipos *range* (*ExtTypeInfoRange*). Essa nova classe possui informação do subtipo do dado *range* e outros utilitários. A construção de objetos *TypeInfo* é gerenciada pelo método "getTypeInfo" da própria classe. Portanto, foi necessário adaptar esse método para reconhecer os novos tipos. As instâncias personalizadas de *TypeInfo* são persistidas de forma íntegra nas definições de colunas de tabelas, carregando consigo a informação adicional do subtipo oriundo do valor *range* antes de sua conversão para *varchar*. Essa informação adicional nas instâncias de *TypeInfo* permite a conversão de objetos *ValueVarchar* para *RangeValue* durante a recuperação de dados das tabelas, bem como a identificação correta dos tipos de dados envolvidos em uma consulta durante a etapa de otimização.

Por outro lado, para a utilização das funções que constroem o dado, foi necessário implementar o comportamento das mesmas.

O parser primário possui um bloco dedicado ao reconhecimento e resolução de funções (Algoritmo 8 na seção 4.4.1). Este bloco engloba funções dentro do padrão SQL e funções exclusivas de acordo com o modo de compatibilidade. No entanto, esse segundo segmento não inclui todas as funções de cada SGBD compatível. Além disso, mesmo para as funções presentes, nem todas têm implementação completa. Percebe-se que o bloco foi concebido visando apenas o reconhecimento de consultas válidas, sem considerar a possibilidade de produzir um resultado diferente. Considerando que este projeto introduz o parser secundário, optou-se por implementar as funções dentro dele em vez de expandir o bloco nativo do H2. Se o projeto fosse aprovado pela equipe do H2, todas as funções de compatibilidade já existentes poderiam ser movidas para o parser secundário, melhorando a organização de responsabilidade das classes. O Algoritmo 8a mostra o Algoritmo 8 adaptado com a mudança, com destaque à chamada de decisão e à chamadas de tratamento do token atual.

Algoritmo 8a: Leitura de expressão de função adaptado

Function *readFunction(sql, identifier, currentToken):*

Output: Objeto Expression que representa a operação de uma palavra-chave ou apenas um identificador.

```

1  begin
2      r ← valor nulo.
3      isCompatibility ← consulta ao parser secundário verificando se o
                          conteúdo de identifier corresponde a alguma função
                          reconhecida com modo de compatibilidade.
4      If isCompatibility then
5          args ← leitura dos argumentos entre tokens "(" e ")",
                          separados por token ",".
6          r ← expressão criada pelo parser secundário com a
                          implementação da função descrita em identifier com
                          argumentos descritos em args.
7          currentToken ← próximo token.
8      end
9      isBuiltIn ← verificação se o conteúdo de identifier corresponde a
                          alguma função dentro do padrão SQL.
10     If isBuiltIn then
11         args ← leitura dos argumentos entre tokens "(" e ")",
                          separados por token ",".
12         r ← expressão com a implementação da função descrita
                          em identifier com argumentos descritos em args.
13         currentToken ← próximo token.
14     end

```

```
15 |      return r  
16 end
```

Cada função possui uma classe com sua implementação, agrupadas pela quantidade e tipo dos argumentos. Essas classes, assim como qualquer elemento da consulta SQL, estendem direta ou indiretamente a classe *Expression* para serem vinculadas à árvore de expressões. O H2 possui classes abstratas que servem como intermediárias na hierarquia de classes entre *Expression* e as classes com implementação das funções. As funções de construção de dados *range* podem ter 2 ou 3 argumentos. Assim, foi criada uma classe *RangeFunction2* que estende a classe *Function2* nativa do H2, dedicada a funções com obrigatoriamente 2 argumentos. As demais funções utilitárias relacionadas ao tipo de dado *range* possuem 1 argumento. Portanto, foi criada uma classe *RangeFunction1* que estende a classe *Function1*, dedicada a funções com 1 argumento. O parser secundário possui um método tradutor do token lido em formato String para uma instância da implementação correspondente de *RangeFunction1* ou *RangeFunction2*.

A decisão de tratar os tipos de dado *range* como *alias* de *varchar* entra em conflito com o fluxo original do H2. Inicialmente, o Parser primário busca pelo nome da função desejada entre as funções dentro do padrão SQL e depois entre as funções exclusivas de cada SGBD. Essa ordem foi estabelecida porque as funções dentro do padrão SQL são mais comuns que as exclusivas de cada SGBD. No entanto, isso se torna um problema para as funções "upper" e "lower", pois já existem funções com esses nomes dentro do padrão SQL. As funções "upper" e "lower" dentro do padrão SQL recebem um argumento do tipo *varchar* e convertem todos os caracteres para maiúsculas ou minúsculas, respectivamente. Já as funções "upper" e "lower" do PostgreSQL, quando recebem como argumento um tipo de dado *range*, retornam o limite superior ou inferior do intervalo, respectivamente. Para o PostgreSQL, essa sobrecarga de funções não é um problema, pois a assinatura dos métodos é diferente. As funções que convertem os caracteres de um *varchar* recebem um argumento do tipo *varchar*, enquanto as funções que retornam os limites do intervalo recebem um argumento do tipo *range*. A abordagem superficial adotada por este projeto, que utiliza os tipos *range* como *alias* de *varchar*, resulta em uma ambiguidade nas funções, todas recebendo um dado do tipo *varchar* como argumento.

Para resolver essa ambiguidade, foi necessário inverter a ordem com a qual o parser primário reconhece as funções (linhas 4 e 10 de ambos Algoritmos 8 e 8a). Ele primeiro consulta o parser secundário, que resolve a ambiguidade, para então retornar ao reconhecimento de funções dentro do padrão SQL.

Caso uma função receba uma variável como argumento, durante a etapa de preparação, não é possível identificar o tipo do dado da variável no momento da sua leitura. Somente na etapa de otimização será identificado seu tipo. Porém, é durante a etapa de preparação que o parser primário e secundário devem decidir quem resolverá aquela função. Diante do impasse, o parser secundário assume o controle do fluxo e instancia sua implementação do método analisado. Durante a etapa de otimização, caso a implementação nova identifique que foi um engano, ela passa por um fluxo de correção que substitui seu elemento na árvore de expressões pela instância da implementação correta.

4.4.4 Interpretação e Resolução dos Operadores

No âmbito da interpretação e resolução dos operadores, é possível identificar três categorias distintas: operadores de condição, soma e multiplicação. Cada uma dessas categorias possui um conjunto específico de operadores, cada qual correspondendo a um bloco de código (método) correspondente (algoritmos 4, 5 e 6 da seção 4.4.1 respectivamente). A chamada desses métodos ocorre de acordo com a prioridade das operações, tratada pelo parser primário com a separação das categorias. O H2 segue a ordem PEMDAS, que significa a execução de operações nesta ordem: parênteses, expoente, multiplicação/divisão, e soma/subtração. Após a execução de operações de soma/subtração, são executadas, nesta ordem, operações de condição geral, operações “AND” e operações “OR”. Os Algoritmos 4a, 5a e 6a mostram os Algoritmos 4, 5 e 6 adaptados com as mudanças, com destaque às chamadas de decisão e às chamadas de tratamento do *token* atual.

Algoritmo 4a: Leitura de expressão de condição adaptado

Function *readCondition(sql, currentToken)*:

Output: Objeto Expression que agrupa uma única operação de condição.

```

1 begin
2   |   r ← readSum(sql, currentToken)
3   |   isCondition ← verificação se o conteúdo de currentToken
   |                       corresponde a uma operação de condição reconhecida

```

```

dentro do padrão SQL.
4   if isCondition then
5       operation ← currentToken
6       currentToken ← próximo token.
7       s ← readSum(sql, currentToken)
8       r ← expressão equivalente à operação descrita em
           operation entre r e s.
9   end
10  isCompatibilityOp ← consulta ao parser secundário
11  verificando se o conteúdo de currentToken
    corresponde a uma operação de condição
    reconhecida pelo modo de compatibilidade.
12  if isCompatibilityOp then
13      operation ← currentToken
14      currentToken ← próximo token.
15      s ← readSum(sql, currentToken)
16      r ← expressão criada pelo parser secundário equivalente à
           operação descrita em operation entre r e s.
17  end
18  return r
19 end

```

Algoritmo 5a: Leitura de expressões de soma e subtração adaptado

Function *readSum(sql, currentToken)*:

Output: Objeto Expression agrupando expressões internas com operações de soma e subtração.

```

1  begin
2      r ← readFactor(sql, currentToken)
3      isSum ← verificação se o conteúdo de currentToken corresponde a
           “+” ou “-”.
4      while isSum do
5          operation ← currentToken
6          currentToken ← próximo token.
7          s ← readFactor(sql, currentToken)
8          isCompatibilityOp ← consulta ao parser secundário
           verificando se o conteúdo de currentToken
           corresponde a uma operação de soma reconhecida
           pelo modo de compatibilidade.
9          if isCompatibilityOp then
10             r ← expressão criada pelo parser secundário
                equivalente à operação descrita em
                operation entre r e s.
11         else
12             r ← expressão equivalente à operação descrita
                em operation entre r e s.
13         end
14         isSum ← verificação se o conteúdo de currentToken
                corresponde a “+” ou “-”.
15     end
16     return r
17 end

```

Algoritmo 6a: Leitura de expressões de multiplicação e divisão adaptado**Function** *readFactor*(*sql*, *currentToken*):**Output:** Objeto Expression agrupando expressões internas com operações de multiplicação, divisão e resto da divisão.

```

1  begin
2      r ← readTerm(sql, currentToken)
3      isFactor ← verificação se o conteúdo de currentToken corresponde
          a “*”, “/” ou “%”.
4      while isSum do
5          operation ← currentToken
6          currentToken ← próximo token.
7          s ← readTerm(sql, currentToken)
8          isCompatibilityOp ← consulta ao parser secundário
          verificando se o conteúdo de currentToken
          corresponde a uma operação de multiplicação
          reconhecida pelo modo de compatibilidade.
9          if isCompatibilityOp then
10             r ← expressão criada pelo parser secundário
                equivalente à operação descrita em
                operation entre r e s.
11          else
12             r ← expressão equivalente à operação descrita
                em operation entre r e s.
13          end
14          isFactor ← verificação se o conteúdo de currentToken
                corresponde a “*”, “/” ou “%”.
15      end
16      return r
17 end

```

Para os operadores de união (+), diferença (-) e intersecção (*), o parser primário os reconhece de forma natural. Não é necessário que o parser identifique qual é a operação, pois isso não afeta a leitura do token em si. Por outro lado, nenhum dos operadores de condição é inicialmente reconhecido como válido e, portanto, requerem uma abordagem especial para sua interpretação.

Para determinar se um caractere que compõe um operador é agrupável ou não, o parser primário classifica os caracteres especiais em dois tipos: tipo 1 e tipo 2. Caracteres do tipo 1 são não agrupáveis, enquanto os do tipo 2 podem ou não ser acompanhados de um ou mais caracteres adicionais. Tomamos, por exemplo, o caractere “(”. Uma consulta SQL pode apresentar múltiplos caracteres “(“ seguidos, porém cada um deles representa uma operação isolada. O parser primário isola os operadores classificando-os como tipo 1. Em outro exemplo, tomamos o caractere “<”. Uma consulta SQL pode apresentar esse caractere isolado para representar uma operação de “menor” (<), ou acompanhado de outro caractere, potencialmente o

caractere “=” para a operação de “menor ou igual” (\leq) ou o caractere “>” para a operação de “diferente” (\neq). O parser primário reconhece a composição de caracteres em um único operador classificando os caracteres envolvidos como tipo 2.

Duas pequenas mudanças na leitura de operadores precisaram ser realizadas para que o H2 reconheça o operador de adjacência (-|-). Ao ler um caractere especial tipo 2, se o caractere seguinte também for do tipo 2, o parser primário agrupa os caracteres para formar um único operador. Originalmente, o caractere “-” é reconhecido como tipo 1, então foi necessário alterar a biblioteca para que ele fosse identificado como tipo 2. Além disso, o parser primário foi originalmente projetado para reconhecer operadores com no máximo 2 caracteres, enquanto o operador de adjacência possui 3 caracteres. O bloco de código responsável pela leitura de operadores precisou ser adaptado para ler mais de 2 caracteres.

Para todos os operadores, foi adicionado no modo de compatibilidade uma lista com os operadores que exigem tratamento especial, separados pela categoria de prioridade (condição, soma ou multiplicação). A leitura de operadores no parser primário foi adaptada para sempre consultar o parser secundário com o propósito de decidir quem deve resolver a construção da expressão (linhas 10, 8 e 8 dos algoritmos 4b, 5b e 6b). O parser secundário, por sua vez, consulta o modo de compatibilidade para identificar quais operadores exigem tratamento especial. Em caso positivo, o fluxo de leitura da consulta é transferido do parser primário para o secundário (linhas 16, 10 e 10 dos algoritmos 4b, 5b e 6b). Em caso negativo, o parser primário continua seu fluxo natural.

Assim como na interpretação de funções, caso um dos operadores de união, diferença ou intersecção (“+”, “-” ou “*”) receba como argumento de tipo indeterminado, o parser primário não é capaz de identificar o tipo de dado envolvido até que seja realizada a etapa de otimização. Na incerteza de qual será a operação necessária, o parser secundário assume o controle do fluxo e instancia sua implementação de operação binária. Na etapa de otimização, caso seja identificado que a operação deveria ser uma operação normal de soma, subtração ou multiplicação, a implementação possui um fluxo de correção que substitui o elemento da árvore de expressões pelo elemento correto.

5 CENÁRIOS COBERTOS E RESULTADOS

A ideia deste projeto é integrar as mudanças no ramo principal do H2, como uma forma de contribuição. Portanto, foram obedecidas as normas para submissão de contribuições. Entre elas, a cobertura de testes automatizados.

O projeto H2 possui uma estrutura própria de testes automatizados. A estrutura possui uma classe abstrata *TestBase*, que precisa ser estendida pelas classes criadas pelos contribuidores, além de outras classes de suporte. Essa classe *TestBase* pode ser resumida em 3 responsabilidades: (1) possui um método de inicialização, que deve ser chamado no método "main" da classe de teste; (2) agrega todos os métodos de asserção, sendo naturalmente herdados pela classe de teste sem necessidade de importação; e (3) possui apenas um método abstrato que deve ser implementado, o método "test". O método de inicialização é responsável por iniciar e vincular a suíte de teste com o ambiente de teste. Os métodos de asserção fazem verificações abstraindo a interação com o ambiente de teste. Já o método "test" compõe todo o código dos testes que devem ser realizados. A organização e separação em cenários de testes fica a encargo do contribuidor, a ser feito da forma que desejar. Vale ressaltar que não é qualquer código que é integrado no projeto. O código do contribuidor, antes de ser integrado ao ramo principal do projeto, passa por uma revisão da equipe do H2. Estes avaliam código limpo, organização, cobertura de pelo menos 90% de todo conteúdo adicionado, entre outras questões como redundância e desempenho.

Para testes unitários, a estrutura de testes do H2 garante o isolamento entre as classes de teste. O isolamento entre cada cenário de teste de uma mesma classe, entretanto, é responsabilidade do contribuidor, na sua implementação do método "test". Já os testes com o BD, ou seja, testes que abrem uma conexão com o SGBD H2, a estrutura não garante o isolamento entre classes de teste, visto que o BD é compartilhado. É responsabilidade do contribuidor escrever os testes de forma a apagar toda a sujeira e rastros deixados pela execução dos testes.

Para testes com o BD, os cenários são realizados providenciando uma consulta SQL e avaliando o resultado. Existe uma classe de suporte *TestDb*, que estende *TestBase* e deve ser estendida pela classe de teste do contribuidor, servindo de intermediário entre elas. Essa classe *TestDb* agrega as funções de criação e exclusão do BD, a serem herdadas pela classe de teste, e têm como argumento

apenas um nome para associar ao banco criado ou excluído. Para garantir o isolamento desses testes, a melhor forma de fazê-lo é criar um BD limpo e depois excluí-lo. Cada BD criado dessa forma inicializa também uma instância independente do SGBD. O H2, em sua natureza, possui rápida inicialização permitindo essa abordagem. A interação com a instância do SGBD é feita com a biblioteca de persistência do Java. Em cada cenário de teste, a instância do H2 recebe a consulta em formato *String*, passa por todas as etapas de preparação, otimização e resolução, e retorna o resultado em um objeto reconhecido pela biblioteca de persistência.

Todos os testes foram realizados utilizando o pacote de persistência do Java, fornecendo uma consulta em formato *String* para uma instância do H2 e avaliando o resultado. Considerando que esses testes com o BD interagem com o SGBD da mesma forma que um software deve utilizá-lo, os cenários de teste servem para demonstrar os resultados obtidos por este trabalho. A única diferença entre o ambiente de testes e um software externo utilizando o H2 como uma biblioteca externa é a presença do código fonte. O software externo necessita importar o H2 como uma dependência, gerado em formato JAR. A construção do JAR é feita utilizando o comando “build jar” na raiz do projeto.

Os cenários foram divididos nas seguintes categorias: construção e declaração de valores, operações com tabela para inserção, recuperação, atualização e remoção de dados (CRUD), operadores especiais e funções especiais. Todos os cenários apresentados neste capítulo estão presentes nos testes automatizados, providenciando cobertura de 90% do código implementado, conforme requisitado pela equipe do H2 (Figura 11). Os blocos de código não cobertos caracterizam verificações de precaução (*fallback* e *fallthrough*) com o propósito de garantir o funcionamento dos módulos conforme novas implementações são adicionadas. Esses cenários foram desconsiderados por serem triviais ou por serem inacessíveis com testes de integração.

Figura 11 – Cobertura de código dos componentes adicionados.

100% classes, 95% lines covered in package 'org.h2.compatibility.postgresql'

Element	Class, %	Method, %	Line, %
ExtTypeInfoRange	100% (1/1)	100% (6/6)	97% (34/35)
PostgreSQLDataConverter	100% (1/1)	100% (5/5)	95% (117/123)
PostgreSQLParser	100% (1/1)	100% (7/7)	100% (29/29)
RangeFunction1	100% (1/1)	100% (5/5)	93% (28/30)
RangeFunction2	100% (1/1)	100% (5/5)	97% (80/82)
RangeOperation2	100% (3/3)	100% (15/15)	96% (221/228)
RangeTypeUtil	100% (1/1)	100% (11/11)	94% (52/55)
RangeValue	100% (1/1)	93% (15/16)	82% (39/47)

Fonte: Elaborado pelo autor.

5.1 CONSTRUÇÃO E DECLARAÇÃO DE VALORES

A construção de valores *range* foi quase inteiramente replicada da forma como é feita no PostgreSQL. No H2 a construção dos valores pode ser feita de 3 formas: (1) declaração explícita com o operador "::"; (2) conversão de valor com a função "cast", ou; (3) utilização das funções exclusivas do PostgreSQL para a construção dos valores, cada uma com mesmo nome do tipo de dado. Internamente, o operador "::" e a função "cast" resultam na conversão do valor declarado no formato *varchar* para o novo tipo de dado, enquanto as funções exclusivas produzem diretamente o valor com o tipo correto. Considerando a ausência da conversão utilizando a função especial, recomenda-se que, ao utilizar a extensão, as consultas utilizem sempre as funções especiais para construção de valores *range*. O Quadro 7 demonstra as 3 formas de construção do intervalo "[1,3)" para o tipo *int4range*. Os demais tipos de *range* são construídos de forma análoga.

Quadro 7 – Demonstração de declaração de valores *range* em seu formato padrão.

```
-- Resultam em '[1,3)'
select '[1,3)::int4range;
select cast('[1,3)' as int4range);
select int4range(1,3);
```

Fonte: Elaborado pelo autor.

Os tipos *int4range*, *int8range* e *daterange* possuem forma canônica (fechado no limite inferior e aberto no limite superior). O Quadro 8 demonstra a construção de

intervalos com delimitadores explícitos para o tipo *int4range*. Em todos os casos apresentados, a função de canonização é acionada.

Quadro 8 – Demonstração de declaração de valores *range* em seu formato não canônico e resultado da canonização.

```
-- Resultam em '[1,3)'
select '(0,3)::int4range;
select '(0,2]::int4range;
select '[1,2]::int4range;
select cast('(0,3)' as int4range);
select cast('(0,2]' as int4range);
select cast('[1,2]' as int4range);
select int4range(0,3, '(');
select int4range(0,2, '[');
select int4range(1,2, '[');
```

Fonte: Elaborado pelo autor.

Limites ausentes oriundos do intervalo vazio ou limites infinitos necessitam tratamento especial. O Quadro 9 demonstra a construção de intervalos vazios e intervalos com limite infinito, onde há ausência de limites e a extensão produz corretamente os valores. O H2 não é capaz de reconhecer palavras-chave dentro de um *varchar*. Portanto, palavras como “today” e “infinity”, reconhecidas pelo PostgreSQL na declaração explícita e na função “cast”, não são reconhecidas pelo H2. Uma análise da interpretação de valores *varchar* poderia ser feita para solucionar esse problema, mas como é um comportamento que não se limita ao tipo de dado *range*, essa análise foi deixada para trabalhos futuros.

Quadro 9 – Demonstração de declaração de valores *range* para intervalo vazio e limites infinitos.

```
-- Resultam em 'empty'
select 'empty)::int4range;
select cast('empty' as int4range);
select int4range(1,1);

-- Resultam em '[1,)'
select '[1,)::int4range;
select cast('[1,)' as int4range);
select int4range(1,NULL);

-- Resultam em '(,3)'
select '(,3)::int4range;
select cast('(,3)' as int4range);
select int4range(NULL,3);

-- Resultam em '(,)'
```

```

select '(,)::int4range;
select cast('(',')' as int4range);
select int4range(NULL,NULL);

-- Erro de sintaxe. Não é reconhecida a palavra-chave "infinity" dentro de um varchar.
select '[1,infinity)::int4range;

```

Fonte: Elaborado pelo autor.

Por fim, existem cenários onde o PostgreSQL não permite construir o intervalo devido à inconsistência dos limites (limite inferior maior que limite superior), resultando em erro. Esses cenários foram replicados e são demonstrados no Quadro 10.

Quadro 10 – Demonstração de declaração de valores *range* com intervalo impossível.

```

-- Exibem erro informando intervalo impossível
select '[2,1)::int4range;
select cast('[2,1)' as int4range);
select int4range(2,1);

```

Fonte: Elaborado pelo autor.

5.2 OPERAÇÕES COM TABELA

Todas as consultas que interagem com tabelas produzem resultados equivalentes às consultas realizadas no PostgreSQL, no que se refere ao uso de tipos de dados *range*. Como foi adotada a abordagem superficial, todos os valores de *range* são reconhecidos, também, como valores *varchar*. Isso significa que operações como a construção e utilização de índices ou consultas que interagem com as colunas especiais (utilizadas em cláusulas *order by*, *group by*, *join*, etc) interpretam os valores como *varchar*. Dessa forma, podem haver diferenças no desempenho das consultas, mas o resultado final é equivalente.

Para demonstrar o funcionamento da extensão com tabelas, primeiramente deve ser construída uma tabela. O Quadro 11 demonstra a construção de uma tabela com colunas do tipo *int4range*.

Quadro 11 – Criação de tabela com coluna de um tipo *range*.

```

-- Cria tabela com colunas do tipo int4range
create table T (R int4range, S int4range);

```

Fonte: Elaborado pelo autor.

O Quadro 12 demonstra a inserção de alguns valores de teste na tabela criada no Quadro 11.

Quadro 12 – Inserção de valores *range*.

```
-- Insere valores na tabela com colunas do tipo int4range
insert into T values ('[1,4)::int4range, '[2,5)::int4range);
insert into T values ('empty)::int4range, '(,)::int4range);
```

Fonte: Elaborado pelo autor.

O Quadro 13 demonstra a recuperação dos dados persistidos na tabela, utilizando também um valor *range* na condição de filtro.

Quadro 13 – Recuperação de valores *range*.

```
-- Resulta em '[2,5)'
select S from T where R = '[1,4)::int4range;

-- Resulta em '(,)'
select S from T where R = 'empty)::int4range;

-- Resulta em 'empty'
select R from T where S = '(,)::int4range;
```

Fonte: Elaborado pelo autor.

O Quadro 14 demonstra a atualização de um registro na tabela, também utilizando um valor *range* no filtro.

Quadro 14 – Atualização de valores *range*.

```
-- Resulta em '[3,6)'
update T set S = '[3,6)::int4range where R = '[1,4)::int4range;
select S from T where R = '[1,4)::int4range;
```

Fonte: Elaborado pelo autor.

Por fim, o Quadro 15 demonstra a exclusão de um registro com filtro.

Quadro 15 – Exclusão de valores *range*.

```
-- Resulta em NULL
delete from T where R = '[1,4)::int4range;
select * from T where R = '[1,4)::int4range;
```

Fonte: Elaborado pelo autor.

Demais consultas que interagem com as colunas do novo tipo as interpretam como tipo *varchar*, como o H2 sempre fez. Sendo assim, não há necessidade de demonstrar seu funcionamento.

5.3 OPERADORES ESPECIAIS

Todos os operadores que lidam com tipos de dado *range* foram inteiramente replicados. São 11 operadores novos e os operadores "contém" (*@>*) e "está contido" (*<@*) podem ser usados de duas formas, para verificar se um elemento pertence a um intervalo ou se um intervalo inteiro pertence a outro intervalo, totalizando 13 novas combinações de operações binárias. Além das operações isoladas, também foi avaliado o enfileiramento de operações de união, diferença e intersecção, a priorização das operações de união e diferença sobre as operações de condição, e a priorização da operação de intersecção sobre as operações de união e diferença.

Devido à ambiguidade dos operadores de união, diferença e intersecção com operadores de soma, subtração e multiplicação, alguns blocos de controles foram necessários para resolver conflitos. Mesmo assim, o resultado final de todas as operações são equivalentes às consultas executadas no PostgreSQL, que é o que importa para o modo de compatibilidade.

5.3.1 União, Diferença e Intersecção

O Quadro 16 demonstra a utilização das operações de união, diferença e intersecção. Os exemplos avaliados apresentam os cenários onde há sobreposição dos intervalos, adjacência de intervalos, e os cenários onde o resultado matemático contém uma lacuna. A lacuna não é representável com um valor *range* e uma mensagem de erro deve ser exibida.

Quadro 16 – Demonstração de operações de união, diferença e intersecção com intervalos canônicos.

```
-- União
-- Resultam em '[1,5)'
select '[1,3)::int4range + '[2,5)::int4range;
select '[1,3)::int4range + '[3,5)::int4range;

-- Exibe erro informando sobre a lacuna
```

```

select '[1,3)::int4range + '[4,5)::int4range;

-- Diferença
-- Resultam em '[1,2)'
select '[1,3)::int4range - '[2,5)::int4range;
select '[1,2)::int4range - '[2,5)::int4range;

-- Resulta em 'empty'
select '[1,3)::int4range - '[1,5)::int4range;

-- Exibe erro informando sobre a lacuna
select '[1,4)::int4range - '[2,3)::int4range;

-- Intersecção
-- Resulta em '[2,4)'
select '[1,5)::int4range * '[2,4)::int4range;

-- Resulta em 'empty'
select '[1,3)::int4range * '[3,5)::int4range;

```

Fonte: Elaborado pelo autor.

O Quadro 17 complementa o Quadro 16 utilizando um tipo de intervalo que não possui forma canônica, permitindo que a lacuna exista mesmo quando o limite inferior de um intervalo coincide com o limite superior do outro intervalo.

Quadro 17 – Demonstração de operações de união, diferença e intersecção com intervalos não canônicos.

```

-- União
-- Resultam em '[1,5)'
select '[1,3)::numrange + '[3,5)::numrange;
select '[1,3)::numrange + '[3,5)::numrange;
select '[1,3)::numrange + '(3,5)::numrange;

-- Exibe erro informando sobre a lacuna
select '[1,3)::numrange + '(3,5)::numrange;

-- Diferença
-- Resulta em '[1,3)'
select '[1,3)::numrange - '[3,5)::numrange;

-- Resulta em '[1,3)'
select '[1,3)::numrange - '(3,5)::numrange;

-- Exibe erro informando sobre a lacuna
select '[1,3)::numrange - '[2,2)::numrange;

-- Intersecção
-- Resulta em '[3,3)'
select '[1,3)::numrange * '[3,5)::numrange;

```

Fonte: Elaborado pelo autor.

O Quadro 18 demonstra os cenários das 3 operações com intervalo vazio e limites infinitos.

Quadro 18 – Demonstração de operações de união, diferença e intersecção com intervalo vazio e limites infinitos.

```
-- União
-- Resulta em '[3,5)'
select 'empty'::int4range + '[3,5)'::int4range;

-- Resulta em '[3,)'
select '[4,)'::int4range + '[3,5)'::int4range;

-- Resulta em '(,5)'
select '(,4)'::int4range + '[3,5)'::int4range;

-- Resulta em '(,)'
select '(,)'::int4range + '[3,5)'::int4range;

-- Diferença
-- Resulta em '[1,3)'
select '[1,3)'::int4range - 'empty'::int4range;

-- Resultam em 'empty'
select 'empty'::int4range - '[1,5)'::int4range;
select '[1,3)'::int4range - '(,)'::int4range;

-- Exibe erro informando sobre a lacuna
select '(,)'::int4range - '[2,3)'::int4range;

-- Intersecção
-- Resulta em 'empty'
select 'empty'::int4range * '[3,5)'::int4range;

-- Resulta em '[3,5)'
select '(,)'::int4range * '[3,5)'::int4range;
```

Fonte: Elaborado pelo autor.

Devido à ambiguidade dos operadores, na etapa de preparação, quando o parser primário lê os operadores "+", "-" ou "*", não é possível identificar imediatamente o tipo do valor envolvido na operação para distinguir soma e união, subtração e diferença, e multiplicação e intersecção respectivamente. Sendo assim, acontece o desvio do fluxo de execução para o parser secundário. Esses cenários de indeterminação acontecem quando o elemento à esquerda da operação é resultado de uma função, é uma variável

ou coluna, ou é uma subconsulta anônima. O Quadro 19 demonstra o funcionamento da extensão para cada um desses cenários.

Quadro 19 – Demonstração de operações de união, diferença e intersecção com cenários de indeterminação.

```
-- União
-- Resultam em '[1,5)'
select int4range(1,3) + '[2,5)::int4range;
select testSq + '[2,5)::int4range from (select '[1,3)::int4range as testSq);
select (select '[1,3)::int4range) + '[2,5)::int4range;

-- Diferença
-- Resultam em '[1,2)'
select int4range(1,3) - '[2,5)::int4range;
select testSq - '[2,5)::int4range from (select '[1,3)::int4range as testSq);
select (select '[1,3)::int4range) - '[2,5)::int4range;

-- Intersecção
-- Resultam em '[2,3)'
select int4range(1,3) * '[2,5)::int4range;
select testSq * '[2,5)::int4range from (select '[1,3)::int4range as testSq);
select (select '[1,3)::int4range) * '[2,5)::int4range;
```

Fonte: Elaborado pelo autor.

5.3.2 Operações Condicionais

Ao contrário das operações de união, diferença e intersecção, as operações de condição resultam em um valor booleano. Para todos os casos de uso, foi possível atingir resultados equivalentes às consultas executadas no PostgreSQL.

O Quadro 20 demonstra a utilização do operador de adjacência. Apenas intervalos onde os limites se tocam têm resultado verdadeiro. Intervalos sobrepostos e demais intervalos disjuntos resultam em falso. Devido à ausência de limites, operações com intervalo vazio ou infinito sempre resultam em falso.

Quadro 20 – Demonstração de operações de adjacência.

```
-- Adjacência
-- Resulta em TRUE
select '[1,3)::int4range -|- '[3,5)::int4range;

-- Resultam em FALSE
select '[1,3)::int4range -|- '[2,5)::int4range;
select '[1,2)::int4range -|- '[3,5)::int4range;
select 'empty)::int4range -|- '[3,5)::int4range;
select '(,)::int4range -|- '[3,5)::int4range;
```

Fonte: Elaborado pelo autor.

O Quadro 21 demonstra a utilização do operador de sobreposição. A operação retorna verdadeiro somente quando há elementos em comum nos intervalos. Por definição, a operação com um intervalo vazio sempre resulta em falso.

Quadro 21 – Demonstração de operações de sobreposição.

```
-- Sobreposição
-- Resultam em TRUE
select '[1,3)::int4range && '[2,5)::int4range;
select '(,)::int4range && '[3,5)::int4range;

-- Resultam em FALSE
select '[1,3)::int4range && '[3,5)::int4range;
select 'empty)::int4range && '[3,5)::int4range;
```

Fonte: Elaborado pelo autor.

O Quadro 22 demonstra a utilização do operador de contenção, onde o elemento ou intervalo da direita pertence ao intervalo da esquerda (sentido direita-esquerda). No caso de comparação com 2 intervalos, o intervalo inteiro da direita deve estar contido no intervalo da direita para que o resultado seja verdadeiro. Por definição, todo intervalo contém o intervalo vazio.

Quadro 22 – Demonstração de operações de contenção, sentido direita-esquerda.

```
-- Contém
-- Resultam em TRUE
select '[1,4)::int4range @> 2;
select '[1,4)::int4range @> '[2,3)::int4range;
select '[1,4)::int4range @> 'empty)::int4range;
select '(,)::int4range @> '[2,3)::int4range;

-- Resultam em FALSE
select '[1,4)::int4range @> 5;
select '[1,4)::int4range @> '[2,5)::int4range;
select 'empty)::int4range @> '[2,5)::int4range;
select '[1,4)::int4range @> '(,)::int4range;
```

Fonte: Elaborado pelo autor.

O Quadro 23 demonstra a utilização do segundo operador de contenção, desta vez onde o elemento ou intervalo da esquerda pertence ao intervalo da direita

(sentido esquerda-direita). Os resultados são equivalentes à inversão dos intervalos envolvidos no operador anterior.

Quadro 23 – Demonstração de operações de contenção, sentido esquerda-direita.

```
-- Está contido
-- Resultam em TRUE
select 2 <@ '[1,4)>::int4range;
select '[2,3)>::int4range <@ '[1,4)>::int4range;
select 'empty'::int4range <@ '[1,4)>::int4range;
select '[2,3)>::int4range <@ '(,)'::int4range;

-- Resultam em FALSE
select 5 <@ '[1,4)>::int4range;
select '[2,5)>::int4range <@ '[1,4)>::int4range;
select '[2,5)>::int4range <@ 'empty'::int4range;
select '(,)'::int4range <@ '[1,4)>::int4range;
```

Fonte: Elaborado pelo autor.

O Quadro 24 demonstra a utilização do operador estritamente à esquerda, onde o intervalo à esquerda do operador deve estar inteiramente à esquerda do intervalo à direita do operador. Em outras palavras, o maior elemento do intervalo à esquerda deve ser menor que o menor elemento do intervalo à direita. Por definição, operações com o intervalo vazio sempre resultam em falso.

Quadro 24 – Demonstração de operações de estritamente à esquerda.

```
-- Estritamente à esquerda
-- Resultam em TRUE
select '[1,3)>::int4range << '[4,5)>::int4range;
select '[1,3)>::int4range << '[3,5)>::int4range;

-- Resultam em FALSE
select '[1,3)>::int4range << '[2,5)>::int4range;
select '[1,3]::numrange << '[3,5]::numrange;
select 'empty'::int4range << '[3,5)>::int4range;
select '[3,5)>::int4range << 'empty'::int4range;
select '(,)'::int4range << '[3,5)>::int4range;
select '[3,5)>::int4range << '(,)'::int4range;
```

Fonte: Elaborado pelo autor.

O Quadro 25 demonstra a utilização do operador estritamente à direita, onde o intervalo à esquerda do operador deve estar inteiramente à direita do intervalo à direita do operador, ou seja, o menor elemento do intervalo à esquerda deve ser maior que o maior elemento do intervalo à direita. Assim como no operador estritamente à esquerda, por definição, operações com o intervalo vazio sempre resultam em falso.

Quadro 25 – Demonstração de operações de estritamente à direita.

```

-- Estritamente à direita
-- Resultam em TRUE
select '[1,3)::int4range >> [-1,0)::int4range;
select '[1,3)::int4range >> [-1,1)::int4range;

-- Resultam em FALSE
select '[1,3)::int4range >> [-1,2)::int4range;
select '[1,3)::numrange >> [-1,1)::numrange;
select 'empty)::int4range >> [-1,1)::int4range;
select '[1,3)::int4range >> 'empty)::int4range;
select '(,)::int4range >> [-1,1)::int4range;
select '[1,3)::int4range >> '(,)::int4range;

```

Fonte: Elaborado pelo autor.

O Quadro 26 demonstra a utilização do operador que não se estende à direita, onde o intervalo à esquerda do operador não deve possuir elementos maiores que o limite superior do intervalo à direita do operador. Por definição, operações com intervalo vazio resultam em falso.

Quadro 26 – Demonstração de operações que não se estendem à direita.

```

-- Não se estende à direita
-- Resultam em TRUE
select '[1,3)::int4range &< '[1,4)::int4range;
select '[1,3)::int4range &< '[1,3)::int4range;
select '[1,3)::int4range &< '(,)::int4range;

-- Resultam em FALSE
select '[1,3)::int4range &< '[1,2)::int4range;
select '[1,3)::numrange &< '[1,3)::numrange;
select 'empty)::int4range &< '[1,3)::int4range;
select '[1,3)::int4range &< 'empty)::int4range;
select '(,)::int4range &< '[1,3)::int4range;

```

Fonte: Elaborado pelo autor.

Por fim, o Quadro 27 demonstra a utilização do operador que não se estende à esquerda. Análogo ao operador anterior, o intervalo à esquerda do operador não deve possuir elementos menores que o limite inferior do intervalo à direita do operador. Por definição, operações com intervalo vazio resultam em falso.

Quadro 27 – Demonstração de operações que não se estendem à esquerda.

```

-- Não se estende à esquerda
-- Resultam em TRUE

```



```

select '[1,3)::int4range &> '[0,5)::int4range;
select '[1,3)::int4range &> '[1,5)::int4range;
select '[1,3)::int4range &> '(,)::int4range;

-- Resultam em FALSE
select '[1,3)::int4range &> '[2,5)::int4range;
select '[1,3)::numrange &> '(1,5)::numrange;
select 'empty)::int4range &> '[1,3)::int4range;
select '[1,3)::int4range &> 'empty)::int4range;
select '(,)::int4range &> '[1,3)::int4range;

```

Fonte: Elaborado pelo autor.

5.3.3 Enfileiramento e Prioridade de Operadores

Os operadores de união, diferença e intersecção recebem intervalos de um determinado tipo e resultam em outro intervalo do mesmo tipo. Dessa forma, é possível enfileirar esses 3 operadores. A avaliação do resultado do enfileiramento dessas operações foi feita com as consultas demonstradas no Quadro 28.

Quadro 28 – Demonstração de enfileiramento de operações.

```

-- União e diferença
-- Enfileiramento de uniões. Resulta em '[1,6)'
select '[1,3)::int4range + '[2,5)::int4range + '[4,6)::int4range;

-- Enfileiramento de diferenças. Resulta em '[4,7)'
select '[3,8)::int4range - '[1,4)::int4range - '[7,9)::int4range;

-- Enfileiramento de diferença seguido de união, resolvido da esquerda para a direita.
Resulta em '[4,6)'
select '[3,5)::int4range - '[1,4)::int4range + '[5,6)::int4range;

-- Enfileiramento de união seguido de diferença, resolvido da esquerda para a direita.
Resulta em '[4,6)'
select '[3,5)::int4range + '[5,6)::int4range - '[1,4)::int4range;

-- Intersecção
-- Enfileiramento de intersecções. Resulta em '[3,5)'
select '[2,6)::int4range * '[3,7)::int4range * '[1,5)::int4range;

```

Fonte: Elaborado pelo autor.

O operador de intersecção tem prioridade mais alta, operadores de união e diferença têm prioridade entre intersecção e demais operadores, e os operadores condicionais têm prioridade mais baixa. Sendo assim, é possível enfileirar operadores de diferentes níveis de prioridade. As prioridades das operações foram avaliadas conforme o Quadro 29. Os valores foram estrategicamente selecionados de forma

que, caso as consultas fossem resolvidas da esquerda para a direita, teriam resultados diferentes.

Quadro 29 – Demonstração de prioridade de operações em diferentes níveis.

```
-- Prioridade de intersecção sobre união. Resulta em '[3,5]'
select '[3,4]>::int4range + '[1,5]>::int4range * '[4,6]>::int4range;

-- Prioridade de intersecção sobre diferença. Resulta em '[1,3]'
select '[1,4]>::int4range - '[2,5]>::int4range * '[3,6]>::int4range;

-- Prioridade de união sobre operadores condicionais. Resultam em valor booleano
correto. Falha na prioridade das operações resultaria em erro de sintaxe.
select '[1,2]>::int4range -| '[3,5]>::int4range + '[4,6]>::int4range;
select '[1,2]>::int4range && '[3,5]>::int4range + '[4,6]>::int4range;
select '[1,2]>::int4range @> '[3,5]>::int4range + '[4,6]>::int4range;
select '[1,2]>::int4range <@ '[3,5]>::int4range + '[4,6]>::int4range;
select '[1,2]>::int4range << '[3,5]>::int4range + '[4,6]>::int4range;
select '[1,2]>::int4range >> '[3,5]>::int4range + '[4,6]>::int4range;
select '[1,2]>::int4range &< '[3,5]>::int4range + '[4,6]>::int4range;
select '[1,2]>::int4range &> '[3,5]>::int4range + '[4,6]>::int4range;

-- Prioridade de diferença sobre operadores condicionais. Resultam em valor booleano
correto. Falha na prioridade das operações resultaria em erro de sintaxe.
select '[1,2]>::int4range -| '[3,5]>::int4range - '[4,6]>::int4range;
select '[1,2]>::int4range && '[3,5]>::int4range - '[4,6]>::int4range;
select '[1,2]>::int4range @> '[3,5]>::int4range - '[4,6]>::int4range;
select '[1,2]>::int4range <@ '[3,5]>::int4range - '[4,6]>::int4range;
select '[1,2]>::int4range << '[3,5]>::int4range - '[4,6]>::int4range;
select '[1,2]>::int4range >> '[3,5]>::int4range - '[4,6]>::int4range;
select '[1,2]>::int4range &< '[3,5]>::int4range - '[4,6]>::int4range;
select '[1,2]>::int4range &> '[3,5]>::int4range - '[4,6]>::int4range;
```

Fonte: Elaborado pelo autor.

5.4 FUNÇÕES ESPECIAIS

As funções especiais incluídas por este trabalho foram inteiramente implementadas, isto é, todas as funções especiais relacionadas aos tipos de dados *range* do PostgreSQL exceto a função de criação de novos tipos, devido a ausência de UDT no H2, conforme explicado no capítulo anterior. As funções de criação de valores *range* já foram apresentadas na seção 5.1, sendo apresentadas nesta seção somente as funções remanescentes.

Particularmente, as funções *upper* e *lower* possuem implementação diferente do PostgreSQL. Elas utilizam um fluxo de correção devido à ambiguidade das funções *upper* e *lower* nativas do padrão SQL, causada devido à abordagem superficial. Além

disso, devido à necessidade de inferência do subtipo de seus parâmetros, também foi avaliada a propagação do resultado, avaliando assim o tipo correto do retorno dessas funções. O Quadro 30 demonstra os testes realizados utilizando consultas que envolvem as funções *upper* e *lower*. Para os cenários de propagação, foi utilizado a função *sqrt* (raiz quadrada) que recebe como argumento um valor numérico, oriundo do subtipo de *int4range*.

Quadro 30 – Demonstração das funções *upper* e *lower*.

```
-- Upper
-- Resulta em 3
select upper('[1,3]::int4range);

-- Resultam em NULL
select upper('[1,]::int4range);
select upper('empty'::int4range);

-- Propagação do resultado. Resulta em 2
select sqrt(upper('[1,4]::int4range));

-- Lower
-- Resulta em 1
select lower('[1,3]::int4range);

-- Resultam em NULL
select lower('[,3]::int4range);
select lower('empty'::int4range);

-- Propagação do resultado. Resulta em 2
select sqrt(lower('[4,6]::int4range));
```

Fonte: Elaborado pelo autor.

As demais funções *isempty*, *lower_inc*, *upper_inc*, *lower_inf*, *upper_inf* e *range_merge* foram implementadas sem complicações. Seus resultados foram avaliados conforme o Quadro 31.

Quadro 31 – Demonstração das funções *isempty*, *lower_inc*, *upper_inc*, *lower_inf*, *upper_inf* e *range_merge*.

```
-- IsEmpty
-- Resulta em TRUE
select isempty('empty'::int4range);

-- Resultam em FALSE
select isempty('[1,3]::int4range);
select isempty('()'::int4range);

-- Lower_Inc
```

```

-- Resulta em TRUE
select lower_inc('[1,3]::numrange);

-- Resultam em FALSE
select lower_inc('(1,3)::numrange);
select lower_inc('empty'::numrange);

-- Upper_Inc
-- Resulta em TRUE
select upper_inc('[1,3]::numrange);

-- Resultam em FALSE
select upper_inc('[1,3]::numrange);
select upper_inc('empty'::numrange);

-- Lower_Inf
-- Resulta em TRUE
select lower_inf('(,3)::int4range);

-- Resultam em FALSE
select lower_inf('[1,3)::int4range);
select lower_inf('empty'::int4range);

-- Upper_Inf
-- Resulta em TRUE
select upper_inf('[1,)::int4range);

-- Resultam em FALSE
select upper_inf('[1,3)::int4range);
select upper_inf('empty'::int4range);

-- Range_Merge
-- Resulta em '[1,5)'
select range_merge('[1,2)::int4range, '[4,5)::int4range);

```

Fonte: Elaborado pelo autor.

5.5 APLICAÇÃO EM UM CASO DE USO

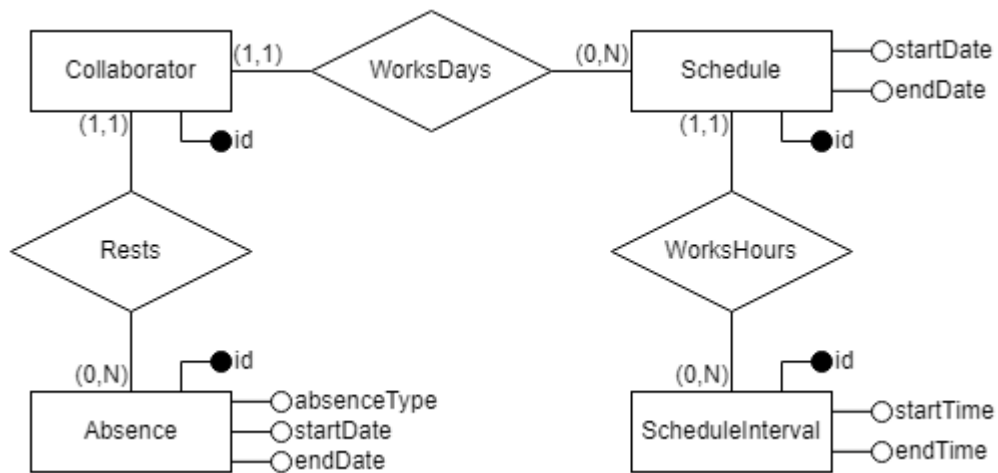
A extensão proposta foi experimentada em um caso de uso de um projeto real. O projeto consiste numa aplicação Web para controle de horas trabalhadas pelos colaboradores de uma empresa. A aplicação possui um cadastro de colaboradores, períodos de contratações, com data de início e fim, itinerários, com a carga horária semanal de cada período de contratação, períodos de ausência, onde consta as férias usufruídas pelo colaborador, e outros utilitários. A aplicação também utiliza o PostgreSQL para armazenamento de dados em ambiente de produção e H2 para execução de testes automatizados em ambiente de desenvolvimento.

Dentre as funções relevantes para este trabalho, a aplicação é capaz de calcular a quantidade de dias de férias que o colaborador tem disponível para usufruir. O cálculo de férias é baseado no total de horas trabalhadas, convertido em dias de acordo com a carga horária atual do colaborador. O cálculo é feito inteiramente em uma consulta SQL, que retorna diretamente a quantidade de dias que o colaborador pode usufruir de férias. Essa consulta utiliza as funções exclusivas do PostgreSQL *daterange*, *upper*, *lower* e *date_part*, além de utilizar o operador de intersecção. Devido ao uso de funções e operadores exclusivos do PostgreSQL, não é possível executar a consulta com o H2, tornando impossível a elaboração de testes automatizados com BD. Diante deste impasse, o projeto foi dado como concluído sem a cobertura de testes de regressão para o módulo de férias.

Essa deficiência do H2 é suprida com a extensão apresentada neste trabalho. O uso das funções *daterange*, *upper* e *lower*, bem como o operador de intersecção, foram apresentados nas seções anteriores deste capítulo. Resta a função *date_part*, que não foi descrita no capítulo de desenvolvimento. A extensão facilita a implementação de novas funções e a função *date_part* foi implementada para experimentação. Sua implementação é trivial devido à função equivalente nativa do Java e sua introdução à extensão é semelhante às demais funções: uma classe foi criada para representar a operação na árvore de expressões e o nome da função foi adicionado ao parser secundário, no bloco correspondente de tradução de funções.

A Figura 11 apresenta o diagrama de entidade-relacionamento das entidades envolvidas. Cada colaborador (*Collaborator*) possui diversos períodos de contratação (*Schedule*) ao longo de sua jornada na empresa, devido à renovação de contrato (data de início de contrato *startDate* e fim *endDate*). Cada período de contratação possui definições de horário de trabalho (*ScheduleInterval*) ao longo da semana, que devem somar a quantidade total de horas acordadas no contrato (início do período *startTime* e término *endTime*). As férias já usufruídas pelo colaborador são armazenadas numa tabela de registro de ausências (*Absence*) com uma classificação de tipo correspondente (*absenceType*) e datas de início e fim (*startDate* e *endDate*).

Figura 11 – Diagrama de entidade-relacionamento do caso de uso.



Fonte: Elaborado pelo autor.

A consulta de cálculo de férias é apresentada no Quadro 32. Ela pode ser dividida em 3 partes: uma consulta interna, intermediária e externa. A consulta interna retorna todos os períodos de trabalho do colaborador em datas, e a carga horária de trabalho para cada período. A consulta intermediária busca os períodos de férias já usufruídos a serem descontados do cálculo total. Como o período de férias é armazenado em dias e o cálculo de férias é em horas, é necessária uma conversão de dias para horas. A consulta intermediária utiliza a operação de intersecção de intervalos de datas *daterange* entre os períodos de férias e os períodos de trabalho. O resultado dessas intersecções é vinculado à carga horária diária atribuída ao período de trabalho. As funções *upper* e *lower* são utilizadas para extrair as datas finais e iniciais resultantes das intersecções. Com as informações extraídas, a consulta intermediária calcula o total de dias a serem descontados do cálculo total de férias. Por fim, a consulta externa soma o total de horas trabalhadas, converte em dias, e subtrai o total de férias usufruídas, calculado na consulta intermediária.

Quadro 32 – Consulta de cálculo de férias.

```
select -- Início da consulta externa
  coalesce(cast(max(case when ativo = 1 then totalDiario else 0 end) as bigint),
  0) as dailyJourney,
  coalesce(case
    when max(case when ativo = 1 then totalDiario else 0 end) > 0 then
    cast(floor(((sum(horasTrabalhadas) / 12) - sum(quantidadeFerias)) / max(case when ativo
    = 1 then totalDiario else 0 end))) as bigint)
    else 0
  end,
  0) as daysAvailable
from
```

```

(
select -- Início da consulta intermediária
  collabld,
  scheduleId,
  totalDiario,
  horasTrabalhadas,
  ativo,
  sum(case when absence.id is not null then (upper((daterange(scheduleStart,
scheduleEnding) * daterange(absence.startDate, absence.endDate))) -
lower((daterange(scheduleStart, scheduleEnding) * daterange(absence.startDate,
absence.endDate))) + 1) * totalDiario else 0 end) as quantidadeFerias
from
  (
  select -- Início da consulta interna
    schedule.id as scheduleId,
    schedule.startDate as scheduleStart,
    schedule.endDate as scheduleEnding,
    collaborator.id as collabld,
    (case
      when schedule.endDate > current_date then current_date
      else schedule.endDate
    end - schedule.startDate + 1) * (date_part('hour',
sum(scheduleInterval.endTime - scheduleInterval.startTime)) / 5) as
horasTrabalhadas,
    (date_part('hour',
sum(scheduleInterval.endTime - scheduleInterval.startTime)) / 5) as totalDiario,
    (case
      when schedule.endDate >= current_date then 1
      else 0
    end) as ativo
  from
    Collaborator collaborator
  inner join Schedule schedule
on
  schedule.idCollaborator = collaborator.id
  inner join ScheduleInterval scheduleInterval
on
  scheduleInterval.idSchedule = schedule.id
group by
  schedule.id,
  schedule.startDate,
  schedule.endDate,
  collaborator.id) as subQuerySchedule -- Fim da consulta interna
left join Absence absence
on
  absence.idCollaborator = collabld
  and absence.absenceType = 'VACATION'
  and daterange(scheduleStart,
scheduleEnding) && daterange(absence.startDate,
absence.endDate)
group by
  collabld,
  scheduleId,
  totalDiario,

```

```
horasTrabalhadas,  
ativo) as subScheduleAbsence -- Fim da consulta intermediária  
group by  
collabld  
; -- Fim da consulta externa
```

Fonte: Elaborado pelo autor.

A extensão apresentada neste trabalho provou-se eficaz para resolver a consulta deste caso de uso em sua íntegra, produzindo o resultado correto. Como o projeto foi dado como concluído, não foram complementados os testes automatizados do projeto. Porém, o caso de uso comprova a eficiência da extensão para projetos futuros.

6 CONCLUSÕES E TRABALHOS FUTUROS

Todos os objetivos propostos por este trabalho foram alcançados. A construção de valores *range*, com seus 6 subtipos padrão, foi possível com a utilização de alias de *varchar*. A persistência e recuperação de dados em tabelas foi alcançada utilizando tratamentos nativos para *varchar*, sendo o resultado produzido por consultas no H2 equivalente aos resultados obtidos em consultas no PostgreSQL. E os operadores e funções relacionadas aos tipos de dados *range* foram inteiramente replicados de forma quase idêntica às suas implementações no PostgreSQL, salvo apenas pelos fluxos de correção para resolução dos casos de ambiguidade.

Existem limitações derivadas da abordagem superficial, todas relacionadas à inferência dos tipos dos valores limites durante a conversão de *varchar* para *range*. Essas limitações poderiam ser resolvidas com a implementação profunda. Porém, considerando que o suporte próprio do H2 para *alias* de tipos de dados já utiliza uma implementação superficial, e considerando as recomendações da equipe do H2, foi decidido fazer a implementação superficial. O propósito dos modos de compatibilidade são a equivalência das consultas, não necessariamente tendo suas implementações idênticas. A abordagem superficial é suficiente para atingir a equivalência das consultas.

A situação com as palavras-chave *today* e *infinity* na declaração explícita de valores *range* poderia ser mitigada no conversor de dados (*DataConverter*), explicado na seção 4.4.2. Esse comportamento, porém, é uma peculiaridade do PostgreSQL em geral, não apenas para os valores *range*. O reconhecimento dessas palavras no *DataConverter* seria uma solução paliativa para um problema maior. Não foi implementado o reconhecimento dessas palavras porque, além de aumentar consideravelmente a complexidade do *DataConverter*, não é uma deficiência da extensão. Todas as consultas onde são usadas essas palavras podem ser reescritas utilizando as funções especiais de construção do novo tipo, eliminando completamente o uso das palavras-chaves. O uso das funções especiais é a forma mais recomendada para o uso da extensão, considerando que os valores são criados diretamente com o tipo correto ao invés de passar por uma conversão. Além disso, é uma característica do código limpo o código ser testável.

Existem cenários extraordinários descartados durante o desenvolvimento da extensão. Um exemplo é a troca do modo de compatibilidade. Da forma com que o

H2 foi implementado, é possível trocar o modo de compatibilidade de um determinado SGBD durante o seu tempo de vida. Isso compromete não somente a extensão produzida neste trabalho, mas também o suporte a novos tipos de dados como *alias* já existente no H2. Cenários como este são absurdos, sem utilidade, e jamais acontecerão para fins práticos. Eles são observados somente explorando intencionalmente as peculiaridades do H2. Vale ressaltar que a utilidade primária do modo de compatibilidade é a realização de testes automatizados, sendo desencorajada sua utilização para outros fins.

Alguns aspectos de desempenho poderiam ser revisados. A extensão depende da interpretação dos valores em formato *varchar* para interação de colunas de tabelas. Teoricamente, a extensão permite a implementação de algoritmos exclusivos para os novos tipos onde normalmente seriam utilizados algoritmos para o tipo de dado *varchar*. Um exemplo é o algoritmo de ordenação nos índices criados para colunas definidas com os novos tipos. Outros aspectos de desempenho a serem avaliados são os fluxos de correção nos casos de ambiguidade de operadores e funções. Atualmente quem resolve a ambiguidade é o parser secundário, durante a etapa de preparação, e é feita a correção na etapa de otimização. Essa responsabilidade poderia ser transferida para os elementos ambíguos criados pelo parser primário e realizada a correção da operação de forma inversa, transferindo o fluxo para o parser secundário somente quando realmente necessário. Essa mudança potencialmente seria mais performática devido à redução do fluxo de execução, mas dificulta a inclusão de novos operadores e funções.

Este trabalho estabeleceu uma base para futuras expansões. Ao longo do desenvolvimento do projeto, novas funcionalidades do PostgreSQL relacionadas aos tipos de dados *range* foram desenvolvidas. Essas novidades incluem os tipos de dados *multirange*, que são conjuntos de *range* capazes de representar lacunas, e, recentemente, a criação de índices especiais que utilizam *multirange* e são capazes de acelerar consultas utilizando os operadores condicionais implementados neste trabalho. Este trabalho pode ser expandido para abranger *multirange* e a criação dos novos índices.

A infraestrutura desta extensão não está limitada ao PostgreSQL, e muito menos ao tipo de dado *range*. Ela pode servir de base para futuras implementações de comportamentos exclusivos do PostgreSQL e qualquer outro SGBD incluído nos modos de compatibilidade do H2.

Por fim, este trabalho descreve em detalhes a representação de tipos de dados no H2. Ele pode servir de objeto de estudo para a implementação de UDT no H2, uma demanda dentro do padrão SQL.

REFERÊNCIAS

ELMASRI, R.; NAVATHE, S. B. **Sistemas de Banco de Dados**. 6ª Edição. São Paulo: Pearson Addison Wesley, 2011.

GLOBAL MARKET INSIGHTS. **Software Testing Market Size By Component (Application Testing, [By Type (Functional and Non-Functional)], Services [Professional and Managed], By Application (IT & Telecom, BFSI, Manufacturing, Retail, Healthcare, Transportation & Logistics, Government), Industry Analysis Report, Regional Outlook, Application Potential, Competitive Market Share & Forecast, 2020 – 2026**. 2019. Disponível em: <https://www.gminsights.com/industry-analysis/software-testing-market>. Acesso em: 20 abr. 2021.

GRANDIN, N. **PostgreSQL range type and functions**. Grupo H2 Database, 2021. Disponível em: <https://groups.google.com/g/h2-database/c/HP396nCkrSA/m/ukyCHa0mBgAJ?pli=1>. Acesso em: 8 fev. 2024.

H2 DATABASE GROUP. **H2 Database**. 2022. Disponível em: <https://www.h2database.com/>. Acesso em: 16 fev. 2022.

HAUER, P. **Don't use In-Memory Databases for Tests**. Philipp Hauer's Blog, 2017. Disponível em: <https://phauer.com/2017/dont-use-in-memory-databases-tests-h2/>. Acesso em: 20 abr. 2021.

LARMAN, C.; BASILI, V. **Iterative and incremental developments: a brief history**. IEEE Computer, v. 36, p. 47-56, 2003.

PARIS TECHNOLOGIES. **In-Memory Database**. 2020. Disponível em: <https://paristech.com/blog/in-memory-data-advantages/>. Acesso em: 20 abr. 2021.

POSTGRES GLOBAL DEVELOPMENT GROUP. **PostgreSQL**. 2022. Disponível em: <https://www.postgresql.org/>. Acesso em: 16 fev. 2022.

SOMMERVILLE, I. **Software Engineering**. 9ª Edição. São Paulo: Pearson Addison Wesley, 2011.

APÊNDICE A – Código Principal

h2/src/main/org/h2/command/Parser.java

	...
	//Classe
389	public class Parser {
	...
	// Atributos estáticos
566	/**
567	* A condition-level operation token known by the compatibility parser.
568	*/
569	private static final int CUSTOM_CONDITION_OP = NOT_TILDE + 1;
570	
571	/**
572	* A sum-level operation token known by the compatibility parser.
573	*/
574	private static final int CUSTOM_SUM_OP = CUSTOM_CONDITION_OP + 1;
575	
576	/**
577	* A factor-level operation token known by the compatibility parser.
578	*/
579	private static final int CUSTOM_FACTOR_OP = CUSTOM_SUM_OP + 1;
	...
	// Atributos do objeto
844	private final CompatibilityParser compatibilityParser;
	...
	// Construtor com sessão
934	/**
935	* Creates a new instance of parser.
936	*
937	* @param session the session
938	*/
939	public Parser(SessionLocal session) {
	...
946	this.session = session;
947	this.compatibilityParser = database.getMode().compatibilityParser.apply(session,
948	this);
	}
	...
	// Construtor sem sessão
950	/**
951	* Creates a new instance of parser for special use cases.
952	*/
953	public Parser() {
	...
959	session = null;
960	this.compatibilityParser = new NeutralCompatibilityParser(null, this);
961	}
	...
	//Método de leitura de condição
3474	private Expression readConditionRightHandSide(Expression r, boolean not, boolean
	whenOperand) {
	...
3514	if (compatibilityParser.shouldHandleConditionBinaryOperation(r, currentToken)) {
3515	String operator = currentToken;
3516	read();
3517	r = compatibilityParser.handleBinaryOperation(operator, r, readConcat());
3518	}

```

...
3528     Return r;
3529 }
...
// Método de leitura de soma
3710 private Expression readSum() {
3711     Expression r = readFactor();
3712     while (true) {
3713         if (compatibilityParser.shouldHandleSumBinaryOperation(r, currentToken)) {
3714             String operation = currentToken;
3715             read();
3716             r = compatibilityParser.handleBinaryOperation(operation, r, readFactor());
3717         } else {
3718             if (readIf(PLUS_SIGN)) {
3719                 r = new BinaryOperation(OpType.PLUS, r, readFactor());
3720             } else if (readIf(MINUS_SIGN)) {
3721                 r = new BinaryOperation(OpType.MINUS, r, readFactor());
3722             } else {
3723                 return r;
3724             }
3725         }
3726     }
3727 }
...
// Método de leitura de multiplicação
3729 private Expression readFactor() {
3730     Expression r = readTerm();
3731     while (true) {
3732         if (compatibilityParser.shouldHandleFactorBinaryOperation(r, currentToken)) {
3733             String operation = currentToken;
3734             read();
3735             r = compatibilityParser.handleBinaryOperation(operation, r, readTerm());
3736         } else {
3737             if (readIf(ASTERISK)) {
3738                 r = new BinaryOperation(OpType.MULTIPLY, r, readTerm());
3739             } else if (readIf(SLASH)) {
3740                 r = new BinaryOperation(OpType.DIVIDE, r, readTerm());
3741             } else if (readIf(PERCENT)) {
3742                 r = new MathFunction(r, readTerm(), MathFunction.MOD);
3743             } else {
3744                 return r;
3745             }
3746         }
3747     }
3748 }
...
// Método de leitura de função
4140 private Expression readFunction(Schema schema, String name) {
...
4152     Expression e;
4153     AggregateType agg = Aggregate.getAggregateType(upperName);
4154     if (agg != null) {
4155         return readAggregate(agg, upperName);
4156     }
4157     e = readCompatibilityFunction(upperName);
4158     if (e != null) {
4159         return e;
4160     }
4161     e = readBuiltinFunctionIf(upperName);

```

```

4162     if (e != null) {
4163         return e;
4164     }
4165     e = readWindowFunction(upperName);
4166     if (e != null) {
4167         return e;
4168     }
    ...
7174 }
...
// Método de leitura de função de compatibilidade
4193 private Expression readCompatibilityFunction(String name) {
4194     if (compatibilityParser != null) {
4195         Expression e = compatibilityParser.readFunction(name);
4196         if (e != null) {
4197             return e;
4198         }
4199     }
    ...
4355 }
...
// Método de leitura de termo isolado (folha)
6182 private void read() {
    ...
6187     int[] types = characterTypes;
6188     lastParseIndex = parseIndex;
6189     int i = parseIndex;
6190     int type;
6191     while ((type = types[i]) == 0) {
6192         i++;
6193     }
6194     int start = i;
6195     char[] chars = sqlCommandChars;
6196     char c = chars[i++];
6197     currentToken = "";
6198     switch (type) {
    ...
6278     case CHAR_SPECIAL_2:
6279         String token = "" + c;
6280         while(types[i] == CHAR_SPECIAL_2) {
6281             if (token.endsWith("=") && chars[i] == '-') {
6282                 break;
6283             }
6284             token += chars[i++];
6285         }
6286         if (token.length() > 1) {
6287             currentTokenType = getSpecialType2(token);
6288         } else {
6289             currentTokenType = getSpecialType1(c);
6290         }
6291         currentToken = token;
6292         parseIndex = i;
6293         return;
6294     case CHAR_SPECIAL_1:
6295         currentTokenType = getSpecialType1(c);
6296         currentToken = "" + c;
6297         parseIndex = i;
6298         return;
    ...

```

```

6616     }
6618     ...
6618     }
6620     // Método de classificação de caracteres
6620     private void initialize(String sql) {
6624         ...
6624         originalSQL = sql;
6625         sqlCommand = sql;
6626         int len = sql.length() + 1;
6627         char[] command = new char[len];
6628         int[] types = new int[len];
6635         ...
6635         for (int i = 0; i < len; i++) {
6636             char c = command[i];
6637             int type = 0;
6638             switch (c) {
6676                 case '-':
6677                     if (command[i + 1] == '-') {
6678                         // single line comment
6685                         ...
6685                         } else {
6686                             type = CHAR_SPECIAL_2;
6687                         }
6688                         break;
6735                 case '@':
6736                     type = CHAR_SPECIAL_2;
6737                     break;
6842                 ...
6842                 }
6843                 types[i] = type;
6845             ...
6845             }
6853         ...
6853     }
6912     // Método de tradução de caractere especial tipo 2 para operador
6912     private int getSpecialType2(String input) {
6913         switch (input) {
6914             case "&&":
6915                 return SPATIAL_INTERSECTS;
6916             case "::":
6917                 return COLON_COLON;
6918             case "!=":
6919                 return COLON_EQ;
6920             case "<=":
6921                 return SMALLER_EQUAL;
6922             case ">=":
6923                 return BIGGER_EQUAL;
6924             case "<>":
6925                 return NOT_EQUAL;
6926             case "!=":
6927                 return NOT_EQUAL;
6928             case "!~":
6929                 return NOT_TILDE;
6930             case "||":
6931                 return CONCATENATION;

```


6932	default:
6933	if (database.getMode().customConditionOperators.contains(input)) {
6934	return CUSTOM_CONDITION_OP;
6935	} else if (database.getMode().customSumOperators.contains(input)) {
6936	return CUSTOM_SUM_OP;
6937	} else if (database.getMode().customFactorOperators.contains(input)) {
6938	return CUSTOM_FACTOR_OP;
6939	} else {
6940	throw getSyntaxError();
6941	}
6942	}
6943	}
	...
	// Método de leitura de tipo de dado
7138	private TypeInfo readIfDataType1() {
	...
7277	Mode mode = database.getMode();
7278	DataType dataType = DataType.getTypeByName(original, mode);
	...
7282	long precision;
7283	int scale;
7284	if (dataType.specialPrecisionScale) {
7285	precision = dataType.defaultPrecision;
7286	scale = dataType.defaultScale;
7287	} else {
7288	precision = -1L;
7289	scale = -1;
7290	}
7291	int t = dataType.type;
	...
7353	return TypeInfo.getTypeInfo(t, precision, scale, dataType.extTypeInfo);
7354	}
	...
10753	}

h2/src/main/org/h2/compatibility/CompatibilityParser.java

1	package org.h2.compatibility;
2	
3	import org.h2.expression.Expression;
4	
5	public interface CompatibilityParser {
6	
7	Expression readFunction(String name);
8	
9	boolean shouldHandleConditionBinaryOperation(Expression expression, String operation);
10	
11	boolean shouldHandleSumBinaryOperation(Expression expression, String operation);
12	
13	boolean shouldHandleFactorBinaryOperation(Expression expression, String operation);
14	
15	Expression handleBinaryOperation(String token, Expression left, Expression right);
16	}

h2/src/main/org/h2/compatibility/DataConverter.java

1	package org.h2.compatibility;
---	-------------------------------

2 3 4 5 6 7 8 9	<pre> import org.h2.value.TypeInfo; import org.h2.value.Value; public interface DataConverter { Value convert(Value originalValue, TypeInfo targetType); } </pre>
--------------------------------------	---

h2/src/main/org/h2/compatibility/NeutralCompatibilityParser.java

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41	<pre> package org.h2.compatibility; import org.h2.command.Parser; import org.h2.engine.SessionLocal; import org.h2.expression.Expression; public class NeutralCompatibilityParser implements CompatibilityParser { private SessionLocal session; private Parser parser; public NeutralCompatibilityParser(SessionLocal session, Parser parser) { this.session = session; this.parser = parser; } @Override public Expression readFunction(String name) { return null; } @Override public boolean shouldHandleConditionBinaryOperation(Expression expression, String operation) { return false; } @Override public boolean shouldHandleSumBinaryOperation(Expression expression, String operation) { return false; } @Override public boolean shouldHandleFactorBinaryOperation(Expression expression, String operation) { return false; } @Override public Expression handleBinaryOperation(String token, Expression left, Expression right) { return null; } } </pre>
---	---

h2/src/main/org/h2/compatibility/NeutralDataConverter.java

```

1 package org.h2.compatibility;
2
3 import org.h2.value.TypeInfo;
4 import org.h2.value.Value;
5
6 public class NeutralDataConverter implements DataConverter {
7     @Override
8     public Value convert(Value originalValue, TypeInfo targetType) {
9         return null;
10    }
11 }

```

h2/src/main/org/h2/compatibility/postgresql/DateFunction2.java

```

1 package org.h2.compatibility.postgresql;
2
3 import org.h2.engine.SessionLocal;
4 import org.h2.expression.Expression;
5 import org.h2.expression.function.Function2;
6 import org.h2.message.DbException;
7 import org.h2.value.*;
8
9 public class DateFunction2 extends Function2 {
10
11     public static final int DATE_PART = 0;
12
13     private static final String[] NAMES = { "DATE_PART" };
14
15     private final int function;
16
17     public DateFunction2(Expression arg1, Expression arg2, int function) {
18         super(arg1, arg2);
19         this.function = function;
20     }
21
22     public int getFunction() {
23         return function;
24     }
25
26     @Override
27     public Value getValue(SessionLocal session) {
28         ValueVarchar v1 = (ValueVarchar) left.getValue(session);
29         ValueInterval v2 = (ValueInterval) right.getValue(session);
30
31         switch(function) {
32             case DATE_PART:
33                 String part = v1.getString();
34                 long time;
35                 switch(part) {
36                     case "hour":
37                         time = v2.getInterval().getHours();
38                         return ValueDouble.get(time);
39                     case "day":
40                         time = v2.getInterval().getDays();
41                         return ValueDouble.get(time);
42                     default:
43                         throw DbException.getInternalError("date_part called with not implemented
time part.");

```

```

44     }
45     default:
46         throw DbException.getInternalError("function=" + function);
47     }
48 }
49 }
50
51 @Override
52 public Expression optimize(SessionLocal session) {
53     left = left.optimize(session);
54     right = right.optimize(session);
55     type = TypeInfo.TYPE_DOUBLE;
56     return this;
57 }
58
59 @Override
60 public String getName() {
61     return NAMES[function];
62 }
63 }

```

h2/src/main/org/h2/compatibility/postgresql/ExtTypeInfoRange.java

```

1 package org.h2.compatibility.postgresql;
2
3 import org.h2.message.DbException;
4 import org.h2.value.ExtTypeInfo;
5 import org.h2.value.TypeInfo;
6 import org.h2.value.Value;
7
8 public class ExtTypeInfoRange extends ExtTypeInfo {
9
10     public static final ExtTypeInfoRange INT4RANGE = new
11     ExtTypeInfoRange(TypeInfo.TYPE_INTEGER);
12     public static final ExtTypeInfoRange INT8RANGE = new
13     ExtTypeInfoRange(TypeInfo.TYPE_BIGINT);
14     public static final ExtTypeInfoRange NUMRANGE = new
15     ExtTypeInfoRange(TypeInfo.TYPE_NUMERIC_FLOATING_POINT);
16     public static final ExtTypeInfoRange TSRANGE = new
17     ExtTypeInfoRange(TypeInfo.TYPE_TIMESTAMP);
18     public static final ExtTypeInfoRange TSTZRANGE = new
19     ExtTypeInfoRange(TypeInfo.TYPE_TIMESTAMP_TZ);
20     public static final ExtTypeInfoRange DATERANGE = new
21     ExtTypeInfoRange(TypeInfo.TYPE_DATE);
22
23     private TypeInfo subtype;
24
25     private ExtTypeInfoRange(TypeInfo _subtype) {
26         subtype = _subtype;
27     }
28
29     public static ExtTypeInfoRange getExtTypeBySubtype(TypeInfo subtype) {
30         switch (subtype.getValueType()) {
31             case Value.INTEGER:
32                 return INT4RANGE;
33             case Value.BIGINT:
34                 return INT8RANGE;
35             case Value.NUMERIC:
36                 return NUMRANGE;

```

```

31         case Value.TIMESTAMP:
32             return TSRANGE;
33         case Value.TIMESTAMP_TZ:
34             return TSTZRANGE;
35         case Value.DATE:
36             return DATERANGE;
37         default:
38             return null;
39     }
40 }
41
42 public TypeInfo getSubtype() {
43     return subtype;
44 }
45
46 @Override
47 public boolean overrideSQL() {
48     return true;
49 }
50
51 @Override
52 public StringBuilder getSQL(StringBuilder builder, int sqlFlags) {
53     String appended = "";
54     switch (subtype.getValueType()) {
55         case Value.INTEGER:
56             appended = "INT4RANGE";
57             break;
58         case Value.BIGINT:
59             appended = "INT8RANGE";
60             break;
61         case Value.NUMERIC:
62             appended = "NUMRANGE";
63             break;
64         case Value.TIMESTAMP:
65             appended = "TSRANGE";
66             break;
67         case Value.TIMESTAMP_TZ:
68             appended = "TSTZRANGE";
69             break;
70         case Value.DATE:
71             appended = "DATERANGE";
72             break;
73         default:
74             throw DbException.getInternalError("Unkown range subtype: " + subtype);
75     }
76     return builder.append(appended);
77 }
78
79 }

```

h2/src/main/org/h2/compatibility/postgresql/PostgreSQLDataConverter.java

```

1 package org.h2.compatibility.postgresql;
2
3 import org.h2.command.Parser;
4 import org.h2.compatibility.DataConverter;
5 import org.h2.expression.Expression;
6 import org.h2.util.StringUtils;
7 import org.h2.value.*;

```

```

8
9 import java.util.ArrayList;
10 import java.util.List;
11
12 public class PostgreSQLDataConverter implements DataConverter {
13
14     private List<ExtTypeInfoRange> canonicalTypes;
15     private CompareMode compareMode;
16
17     public PostgreSQLDataConverter() {
18         canonicalTypes = new ArrayList<>();
19         canonicalTypes.add(ExtTypeInfoRange.INT4RANGE);
20         canonicalTypes.add(ExtTypeInfoRange.INT8RANGE);
21         canonicalTypes.add(ExtTypeInfoRange.DATERANGE);
22         compareMode = CompareMode.getInstance(null, 0);
23     }
24
25     @Override
26     public Value convert(Value originalValue, TypeInfo targetType) {
27         String raw = originalValue.getString();
28         if (originalValue.getValueType() != Value.VARCHAR || targetType.getValueType() !=
Value.VARCHAR || raw.length() < 3) {
29             return null;
30         }
31
32         if ("empty".equals(raw)) {
33             return RangeValue.EMPTY;
34         }
35         char prefix = raw.charAt(0);
36         char suffix = raw.charAt(raw.length() - 1);
37         String[] values = raw.substring(1, raw.length() - 1).split(",");
38         boolean hasPrefix = prefix == '[' || prefix == '(';
39         boolean hasSuffix = suffix == ']' || suffix == ')';
40         boolean hasValues = values.length == 1 || values.length == 2;
41         if (hasPrefix && hasSuffix && hasValues && raw.length() > 3) {
42             String lowerBound = "";
43             String upperBound = "";
44             if (values.length == 2) {
45                 lowerBound = values[0];
46                 upperBound = values[1];
47                 if (raw.indexOf(',') == 1) {
48                     if (prefix != '(') {
49                         return null;
50                     }
51                 }
52             } else {
53                 if (raw.indexOf(',') == raw.length() - 2) {
54                     if (suffix != ')') {
55                         return null;
56                     }
57                     lowerBound = values[0];
58                 }
59             }
60             int dateLowerBound = isDate(lowerBound);
61             int dateUpperBound = isDate(upperBound);
62             if (dateLowerBound != -1) {
63                 lowerBound = "" + lowerBound + "";
64             }
65             if (dateUpperBound != -1) {

```

```

66         upperBound = "" + upperBound + "";
67     }
68     Parser p = new Parser();
69     Value lowerValue = StringUtils.isEmpty(lowerBound) ? ValueNull.INSTANCE :
p.parseExpression(lowerBound).getValue(null);
70     Value upperValue = StringUtils.isEmpty(upperBound) ? ValueNull.INSTANCE
: p.parseExpression(upperBound).getValue(null);
71     lowerValue = convertToDate(lowerValue, dateLowerBound);
72     upperValue = convertToDate(upperValue, dateUpperBound);
73     RangeValue value = new RangeValue(lowerValue, upperValue, prefix == '[', suffix ==
']', targetType);
74     if (lowerValue != ValueNull.INSTANCE && upperValue != ValueNull.INSTANCE &&
compareMode.compare(lowerValue, upperValue) > 0) {
75         throw new IllegalArgumentException("range lower bound must be less than or
equal to range upper bound");
76     }
77     if (canonicalTypes.contains(targetType.getExtTypeInfo())) {
78         value = RangeTypeUtil.canonicize(value);
79     }
80     return value;
81 } else if (hasPrefix && hasSuffix && values.length == 0) {
82     return new RangeValue(ValueNull.INSTANCE, ValueNull.INSTANCE, targetType);
83 }
84 return null;
85 }
86
87 public RangeValue convertTypelessToRangeValue(Value originalValue) {
88     if (originalValue.getValueType() != Value.VARCHAR) {
89         return null;
90     }
91
92     String raw = originalValue.getString();
93     if ("empty".equals(raw)) {
94         return RangeValue.EMPTY;
95     }
96     char prefix = raw.charAt(0);
97     char suffix = raw.charAt(raw.length() - 1);
98     String[] values = raw.substring(1, raw.length() - 1).split(",");
99     boolean hasPrefix = prefix == '[' || prefix == '(';
100    boolean hasSuffix = suffix == ']' || suffix == ')';
101    boolean hasValues = values.length == 1 || values.length == 2;
102    if (hasPrefix && hasSuffix && hasValues && raw.length() > 3) {
103        String lowerBound = "";
104        String upperBound = "";
105        if (values.length == 2) {
106            lowerBound = values[0];
107            upperBound = values[1];
108            if (raw.indexOf(',') == 1) {
109                if (prefix != '(') {
110                    return null;
111                }
112            }
113        } else {
114            if (raw.indexOf(',') == raw.length() - 2) {
115                if (suffix != ')') {
116                    return null;
117                }
118                lowerBound = values[0];
119            }

```

```

120     }
121     int dateLowerBound = isDate(lowerBound);
122     int dateUpperBound = isDate(upperBound);
123     if (dateLowerBound != -1) {
124         lowerBound = "" + lowerBound + "";
125     }
126     if (dateUpperBound != -1) {
127         upperBound = "" + upperBound + "";
128     }
129     Parser p = new Parser();
130     TypeInfo targetType = null;
131     Value lowerValue;
132     Value upperValue;
133     if (StringUtils.isEmpty(lowerBound)) {
134         lowerValue = ValueNull.INSTANCE;
135     } else {
136         Expression e = p.parseExpression(lowerBound);
137         lowerValue = e.getValue(null);
138         TypeInfo lowerType = e.getType();
139         if (lowerType != TypeInfo.TYPE_NULL) {
140             targetType = lowerType;
141         }
142     }
143     if (StringUtils.isEmpty(upperBound)) {
144         upperValue = ValueNull.INSTANCE;
145     } else {
146         Expression e = p.parseExpression(upperBound);
147         upperValue = e.getValue(null);
148         TypeInfo upperType = e.getType();
149         if (upperType != TypeInfo.TYPE_NULL) {
150             targetType = upperType;
151         }
152     }
153     lowerValue = convertToDate(lowerValue, dateLowerBound);
154     upperValue = convertToDate(upperValue, dateUpperBound);
155     RangeValue value = new RangeValue(lowerValue, upperValue, prefix == '[', suffix ==
156     ']', targetType);
157     if (lowerValue != ValueNull.INSTANCE && upperValue != ValueNull.INSTANCE &&
158     compareMode.compare(lowerValue, upperValue) > 0) {
159         throw new IllegalArgumentException("range lower bound must be less than or
160     equal to range upper bound");
161     }
162     if (targetType != null && canonicalTypes.contains(targetType.getExtTypeInfo())) {
163         value = RangeTypeUtil.canonize(value);
164     }
165     return value;
166 } else if (hasPrefix && hasSuffix && values.length == 0) {
167     return new RangeValue(ValueNull.INSTANCE, ValueNull.INSTANCE);
168 }
169 return null;
170 }
171 private int isDate(String value) {
172     String[] splitSpace = value.split(" ");
173     String[] splitDate = splitSpace[0].split("-");
174     boolean isDate = splitDate.length > 2 && splitDate[0].length() == 4 &&
175     splitDate[1].length() == 2 && splitDate[2].length() == 2;
176     if (isDate) {
177         if (splitSpace.length > 1) {

```



```

175         if (splitSpace.length > 2 || splitSpace[1].contains("+") || splitSpace[1].contains("-"))
176     {
177         return 2;
178     } else {
179         return 1;
180     }
181     return 0;
182 }
183 return -1;
184 }
185
186 private Value convertToDate(Value value, int dateCase) {
187     switch (dateCase) {
188     case 2:
189         return value.convertTo(TypeInfo.TYPE_TIMESTAMP_TZ);
190     case 1:
191         return value.convertTo(TypeInfo.TYPE_TIMESTAMP);
192     case 0:
193         return value.convertTo(TypeInfo.TYPE_DATE);
194     default:
195         return value;
196     }
197 }
198 }

```

h2/src/main/org/h2/compatibility/postgresql/PostgreSQLParser.java

```

1 package org.h2.compatibility.postgresql;
2
3 import org.h2.command.Parser;
4 import org.h2.compatibility.CompatibilityParser;
5 import org.h2.engine.Constants;
6 import org.h2.engine.SessionLocal;
7 import org.h2.expression.Expression;
8 import org.h2.value.TypeInfo;
9 import org.h2.value.Value;
10
11 public class PostgreSQLParser implements CompatibilityParser {
12
13     private SessionLocal session;
14     private Parser parser;
15
16     public PostgreSQLParser(SessionLocal session, Parser parser) {
17         this.session = session;
18         this.parser = parser;
19     }
20
21     public Expression readFunction(String name) {
22         switch(name) {
23         case "INT4RANGE":
24             return new RangeFunction2(parser.readExpression(), parser.readNextArgument(),
25             parser.readIfArgument(), RangeFunction2.INT4RANGE);
26         case "INT8RANGE":
27             return new RangeFunction2(parser.readExpression(), parser.readNextArgument(),
28             parser.readIfArgument(), RangeFunction2.INT8RANGE);
29         case "NUMRANGE":
30             return new RangeFunction2(parser.readExpression(), parser.readNextArgument(),
31             parser.readIfArgument(), RangeFunction2.NUMRANGE);

```

```

29         case "DATERANGE":
30             return new RangeFunction2(parser.readExpression(), parser.readNextArgument(),
parser.readIfArgument(), RangeFunction2.DATERANGE);
31         case "TSRANGE":
32             return new RangeFunction2(parser.readExpression(), parser.readNextArgument(),
parser.readIfArgument(), RangeFunction2.TSRANGE);
33         case "TSTZRANGE":
34             return new RangeFunction2(parser.readExpression(), parser.readNextArgument(),
parser.readIfArgument(), RangeFunction2.TSTZRANGE);
35         case "LOWER":
36             return new RangeFunction1(parser.readSingleArgument(),
RangeFunction1.LOWER);
37         case "UPPER":
38             return new RangeFunction1(parser.readSingleArgument(),
RangeFunction1.UPPER);
39         case "ISEMPTY":
40             return new RangeFunction1(parser.readSingleArgument(),
RangeFunction1.ISEMPTY);
41         case "LOWER_INC":
42             return new RangeFunction1(parser.readSingleArgument(),
RangeFunction1.LOWER_INC);
43         case "UPPER_INC":
44             return new RangeFunction1(parser.readSingleArgument(),
RangeFunction1.UPPER_INC);
45         case "LOWER_INF":
46             return new RangeFunction1(parser.readSingleArgument(),
RangeFunction1.LOWER_INF);
47         case "UPPER_INF":
48             return new RangeFunction1(parser.readSingleArgument(),
RangeFunction1.UPPER_INF);
49         case "RANGE_MERGE":
50             return new RangeFunction2(parser.readExpression(), parser.readLastArgument(),
null, RangeFunction2.RANGE_MERGE);
51         case "DATE_PART":
52             return new DateFunction2(parser.readExpression(), parser.readLastArgument(),
DateFunction2.DATE_PART);
53         default:
54             return null;
55     }
56 }
57
58 private boolean isRangeValue(Expression expression) {
59     return expression instanceof RangeExpression || expression.getType() == null ||
expression.getType() == TypeInfo.TYPE_UNKNOWN || (
60         expression.getType().getValueType() == Value.VARCHAR &&
expression.getType().getPrecision() == Constants.MAX_STRING_LENGTH
61     );
62 }
63
64 public boolean shouldHandleConditionBinaryOperation(Expression expression, String
operation) {
65     boolean knownOp = session.getMode().customConditionOperators.contains(operation);
66     return knownOp && (operation.equals("<@" ) || isRangeValue(expression));
67 }
68
69 public boolean shouldHandleSumBinaryOperation(Expression expression, String
operation) {
70     boolean knownOp = session.getMode().customSumOperators.contains(operation);
71     return knownOp && isRangeValue(expression);

```

72	}
73	
74	public boolean shouldHandleFactorBinaryOperation(Expression expression, String operation) {
75	boolean knownOp = session.getMode().customFactorOperators.contains(operation);
76	return knownOp && isRangeValue(expression);
77	}
78	
79	public Expression handleBinaryOperation(String token, Expression left, Expression right) {
80	return new RangeOperation2(token, left, right);
81	}
82	
83	}

h2/src/main/org/h2/compatibility/postgresql/RangeExpression.java

1	package org.h2.compatibility.postgresql;
2	
3	public interface RangeExpression {
4	}

h2/src/main/org/h2/compatibility/postgresql/RangeFunction1.java

1	package org.h2.compatibility.postgresql;
2	
3	import org.h2.engine.SessionLocal;
4	import org.h2.expression.Expression;
5	import org.h2.expression.function.Function1;
6	import org.h2.expression.function.StringFunction1;
7	import org.h2.message.DbException;
8	import org.h2.value.*;
9	
10	public class RangeFunction1 extends Function1 {
11	
12	public static final int LOWER = 0;
13	
14	public static final int UPPER = LOWER + 1;
15	
16	public static final int ISEMPY = UPPER + 1;
17	
18	public static final int LOWER_INC = ISEMPY + 1;
19	
20	public static final int UPPER_INC = LOWER_INC + 1;
21	
22	public static final int LOWER_INF = UPPER_INC + 1;
23	
24	public static final int UPPER_INF = LOWER_INF + 1;
25	
26	private static final String[] NAMES = {
27	"LOWER", "UPPER", "ISEMPY", "LOWER_INC", "UPPER_INC", "LOWER_INF",
28	"UPPER_INF"
29	};
30	private final int function;
31	
32	private Expression correction;
33	
34	public RangeFunction1(Expression arg1, int function) {

```

35     super(arg1);
36     this.function = function;
37 }
38
39 @Override
40 public Value getValue(SessionLocal session) {
41     if (correction != null) {
42         return correction.getValue(session);
43     }
44     RangeValue v = RangeTypeUtil.castRangeValue(arg.getValue(session));
45     switch (function) {
46         case LOWER:
47             return v.getLowerBound();
48         case UPPER:
49             return v.getUpperBound();
50         case ISEMPTY:
51             return new ValueBoolean(v.isEmpty());
52         case LOWER_INC:
53             return new ValueBoolean(v.isLowerInclusive());
54         case UPPER_INC:
55             return new ValueBoolean(v.isUpperInclusive());
56         case LOWER_INF:
57             return v == RangeValue.EMPTY ? new ValueBoolean(false) : new
ValueBoolean(v.getLowerBound() == ValueNull.INSTANCE);
58         case UPPER_INF:
59             return v == RangeValue.EMPTY ? new ValueBoolean(false) : new
ValueBoolean(v.getUpperBound() == ValueNull.INSTANCE);
60         default:
61             throw DbException.getInternalError("function=" + function);
62     }
63 }
64
65 @Override
66 public Expression optimize(SessionLocal session) {
67     arg = arg.optimize(session);
68     boolean isLower = function == LOWER;
69     boolean isUpper = function == UPPER;
70     if (isLower || isUpper) {
71         ExtTypeInfo extType = arg.getType().getExtTypeInfo();
72         if (extType instanceof ExtTypeInfoRange)
73             type = ((ExtTypeInfoRange) extType).getSubtype();
74         else {
75             correction = new StringFunction1(arg, isLower ? StringFunction1.LOWER :
StringFunction1.UPPER);
76             correction.optimize(session);
77             return correction;
78         }
79     } else {
80         type = TypeInfo.TYPE_BOOLEAN;
81     }
82     return this;
83 }
84
85 @Override
86 public String getName() {
87     return NAMES[function];
88 }
89 }

```

h2/src/main/org/h2/compatibility/postgresql/RangeFunction2.java

```

1 package org.h2.compatibility.postgresql;
2
3 import org.h2.engine.SessionLocal;
4 import org.h2.expression.Expression;
5 import org.h2.expression.function.Function2;
6 import org.h2.message.DbException;
7 import org.h2.util.DateUtils;
8 import org.h2.value.*;
9
10 public class RangeFunction2 extends Function2 implements RangeExpression {
11
12     public static final int INT4RANGE = 0;
13
14     public static final int INT8RANGE = INT4RANGE + 1;
15
16     public static final int NUMRANGE = INT8RANGE + 1;
17
18     public static final int TSRANGE = NUMRANGE + 1;
19
20     public static final int TSTZRANGE = TSRANGE + 1;
21
22     public static final int DATERANGE = TSTZRANGE + 1;
23
24     public static final int RANGE_MERGE = DATERANGE + 1;
25
26     private static final String[] NAMES = {
27         "INT4RANGE", "INT8RANGE", "NUMRANGE", "TSRANGE", "TSTZRANGE",
28         "DATERANGE", "RANGE_MERGE"
29     };
30
31     private final int function;
32     private final Expression boundFlags;
33
34     public RangeFunction2(Expression arg1, Expression arg2, Expression arg3, int function) {
35         super(arg1, arg2);
36         this.boundFlags = arg3;
37         this.function = function;
38     }
39
40     @Override
41     public Value getValue(SessionLocal session) {
42         Value v1 = left.getValue(session);
43         Value v2 = right.getValue(session);
44         boolean lowerInclusive = true;
45         boolean upperInclusive = false;
46         if (boundFlags != null) {
47             String bounds = boundFlags.getValue(session).getString();
48             switch (bounds) {
49                 case "[]":
50                     lowerInclusive = true;
51                     upperInclusive = false;
52                     break;
53                 case "[)":
54                     lowerInclusive = true;
55                     upperInclusive = true;
56                     break;
57                 case "()":
58                     lowerInclusive = false;

```

```

58         upperInclusive = false;
59         break;
60     case "[)":
61         lowerInclusive = false;
62         upperInclusive = true;
63         break;
64     default:
65         throw new IllegalArgumentException("invalid range bound flags Hint: Valid
values are \"[]\", \"()\", \"[)\", and \"()\"");
66     }
67 }
68 RangeValue rangeValue;
69 switch (function) {
70     case DATERANGE:
71         if (v1 != ValueNull.INSTANCE) {
72             String v1String = v1.getString();
73             v1 = ValueDate.fromDateValue(DateTimeUtils.parseDateValue(v1String, 0,
v1String.length()));
74         }
75         if (v2 != ValueNull.INSTANCE) {
76             String v2String = v2.getString();
77             v2 = ValueDate.fromDateValue(DateTimeUtils.parseDateValue(v2String, 0,
v2String.length()));
78         }
79     case INT4RANGE:
80     case INT8RANGE:
81         rangeValue = RangeTypeUtil.canonicize(new RangeValue(v1, v2, lowerInclusive,
upperInclusive, type));
82         break;
83     case NUMRANGE:
84     case TSRANGE:
85     case TSTZRANGE:
86         rangeValue = new RangeValue(v1, v2, lowerInclusive, upperInclusive, type);
87         break;
88     case RANGE_MERGE:
89         RangeValue vr1 = RangeTypeUtil.castRangeValue(v1);
90         RangeValue vr2 = RangeTypeUtil.castRangeValue(v2);
91         int lowToLowComparison =
RangeTypeUtil.compareLowerToLowerBounds(session, vr1, vr2);
92         int upToUpComparison = RangeTypeUtil.compareUpperToUpperBounds(session,
vr1, vr2);
93         Value min;
94         Value max;
95         if (lowToLowComparison < 0) {
96             min = vr1.getLowerBound();
97             lowerInclusive = vr1.isLowerInclusive();
98         } else if (lowToLowComparison == 0) {
99             min = vr1.getLowerBound();
100            lowerInclusive = vr1.isLowerInclusive() || vr2.isLowerInclusive();
101        } else {
102            min = vr2.getLowerBound();
103            lowerInclusive = vr2.isLowerInclusive();
104        }
105        if (upToUpComparison > 0) {
106            max = vr1.getUpperBound();
107            upperInclusive = vr1.isUpperInclusive();
108        } else if (upToUpComparison == 0) {
109            max = vr1.getUpperBound();
110            upperInclusive = vr1.isUpperInclusive() || vr2.isUpperInclusive();

```

```

111         } else {
112             max = vr2.getUpperBound();
113             upperInclusive = vr2.isUpperInclusive();
114         }
115         return new RangeValue(min, max, lowerInclusive, upperInclusive, type);
116     default:
117         throw DbException.getInternalError("function=" + function);
118     }
119     if (RangeTypeUtil.compareLowerToUpperBounds(session, rangeValue, rangeValue) >
120 0) {
121         throw new IllegalArgumentException("range lower bound must be less than or equal
to range upper bound");
122     } else if (RangeTypeUtil.compareLowerToUpperBounds(session, rangeValue,
rangeValue) == 0
123         && rangeValue.isLowerInclusive() && !rangeValue.isUpperInclusive()) {
124         return RangeValue.EMPTY;
125     }
126     return rangeValue;
127 }
128 @Override
129 public Expression optimize(SessionLocal session) {
130     left = left.optimize(session);
131     right = right.optimize(session);
132     switch (function) {
133         case INT4RANGE:
134             type = TypeInfo.getTypeInfo(Value.VARCHAR, -1, -1,
ExtTypeInfoRange.INT4RANGE);
135             break;
136         case INT8RANGE:
137             type = TypeInfo.getTypeInfo(Value.VARCHAR, -1, -1,
ExtTypeInfoRange.INT8RANGE);
138             break;
139         case NUMRANGE:
140             type = TypeInfo.getTypeInfo(Value.VARCHAR, -1, -1,
ExtTypeInfoRange.NUMRANGE);
141             break;
142         case TSRANGE:
143             type = TypeInfo.getTypeInfo(Value.VARCHAR, -1, -1,
ExtTypeInfoRange.TSRANGE);
144             break;
145         case TSTZRANGE:
146             type = TypeInfo.getTypeInfo(Value.VARCHAR, -1, -1,
ExtTypeInfoRange.TSTZRANGE);
147             break;
148         case DATERANGE:
149             type = TypeInfo.getTypeInfo(Value.VARCHAR, -1, -1,
ExtTypeInfoRange.DATERANGE);
150             break;
151         default:
152             type = TypeInfo.TYPE_VARCHAR;
153     }
154     return this;
155 }
156
157 @Override
158 public String getName() {
159     return NAMES[function];
160 }

```

161	}
-----	---

h2/src/main/org/h2/compatibility/postgresql/RangeOperation2.java

```

1  package org.h2.compatibility.postgresql;
2
3  import org.h2.engine.SessionLocal;
4  import org.h2.expression.BinaryOperation;
5  import org.h2.expression.Expression;
6  import org.h2.expression.Operation2;
7  import org.h2.message.DbException;
8  import org.h2.value.TypeInfo;
9  import org.h2.value.Value;
10 import org.h2.value.ValueBoolean;
11
12 public class RangeOperation2 extends Operation2 implements RangeExpression {
13
14     public enum RangeOpType {
15         UNION, // +
16         DIFFERENCE, // -
17         INTERSECTION, // *
18         ADJACENT, // -|-
19         OVERLAPS, // &&
20         CONTAINS, // @>
21         CONTAINED_BY, // <@
22         STRICT_LEFT, // <<
23         STRICT_RIGHT, // >>
24         NOT_EXTEND_RIGHT, // &<
25         NOT_EXTEND_LEFT // &>
26     }
27
28     private RangeOpType opType;
29     private BinaryOperation correction;
30
31     public RangeOperation2(String opType, Expression left, Expression right) {
32         super(left, right);
33         this.opType = getOperationEnum(opType);
34     }
35
36     @Override
37     public Value getValue(SessionLocal session) {
38         if (correction != null) {
39             return correction.getValue(session);
40         }
41         Value rawLeft = left.getValue(session);
42         Value rawRight = right.getValue(session);
43         boolean isRangeLeft = RangeTypeUtil.isRangeValue(rawLeft);
44         boolean isRangeRight = RangeTypeUtil.isRangeValue(rawRight);
45         if (isRangeLeft && !isRangeRight) {
46             RangeValue leftValue = RangeTypeUtil.castRangeValue(rawLeft);
47             TypeInfo leftSubType = leftValue.getSubType();
48             TypeInfo rightType = rawRight.getType();
49             if (opType != RangeOpType.CONTAINS || !(leftSubType == TypeInfo.TYPE_NULL ||
50 leftSubType == rightType)) {
51                 throw DbException.getInternalError("operator does not exist: " +
52 leftValue.getNamedRangeType() + " " + getOperationToken() + " " + rightType);
53             }
54             // case range CONTAINS element
55             return new ValueBoolean(containsElement(session, leftValue, rawRight));

```



```

54     } else if(isRangeRight && !isRangeLeft) {
55         RangeValue rightValue = RangeTypeUtil.castRangeValue(rawRight);
56         TypeInfo rightSubType = rightValue.getSubType();
57         TypeInfo leftType = rawLeft.getType();
58         if (opType != RangeOpType.CONTAINED_BY || !(rightSubType ==
TypeInfo.TYPE_NULL || rightSubType == leftType)) {
59             throw DbException.getInternalError("operator does not exist: " + leftType + " " +
getOperationToken() + " " + rightValue.getNamedRangeType());
60         }
61         // case element CONTAINED_BY range
62         return new ValueBoolean(containsElement(session, rightValue, rawLeft));
63     } else {
64         RangeValue leftValue = RangeTypeUtil.castRangeValue(rawLeft);
65         RangeValue rightValue = RangeTypeUtil.castRangeValue(rawRight);
66         TypeInfo leftType = leftValue.getSubType();
67         TypeInfo rightType = rightValue.getSubType();
68         if (leftType != rightType && leftType != TypeInfo.TYPE_NULL && rightType !=
TypeInfo.TYPE_NULL) {
69             throw DbException.getInternalError("range types do not match");
70         }
71         switch(opType) {
72             case UNION:
73                 if (leftValue.isEmpty()) {
74                     return rightValue.getValue(session);
75                 } else if (rightValue.isEmpty()) {
76                     return leftValue.getValue(session);
77                 } else {
78                     Value leftLower = leftValue.getLowerBound();
79                     Value leftUpper = leftValue.getUpperBound();
80                     Value rightLower = rightValue.getLowerBound();
81                     Value rightUpper = rightValue.getUpperBound();
82                     if (!overlaps(session, leftValue, rightValue) && !adjacent(session, leftValue,
rightValue)) {
83                         throw DbException.getInternalError("result of range union would not be
contiguous");
84                     }
85                     int comparisonLeftRightLow =
RangeTypeUtil.compareLowerToLowerBounds(session, leftValue, rightValue);
86                     int comparisonLeftRightUp =
RangeTypeUtil.compareUpperToUpperBounds(session, leftValue, rightValue);
87                     Value min;
88                     Value max;
89                     boolean lowerInclusive;
90                     boolean upperInclusive;
91                     if (comparisonLeftRightLow < 0) {
92                         min = leftLower;
93                         lowerInclusive = leftValue.isLowerInclusive();
94                     } else if (comparisonLeftRightLow == 0) {
95                         min = leftLower;
96                         lowerInclusive = leftValue.isLowerInclusive() ||
rightValue.isLowerInclusive();
97                     } else {
98                         min = rightLower;
99                         lowerInclusive = rightValue.isLowerInclusive();
100                    }
101                    if (comparisonLeftRightUp > 0) {
102                        max = leftUpper;
103                        upperInclusive = leftValue.isUpperInclusive();
104                    } else if (comparisonLeftRightUp == 0) {

```

```

105         max = leftUpper;
106         upperInclusive = leftValue.isUpperInclusive() ||
rightValue.isUpperInclusive();
107     } else {
108         max = rightUpper;
109         upperInclusive = rightValue.isUpperInclusive();
110     }
111     return new RangeValue(min, max, lowerInclusive, upperInclusive);
112 }
113 case DIFFERENCE:
114     if (leftValue.isEmpty()) {
115         return RangeValue.EMPTY;
116     } else if (rightValue.isEmpty()) {
117         return left.getValue(session);
118     } else {
119         Value leftLower = leftValue.getLowerBound();
120         Value leftUpper = leftValue.getUpperBound();
121         Value rightLower = rightValue.getLowerBound();
122         Value rightUpper = rightValue.getUpperBound();
123         int comparisonLeftRightLow =
RangeTypeUtil.compareLowerToLowerBounds(session, leftValue, rightValue);
124         int comparisonLeftRightUp =
RangeTypeUtil.compareUpperToUpperBounds(session, leftValue, rightValue);
125         boolean edgedLeft = comparisonLeftRightLow == 0 &&
leftValue.isLowerInclusive() && rightValue.isLowerInclusive();
126         boolean edgedRight = comparisonLeftRightUp == 0 &&
leftValue.isUpperInclusive() && rightValue.isUpperInclusive();
127         if (containsRange(session, leftValue, rightValue) && !edgedLeft &&
!edgedRight) {
128             throw DbException.getInternalError("result of range difference would not
be contiguous");
129         }
130         if (containsRange(session, rightValue, leftValue)) {
131             return RangeValue.EMPTY;
132         }
133         if (!overlaps(session, leftValue, rightValue)) {
134             return leftValue;
135         }
136         Value min;
137         Value max;
138         boolean lowerInclusive;
139         boolean upperInclusive;
140         if (comparisonLeftRightLow < 0) {
141             // right cut
142             min = leftLower;
143             max = rightLower;
144             lowerInclusive = leftValue.isLowerInclusive();
145             upperInclusive = !rightValue.isLowerInclusive();
146         }
147         } else {
148             // left cut
149             min = rightUpper;
150             max = leftUpper;
151             lowerInclusive = !rightValue.isUpperInclusive();
152             upperInclusive = leftValue.isUpperInclusive();
153         }
154         return new RangeValue(min, max, lowerInclusive, upperInclusive);
155     }
156 case INTERSECTION:

```

```

157         if (leftValue.isEmpty() || rightValue.isEmpty() || !overlaps(session, leftValue,
rightValue)) {
158             return RangeValue.EMPTY;
159         } else {
160             Value leftLower = leftValue.getLowerBound();
161             Value leftUpper = leftValue.getUpperBound();
162             Value rightLower = rightValue.getLowerBound();
163             Value rightUpper = rightValue.getUpperBound();
164             Value min;
165             Value max;
166             boolean lowerInclusive;
167             boolean upperInclusive;
168             if (RangeTypeUtil.compareLowerToLowerBounds(session, leftValue,
rightValue) < 0) {
169                 min = rightLower;
170                 lowerInclusive = rightValue.isLowerInclusive();
171             } else {
172                 min = leftLower;
173                 lowerInclusive = leftValue.isLowerInclusive();
174             }
175             if (RangeTypeUtil.compareUpperToUpperBounds(session, leftValue,
rightValue) > 0) {
176                 max = rightUpper;
177                 upperInclusive = rightValue.isUpperInclusive();
178             } else {
179                 max = leftUpper;
180                 upperInclusive = leftValue.isUpperInclusive();
181             }
182             return new RangeValue(min, max, lowerInclusive, upperInclusive);
183         }
184     case ADJACENT:
185         if (leftValue.isEmpty() || rightValue.isEmpty()) {
186             return new ValueBoolean(false);
187         } else {
188             return new ValueBoolean(adjacent(session, leftValue, rightValue));
189         }
190     case OVERLAPS:
191         if (leftValue.isEmpty() || rightValue.isEmpty()) {
192             return new ValueBoolean(false);
193         } else {
194             return new ValueBoolean(overlaps(session, leftValue, rightValue));
195         }
196     case CONTAINS:
197         if (rightValue.isEmpty()) {
198             return new ValueBoolean(true);
199         } else if (leftValue.isEmpty()) {
200             return new ValueBoolean(false);
201         } else {
202             return new ValueBoolean(containsRange(session, leftValue, rightValue));
203         }
204     case CONTAINED_BY:
205         if (leftValue.isEmpty()) {
206             return new ValueBoolean(true);
207         } else if (rightValue.isEmpty()) {
208             return new ValueBoolean(false);
209         } else {
210             return new ValueBoolean(containsRange(session, rightValue, leftValue));
211         }
212     case STRICT_LEFT:

```

```

213         if (leftValue.isEmpty() || rightValue.isEmpty()) {
214             return new ValueBoolean(false);
215         } else {
216             int rightLowToLeftUpComparison =
RangeTypeUtil.compareLowerToUpperBounds(session, rightValue, leftValue);
217             if (rightLowToLeftUpComparison == 0) {
218                 return new ValueBoolean(!leftValue.isUpperInclusive() ||
!rightValue.isLowerInclusive());
219             }
220             return new ValueBoolean(rightLowToLeftUpComparison > 0);
221         }
222     case STRICT_RIGHT:
223         if (leftValue.isEmpty() || rightValue.isEmpty()) {
224             return new ValueBoolean(false);
225         } else {
226             int leftLowToRightUpComparison =
RangeTypeUtil.compareLowerToUpperBounds(session, leftValue, rightValue);
227             if (leftLowToRightUpComparison == 0) {
228                 return new ValueBoolean(!leftValue.isLowerInclusive() ||
!rightValue.isUpperInclusive());
229             }
230             return new ValueBoolean(leftLowToRightUpComparison > 0);
231         }
232     case NOT_EXTEND_RIGHT:
233         if (leftValue.isEmpty() || rightValue.isEmpty()) {
234             return new ValueBoolean(false);
235         } else {
236             return new ValueBoolean(notExtendRight(session, leftValue, rightValue));
237         }
238     case NOT_EXTEND_LEFT:
239         if (leftValue.isEmpty() || rightValue.isEmpty()) {
240             return new ValueBoolean(false);
241         } else {
242             return new ValueBoolean(notExtendLeft(session, leftValue, rightValue));
243         }
244     default:
245         return null;
246     }
247 }
248
249 }
250
251 private boolean adjacent(SessionLocal session, RangeValue leftValue, RangeValue
rightValue) {
252     int leftLowToRightUpComparison =
RangeTypeUtil.compareLowerToUpperBounds(session, leftValue, rightValue);
253     int rightLowToLeftUpComparison =
RangeTypeUtil.compareLowerToUpperBounds(session, rightValue, leftValue);
254     if (leftLowToRightUpComparison == 0) {
255         return leftValue.isLowerInclusive() ^ rightValue.isUpperInclusive();
256     } else if (rightLowToLeftUpComparison == 0) {
257         return rightValue.isLowerInclusive() ^ leftValue.isUpperInclusive();
258     }
259     return false;
260 }
261
262 private boolean overlaps(SessionLocal session, RangeValue leftValue, RangeValue
rightValue) {
263     int leftLowToRightUpComparison =

```

```

RangeTypeUtil.compareLowerToUpperBounds(session, leftValue, rightValue);
264     int rightLowToLeftUpComparison =
RangeTypeUtil.compareLowerToUpperBounds(session, rightValue, leftValue);
265     if (leftLowToRightUpComparison == 0) {
266         return leftValue.isLowerInclusive() && rightValue.isUpperInclusive();
267     } else if (rightLowToLeftUpComparison == 0) {
268         return rightValue.isLowerInclusive() && leftValue.isUpperInclusive();
269     }
270     return leftLowToRightUpComparison < 0 && rightLowToLeftUpComparison < 0;
271 }
272
273     private <E extends Value> boolean containsElement(SessionLocal session,
RangeValue<E> range, E element) {
274         if (range == RangeValue.EMPTY) {
275             return false;
276         }
277         int lowToElementComparison =
RangeTypeUtil.compareLowerBoundToElement(session, range, element);
278         int upToElementComparison =
RangeTypeUtil.compareUpperBoundToElement(session, range, element);
279         if (lowToElementComparison == 0 && upToElementComparison > 0) {
280             return range.isLowerInclusive();
281         } else if (upToElementComparison == 0 && lowToElementComparison < 0) {
282             return range.isUpperInclusive();
283         }
284         return lowToElementComparison < 0 && upToElementComparison > 0;
285     }
286
287     private boolean containsRange(SessionLocal session, RangeValue outerRange,
RangeValue innerRange) {
288         return notExtendLeft(session, innerRange, outerRange) && notExtendRight(session,
innerRange, outerRange);
289     }
290
291     private boolean notExtendRight(SessionLocal session, RangeValue leftRange,
RangeValue rightRange) {
292         int leftUpToRightUpComparison =
RangeTypeUtil.compareUpperToUpperBounds(session, leftRange, rightRange);
293         if (leftUpToRightUpComparison == 0) {
294             return rightRange.isUpperInclusive() || !leftRange.isUpperInclusive();
295         }
296         return leftUpToRightUpComparison < 0;
297     }
298
299     private boolean notExtendLeft(SessionLocal session, RangeValue leftRange,
RangeValue rightRange) {
300         int leftLowToRightLowComparison =
RangeTypeUtil.compareLowerToLowerBounds(session, leftRange, rightRange);
301         if (leftLowToRightLowComparison == 0) {
302             return rightRange.isLowerInclusive() || !leftRange.isLowerInclusive();
303         }
304         return leftLowToRightLowComparison > 0;
305     }
306
307     @Override
308     public Expression optimize(SessionLocal session) {
309         left = left.optimize(session);
310         right = right.optimize(session);
311         if (left.getType().getExtTypeInfo() instanceof ExtTypeInfoRange) {

```

```

312     type = left.getType();
313 } else if (right.getType().getExtTypeInfo() instanceof ExtTypeInfoRange) {
314     type = right.getType();
315 } else {
316     BinaryOperation.OpType originalOpType = null;
317     switch (opType) {
318         case UNION:
319             originalOpType = BinaryOperation.OpType.PLUS;
320             break;
321         case DIFFERENCE:
322             originalOpType = BinaryOperation.OpType.MINUS;
323             break;
324         case INTERSECTION:
325             originalOpType = BinaryOperation.OpType.MULTIPLY;
326     }
327     correction = new BinaryOperation(originalOpType, left, right);
328     return correction.optimize(session);
329 }
330 return this;
331 }
332
333 @Override
334 public StringBuilder getUnenclosedSQL(StringBuilder builder, int sqlFlags) {
335     left.getSQL(builder, sqlFlags, AUTO_PARENTHESSES).append('
').append(getOperationToken()).append(' ');
336     return right.getSQL(builder, sqlFlags, AUTO_PARENTHESSES);
337 }
338
339 private String getOperationToken() {
340     switch (opType) {
341         case UNION:
342             return "+";
343         case DIFFERENCE:
344             return "-";
345         case INTERSECTION:
346             return "*";
347         case ADJACENT:
348             return "-|";
349         case OVERLAPS:
350             return "&&";
351         case CONTAINS:
352             return "@>";
353         case CONTAINED_BY:
354             return "<@";
355         case STRICT_LEFT:
356             return "<<";
357         case STRICT_RIGHT:
358             return ">>";
359         case NOT_EXTEND_RIGHT:
360             return "&<";
361         case NOT_EXTEND_LEFT:
362             return "&>";
363         default:
364             throw DbException.getInternalError("Unknown operator. opType=" + opType);
365     }
366 }
367
368 private RangeOpType getOperationEnum(String operation) {
369     switch (operation) {

```

```

370     case "+":
371         return RangeOpType.UNION;
372     case "-":
373         return RangeOpType.DIFFERENCE;
374     case "*":
375         return RangeOpType.INTERSECTION;
376     case "-|":
377         return RangeOpType.ADJACENT;
378     case "&&":
379         return RangeOpType.OVERLAPS;
380     case "@>":
381         return RangeOpType.CONTAINS;
382     case "<@" :
383         return RangeOpType.CONTAINED_BY;
384     case "<<":
385         return RangeOpType.STRICT_LEFT;
386     case ">>":
387         return RangeOpType.STRICT_RIGHT;
388     case "&<":
389         return RangeOpType.NOT_EXTEND_RIGHT;
390     case "&>":
391         return RangeOpType.NOT_EXTEND_LEFT;
392     default:
393         throw DbException.getInternalError("Unknown operator. opType=" + operation);
394     }
395 }
396 }

```

h2/src/main/org/h2/compatibility/postgresql/RangeTypeUtil.java

```

1  package org.h2.compatibility.postgresql;
2
3  import org.h2.command.Parser;
4  import org.h2.engine.SessionLocal;
5  import org.h2.expression.Expression;
6  import org.h2.expression.ValueExpression;
7  import org.h2.message.DbException;
8  import org.h2.util.DateTimeUtils;
9  import org.h2.util.StringUtils;
10 import org.h2.value.*;
11
12 public class RangeTypeUtil {
13
14     private static PostgreSQLDataConverter converter = new PostgreSQLDataConverter();
15
16     public static boolean isRangeValue(Value value) {
17         if (value instanceof RangeValue) {
18             return true;
19         }
20         return isRangeValue(value.getString());
21     }
22 }
23
24 public static boolean isRangeValue(String input) {
25     if ("empty".equals(input)) {
26         return true;
27     }
28     String split[] = input.split(",");
29     if (split.length != 2) {

```

```

30         return false;
31     }
32     String opening = input.substring(0, 1);
33     String closing = input.substring(input.length() - 1);
34     String lowerBound = split[0].substring(1);
35     String upperBound = split[1].substring(0, split[1].length() - 1);
36     Parser p = new Parser();
37     Expression lowerExp = StringUtils.isEmpty(lowerBound) ?
ValueExpression.get(ValueNull.INSTANCE) : p.parseExpression(lowerBound);
38     Expression upperExp = StringUtils.isEmpty(upperBound) ?
ValueExpression.get(ValueNull.INSTANCE) : p.parseExpression(upperBound);
39     return (lowerExp.getType() == TypeInfo.TYPE_NULL || upperExp.getType() ==
TypeInfo.TYPE_NULL || lowerExp.getType() == upperExp.getType()) &&
40         (opening.equals("[") || opening.equals("(") &&
41         (closing.equals("]") || closing.equals(")")));
42     }
43
44     public static RangeValue castRangeValue(Value value) {
45         if (value instanceof RangeValue) {
46             return (RangeValue) value;
47         } else if (value instanceof ValueVarchar) {
48             return converter.convertTypelessToRangeValue(value);
49         }
50         throw DbException.getInternalError("not a range value: " + value);
51     }
52
53     public static int compareLowerToLowerBounds(SessionLocal session, RangeValue
range1, RangeValue range2) {
54         return session.compare(range1.getLowerBound(), range2.getLowerBound());
55     }
56
57     public static int compareLowerToUpperBounds(SessionLocal session, RangeValue
range1, RangeValue range2) {
58         int comparison = session.compare(range1.getLowerBound(),
range2.getUpperBound());
59         if (comparison == 1 && range2.getUpperBound() == ValueNull.INSTANCE) {
60             comparison = -1;
61         }
62         return comparison;
63     }
64
65     public static int compareUpperToUpperBounds(SessionLocal session, RangeValue
range1, RangeValue range2) {
66         int comparison = session.compare(range1.getUpperBound(),
range2.getUpperBound());
67         if (comparison == 1 && range2.getUpperBound() == ValueNull.INSTANCE) {
68             comparison = -1;
69         } else if (comparison == -1 && range1.getUpperBound() == ValueNull.INSTANCE) {
70             comparison = 1;
71         }
72         return comparison;
73     }
74
75     public static <E extends Value> int compareLowerBoundToElement(SessionLocal
session, RangeValue<E> range, E element) {
76         int comparison = session.compare(range.getLowerBound(), element);
77         return comparison;
78     }
79

```



```

80     public static <E extends Value> int compareUpperBoundToElement(SessionLocal
      session, RangeValue<E> range, E element) {
81         int comparison = session.compare(range.getUpperBound(), element);
82         if (comparison == -1 && range.getUpperBound() == ValueNull.INSTANCE) {
83             comparison = 1;
84         }
85         return comparison;
86     }
87
88     public static RangeValue canonize(RangeValue rangeValue) {
89         Value v1 = rangeValue.getLowerBound();
90         Value v2 = rangeValue.getUpperBound();
91         if (!rangeValue.isLowerInclusive()) {
92             v1 = addOneDiscreteTo(v1);
93         }
94         if (rangeValue.isUpperInclusive()) {
95             v2 = addOneDiscreteTo(v2);
96         }
97         return new RangeValue(v1, v2, rangeValue.getType());
98     }
99
100    private static Value addOneDiscreteTo(Value v) {
101        switch (v.getValueType()) {
102            case Value.INTEGER:
103                return v.add(ValueInteger.get(1));
104            case Value.BIGINT:
105                return v.add(ValueBigint.get(1));
106            case Value.DATE:
107                long dateValue = ((ValueDate) v).getDateValue();
108                long oneDay = DateTimeUtils.dateValue(0, 0, 1);
109                return ValueDate.fromDateValue(dateValue + oneDay);
110            default:
111                return v;
112        }
113    }
114
115 }

```

h2/src/main/org/h2/compatibility/postgresql/RangeValue.java

```

1     package org.h2.compatibility.postgresql;
2
3     import org.h2.util.StringUtils;
4     import org.h2.value.*;
5
6     public final class RangeValue<E extends Value> extends ValueStringBase {
7
8         /**
9          * Empty range.
10        */
11        public static final RangeValue EMPTY = new RangeValue();
12
13        private E lowerBound;
14        private E upperBound;
15        private boolean lowerInclusive;
16        private boolean upperInclusive;
17        private boolean empty;
18
19        private RangeValue() {

```

```

20     super("empty");
21     this.empty = true;
22     this.lowerBound = (E) ValueNull.INSTANCE;
23     this.upperBound = (E) ValueNull.INSTANCE;
24     this.lowerInclusive = false;
25     this.upperInclusive = false;
26 }
27
28 public RangeValue(E lowerBound, E upperBound) {
29     this(lowerBound, upperBound, true, false);
30 }
31
32 public RangeValue(E lowerBound, E upperBound, TypeInfo type) {
33     this(lowerBound, upperBound, true, false, type);
34 }
35
36 public RangeValue(E lowerBound, E upperBound, boolean lowerInclusive, boolean
upperInclusive) {
37     this(lowerBound, upperBound, lowerInclusive, upperInclusive, null);
38     getType();
39 }
40
41 public RangeValue(E lowerBound, E upperBound, boolean lowerInclusive, boolean
upperInclusive, TypeInfo type) {
42     super("");
43     empty = false;
44     this.lowerBound = lowerBound;
45     this.upperBound = upperBound;
46     this.lowerInclusive = lowerBound == ValueNull.INSTANCE ? false : lowerInclusive;
47     this.upperInclusive = upperBound == ValueNull.INSTANCE ? false : upperInclusive;
48     this.value = (this.lowerInclusive ? '[' : '(') +
49         (this.lowerBound == ValueNull.INSTANCE ? "" : this.lowerBound.getString()) +
50         ',' +
51         (this.upperBound == ValueNull.INSTANCE ? "" : this.upperBound.getString()) +
52         (this.upperInclusive ? ']' : ')');
53     this.type = type;
54 }
55
56 public E getLowerBound() {
57     return lowerBound;
58 }
59
60 public E getUpperBound() {
61     return upperBound;
62 }
63
64 public boolean isLowerInclusive() {
65     return lowerInclusive;
66 }
67
68 public boolean isUpperInclusive() {
69     return upperInclusive;
70 }
71
72 public boolean isEmpty() {
73     return empty;
74 }
75
76 @Override

```

```

77     public StringBuilder getSQL(StringBuilder builder, int sqlFlags) {
78         return StringUtils.quoteStringSQL(builder, value);
79     }
80
81     @Override
82     public int getValueType() {
83         return VARCHAR;
84     }
85
86     public String getNamedRangeType() {
87         switch(getSubType().getValueType()) {
88             case INTEGER:
89                 return "int4range";
90             case BIGINT:
91                 return "int8range";
92             case NUMERIC:
93                 return "numrange";
94             case TIMESTAMP:
95                 return "tsrange";
96             case TIMESTAMP_TZ:
97                 return "tstzrange";
98             case DATE:
99                 return "daterange";
100             default:
101                 return null;
102         }
103     }
104
105     public TypeInfo getSubType() {
106         return lowerBound != ValueNull.INSTANCE ? lowerBound.getType() :
107         upperBound.getType();
108     }
109
110     @Override
111     public TypeInfo getType() {
112         if (type == null) {
113             TypeInfo subType = getSubType();
114             ExtTypeInfo extType = ExtTypeInfoRange.getExtTypeBySubtype(subType);
115             this.type = new TypeInfo(getValueType(), -1, -1, extType);
116         }
117         return type;
118     }

```

h2/src/main/org/h2/engine/Mode.java

```

...
28     // Classe
29     /**
30      * The compatibility modes. There is a fixed set of modes (for example
31      * PostgreSQL, MySQL). Each mode has different settings.
32      */
33     public class Mode {
34         public enum ModeEnum {
35             REGULAR, DB2, Derby, MSSQLServer, HSQLDB, MySQL, Oracle, PostgreSQL
36         }
37     }
38
39     ...
140     // Atributos estáticos
    private static final HashMap<String, Mode> MODES = new HashMap<>();

```

```

...
// Atributos do objeto
405 /**
406  * Custom condition-level binary operators.
407  */
408 public Set<String> customConditionOperators = Collections.emptySet();
409
410 /**
411  * Custom sum-level binary operators.
412  */
413 public Set<String> customSumOperators = Collections.emptySet();
414
415 /**
416  * Custom factor-level binary operators.
417  */
418 public Set<String> customFactorOperators = Collections.emptySet();
...
432 /**
433  * Parser with exclusive parsing capabilities.
434  */
435 public BiFunction<SessionLocal, Parser, CompatibilityParser> compatibilityParser =
(SessionLocal session, Parser parser) -> new NeutralCompatibilityParser(session, parser);
436
437 public Supplier<DataConverter> dataConverter = () -> new NeutralDataConverter();
438
439 private final String name;
440
441 private final ModeEnum modeEnum;
442
443 static {
444     Mode mode;
...
575     mode = new Mode(ModeEnum.PostgreSQL);
576     mode.compatibilityParser = PostgreSQLParser::new;
577     mode.dataConverter = PostgreSQLDataConverter::new;
...
594     Set<String> customConditionOperators = new java.util.HashSet<>();
595     customConditionOperators.add("&&");
596     customConditionOperators.add("<@");
597     customConditionOperators.add("@>");
598     customConditionOperators.add("<<");
599     customConditionOperators.add(">>");
600     customConditionOperators.add("&<");
601     customConditionOperators.add("&>");
602     customConditionOperators.add("-|-");
603     mode.customConditionOperators = customConditionOperators;
604     Set<String> customSumOperators = new java.util.HashSet<>();
605     customSumOperators.add("-");
606     customSumOperators.add("+");
607     mode.customSumOperators = customSumOperators;
608     Set<String> customFactorOperators = new java.util.HashSet<>();
609     customFactorOperators.add("**");
610     mode.customFactorOperators = customFactorOperators;
...
629     dt = new DataType();
630     dt.type = Value.VARCHAR;
631     dt.sqlType = Types.VARCHAR;
632     dt.extTypeInfo = ExtTypeInfoRange.INT4RANGE;
633     mode.typeByNameMap.put("INT4RANGE", dt);

```

```

634     dt = new DataType();
635     dt.type = Value.VARCHAR;
636     dt.sqlType = Types.VARCHAR;
637     dt.extTypeInfo = ExtTypeInfoRange.INT8RANGE;
638     mode.typeByNameMap.put("INT8RANGE", dt);
639     dt = new DataType();
640     dt.type = Value.VARCHAR;
641     dt.sqlType = Types.VARCHAR;
642     dt.extTypeInfo = ExtTypeInfoRange.NUMRANGE;
643     mode.typeByNameMap.put("NUMRANGE", dt);
644     dt = new DataType();
645     dt.type = Value.VARCHAR;
646     dt.sqlType = Types.VARCHAR;
647     dt.extTypeInfo = ExtTypeInfoRange.TSRANGE;
648     mode.typeByNameMap.put("TSRANGE", dt);
649     dt = new DataType();
650     dt.type = Value.VARCHAR;
651     dt.sqlType = Types.VARCHAR;
652     dt.extTypeInfo = ExtTypeInfoRange.TSTZRANGE;
653     mode.typeByNameMap.put("TSTZRANGE", dt);
654     dt = new DataType();
655     dt.type = Value.VARCHAR;
656     dt.sqlType = Types.VARCHAR;
657     dt.extTypeInfo = ExtTypeInfoRange.DATERANGE;
658     mode.typeByNameMap.put("DATERANGE", dt);
659     add(mode);
660 }
...
// Construtor
662 private Mode(ModeEnum modeEnum) {
663     this.name = modeEnum.name();
664     this.modeEnum = modeEnum;
665 }
666 private static void add(Mode mode) {
667     MODES.put(StringUtils.toUpperEnglish(mode.name), mode);
668 }
...
698 }

```

h2/src/main/org/h2/value/DataType.java

```

24     ...
25     /**
26     * This class contains meta data information about data types,
27     * and can convert between Java objects and Values.
28     */
29     public class DataType {
30         ...
115         /**
116         * Direct approach to custom data type implementation.
117         */
118         public ExtTypeInfo extTypeInfo;
...
830     }

```

h2/src/main/org/h2/value/ExtTypeInfo.java

```

...
10  /**
11  * Extended parameters of a data type.
12  */
13  public abstract class ExtTypeInfo implements HasSQL {
...
20  /**
21  * Overriding this method returning true forces TypeInfo's getSQL to return it's companion
  ExtTypeInfo getSQL instead.
22  */
23  public boolean overrideSQL() {
24      return false;
25  }
...
27  }

```

h2/src/main/org/h2/value/TypeInfo.java

```

...
21  // Classe
22  /**
23  * Data type with parameters.
24  */
  public class TypeInfo extends ExtTypeInfo implements Typed {
...
  // Atributos estáticos
41  /**
42  * CHARACTER VARYING type with maximum parameters.
43  */
44  public static final TypeInfo TYPE_VARCHAR;
...
  // Atributos do objeto
213 private final int valueType;
214
215 private final long precision;
216
217 private final int scale;
218
219 private final ExtTypeInfo extTypeInfo;
...
221 static {
...
228     TYPE_VARCHAR = new TypeInfo(Value.VARCHAR);
...
277 }
  // Método gerenciador de instâncias de TypeInfo
300 /**
301 * Get the data type with parameters object for the given value type and the
302 * specified parameters.
303 *
304 * @param type
305 *     the value type
306 * @param precision
307 *     the precision or {@code -1L} for default
308 * @param scale
309 *     the scale or {@code -1} for default
310 * @param extTypeInfo
311 *     the extended type information or null

```

```

312     * @return the data type with parameters object
313     */
314     public static TypeInfo getTypeInfo(int type, long precision, int scale, ExtTypeInfo
extTypeInfo) {
315         switch (type) {
316             ...
335         case Value.VARCHAR:
336             if (extTypeInfo instanceof ExtTypeInfoRange) {
337                 return new TypeInfo(Value.VARCHAR, -1, -1, extTypeInfo);
338             }
339             if (precision < 1 || precision >= Constants.MAX_STRING_LENGTH) {
340                 if (precision != 0) {
341                     return TYPE_VARCHAR;
342                 }
343                 precision = 1;
344             }
345             return new TypeInfo(Value.VARCHAR, precision);
346         }
347         ...
348     }
349     ...
350 }
351 ...
352 // Método de conversão objeto TypeInfo para String
1231 @Override
1232 public StringBuilder getSQL(StringBuilder builder, int sqlFlags) {
1233     if (extTypeInfo != null && extTypeInfo.overrideSQL()) {
1234         return extTypeInfo.getSQL(builder, sqlFlags);
1235     }
1236     ...
1237 }
1238 ...
1239 }
1240 ...
1241 }
1493 }

```

h2/src/main/org/h2/value/Value.java

```

...
// Classe
39 /**
40  * This is the base class for all value classes.
41  * It provides conversion and comparison methods.
42  *
43  * @author Thomas Mueller
44  * @author Noel Grandin
45  * @author Nicolas Fortin, Atelier SIG, IRSTV FR CNRS 24888
46  */
47 public abstract class Value extends VersionedValue<Value> implements HasSQL, Typed {
48     ...
49     // Atributos estáticos
50     /**
51      * The data type is unknown at this time.
52      */
53     public static final int UNKNOWN = -1;
54     /**
55      * The value type for NULL.
56      */
57     public static final int NULL = UNKNOWN + 1;
58     /**
59

```

```

60      * The value type for CHARACTER values.
61      */
62      public static final int CHAR = NULL + 1;
63
64      /**
65       * The value type for CHARACTER VARYING values.
66       */
67      public static final int VARCHAR = CHAR + 1;
        ...
        // Método de conversão de tipos
1033     /**
1034     * Convert a value to the specified type.
1035     *
1036     * @param targetType the type of the returned value
1037     * @param provider the cast information provider
1038     * @param conversionMode conversion mode
1039     * @param column the column (if any), used to improve the error message if conversion
        fails
1040     * @return the converted value
1041     */
1042     private Value convertTo(TypeInfo targetType, CastDataProvider provider, int
conversionMode, Object column) {
1043         if (provider != null) {
1044             Value converted = provider.getMode().dataConverter.get().convert(this, targetType);
1045             if (converted != null) {
1046                 return converted;
1047             }
1048         }
        ...
1133     }
        ...
2698 }

```


APÊNDICE B – Código de Testes

h2/src/test/org/h2/test/db/compatibility/postgresql/TestRangeType.java

```
1  /*
2  * Copyright 2004-2021 H2 Group. Multiple-Licensed under the MPL 2.0,
3  * and the EPL 1.0 (https://h2database.com/html/license.html).
4  * Initial Developer: H2 Group
5  */
6  package org.h2.test.db.compatibility.postgresql;
7
8  import org.h2.api.ErrorCode;
9  import org.h2.test.TestBase;
10 import org.h2.test.TestDb;
11
12 import java.sql.*;
13
14 /**
15  * Test range datatype compatibility.
16  */
17 public class TestRangeType extends TestDb {
18
19     /**
20      * Run just this test.
21      *
22      * @param a ignored
23      */
24     public static void main(String... a) throws Exception {
25         TestBase.createCaller().init().testFromMain();
26     }
27
28     @Override
29     public void test() throws Exception {
30         deleteDb("postgres");
31         testDeclaration();
32         testCRUD();
33         testOperators();
34         testFunctions();
35         deleteDb("postgres");
36         System.out.println("end of tests");
37     }
38
39     private void testDeclaration() throws SQLException {
40         testDeclarationInt4range();
41         testDeclarationInt8range();
42         testDeclarationNumrange();
43         testDeclarationDaterange();
44         testDeclarationTsrangle();
45         testDeclarationTstzrange();
46     }
47
48     private void testCRUD() throws SQLException {
49         testCRUDint4range();
50         testCRUDint8range();
51         testCRUDnumrange();
52         testCRUDdaterange();
53         testCRUDtsrange();
54         testCRUDtstzrange();
55     }
56 }
```

```

56
57     private void testOperators() throws SQLException {
58         testUnion();
59         testDifference();
60         testIntersection();
61         testAdjacent();
62         testOverlaps();
63         testContainsRange();
64         testContainsElement();
65         testContainedRange();
66         testContainedElement();
67         testStrictLeft();
68         testStrictRight();
69         testNotExtendRight();
70         testNotExtendLeft();
71         testSumPriorityUnion();
72         testSumPriorityDifference();
73         testSumChain();
74         testFactorPriority();
75         testFactorChain();
76     }
77
78     private void testFunctions() throws SQLException {
79         testLower();
80         testLowerReturnPropagation();
81         testUpper();
82         testUpperReturnPropagation();
83         testIsEmpty();
84         testLowerInc();
85         testUpperInc();
86         testLowerInf();
87         testUpperInf();
88         testRangeMerge();
89     }
90
91     private void testDeclarationInt4range() throws SQLException {
92         testDeclarationInt4rangeEmpty();
93         testDeclarationInt4rangeDefaultCanonic();
94         testDeclarationInt4rangeExplicitBoundsCaninicise();
95         testDeclarationInt4rangeInfinity();
96         testDeclarationInt4rangeIllegalBounds();
97     }
98
99     private void testDeclarationInt4rangeEmpty() throws SQLException {
100         deleteDb("postgres");
101         Connection conn = getConnection("postgres;MODE=PostgreSQL");
102         Statement stat = conn.createStatement();
103
104         assertResult("empty", stat, "select 'empty'::int4range");
105         assertResult("empty", stat, "select cast('empty' as int4range)");
106         assertResult("empty", stat, "select int4range(1,1)");
107
108         conn.close();
109     }
110
111     private void testDeclarationInt4rangeDefaultCanonic() throws SQLException {
112         deleteDb("postgres");
113         Connection conn = getConnection("postgres;MODE=PostgreSQL");
114         Statement stat = conn.createStatement();

```

```

115
116     assertResult("[1,3]", stat, "select '[1,3]':int4range");
117     assertResult("[1,3]", stat, "select cast('[1,3]' as int4range)");
118     assertResult("[1,3]", stat, "select int4range(1,3)");
119
120     conn.close();
121 }
122
123 private void testDeclarationInt4rangeExplicitBoundsCaninicise() throws SQLException {
124     deleteDb("postgres");
125     Connection conn = getConnection("postgres;MODE=PostgreSQL");
126     Statement stat = conn.createStatement();
127
128     assertResult("[2,3]", stat, "select '(1,3)':int4range");
129     assertResult("[2,4]", stat, "select '(1,3)':int4range");
130     assertResult("[1,4]", stat, "select '[1,3]':int4range");
131     assertResult("[2,3]", stat, "select cast('(1,3)' as int4range)");
132     assertResult("[2,4]", stat, "select cast('(1,3]' as int4range)");
133     assertResult("[1,4]", stat, "select cast('[1,3]' as int4range)");
134     assertResult("[1,3]", stat, "select int4range(1,3,'I')");
135     assertResult("[2,3]", stat, "select int4range(1,3,'()')");
136     assertResult("[2,4]", stat, "select int4range(1,3,'()')");
137     assertResult("[1,4]", stat, "select int4range(1,3,'[]')");
138
139     conn.close();
140 }
141
142 private void testDeclarationInt4rangeInfinity() throws SQLException {
143     deleteDb("postgres");
144     Connection conn = getConnection("postgres;MODE=PostgreSQL");
145     Statement stat = conn.createStatement();
146
147     assertResult("[1,)", stat, "select '[1,)':int4range");
148     assertResult("(,3)", stat, "select '(,3)':int4range");
149     assertResult("(,)", stat, "select '(,)':int4range");
150     assertResult("[1,)", stat, "select cast('[1,)' as int4range)");
151     assertResult("(,3)", stat, "select cast('(,3)' as int4range)");
152     assertResult("(,)", stat, "select cast('(,)' as int4range)");
153     assertResult("[1,)", stat, "select int4range(1,NULL)");
154     assertResult("(,3)", stat, "select int4range(NULL,3)");
155     assertResult("(,)", stat, "select int4range(NULL,NULL)");
156
157     conn.close();
158 }
159
160 private void testDeclarationInt4rangeIllegalBounds() throws SQLException {
161     deleteDb("postgres");
162     Connection conn = getConnection("postgres;MODE=PostgreSQL");
163     Statement stat = conn.createStatement();
164
165     assertThrows(ErrorCode.GENERAL_ERROR_1, stat, "select '[2,1)':int4range");
166     assertThrows(ErrorCode.GENERAL_ERROR_1, stat, "select cast('[2,1]' as
int4range)");
167     assertThrows(ErrorCode.GENERAL_ERROR_1, stat, "select int4range(2,1)");
168
169     conn.close();
170 }
171
172 private void testDeclarationInt8range() throws SQLException {

```

```

173     testDeclarationInt8rangeEmpty();
174     testDeclarationInt8rangeDefaultCanonic();
175     testDeclarationInt8rangeExplicitBoundsCanonicise();
176     testDeclarationInt8rangeInfinity();
177     testDeclarationInt8rangeIllegalBounds();
178 }
179
180 private void testDeclarationInt8rangeEmpty() throws SQLException {
181     deleteDb("postgres");
182     Connection conn = getConnection("postgres;MODE=PostgreSQL");
183     Statement stat = conn.createStatement();
184
185     assertResult("empty", stat, "select 'empty'::int8range");
186     assertResult("empty", stat, "select cast('empty' as int8range)");
187     assertResult("empty", stat, "select int8range(1,1)");
188
189     conn.close();
190 }
191
192 private void testDeclarationInt8rangeDefaultCanonic() throws SQLException {
193     deleteDb("postgres");
194     Connection conn = getConnection("postgres;MODE=PostgreSQL");
195     Statement stat = conn.createStatement();
196
197     assertResult("[1,3]", stat, "select '[1,3]'::int8range");
198     assertResult("[1,3]", stat, "select cast('[1,3]' as int8range)");
199     assertResult("[1,3]", stat, "select int8range(1,3)");
200
201     conn.close();
202 }
203
204 private void testDeclarationInt8rangeExplicitBoundsCanonicise() throws SQLException {
205     deleteDb("postgres");
206     Connection conn = getConnection("postgres;MODE=PostgreSQL");
207     Statement stat = conn.createStatement();
208
209     assertResult("[2,3]", stat, "select '(1,3)'::int8range");
210     assertResult("[2,4]", stat, "select '(1,3)'::int8range");
211     assertResult("[1,4]", stat, "select '[1,3]'::int8range");
212     assertResult("[2,3]", stat, "select cast('(1,3)' as int8range)");
213     assertResult("[2,4]", stat, "select cast('(1,3)' as int8range)");
214     assertResult("[1,4]", stat, "select cast('[1,3]' as int8range)");
215     assertResult("[1,3]", stat, "select int8range(1,3,'[])");
216     assertResult("[2,3]", stat, "select int8range(1,3,'()')");
217     assertResult("[2,4]", stat, "select int8range(1,3,'[])");
218     assertResult("[1,4]", stat, "select int8range(1,3,'[])");
219
220     conn.close();
221 }
222
223 private void testDeclarationInt8rangeInfinity() throws SQLException {
224     deleteDb("postgres");
225     Connection conn = getConnection("postgres;MODE=PostgreSQL");
226     Statement stat = conn.createStatement();
227
228     assertResult("[1,)", stat, "select '[1,)'::int8range");
229     assertResult("(,3]", stat, "select '(,3]'::int8range");
230     assertResult("(,)", stat, "select '(,)'::int8range");
231     assertResult("[1,)", stat, "select cast('[1,)' as int8range)");

```

```

232     assertResult("(,3)", stat, "select cast('(,3)' as int8range)");
233     assertResult("(", stat, "select cast('(') as int8range)");
234     assertResult("[1,)", stat, "select int8range(1,NULL)");
235     assertResult("(,3)", stat, "select int8range(NULL,3)");
236     assertResult("(", stat, "select int8range(NULL,NULL)");
237
238     conn.close();
239 }
240
241 private void testDeclarationInt8rangeIllegalBounds() throws SQLException {
242     deleteDb("postgres");
243     Connection conn = getConnection("postgres;MODE=PostgreSQL");
244     Statement stat = conn.createStatement();
245
246     assertThrows(ErrorCode.GENERAL_ERROR_1, stat, "select '[2,1)::int8range)");
247     assertThrows(ErrorCode.GENERAL_ERROR_1, stat, "select cast('[2,1)' as
int8range)");
248     assertThrows(ErrorCode.GENERAL_ERROR_1, stat, "select int8range(2,1)");
249
250     conn.close();
251 }
252
253 private void testDeclarationNumrange() throws SQLException {
254     testDeclarationNumrangeEmpty();
255     testDeclarationNumrangeDefaultCanonic();
256     testDeclarationNumrangeExplicitBoundsDoesNotCanonicise();
257     testDeclarationNumrangeInfinity();
258     testDeclarationNumrangeIllegalBounds();
259 }
260
261 private void testDeclarationNumrangeEmpty() throws SQLException {
262     deleteDb("postgres");
263     Connection conn = getConnection("postgres;MODE=PostgreSQL");
264     Statement stat = conn.createStatement();
265
266     assertResult("empty", stat, "select 'empty)::numrange)");
267     assertResult("empty", stat, "select cast('empty' as numrange)");
268     assertResult("empty", stat, "select numrange(1.0,1.0)");
269
270     conn.close();
271 }
272
273 private void testDeclarationNumrangeDefaultCanonic() throws SQLException {
274     deleteDb("postgres");
275     Connection conn = getConnection("postgres;MODE=PostgreSQL");
276     Statement stat = conn.createStatement();
277
278     assertResult("[1.0,3.0)", stat, "select '[1.0,3.0)::numrange)");
279     assertResult("[1.0,3.0)", stat, "select cast('[1.0,3.0)' as numrange)");
280     assertResult("[1.0,3.0)", stat, "select numrange(1.0,3.0)");
281
282     conn.close();
283 }
284
285 private void testDeclarationNumrangeExplicitBoundsDoesNotCanonicise() throws
SQLException {
286     deleteDb("postgres");
287     Connection conn = getConnection("postgres;MODE=PostgreSQL");
288     Statement stat = conn.createStatement();

```

```

289
290     assertResult("(1.0,3.0)", stat, "select '(1.0,3.0)::numrange");
291     assertResult("(1.0,3.0]", stat, "select '(1.0,3.0]::numrange");
292     assertResult("[1.0,3.0]", stat, "select '[1.0,3.0]::numrange");
293     assertResult("(1.0,3.0)", stat, "select cast('(1.0,3.0)' as numrange)");
294     assertResult("(1.0,3.0]", stat, "select cast('(1.0,3.0]' as numrange)");
295     assertResult("[1.0,3.0]", stat, "select cast('[1.0,3.0]' as numrange)");
296     assertResult("(1.0,3.0)", stat, "select numrange(1.0,3.0,'()')");
297     assertResult("(1.0,3.0)", stat, "select numrange(1.0,3.0,'()')");
298     assertResult("(1.0,3.0]", stat, "select numrange(1.0,3.0,'()]')");
299     assertResult("[1.0,3.0]", stat, "select numrange(1.0,3.0,'[]')");
300
301     conn.close();
302 }
303
304 private void testDeclarationNumrangeInfinity() throws SQLException {
305     deleteDb("postgres");
306     Connection conn = getConnection("postgres;MODE=PostgreSQL");
307     Statement stat = conn.createStatement();
308
309     assertResult("[1.0,)", stat, "select '[1.0,)::numrange");
310     assertResult("(,3.0)", stat, "select '(,3.0)::numrange");
311     assertResult("(,)", stat, "select '(,)::numrange");
312     assertResult("[1.0,)", stat, "select cast('[1.0,)' as numrange");
313     assertResult("(,3.0)", stat, "select cast('(,3.0)' as numrange");
314     assertResult("(,)", stat, "select cast('(,)' as numrange");
315     assertResult("[1.0,)", stat, "select numrange(1.0,NULL)");
316     assertResult("(,3.0)", stat, "select numrange(NULL,3.0)");
317     assertResult("(,)", stat, "select numrange(NULL,NULL)");
318
319     conn.close();
320 }
321
322 private void testDeclarationNumrangeIllegalBounds() throws SQLException {
323     deleteDb("postgres");
324     Connection conn = getConnection("postgres;MODE=PostgreSQL");
325     Statement stat = conn.createStatement();
326
327     assertThrows(ErrorCode.GENERAL_ERROR_1, stat, "select '[2,1)::numrange");
328     assertThrows(ErrorCode.GENERAL_ERROR_1, stat, "select cast('[2,1)' as
numrange)");
329     assertThrows(ErrorCode.GENERAL_ERROR_1, stat, "select numrange(2.0,1.0)");
330
331     conn.close();
332 }
333
334 private void testDeclarationDaterange() throws SQLException {
335     testDeclarationDaterangeEmpty();
336     testDeclarationDaterangeDefaultCanonic();
337     testDeclarationDaterangeExplicitBoundsCanonicise();
338     testDeclarationDaterangeInfinity();
339     testDeclarationDaterangeIllegalBounds();
340 }
341
342 private void testDeclarationDaterangeEmpty() throws SQLException {
343     deleteDb("postgres");
344     Connection conn = getConnection("postgres;MODE=PostgreSQL");
345     Statement stat = conn.createStatement();
346

```

```

347     assertResult("empty", stat, "select 'empty'::daterange");
348     assertResult("empty", stat, "select cast('empty' as daterange)");
349     assertResult("empty", stat, "select daterange('2020-01-01','2020-01-01')");
350
351     conn.close();
352 }
353
354 private void testDeclarationDaterangeDefaultCanonic() throws SQLException {
355     deleteDb("postgres");
356     Connection conn = getConnection("postgres;MODE=PostgreSQL");
357     Statement stat = conn.createStatement();
358
359     assertResult("[2020-01-01,2020-01-03]", stat, "select '[2020-01-01,2020-01-03]'::daterange");
360     assertResult("[2020-01-01,2020-01-03]", stat, "select cast('[2020-01-01,2020-01-03]' as daterange)");
361     assertResult("[2020-01-01,2020-01-03]", stat, "select daterange('2020-01-01','2020-01-03')");
362
363     conn.close();
364 }
365
366 private void testDeclarationDaterangeExplicitBoundsCanonicise() throws SQLException {
367     deleteDb("postgres");
368     Connection conn = getConnection("postgres;MODE=PostgreSQL");
369     Statement stat = conn.createStatement();
370
371     assertResult("(2020-01-02,2020-01-03)", stat, "select '(2020-01-01,2020-01-03)'::daterange");
372     assertResult("(2020-01-02,2020-01-04)", stat, "select '(2020-01-01,2020-01-03]'::daterange");
373     assertResult("[2020-01-01,2020-01-04)", stat, "select '[2020-01-01,2020-01-03]'::daterange");
374     assertResult("[2020-01-02,2020-01-03)", stat, "select cast('(2020-01-01,2020-01-03)' as daterange)");
375     assertResult("(2020-01-02,2020-01-04)", stat, "select cast('(2020-01-01,2020-01-03]' as daterange)");
376     assertResult("(2020-01-01,2020-01-04)", stat, "select cast('[2020-01-01,2020-01-03]' as daterange)");
377     assertResult("[2020-01-01,2020-01-03)", stat, "select daterange('2020-01-01','2020-01-03','()')");
378     assertResult("(2020-01-02,2020-01-03)", stat, "select daterange('2020-01-01','2020-01-03','()')");
379     assertResult("[2020-01-02,2020-01-04)", stat, "select daterange('2020-01-01','2020-01-03','()')");
380     assertResult("[2020-01-01,2020-01-04)", stat, "select daterange('2020-01-01','2020-01-03','[]')");
381
382     conn.close();
383 }
384
385 private void testDeclarationDaterangeInfinity() throws SQLException {
386     deleteDb("postgres");
387     Connection conn = getConnection("postgres;MODE=PostgreSQL");
388     Statement stat = conn.createStatement();
389
390     assertResult("[2020-01-01,)", stat, "select '[2020-01-01,)'::daterange");
391     assertResult("(,2020-01-03)", stat, "select '(,2020-01-03)'::daterange");
392     assertResult("(,)", stat, "select '(,)'::daterange");

```

```

393     assertResult("[2020-01-01,)", stat, "select cast('[2020-01-01,)' as daterange");
394     assertResult("(,2020-01-03)", stat, "select cast('(,2020-01-03)' as daterange");
395     assertResult(",)", stat, "select cast(',)' as daterange");
396     assertResult("[2020-01-01,)", stat, "select daterange('2020-01-01',NULL)");
397     assertResult("(,2020-01-03)", stat, "select daterange(NULL,'2020-01-03')");
398     assertResult(",)", stat, "select daterange(NULL,NULL)");
399
400     conn.close();
401 }
402
403 private void testDeclarationDaterangeIllegalBounds() throws SQLException {
404     deleteDb("postgres");
405     Connection conn = getConnection("postgres;MODE=PostgreSQL");
406     Statement stat = conn.createStatement();
407
408     assertThrows(ErrorCode.GENERAL_ERROR_1, stat, "select '[2020-01-02,2020-01-01]::daterange");
409     assertThrows(ErrorCode.GENERAL_ERROR_1, stat, "select cast('[2020-01-02,2020-01-01]' as daterange)");
410     assertThrows(ErrorCode.GENERAL_ERROR_1, stat, "select daterange('2020-01-02','2020-01-01')");
411
412     conn.close();
413 }
414
415 private void testDeclarationTsrangle() throws SQLException {
416     testDeclarationTsrangleEmpty();
417     testDeclarationTsrangleDefaultCanonic();
418     testDeclarationTsrangleExplicitBoundsDoesNotCanonicise();
419     testDeclarationTsrangleInfinity();
420     testDeclarationTsrangleIllegalBounds();
421 }
422
423 private void testDeclarationTsrangleEmpty() throws SQLException {
424     deleteDb("postgres");
425     Connection conn = getConnection("postgres;MODE=PostgreSQL");
426     Statement stat = conn.createStatement();
427
428     assertResult("empty", stat, "select 'empty'::tsrange");
429     assertResult("empty", stat, "select cast('empty' as tsrange)");
430     assertResult("empty", stat, "select tsrange('2020-01-01 00:00:01','2020-01-01 00:00:01')");
431
432     conn.close();
433 }
434
435 private void testDeclarationTsrangleDefaultCanonic() throws SQLException {
436     deleteDb("postgres");
437     Connection conn = getConnection("postgres;MODE=PostgreSQL");
438     Statement stat = conn.createStatement();
439
440     assertResult("[2020-01-01 00:00:01,2020-01-01 00:00:03)", stat, "select '[2020-01-01 00:00:01,2020-01-01 00:00:03]::tsrange");
441     assertResult("[2020-01-01 00:00:01,2020-01-01 00:00:03)", stat, "select cast('[2020-01-01 00:00:01,2020-01-01 00:00:03]' as tsrange)");
442     assertResult("[2020-01-01 00:00:01,2020-01-01 00:00:03)", stat, "select tsrange('2020-01-01 00:00:01','2020-01-01 00:00:03')");
443
444     conn.close();

```



```

445     }
446
447     private void testDeclarationTsrangleExplicitBoundsDoesNotCanonicise() throws
SQLException {
448         deleteDb("postgres");
449         Connection conn = getConnection("postgres;MODE=PostgreSQL");
450         Statement stat = conn.createStatement();
451
452         assertResult("(2020-01-01 00:00:01,2020-01-01 00:00:03)", stat, "select '(2020-01-01-
00:00:01,2020-01-01 00:00:03)::tsrange");
453         assertResult("(2020-01-01 00:00:01,2020-01-01 00:00:03]", stat, "select '(2020-01-01
00:00:01,2020-01-01 00:00:03]::tsrange");
454         assertResult("[2020-01-01 00:00:01,2020-01-01 00:00:03]", stat, "select '[2020-01-01
00:00:01,2020-01-01 00:00:03]::tsrange");
455         assertResult("(2020-01-01 00:00:01,2020-01-01 00:00:03)", stat, "select cast('(2020-01-
01 00:00:01,2020-01-01 00:00:03)' as tsrange");
456         assertResult("(2020-01-01 00:00:01,2020-01-01 00:00:03]", stat, "select cast('(2020-01-
01 00:00:01,2020-01-01 00:00:03]' as tsrange");
457         assertResult("[2020-01-01 00:00:01,2020-01-01 00:00:03]", stat, "select cast('[2020-01-
01 00:00:01,2020-01-01 00:00:03]' as tsrange");
458         assertResult("[2020-01-01 00:00:01,2020-01-01 00:00:03)", stat, "select tsrange('2020-
01-01 00:00:01','2020-01-01 00:00:03',')");
459         assertResult("(2020-01-01 00:00:01,2020-01-01 00:00:03)", stat, "select tsrange('2020-
01-01 00:00:01','2020-01-01 00:00:03',')");
460         assertResult("(2020-01-01 00:00:01,2020-01-01 00:00:03]", stat, "select tsrange('2020-
01-01 00:00:01','2020-01-01 00:00:03',']");
461         assertResult("[2020-01-01 00:00:01,2020-01-01 00:00:03)", stat, "select tsrange('2020-
01-01 00:00:01','2020-01-01 00:00:03',')");
462
463         conn.close();
464     }
465
466     private void testDeclarationTsrangleInfinity() throws SQLException {
467         deleteDb("postgres");
468         Connection conn = getConnection("postgres;MODE=PostgreSQL");
469         Statement stat = conn.createStatement();
470
471         assertResult("[2020-01-01 00:00:01,)", stat, "select '[2020-01-01 00:00:01,)::tsrange");
472         assertResult("(,2020-01-01 00:00:03)", stat, "select '(,2020-01-01 00:00:03)::tsrange");
473         assertResult("(,)", stat, "select '(,)::tsrange");
474         assertResult("[2020-01-01 00:00:01,)", stat, "select cast('[2020-01-01 00:00:01,)' as
tsrange");
475         assertResult("(,2020-01-01 00:00:03)", stat, "select cast('(,2020-01-01 00:00:03)' as
tsrange");
476         assertResult("(,)", stat, "select cast('(,)' as tsrange");
477         assertResult("[2020-01-01 00:00:01,)", stat, "select tsrange('2020-01-01
00:00:01',NULL)");
478         assertResult("(,2020-01-01 00:00:03)", stat, "select tsrange(NULL,'2020-01-01
00:00:03')");
479         assertResult("(,)", stat, "select tsrange(NULL,NULL)");
480
481         conn.close();
482     }
483
484     private void testDeclarationTsrangleIllegalBounds() throws SQLException {
485         deleteDb("postgres");
486         Connection conn = getConnection("postgres;MODE=PostgreSQL");
487         Statement stat = conn.createStatement();
488

```

```

489     assertThrows(ErrorCode.GENERAL_ERROR_1, stat, "select '[2020-01-01
490     00:00:02,2020-01-01 00:00:01]':tstrange");
491     assertThrows(ErrorCode.GENERAL_ERROR_1, stat, "select cast('[2020-01-01
492     00:00:02,2020-01-01 00:00:01]' as tstrange)");
493     assertThrows(ErrorCode.GENERAL_ERROR_1, stat, "select tstrange('2020-01-01
494     00:00:02','2020-01-01 00:00:01')");
495
496     conn.close();
497 }
498
499 private void testDeclarationTstzrange() throws SQLException {
500     testDeclarationTstzrangeEmpty();
501     testDeclarationTstzrangeDefaultCanonic();
502     testDeclarationTstzrangeExplicitBoundsDoesNotCanonicise();
503     testDeclarationTstzrangeInfinity();
504     testDeclarationTstzrangeIllegalBounds();
505 }
506
507 private void testDeclarationTstzrangeEmpty() throws SQLException {
508     deleteDb("postgres");
509     Connection conn = getConnection("postgres;MODE=PostgreSQL");
510     Statement stat = conn.createStatement();
511
512     assertResult("empty", stat, "select 'empty':tstzrange");
513     assertResult("empty", stat, "select cast('empty' as tstzrange)");
514     assertResult("empty", stat, "select tstzrange('2020-01-01 00:00:01-05','2020-01-01
515     00:00:01-05')");
516
517     conn.close();
518 }
519
520 private void testDeclarationTstzrangeDefaultCanonic() throws SQLException {
521     deleteDb("postgres");
522     Connection conn = getConnection("postgres;MODE=PostgreSQL");
523     Statement stat = conn.createStatement();
524
525     assertResult("[2020-01-01 00:00:01-05,2020-01-01 00:00:03-05]", stat, "select '[2020-
526     01-01 00:00:01-05,2020-01-01 00:00:03-05]':tstzrange");
527     assertResult("[2020-01-01 00:00:01-05,2020-01-01 00:00:03-05]", stat, "select
528     cast('[2020-01-01 00:00:01-05,2020-01-01 00:00:03-05]' as tstzrange)");
529     assertResult("[2020-01-01 00:00:01-05,2020-01-01 00:00:03-05]", stat, "select
530     tstzrange('2020-01-01 00:00:01-05','2020-01-01 00:00:03-05')");
531
532     conn.close();
533 }
534
535 private void testDeclarationTstzrangeExplicitBoundsDoesNotCanonicise() throws
536     SQLException {
537     deleteDb("postgres");
538     Connection conn = getConnection("postgres;MODE=PostgreSQL");
539     Statement stat = conn.createStatement();
540
541     assertResult("(2020-01-01 00:00:01-05,2020-01-01 00:00:03-05)", stat, "select '(2020-
542     01-01 00:00:01-05,2020-01-01 00:00:03-05)':tstzrange");
543     assertResult("(2020-01-01 00:00:01-05,2020-01-01 00:00:03-05)", stat, "select '(2020-
544     01-01 00:00:01-05,2020-01-01 00:00:03-05)':tstzrange");
545     assertResult("[2020-01-01 00:00:01-05,2020-01-01 00:00:03-05]", stat, "select '[2020-
546     01-01 00:00:01-05,2020-01-01 00:00:03-05]':tstzrange");
547     assertResult("(2020-01-01 00:00:01-05,2020-01-01 00:00:03-05)", stat, "select

```

```

537 cast('(2020-01-01 00:00:01-05,2020-01-01 00:00:03-05)' as tstzrange));
    assertResult("(2020-01-01 00:00:01-05,2020-01-01 00:00:03-05]", stat, "select
538 cast('(2020-01-01 00:00:01-05,2020-01-01 00:00:03-05]' as tstzrange)");
    assertResult("[2020-01-01 00:00:01-05,2020-01-01 00:00:03-05]", stat, "select
539 cast('[2020-01-01 00:00:01-05,2020-01-01 00:00:03-05]' as tstzrange)");
    assertResult("[2020-01-01 00:00:01-05,2020-01-01 00:00:03-05)", stat, "select
540 tstzrange('2020-01-01 00:00:01-05','2020-01-01 00:00:03-05',')')");
    assertResult("(2020-01-01 00:00:01-05,2020-01-01 00:00:03-05)", stat, "select
541 tstzrange('2020-01-01 00:00:01-05','2020-01-01 00:00:03-05',')')");
    assertResult("(2020-01-01 00:00:01-05,2020-01-01 00:00:03-05]", stat, "select
542 tstzrange('2020-01-01 00:00:01-05','2020-01-01 00:00:03-05',']')");
543
544     conn.close();
545 }
546
547 private void testDeclarationTstzrangeInfinity() throws SQLException {
548     deleteDb("postgres");
549     Connection conn = getConnection("postgres;MODE=PostgreSQL");
550     Statement stat = conn.createStatement();
551
552     assertResult("[2020-01-01 00:00:01-05)", stat, "select '[2020-01-01 00:00:01-
553 05,):::tstzrange)");
    assertResult("(,2020-01-01 00:00:03-05)", stat, "select '(,2020-01-01 00:00:03-
554 05)::tstzrange)");
    assertResult(",)", stat, "select '(,):::tstzrange)");
555     assertResult("[2020-01-01 00:00:01-05)", stat, "select cast('[2020-01-01 00:00:01-05,)'
as tstzrange)");
556     assertResult("(,2020-01-01 00:00:03-05)", stat, "select cast('(,2020-01-01 00:00:03-05)'
as tstzrange)");
557     assertResult(",)", stat, "select cast('(,) as tstzrange)");
558     assertResult("[2020-01-01 00:00:01-05)", stat, "select tstzrange('2020-01-01 00:00:01-
05',NULL)");
559     assertResult("(,2020-01-01 00:00:03-05)", stat, "select tstzrange(NULL,'2020-01-01
00:00:03-05)");
560     assertResult(",)", stat, "select tstzrange(NULL,NULL)");
561
562     conn.close();
563 }
564
565 private void testDeclarationTstzrangeIllegalBounds() throws SQLException {
566     deleteDb("postgres");
567     Connection conn = getConnection("postgres;MODE=PostgreSQL");
568     Statement stat = conn.createStatement();
569
570     assertThrows(ErrorCode.GENERAL_ERROR_1, stat, "select '[2020-01-01 00:00:02-
571 05,2020-01-01 00:00:01-05)::tstzrange)");
    assertThrows(ErrorCode.GENERAL_ERROR_1, stat, "select cast('[2020-01-01
00:00:02-05,2020-01-01 00:00:01-05]' as tstzrange)");
572     assertThrows(ErrorCode.GENERAL_ERROR_1, stat, "select tstzrange('2020-01-01
00:00:02-05','2020-01-01 00:00:01-05)");
573
574     conn.close();
575 }
576
577 private void testCRUDint4range() throws SQLException {
578     deleteDb("postgres");
579     Connection conn = getConnection("postgres;MODE=PostgreSQL");

```

```

580 Statement stat = conn.createStatement();
581 //Create
582 stat.execute("create table T (R int4range, S int4range)");
583 stat.execute("insert into T values ([1,4)::int4range, [2,5)::int4range)");
584 stat.execute("insert into T values ('empty)::int4range, '(.)::int4range)");
585 //Read
586 assertResult("[2,5)", stat, "select S from T where R = '[1,4)::int4range)");
587 assertResult("(,)", stat, "select S from T where R = 'empty)::int4range)");
588 assertResult("empty", stat, "select R from T where S = '(.)::int4range)");
589 //Update
590 stat.execute("update T set S = '[3,6)::int4range where R = '[1,4)::int4range)");
591 assertResult("[3,6)", stat, "select S from T where R = '[1,4)::int4range)");
592 //Delete
593 stat.execute("delete from T where R = '[1,4)::int4range)");
594 assertResult(null, stat, "select * from T where R = '[1,4)::int4range)");
595
596 conn.close();
597 }
598
599 private void testCRUDint8range() throws SQLException {
600     deleteDb("postgres");
601     Connection conn = getConnection("postgres;MODE=PostgreSQL");
602     Statement stat = conn.createStatement();
603     //Create
604     stat.execute("create table T (R int8range, S int8range)");
605     stat.execute("insert into T values ([1,4)::int8range, [2,5)::int8range)");
606     stat.execute("insert into T values ('empty)::int8range, '(.)::int8range)");
607     //Read
608     assertResult("[2,5)", stat, "select S from T where R = '[1,4)::int8range)");
609     assertResult("(,)", stat, "select S from T where R = 'empty)::int8range)");
610     assertResult("empty", stat, "select R from T where S = '(.)::int8range)");
611     //Update
612     stat.execute("update T set S = '[3,6)::int8range where R = '[1,4)::int8range)");
613     assertResult("[3,6)", stat, "select S from T where R = '[1,4)::int8range)");
614     //Delete
615     stat.execute("delete from T where R = '[1,4)::int8range)");
616     assertResult(null, stat, "select * from T where R = '[1,4)::int8range)");
617
618     conn.close();
619 }
620
621 private void testCRUDnumrange() throws SQLException {
622     deleteDb("postgres");
623     Connection conn = getConnection("postgres;MODE=PostgreSQL");
624     Statement stat = conn.createStatement();
625     //Create
626     stat.execute("create table T (R numrange, S numrange)");
627     stat.execute("insert into T values ([1.0,4.0)::numrange, [2.0,5.0)::numrange)");
628     stat.execute("insert into T values ('empty)::numrange, '(.)::numrange)");
629     //Read
630     assertResult("[2.0,5.0)", stat, "select S from T where R = '[1.0,4.0)::numrange)");
631     assertResult("(,)", stat, "select S from T where R = 'empty)::numrange)");
632     assertResult("empty", stat, "select R from T where S = '(.)::numrange)");
633     //Update
634     stat.execute("update T set S = '[3.0,6.0)::numrange where R = '[1.0,4.0)::numrange)");
635     assertResult("[3.0,6.0)", stat, "select S from T where R = '[1.0,4.0)::numrange)");
636     //Delete
637     stat.execute("delete from T where R = '[1.0,4.0)::numrange)");
638     assertResult(null, stat, "select * from T where R = '[1.0,4.0)::numrange)");

```

```

639     conn.close();
640 }
641
642 private void testCRUDdaterange() throws SQLException {
643     deleteDb("postgres");
644     Connection conn = getConnection("postgres;MODE=PostgreSQL");
645     Statement stat = conn.createStatement();
646     //Create
647     stat.execute("create table T (R daterange, S daterange)");
648     stat.execute("insert into T values ('[2020-01-01,2020-01-04]':daterange, '[2020-01-02,2020-01-05]':daterange)");
649     stat.execute("insert into T values ('empty':daterange, '(,)':daterange)");
650     //Read
651     assertResult("[2020-01-02,2020-01-05]", stat, "select S from T where R = '[2020-01-01,2020-01-04]':daterange");
652     assertResult("(,)", stat, "select S from T where R = 'empty':daterange");
653     assertResult("empty", stat, "select R from T where S = '(,)':daterange");
654     //Update
655     stat.execute("update T set S = '[2020-01-03,2020-01-06]':daterange where R = '[2020-01-01,2020-01-04]':daterange");
656     assertResult("[2020-01-03,2020-01-06]", stat, "select S from T where R = '[2020-01-01,2020-01-04]':daterange");
657     //Delete
658     stat.execute("delete from T where R = '[2020-01-01,2020-01-04]':daterange");
659     assertResult(null, stat, "select * from T where R = '[2020-01-01,2020-01-04]':daterange");
660
661     conn.close();
662 }
663
664 private void testCRUDtsrange() throws SQLException {
665     deleteDb("postgres");
666     Connection conn = getConnection("postgres;MODE=PostgreSQL");
667     Statement stat = conn.createStatement();
668     //Create
669     stat.execute("create table T (R tsrange, S tsrange)");
670     stat.execute("insert into T values ('[2020-01-01 00:00:01,2020-01-01 00:00:04]':tsrange, '[2020-01-01 00:00:02,2020-01-01 00:00:05]':tsrange)");
671     stat.execute("insert into T values ('empty':tsrange, '(,)':tsrange)");
672     //Read
673     assertResult("[2020-01-01 00:00:02,2020-01-01 00:00:05]", stat, "select S from T where R = '[2020-01-01 00:00:01,2020-01-01 00:00:04]':tsrange");
674     assertResult("(,)", stat, "select S from T where R = 'empty':tsrange");
675     assertResult("empty", stat, "select R from T where S = '(,)':tsrange");
676     //Update
677     stat.execute("update T set S = '[2020-01-01 00:00:03,2020-01-01 00:00:06]':tsrange where R = '[2020-01-01 00:00:01,2020-01-01 00:00:04]':tsrange");
678     assertResult("[2020-01-01 00:00:03,2020-01-01 00:00:06]", stat, "select S from T where R = '[2020-01-01 00:00:01,2020-01-01 00:00:04]':tsrange");
679     //Delete
680     stat.execute("delete from T where R = '[2020-01-01 00:00:01,2020-01-01 00:00:04]':tsrange");
681     assertResult(null, stat, "select * from T where R = '[2020-01-01 00:00:01,2020-01-01 00:00:04]':tsrange");
682
683     conn.close();
684 }
685
686

```

```

687 private void testCRUDtstzrange() throws SQLException {
688     deleteDb("postgres");
689     Connection conn = getConnection("postgres;MODE=PostgreSQL");
690     Statement stat = conn.createStatement();
691     //Create
692     stat.execute("create table T (R tstzrange, S tstzrange)");
693     stat.execute("insert into T values ('[2020-01-01 00:00:01-05,2020-01-01 00:00:04-05]':tstzrange, '[2020-01-01 00:00:02-05,2020-01-01 00:00:05-05]':tstzrange)");
694     stat.execute("insert into T values ('empty':tstzrange, '(,)':tstzrange)");
695     //Read
696     assertResult("[2020-01-01 00:00:02-05,2020-01-01 00:00:05-05]", stat, "select S from T where R = '[2020-01-01 00:00:01-05,2020-01-01 00:00:04-05]':tstzrange");
697     assertResult("(,)", stat, "select S from T where R = 'empty':tstzrange");
698     assertResult("empty", stat, "select R from T where S = '(,)':tstzrange");
699     //Update
700     stat.execute("update T set S = '[2020-01-01 00:00:03-05,2020-01-01 00:00:06-05]':tstzrange where R = '[2020-01-01 00:00:01-05,2020-01-01 00:00:04-05]':tstzrange");
701     assertResult("[2020-01-01 00:00:03-05,2020-01-01 00:00:06-05]", stat, "select S from T where R = '[2020-01-01 00:00:01-05,2020-01-01 00:00:04-05]':tstzrange");
702     //Delete
703     stat.execute("delete from T where R = '[2020-01-01 00:00:01-05,2020-01-01 00:00:04-05]':tstzrange");
704     assertResult(null, stat, "select * from T where R = '[2020-01-01 00:00:01-05,2020-01-01 00:00:04-05]':tstzrange");
705
706     conn.close();
707 }
708
709 public void testUnion() throws SQLException {
710     testUnionLiteralCanonicalBorders();
711     testUnionLiteralNotCanonicalBorders();
712     testUnionWithEmpty();
713     testUnionWithInfinity();
714     testUnionIdentifierFunction();
715     testUnionIdentifierDot();
716     testUnionIdentifierSubquery();
717     testUnionNamelessSubquery();
718 }
719
720 public void testUnionLiteralCanonicalBorders() throws SQLException {
721     deleteDb("postgres");
722     Connection conn = getConnection("postgres;MODE=PostgreSQL");
723     Statement stat = conn.createStatement();
724
725     assertResult("[1,5]", stat, "select '[1,3]':int4range + '[2,5]':int4range");
726     assertResult("[1,5]", stat, "select '[1,3]':int4range + '[3,5]':int4range");
727     assertThrows(ErrorCode.GENERAL_ERROR_1, stat, "select '[1,3]':int4range + '[4,5]':int4range");
728
729     conn.close();
730 }
731
732 public void testUnionLiteralNotCanonicalBorders() throws SQLException {
733     deleteDb("postgres");
734     Connection conn = getConnection("postgres;MODE=PostgreSQL");
735     Statement stat = conn.createStatement();
736
737     assertResult("[1,5]", stat, "select '[1,3]':numrange + '[3,5]':numrange");
738     assertResult("[1,5]", stat, "select '[1,3]':numrange + '[3,5]':numrange");

```

```

739     assertResult("[1,5]", stat, "select '[1,3]':::numrange + '(3,5)':::numrange");
740     assertThrows(ErrorCode.GENERAL_ERROR_1, stat, "select '[1,3]':::numrange +
'(3,5)':::numrange");
741     assertResult("[1,5]", stat, "select '(1,5)':::numrange + '[1,3]':::numrange");
742     assertResult("[1,5]", stat, "select '[3,5]':::numrange + '[1,5]':::numrange");
743
744     conn.close();
745 }
746
747 public void testUnionWithEmpty() throws SQLException {
748     deleteDb("postgres");
749     Connection conn = getConnection("postgres;MODE=PostgreSQL");
750     Statement stat = conn.createStatement();
751
752     assertResult("[3,5]", stat, "select 'empty':::int4range + '[3,5]':::int4range");
753     assertResult("[3,5]", stat, "select '[3,5]':::int4range + 'empty':::int4range");
754
755     conn.close();
756 }
757
758 public void testUnionWithInfinity() throws SQLException {
759     deleteDb("postgres");
760     Connection conn = getConnection("postgres;MODE=PostgreSQL");
761     Statement stat = conn.createStatement();
762
763     assertResult("[3,)", stat, "select '[4,)':::int4range + '[3,5]':::int4range");
764     assertResult("(,5)", stat, "select '(,4)':::int4range + '[3,5]':::int4range");
765     assertResult("(,)", stat, "select '(,)':::int4range + '[3,5]':::int4range");
766
767     conn.close();
768 }
769
770 public void testUnionIdentifierFunction() throws SQLException {
771     deleteDb("postgres");
772     Connection conn = getConnection("postgres;MODE=PostgreSQL");
773     Statement stat = conn.createStatement();
774
775     assertResult("[1,5]", stat, "select int4range(1,3) + '[2,5]':::int4range");
776
777     conn.close();
778 }
779
780 public void testUnionIdentifierDot() throws SQLException {
781     deleteDb("postgres");
782     Connection conn = getConnection("postgres;MODE=PostgreSQL");
783     Statement stat = conn.createStatement();
784
785     stat.execute("create table testTb (testKey integer primary key, testCol int4range)");
786     stat.execute("insert into testTb values (1, '[1,3]':::int4range)");
787     assertResult("[1,5]", stat, "select testSq.testCol + '[2,5]':::int4range from (select
tb.testCol from testTb tb where tb.testKey = 1) as testSq");
788
789     conn.close();
790 }
791
792 public void testUnionIdentifierSubquery() throws SQLException {
793     deleteDb("postgres");
794     Connection conn = getConnection("postgres;MODE=PostgreSQL");
795     Statement stat = conn.createStatement();

```

```

796
797     assertResult("[1,5]", stat, "select testSq + '[2,5]':int4range from (select '[1,3]':int4range
as testSq)");
798
799     conn.close();
800 }
801
802 public void testUnionNamelessSubquery() throws SQLException {
803     deleteDb("postgres");
804     Connection conn = getConnection("postgres;MODE=PostgreSQL");
805     Statement stat = conn.createStatement();
806
807     assertResult("[1,5]", stat, "select (select '[1,3]':int4range) + '[2,5]':int4range");
808
809     conn.close();
810 }
811
812 public void testDifference() throws SQLException {
813     testDifferenceLiteralCanonicalBorders();
814     testDifferenceLiteralNotCanonicalBorders();
815     testDifferenceWithEmpty();
816     testDifferenceWithInfinity();
817     testDifferenceIdentifierFunction();
818     testDifferenceIdentifierDot();
819     testDifferenceIdentifierSubquery();
820     testDifferenceNamelessSubquery();
821 }
822
823 public void testDifferenceLiteralCanonicalBorders() throws SQLException {
824     deleteDb("postgres");
825     Connection conn = getConnection("postgres;MODE=PostgreSQL");
826     Statement stat = conn.createStatement();
827
828     assertResult("[1,2]", stat, "select '[1,3]':int4range - '[2,5]':int4range");
829     assertResult("empty", stat, "select '[1,3]':int4range - '[1,5]':int4range");
830     assertThrows(ErrorCode.GENERAL_ERROR_1, stat, "select '[1,4]':int4range -
'[2,3]':int4range");
831
832     conn.close();
833 }
834
835 public void testDifferenceLiteralNotCanonicalBorders() throws SQLException {
836     deleteDb("postgres");
837     Connection conn = getConnection("postgres;MODE=PostgreSQL");
838     Statement stat = conn.createStatement();
839
840     assertResult("[1,3]", stat, "select '[1,3]':numrange - '[3,5]':numrange");
841     assertResult("[1,3]", stat, "select '[1,3]':numrange - '[3,5]':numrange");
842     assertResult("[1,3]", stat, "select '[1,3]':numrange - '(3,5)':numrange");
843     assertResult("[1,3]", stat, "select '[1,3]':numrange - '(3,5)':numrange");
844
845     conn.close();
846 }
847
848 public void testDifferenceWithEmpty() throws SQLException {
849     deleteDb("postgres");
850     Connection conn = getConnection("postgres;MODE=PostgreSQL");
851     Statement stat = conn.createStatement();
852

```



```

853     assertResult("empty", stat, "select 'empty'::int4range - '[1,5]'::int4range");
854     assertResult("[1,3)", stat, "select '[1,3]'::int4range - 'empty'::int4range");
855
856     conn.close();
857 }
858
859 public void testDifferenceWithInfinity() throws SQLException {
860     deleteDb("postgres");
861     Connection conn = getConnection("postgres;MODE=PostgreSQL");
862     Statement stat = conn.createStatement();
863
864     assertResult("empty", stat, "select '[1,3]'::int4range - '(,)'::int4range");
865     assertThrows(ErrorCode.GENERAL_ERROR_1, stat, "select '(,)'::int4range -
866 '[2,3]'::int4range");
867     conn.close();
868 }
869
870 public void testDifferenceIdentifierFunction() throws SQLException {
871     deleteDb("postgres");
872     Connection conn = getConnection("postgres;MODE=PostgreSQL");
873     Statement stat = conn.createStatement();
874
875     assertResult("[1,2)", stat, "select int4range(1,3) - '[2,5]'::int4range");
876
877     conn.close();
878 }
879
880 public void testDifferenceIdentifierDot() throws SQLException {
881     deleteDb("postgres");
882     Connection conn = getConnection("postgres;MODE=PostgreSQL");
883     Statement stat = conn.createStatement();
884
885     stat.execute("create table testTb (testKey integer primary key, testCol int4range)");
886     stat.execute("insert into testTb values (1, '[1,3]'::int4range)");
887     assertResult("[1,2)", stat, "select testSq.testCol - '[2,5]'::int4range from (select
888 tb.testCol from testTb tb where tb.testKey = 1) as testSq");
889     conn.close();
890 }
891
892 public void testDifferenceIdentifierSubquery() throws SQLException {
893     deleteDb("postgres");
894     Connection conn = getConnection("postgres;MODE=PostgreSQL");
895     Statement stat = conn.createStatement();
896
897     assertResult("[1,2)", stat, "select testSq - '[2,5]'::int4range from (select '[1,3]'::int4range
898 as testSq)");
899     conn.close();
900 }
901
902 public void testDifferenceNamelessSubquery() throws SQLException {
903     deleteDb("postgres");
904     Connection conn = getConnection("postgres;MODE=PostgreSQL");
905     Statement stat = conn.createStatement();
906
907     assertResult("[1,2)", stat, "select (select '[1,3]'::int4range) - '[2,5]'::int4range");
908

```

```

909     conn.close();
910 }
911
912 public void testIntersection() throws SQLException {
913     testIntersectionLiteralCanonicalBorders();
914     testIntersectionLiteralNotCanonicalBorders();
915     testIntersectionWithEmpty();
916     testIntersectionWithInfinity();
917     testIntersectionIdentifierFunction();
918     testIntersectionIdentifierDot();
919     testIntersectionIdentifierSubquery();
920     testIntersectionNamelessSubquery();
921 }
922
923 public void testIntersectionLiteralCanonicalBorders() throws SQLException {
924     deleteDb("postgres");
925     Connection conn = getConnection("postgres;MODE=PostgreSQL");
926     Statement stat = conn.createStatement();
927
928     assertResult("[2,4]", stat, "select '[1,5]::int4range * '[2,4]::int4range");
929     assertResult("[2,4]", stat, "select '[2,4]::int4range * '[1,5]::int4range");
930     assertResult("[2,3]", stat, "select '[1,3]::int4range * '[2,5]::int4range");
931     assertResult("[2,3]", stat, "select '[2,5]::int4range * '[1,3]::int4range");
932     assertResult("empty", stat, "select '[1,3]::int4range * '[3,5]::int4range");
933
934     conn.close();
935 }
936
937 public void testIntersectionLiteralNotCanonicalBorders() throws SQLException {
938     deleteDb("postgres");
939     Connection conn = getConnection("postgres;MODE=PostgreSQL");
940     Statement stat = conn.createStatement();
941
942     assertResult("[3,3]", stat, "select '[1,3]::numrange * '[3,5]::numrange");
943     assertResult("empty", stat, "select '[1,3]::numrange * '(3,5)::numrange");
944     assertResult("empty", stat, "select '[1,3]::numrange * '(3,5)::numrange");
945     assertResult("[3,4]", stat, "select '[1,4]::numrange * '[3,5]::numrange");
946     assertResult("(3,4)", stat, "select '[1,4]::numrange * '(3,5)::numrange");
947     assertResult("(3,4)", stat, "select '[1,4]::numrange * '[3,5]::numrange");
948     assertResult("(3,4)", stat, "select '[1,4]::numrange * '(3,5)::numrange");
949
950     conn.close();
951 }
952
953 public void testIntersectionWithEmpty() throws SQLException {
954     deleteDb("postgres");
955     Connection conn = getConnection("postgres;MODE=PostgreSQL");
956     Statement stat = conn.createStatement();
957
958     assertResult("empty", stat, "select 'empty'::int4range * '[3,5]::int4range");
959     assertResult("empty", stat, "select '[3,5]::int4range * 'empty'::int4range");
960
961     conn.close();
962 }
963
964 public void testIntersectionWithInfinity() throws SQLException {
965     deleteDb("postgres");
966     Connection conn = getConnection("postgres;MODE=PostgreSQL");
967     Statement stat = conn.createStatement();

```

```

968
969     assertResult("[3,5]", stat, "select '(,)'::int4range * '[3,5]'::int4range");
970     assertResult("[3,5]", stat, "select '[3,5]'::int4range * '(,)'::int4range");
971
972     conn.close();
973 }
974
975 public void testIntersectionIdentifierFunction() throws SQLException {
976     deleteDb("postgres");
977     Connection conn = getConnection("postgres;MODE=PostgreSQL");
978     Statement stat = conn.createStatement();
979
980     assertResult("[2,3]", stat, "select int4range(1,3) * '[2,5]'::int4range");
981
982     conn.close();
983 }
984
985 public void testIntersectionIdentifierDot() throws SQLException {
986     deleteDb("postgres");
987     Connection conn = getConnection("postgres;MODE=PostgreSQL");
988     Statement stat = conn.createStatement();
989
990     stat.execute("create table testTb (testKey integer primary key, testCol int4range)");
991     stat.execute("insert into testTb values (1, '[1,3]'::int4range)");
992     assertResult("[2,3]", stat, "select testSq.testCol * '[2,5]'::int4range from (select
993     tb.testCol from testTb tb where tb.testKey = 1) as testSq");
994
995     conn.close();
996 }
997
998 public void testIntersectionIdentifierSubquery() throws SQLException {
999     deleteDb("postgres");
1000     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1001     Statement stat = conn.createStatement();
1002
1003     assertResult("[2,3]", stat, "select testSq * '[2,5]'::int4range from (select '[1,3]'::int4range
1004     as testSq)");
1005
1006     conn.close();
1007 }
1008
1009 public void testIntersectionNamelessSubquery() throws SQLException {
1010     deleteDb("postgres");
1011     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1012     Statement stat = conn.createStatement();
1013
1014     assertResult("[2,3]", stat, "select (select '[1,3]'::int4range) * '[2,5]'::int4range");
1015
1016     conn.close();
1017 }
1018
1019 public void testAdjacent() throws SQLException {
1020     testAdjacentLiteralCanonicalBorders();
1021     testAdjacentLiteralNotCanonicalBorders();
1022     testAdjacentWithEmpty();
1023     testAdjacentWithInfinity();
1024     testAdjacentIdentifierFunction();
1025     testAdjacentIdentifierDot();
1026     testAdjacentIdentifierSubquery();

```

```

1025     testAdjacentNamelessSubquery();
1026 }
1027
1028 public void testAdjacentLiteralCanonicalBorders() throws SQLException {
1029     deleteDb("postgres");
1030     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1031     Statement stat = conn.createStatement();
1032
1033     assertResult("FALSE", stat, "select '[1,3]::int4range -|- '[2,5]::int4range");
1034     assertResult("TRUE", stat, "select '[1,3]::int4range -|- '[3,5]::int4range");
1035     assertResult("FALSE", stat, "select '[1,2]::int4range -|- '[3,5]::int4range");
1036
1037     conn.close();
1038 }
1039
1040 public void testAdjacentLiteralNotCanonicalBorders() throws SQLException {
1041     deleteDb("postgres");
1042     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1043     Statement stat = conn.createStatement();
1044
1045     assertResult("FALSE", stat, "select '[1,3]::numrange -|- '[3,5]::numrange");
1046     assertResult("TRUE", stat, "select '[1,3]::numrange -|- '(3,5)::numrange");
1047     assertResult("FALSE", stat, "select '[1,3]::numrange -|- '(3,5)::numrange");
1048     assertResult("FALSE", stat, "select '[3,5]::numrange -|- '[1,3]::numrange");
1049     assertResult("TRUE", stat, "select '(3,5)::numrange -|- '[1,3]::numrange");
1050     assertResult("FALSE", stat, "select '(3,5)::numrange -|- '[1,3]::numrange");
1051
1052     conn.close();
1053 }
1054
1055 public void testAdjacentWithEmpty() throws SQLException {
1056     deleteDb("postgres");
1057     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1058     Statement stat = conn.createStatement();
1059
1060     assertResult("FALSE", stat, "select 'empty'::int4range -|- '[3,5]::int4range");
1061     assertResult("FALSE", stat, "select '[3,5]::int4range -|- 'empty'::int4range");
1062
1063     conn.close();
1064 }
1065
1066 public void testAdjacentWithInfinity() throws SQLException {
1067     deleteDb("postgres");
1068     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1069     Statement stat = conn.createStatement();
1070
1071     assertResult("FALSE", stat, "select '(,)'::int4range -|- '[3,5]::int4range");
1072     assertResult("FALSE", stat, "select '[3,5]::int4range -|- '(,)'::int4range");
1073
1074     conn.close();
1075 }
1076
1077 public void testAdjacentIdentifierFunction() throws SQLException {
1078     deleteDb("postgres");
1079     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1080     Statement stat = conn.createStatement();
1081
1082     assertResult("FALSE", stat, "select int4range(1,3) -|- '[2,5]::int4range");
1083     assertResult("TRUE", stat, "select int4range(1,3) -|- '[3,5]::int4range");

```

```

1084     conn.close();
1085 }
1086
1087
1088 public void testAdjacentIdentifierDot() throws SQLException {
1089     deleteDb("postgres");
1090     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1091     Statement stat = conn.createStatement();
1092
1093     stat.execute("create table testTb (testKey integer primary key, testCol int4range)");
1094     stat.execute("insert into testTb values (1, '[1,3]::int4range)");
1095     assertResult("FALSE", stat, "select testSq.testCol -|- '[2,5]::int4range from (select
tb.testCol from testTb tb where tb.testKey = 1) as testSq");
1096     assertResult("TRUE", stat, "select testSq.testCol -|- '[3,5]::int4range from (select
tb.testCol from testTb tb where tb.testKey = 1) as testSq");
1097
1098     conn.close();
1099 }
1100
1101 public void testAdjacentIdentifierSubquery() throws SQLException {
1102     deleteDb("postgres");
1103     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1104     Statement stat = conn.createStatement();
1105
1106     assertResult("FALSE", stat, "select testSq -|- '[2,5]::int4range from (select
'[1,3]::int4range as testSq)");
1107     assertResult("TRUE", stat, "select testSq -|- '[3,5]::int4range from (select
'[1,3]::int4range as testSq)");
1108
1109     conn.close();
1110 }
1111
1112 public void testAdjacentNamelessSubquery() throws SQLException {
1113     deleteDb("postgres");
1114     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1115     Statement stat = conn.createStatement();
1116
1117     assertResult("FALSE", stat, "select (select '[1,3]::int4range) -|- '[2,5]::int4range");
1118     assertResult("TRUE", stat, "select (select '[1,3]::int4range) -|- '[3,5]::int4range");
1119
1120     conn.close();
1121 }
1122
1123 public void testOverlaps() throws SQLException {
1124     testOverlapsLiteralCanonicalBorders();
1125     testOverlapsLiteralNotCanonicalBorders();
1126     testOverlapsWithEmpty();
1127     testOverlapsWithInfinity();
1128     testOverlapsIdentifierFunction();
1129     testOverlapsIdentifierDot();
1130     testOverlapsIdentifierSubquery();
1131     testOverlapsNamelessSubquery();
1132 }
1133
1134 public void testOverlapsLiteralCanonicalBorders() throws SQLException {
1135     deleteDb("postgres");
1136     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1137     Statement stat = conn.createStatement();
1138

```

```

1139     assertResult("FALSE", stat, "select '[1,3)::int4range && '[4,5)::int4range");
1140     assertResult("FALSE", stat, "select '[1,3)::int4range && '[3,5)::int4range");
1141     assertResult("TRUE", stat, "select '[1,3)::int4range && '[2,5)::int4range");
1142     assertResult("TRUE", stat, "select '[1,3)::int4range && '[1,5)::int4range");
1143     assertResult("TRUE", stat, "select '[1,3)::int4range && '[-1,5)::int4range");
1144     assertResult("TRUE", stat, "select '[1,3)::int4range && '[1,3)::int4range");
1145     assertResult("TRUE", stat, "select '[1,3)::int4range && '[-1,3)::int4range");
1146     assertResult("TRUE", stat, "select '[1,3)::int4range && '[-1,2)::int4range");
1147     assertResult("FALSE", stat, "select '[1,3)::int4range && '[-1,1)::int4range");
1148     assertResult("FALSE", stat, "select '[1,3)::int4range && '[-1,0)::int4range");
1149
1150     conn.close();
1151 }
1152
1153 public void testOverlapsLiteralNotCanonicalBorders() throws SQLException {
1154     deleteDb("postgres");
1155     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1156     Statement stat = conn.createStatement();
1157
1158     assertResult("TRUE", stat, "select '[1,3)::numrange && '[3,5)::numrange");
1159     assertResult("FALSE", stat, "select '[1,3)::numrange && '(3,5)::numrange");
1160     assertResult("FALSE", stat, "select '[1,3)::numrange && '(3,5)::numrange");
1161
1162     conn.close();
1163 }
1164
1165 public void testOverlapsWithEmpty() throws SQLException {
1166     deleteDb("postgres");
1167     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1168     Statement stat = conn.createStatement();
1169
1170     assertResult("FALSE", stat, "select 'empty)::int4range && '[3,5)::int4range");
1171     assertResult("FALSE", stat, "select '[3,5)::int4range && 'empty)::int4range");
1172
1173     conn.close();
1174 }
1175
1176 public void testOverlapsWithInfinity() throws SQLException {
1177     deleteDb("postgres");
1178     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1179     Statement stat = conn.createStatement();
1180
1181     assertResult("TRUE", stat, "select '(,)::int4range && '[3,5)::int4range");
1182     assertResult("TRUE", stat, "select '[3,5)::int4range && '(,)::int4range");
1183
1184     conn.close();
1185 }
1186
1187 public void testOverlapsIdentifierFunction() throws SQLException {
1188     deleteDb("postgres");
1189     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1190     Statement stat = conn.createStatement();
1191
1192     assertResult("TRUE", stat, "select int4range(1,3) && '[2,5)::int4range");
1193     assertResult("FALSE", stat, "select int4range(1,3) && '[3,5)::int4range");
1194
1195     conn.close();
1196 }
1197

```

```

1198 public void testOverlapsIdentifierDot() throws SQLException {
1199     deleteDb("postgres");
1200     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1201     Statement stat = conn.createStatement();
1202
1203     stat.execute("create table testTb (testKey integer primary key, testCol int4range)");
1204     stat.execute("insert into testTb values (1, '[1,3)::int4range)");
1205     assertResult("TRUE", stat, "select testSq.testCol && '[2,5)::int4range from (select
tb.testCol from testTb tb where tb.testKey = 1) as testSq");
1206     assertResult("FALSE", stat, "select testSq.testCol && '[3,5)::int4range from (select
tb.testCol from testTb tb where tb.testKey = 1) as testSq");
1207
1208     conn.close();
1209 }
1210
1211 public void testOverlapsIdentifierSubquery() throws SQLException {
1212     deleteDb("postgres");
1213     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1214     Statement stat = conn.createStatement();
1215
1216     assertResult("TRUE", stat, "select testSq && '[2,5)::int4range from (select
'[1,3)::int4range as testSq)");
1217     assertResult("FALSE", stat, "select testSq && '[3,5)::int4range from (select
'[1,3)::int4range as testSq)");
1218
1219     conn.close();
1220 }
1221
1222 public void testOverlapsNamelessSubquery() throws SQLException {
1223     deleteDb("postgres");
1224     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1225     Statement stat = conn.createStatement();
1226
1227     assertResult("TRUE", stat, "select (select '[1,3)::int4range) && '[2,5)::int4range");
1228     assertResult("FALSE", stat, "select (select '[1,3)::int4range) && '[3,5)::int4range");
1229
1230     conn.close();
1231 }
1232
1233 public void testContainsRange() throws SQLException {
1234     testContainsRangeLiteralCanonicalBorders();
1235     testContainsRangeLiteralNotCanonicalBorders();
1236     testContainsRangeWithEmpty();
1237     testContainsRangeWithInfinity();
1238     testContainsRangeIdentifierFunction();
1239     testContainsRangeIdentifierDot();
1240     testContainsRangeIdentifierSubquery();
1241     testContainsRangeNamelessSubquery();
1242 }
1243
1244 public void testContainsRangeLiteralCanonicalBorders() throws SQLException {
1245     deleteDb("postgres");
1246     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1247     Statement stat = conn.createStatement();
1248
1249     assertResult("FALSE", stat, "select '[1,4)::int4range @> '[0,5)::int4range");
1250     assertResult("FALSE", stat, "select '[1,4)::int4range @> '[0,4)::int4range");
1251     assertResult("FALSE", stat, "select '[1,4)::int4range @> '[1,5)::int4range");
1252     assertResult("TRUE", stat, "select '[1,4)::int4range @> '[1,4)::int4range");

```

```

1253     assertResult("TRUE", stat, "select '[1,4]':int4range @> '[1,3]':int4range");
1254     assertResult("TRUE", stat, "select '[1,4]':int4range @> '[2,4]':int4range");
1255     assertResult("TRUE", stat, "select '[1,4]':int4range @> '[2,3]':int4range");
1256
1257     conn.close();
1258 }
1259
1260 public void testContainsRangeLiteralNotCanonicalBorders() throws SQLException {
1261     deleteDb("postgres");
1262     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1263     Statement stat = conn.createStatement();
1264
1265     assertResult("TRUE", stat, "select '[1,3]':numrange @> '[1,3]':numrange");
1266     assertResult("TRUE", stat, "select '[1,3]':numrange @> '[1,3]':numrange");
1267     assertResult("TRUE", stat, "select '[1,3]':numrange @> '(1,3]':numrange");
1268     assertResult("TRUE", stat, "select '[1,3]':numrange @> '(1,3)':numrange");
1269     assertResult("FALSE", stat, "select '[1,3]':numrange @> '[1,3]':numrange");
1270     assertResult("TRUE", stat, "select '[1,3]':numrange @> '[1,3]':numrange");
1271     assertResult("FALSE", stat, "select '[1,3]':numrange @> '(1,3]':numrange");
1272     assertResult("TRUE", stat, "select '[1,3]':numrange @> '(1,3)':numrange");
1273     assertResult("FALSE", stat, "select '(1,3]':numrange @> '[1,3]':numrange");
1274     assertResult("FALSE", stat, "select '(1,3]':numrange @> '[1,3]':numrange");
1275     assertResult("TRUE", stat, "select '(1,3]':numrange @> '(1,3]':numrange");
1276     assertResult("TRUE", stat, "select '(1,3]':numrange @> '(1,3]':numrange");
1277     assertResult("FALSE", stat, "select '(1,3]':numrange @> '[1,3]':numrange");
1278     assertResult("FALSE", stat, "select '(1,3]':numrange @> '[1,3]':numrange");
1279     assertResult("FALSE", stat, "select '(1,3]':numrange @> '(1,3]':numrange");
1280     assertResult("TRUE", stat, "select '(1,3]':numrange @> '(1,3]':numrange");
1281
1282     conn.close();
1283 }
1284
1285 public void testContainsRangeWithEmpty() throws SQLException {
1286     deleteDb("postgres");
1287     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1288     Statement stat = conn.createStatement();
1289
1290     assertResult("FALSE", stat, "select 'empty':int4range @> '[3,5]':int4range");
1291     assertResult("TRUE", stat, "select '[3,5]':int4range @> 'empty':int4range");
1292
1293     conn.close();
1294 }
1295
1296 public void testContainsRangeWithInfinity() throws SQLException {
1297     deleteDb("postgres");
1298     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1299     Statement stat = conn.createStatement();
1300
1301     assertResult("TRUE", stat, "select '(,)':int4range @> '[3,5]':int4range");
1302     assertResult("FALSE", stat, "select '[3,5]':int4range @> '(,)':int4range");
1303
1304     conn.close();
1305 }
1306
1307 public void testContainsRangeIdentifierFunction() throws SQLException {
1308     deleteDb("postgres");
1309     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1310     Statement stat = conn.createStatement();
1311

```



```

1312     assertResult("FALSE", stat, "select int4range(1,3) @> '[2,5]':int4range");
1313     assertResult("TRUE", stat, "select int4range(1,3) @> '[1,3]':int4range");
1314
1315     conn.close();
1316 }
1317
1318 public void testContainsRangeIdentifierDot() throws SQLException {
1319     deleteDb("postgres");
1320     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1321     Statement stat = conn.createStatement();
1322
1323     stat.execute("create table testTb (testKey integer primary key, testCol int4range)");
1324     stat.execute("insert into testTb values (1, '[1,3]':int4range)");
1325     assertResult("FALSE", stat, "select testSq.testCol @> '[2,5]':int4range from (select
tb.testCol from testTb tb where tb.testKey = 1) as testSq");
1326     assertResult("TRUE", stat, "select testSq.testCol @> '[1,3]':int4range from (select
tb.testCol from testTb tb where tb.testKey = 1) as testSq");
1327
1328     conn.close();
1329 }
1330
1331 public void testContainsRangeIdentifierSubquery() throws SQLException {
1332     deleteDb("postgres");
1333     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1334     Statement stat = conn.createStatement();
1335
1336     assertResult("FALSE", stat, "select testSq @> '[2,5]':int4range from (select
'[1,3]':int4range as testSq)");
1337     assertResult("TRUE", stat, "select testSq @> '[1,3]':int4range from (select
'[1,3]':int4range as testSq)");
1338
1339     conn.close();
1340 }
1341
1342 public void testContainsRangeNamelessSubquery() throws SQLException {
1343     deleteDb("postgres");
1344     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1345     Statement stat = conn.createStatement();
1346
1347     assertResult("FALSE", stat, "select (select '[1,3]':int4range) @> '[2,5]':int4range");
1348     assertResult("TRUE", stat, "select (select '[1,3]':int4range) @> '[1,3]':int4range");
1349
1350     conn.close();
1351 }
1352
1353 public void testContainsElement() throws SQLException {
1354     testContainsElementLiteralCanonicalBorders();
1355     testContainsElementLiteralNotCanonicalBorders();
1356     testContainsElementWithEmpty();
1357     testContainsElementWithInfinity();
1358     testContainsElementIdentifierFunction();
1359     testContainsElementIdentifierDot();
1360     testContainsElementIdentifierSubquery();
1361     testContainsElementNamelessSubquery();
1362 }
1363
1364 public void testContainsElementLiteralCanonicalBorders() throws SQLException {
1365     deleteDb("postgres");
1366     Connection conn = getConnection("postgres;MODE=PostgreSQL");

```

```
1367     Statement stat = conn.createStatement();
1368
1369     assertResult("FALSE", stat, "select '[1,2]::int4range @> 0");
1370     assertResult("TRUE", stat, "select '[1,2]::int4range @> 1");
1371     assertResult("FALSE", stat, "select '[1,2]::int4range @> 2");
1372     assertResult("FALSE", stat, "select '[1,2]::int4range @> 3");
1373
1374     conn.close();
1375 }
1376
1377 public void testContainsElementLiteralNotCanonicalBorders() throws SQLException {
1378     deleteDb("postgres");
1379     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1380     Statement stat = conn.createStatement();
1381
1382     assertResult("FALSE", stat, "select '[1,2]::numrange @> 0");
1383     assertResult("TRUE", stat, "select '[1,2]::numrange @> 1");
1384     assertResult("TRUE", stat, "select '[1,2]::numrange @> 2");
1385     assertResult("FALSE", stat, "select '[1,2]::numrange @> 3");
1386
1387     conn.close();
1388 }
1389
1390 public void testContainsElementWithEmpty() throws SQLException {
1391     deleteDb("postgres");
1392     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1393     Statement stat = conn.createStatement();
1394
1395     assertResult("FALSE", stat, "select 'empty'::int4range @> 1");
1396
1397     conn.close();
1398 }
1399
1400 public void testContainsElementWithInfinity() throws SQLException {
1401     deleteDb("postgres");
1402     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1403     Statement stat = conn.createStatement();
1404
1405     assertResult("TRUE", stat, "select '(,)'::int4range @> 1");
1406
1407     conn.close();
1408 }
1409
1410 public void testContainsElementIdentifierFunction() throws SQLException {
1411     deleteDb("postgres");
1412     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1413     Statement stat = conn.createStatement();
1414
1415     assertResult("FALSE", stat, "select int4range(1,3) @> 0");
1416     assertResult("TRUE", stat, "select int4range(1,3) @> 1");
1417
1418     conn.close();
1419 }
1420
1421 public void testContainsElementIdentifierDot() throws SQLException {
1422     deleteDb("postgres");
1423     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1424     Statement stat = conn.createStatement();
1425 }
```

```

1426     stat.execute("create table testTb (testKey integer primary key, testCol int4range)");
1427     stat.execute("insert into testTb values (1, '[1,3]::int4range)");
1428     assertResult("FALSE", stat, "select testSq.testCol @> 0 from (select tb.testCol from
testTb tb where tb.testKey = 1) as testSq");
1429     assertResult("TRUE", stat, "select testSq.testCol @> 1 from (select tb.testCol from
testTb tb where tb.testKey = 1) as testSq");

1430
1431     conn.close();
1432 }
1433
1434 public void testContainsElementIdentifierSubquery() throws SQLException {
1435     deleteDb("postgres");
1436     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1437     Statement stat = conn.createStatement();

1438     assertResult("FALSE", stat, "select testSq @> 0 from (select '[1,3]::int4range as
testSq)");
1439     assertResult("TRUE", stat, "select testSq @> 1 from (select '[1,3]::int4range as
testSq)");
1440
1441     conn.close();
1442 }
1443
1444 public void testContainsElementNamelessSubquery() throws SQLException {
1445     deleteDb("postgres");
1446     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1447     Statement stat = conn.createStatement();
1448
1449     assertResult("FALSE", stat, "select (select '[1,3]::int4range) @> 0");
1450     assertResult("TRUE", stat, "select (select '[1,3]::int4range) @> 1");
1451
1452     conn.close();
1453 }
1454
1455 public void testContainedRange() throws SQLException {
1456     testContainedRangeLiteralCanonicalBorders();
1457     testContainedRangeLiteralNotCanonicalBorders();
1458     testContainedRangeWithEmpty();
1459     testContainedRangeWithInfinity();
1460     testContainedRangeIdentifierFunction();
1461     testContainedRangeIdentifierDot();
1462     testContainedRangeIdentifierSubquery();
1463     testContainedRangeNamelessSubquery();
1464 }
1465
1466 public void testContainedRangeLiteralCanonicalBorders() throws SQLException {
1467     deleteDb("postgres");
1468     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1469     Statement stat = conn.createStatement();
1470
1471     assertResult("TRUE", stat, "select '[1,4]::int4range <@ '[0,5]::int4range");
1472     assertResult("TRUE", stat, "select '[1,4]::int4range <@ '[0,4]::int4range");
1473     assertResult("TRUE", stat, "select '[1,4]::int4range <@ '[1,5]::int4range");
1474     assertResult("TRUE", stat, "select '[1,4]::int4range <@ '[1,4]::int4range");
1475     assertResult("FALSE", stat, "select '[1,4]::int4range <@ '[1,3]::int4range");
1476     assertResult("FALSE", stat, "select '[1,4]::int4range <@ '[2,4]::int4range");
1477     assertResult("FALSE", stat, "select '[1,4]::int4range <@ '[2,3]::int4range");
1478
1479     conn.close();
1480

```

```

1481     }
1482
1483     public void testContainedRangeLiteralNotCanonicalBorders() throws SQLException {
1484         deleteDb("postgres");
1485         Connection conn = getConnection("postgres;MODE=PostgreSQL");
1486         Statement stat = conn.createStatement();
1487
1488         assertResult("TRUE", stat, "select '[1,3]::numrange <@ '[1,3]::numrange)");
1489         assertResult("FALSE", stat, "select '[1,3]::numrange <@ '[1,3]::numrange)");
1490         assertResult("FALSE", stat, "select '[1,3]::numrange <@ '(1,3]::numrange)");
1491         assertResult("FALSE", stat, "select '[1,3]::numrange <@ '(1,3)::numrange)");
1492         assertResult("TRUE", stat, "select '[1,3]::numrange <@ '[1,3]::numrange)");
1493         assertResult("TRUE", stat, "select '[1,3]::numrange <@ '(1,3]::numrange)");
1494         assertResult("FALSE", stat, "select '[1,3]::numrange <@ '(1,3)::numrange)");
1495         assertResult("FALSE", stat, "select '[1,3]::numrange <@ '(1,3]::numrange)");
1496         assertResult("TRUE", stat, "select '(1,3]::numrange <@ '[1,3]::numrange)");
1497         assertResult("FALSE", stat, "select '(1,3]::numrange <@ '[1,3]::numrange)");
1498         assertResult("TRUE", stat, "select '(1,3]::numrange <@ '(1,3]::numrange)");
1499         assertResult("FALSE", stat, "select '(1,3]::numrange <@ '(1,3)::numrange)");
1500         assertResult("TRUE", stat, "select '(1,3)::numrange <@ '[1,3]::numrange)");
1501         assertResult("TRUE", stat, "select '(1,3)::numrange <@ '[1,3]::numrange)");
1502         assertResult("TRUE", stat, "select '(1,3)::numrange <@ '(1,3]::numrange)");
1503         assertResult("TRUE", stat, "select '(1,3)::numrange <@ '(1,3)::numrange)");
1504
1505         conn.close();
1506     }
1507
1508     public void testContainedRangeWithEmpty() throws SQLException {
1509         deleteDb("postgres");
1510         Connection conn = getConnection("postgres;MODE=PostgreSQL");
1511         Statement stat = conn.createStatement();
1512
1513         assertResult("TRUE", stat, "select 'empty'::int4range <@ '[3,5]::int4range)");
1514         assertResult("FALSE", stat, "select '[3,5]::int4range <@ 'empty'::int4range)");
1515
1516         conn.close();
1517     }
1518
1519     public void testContainedRangeWithInfinity() throws SQLException {
1520         deleteDb("postgres");
1521         Connection conn = getConnection("postgres;MODE=PostgreSQL");
1522         Statement stat = conn.createStatement();
1523
1524         assertResult("FALSE", stat, "select '(,)::int4range <@ '[3,5]::int4range)");
1525         assertResult("TRUE", stat, "select '[3,5]::int4range <@ '(,)::int4range)");
1526
1527         conn.close();
1528     }
1529
1530     public void testContainedRangeIdentifierFunction() throws SQLException {
1531         deleteDb("postgres");
1532         Connection conn = getConnection("postgres;MODE=PostgreSQL");
1533         Statement stat = conn.createStatement();
1534
1535         assertResult("FALSE", stat, "select int4range(1,3) <@ '[2,5]::int4range)");
1536         assertResult("TRUE", stat, "select int4range(1,3) <@ '[1,3]::int4range)");
1537
1538         conn.close();
1539     }

```

```

1540
1541 public void testContainedRangeIdentifierDot() throws SQLException {
1542     deleteDb("postgres");
1543     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1544     Statement stat = conn.createStatement();
1545
1546     stat.execute("create table testTb (testKey integer primary key, testCol int4range)");
1547     stat.execute("insert into testTb values (1, '[1,3]::int4range)");
1548     assertResult("FALSE", stat, "select testSq.testCol <@ '[2,5]::int4range from (select
tb.testCol from testTb tb where tb.testKey = 1) as testSq");
1549     assertResult("TRUE", stat, "select testSq.testCol <@ '[1,3]::int4range from (select
tb.testCol from testTb tb where tb.testKey = 1) as testSq");
1550
1551     conn.close();
1552 }
1553
1554 public void testContainedRangeIdentifierSubquery() throws SQLException {
1555     deleteDb("postgres");
1556     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1557     Statement stat = conn.createStatement();
1558
1559     assertResult("FALSE", stat, "select testSq <@ '[2,5]::int4range from (select
'[1,3]::int4range as testSq)");
1560     assertResult("TRUE", stat, "select testSq <@ '[1,3]::int4range from (select
'[1,3]::int4range as testSq)");
1561
1562     conn.close();
1563 }
1564
1565 public void testContainedRangeNamelessSubquery() throws SQLException {
1566     deleteDb("postgres");
1567     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1568     Statement stat = conn.createStatement();
1569
1570     assertResult("FALSE", stat, "select (select '[1,3]::int4range) <@ '[2,5]::int4range");
1571     assertResult("TRUE", stat, "select (select '[1,3]::int4range) <@ '[1,3]::int4range");
1572
1573     conn.close();
1574 }
1575
1576 public void testContainedElement() throws SQLException {
1577     testContainedElementLiteralCanonicalBorders();
1578     testContainedElementLiteralNotCanonicalBorders();
1579     testContainedElementWithEmpty();
1580     testContainedElementWithInfinity();
1581     testContainedElementIdentifierFunction();
1582     testContainedElementIdentifierDot();
1583     testContainedElementIdentifierSubquery();
1584     testContainedElementNamelessSubquery();
1585 }
1586
1587 public void testContainedElementLiteralCanonicalBorders() throws SQLException {
1588     deleteDb("postgres");
1589     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1590     Statement stat = conn.createStatement();
1591
1592     assertResult("FALSE", stat, "select 0 <@ '[1,2]::int4range");
1593     assertResult("TRUE", stat, "select 1 <@ '[1,2]::int4range");
1594     assertResult("FALSE", stat, "select 2 <@ '[1,2]::int4range");

```

```

1595     assertResult("FALSE", stat, "select 3 <@ '[1,2]':int4range");
1596
1597     conn.close();
1598 }
1599
1600 public void testContainedElementLiteralNotCanonicalBorders() throws SQLException {
1601     deleteDb("postgres");
1602     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1603     Statement stat = conn.createStatement();
1604
1605     assertResult("FALSE", stat, "select 0 <@ '[1,2]':numrange");
1606     assertResult("TRUE", stat, "select 1 <@ '[1,2]':numrange");
1607     assertResult("TRUE", stat, "select 2 <@ '[1,2]':numrange");
1608     assertResult("FALSE", stat, "select 3 <@ '[1,2]':numrange");
1609
1610     conn.close();
1611 }
1612
1613 public void testContainedElementWithEmpty() throws SQLException {
1614     deleteDb("postgres");
1615     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1616     Statement stat = conn.createStatement();
1617
1618     assertResult("FALSE", stat, "select 1 <@ 'empty':int4range");
1619
1620     conn.close();
1621 }
1622
1623 public void testContainedElementWithInfinity() throws SQLException {
1624     deleteDb("postgres");
1625     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1626     Statement stat = conn.createStatement();
1627
1628     assertResult("TRUE", stat, "select 1 <@ '(,)':int4range");
1629
1630     conn.close();
1631 }
1632
1633 public void testContainedElementIdentifierFunction() throws SQLException {
1634     deleteDb("postgres");
1635     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1636     Statement stat = conn.createStatement();
1637
1638     assertResult("FALSE", stat, "select 0 <@ int4range(1,3)");
1639     assertResult("TRUE", stat, "select 1 <@ int4range(1,3)");
1640
1641     conn.close();
1642 }
1643
1644 public void testContainedElementIdentifierDot() throws SQLException {
1645     deleteDb("postgres");
1646     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1647     Statement stat = conn.createStatement();
1648
1649     stat.execute("create table testTb (testKey integer primary key, testCol integer)");
1650     stat.execute("insert into testTb values (1, 3)");
1651     assertResult("FALSE", stat, "select testSq.testCol <@ '[1,2]':int4range from (select
tb.testCol from testTb tb where tb.testKey = 1) as testSq");
1652     assertResult("TRUE", stat, "select testSq.testCol <@ '[1,5]':int4range from (select

```

```

tb.testCol from testTb tb where tb.testKey = 1) as testSq");
1653     conn.close();
1654 }
1655
1656 public void testContainedElementIdentifierSubquery() throws SQLException {
1657     deleteDb("postgres");
1658     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1659     Statement stat = conn.createStatement();
1660
1661     assertResult("FALSE", stat, "select testSq <@ '[1,2]':int4range from (select 3 as
1662 testSq)");
1663     assertResult("TRUE", stat, "select testSq <@ '[1,5]':int4range from (select 3 as
1664 testSq)");
1665     conn.close();
1666 }
1667
1668 public void testContainedElementNamelessSubquery() throws SQLException {
1669     deleteDb("postgres");
1670     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1671     Statement stat = conn.createStatement();
1672
1673     assertResult("FALSE", stat, "select (select 3) <@ '[1,2]':int4range");
1674     assertResult("TRUE", stat, "select (select 3) <@ '[1,5]':int4range");
1675
1676     conn.close();
1677 }
1678
1679 public void testStrictLeft() throws SQLException {
1680     testStrictLeftLiteralCanonicalBorders();
1681     testStrictLeftLiteralNotCanonicalBorders();
1682     testStrictLeftWithEmpty();
1683     testStrictLeftWithInfinity();
1684     testStrictLeftIdentifierFunction();
1685     testStrictLeftIdentifierDot();
1686     testStrictLeftIdentifierSubquery();
1687     testStrictLeftNamelessSubquery();
1688 }
1689
1690 public void testStrictLeftLiteralCanonicalBorders() throws SQLException {
1691     deleteDb("postgres");
1692     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1693     Statement stat = conn.createStatement();
1694
1695     assertResult("TRUE", stat, "select '[1,3]':int4range << '[4,5]':int4range");
1696     assertResult("TRUE", stat, "select '[1,3]':int4range << '[3,5]':int4range");
1697     assertResult("FALSE", stat, "select '[1,3]':int4range << '[2,5]':int4range");
1698
1699     conn.close();
1700 }
1701
1702 public void testStrictLeftLiteralNotCanonicalBorders() throws SQLException {
1703     deleteDb("postgres");
1704     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1705     Statement stat = conn.createStatement();
1706
1707     assertResult("TRUE", stat, "select '[1,3]':numrange << '[4,5]':numrange");
1708     assertResult("FALSE", stat, "select '[1,3]':numrange << '[3,5]':numrange");

```

```

1709     assertResult("FALSE", stat, "select '[1,3]::numrange << '[2,5]::numrange");
1710     assertResult("TRUE", stat, "select '[1,3]::numrange << '[4,5]::numrange");
1711     assertResult("TRUE", stat, "select '[1,3]::numrange << '[3,5]::numrange");
1712     assertResult("FALSE", stat, "select '[1,3]::numrange << '(2,5]::numrange");
1713     assertResult("TRUE", stat, "select '[1,3]::numrange << '(4,5]::numrange");
1714     assertResult("TRUE", stat, "select '[1,3]::numrange << '(3,5]::numrange");
1715     assertResult("FALSE", stat, "select '[1,3]::numrange << '(2,5]::numrange");
1716
1717     conn.close();
1718 }
1719
1720 public void testStrictLeftWithEmpty() throws SQLException {
1721     deleteDb("postgres");
1722     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1723     Statement stat = conn.createStatement();
1724
1725     assertResult("FALSE", stat, "select 'empty'::int4range << '[3,5]::int4range");
1726     assertResult("FALSE", stat, "select '[3,5]::int4range << 'empty'::int4range");
1727
1728     conn.close();
1729 }
1730
1731 public void testStrictLeftWithInfinity() throws SQLException {
1732     deleteDb("postgres");
1733     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1734     Statement stat = conn.createStatement();
1735
1736     assertResult("FALSE", stat, "select '(,)'::int4range << '[3,5]::int4range");
1737     assertResult("FALSE", stat, "select '[3,5]::int4range << '(,)'::int4range");
1738
1739     conn.close();
1740 }
1741
1742 public void testStrictLeftIdentifierFunction() throws SQLException {
1743     deleteDb("postgres");
1744     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1745     Statement stat = conn.createStatement();
1746
1747     assertResult("FALSE", stat, "select int4range(1,3) << '[2,5]::int4range");
1748     assertResult("TRUE", stat, "select int4range(1,3) << '[3,5]::int4range");
1749
1750     conn.close();
1751 }
1752
1753 public void testStrictLeftIdentifierDot() throws SQLException {
1754     deleteDb("postgres");
1755     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1756     Statement stat = conn.createStatement();
1757
1758     stat.execute("create table testTb (testKey integer primary key, testCol int4range)");
1759     stat.execute("insert into testTb values (1, '[1,3]::int4range)");
1760     assertResult("FALSE", stat, "select testSq.testCol << '[2,5]::int4range from (select
tb.testCol from testTb tb where tb.testKey = 1) as testSq");
1761     assertResult("TRUE", stat, "select testSq.testCol << '[3,5]::int4range from (select
tb.testCol from testTb tb where tb.testKey = 1) as testSq");
1762
1763     conn.close();
1764 }
1765

```



```

1766 public void testStrictLeftIdentifierSubquery() throws SQLException {
1767     deleteDb("postgres");
1768     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1769     Statement stat = conn.createStatement();
1770
1771     assertResult("FALSE", stat, "select testSq << '[2,5]':int4range from (select
1772 '[1,3]':int4range as testSq)");
1773     assertResult("TRUE", stat, "select testSq << '[3,5]':int4range from (select
1774 '[1,3]':int4range as testSq)");
1775
1776     conn.close();
1777 }
1778
1779 public void testStrictLeftNamelessSubquery() throws SQLException {
1780     deleteDb("postgres");
1781     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1782     Statement stat = conn.createStatement();
1783
1784     assertResult("FALSE", stat, "select (select '[1,3]':int4range) << '[2,5]':int4range");
1785     assertResult("TRUE", stat, "select (select '[1,3]':int4range) << '[3,5]':int4range");
1786
1787     conn.close();
1788 }
1789
1790 public void testStrictRight() throws SQLException {
1791     testStrictRightLiteralCanonicalBorders();
1792     testStrictRightLiteralNotCanonicalBorders();
1793     testStrictRightWithEmpty();
1794     testStrictRightWithInfinity();
1795     testStrictRightIdentifierFunction();
1796     testStrictRightIdentifierDot();
1797     testStrictRightIdentifierSubquery();
1798     testStrictRightNamelessSubquery();
1799 }
1800
1801 public void testStrictRightLiteralCanonicalBorders() throws SQLException {
1802     deleteDb("postgres");
1803     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1804     Statement stat = conn.createStatement();
1805
1806     assertResult("FALSE", stat, "select '[1,3]':int4range >> '[-1,2]':int4range");
1807     assertResult("TRUE", stat, "select '[1,3]':int4range >> '[-1,1]':int4range");
1808     assertResult("TRUE", stat, "select '[1,3]':int4range >> '[-1,0]':int4range");
1809
1810     conn.close();
1811 }
1812
1813 public void testStrictRightLiteralNotCanonicalBorders() throws SQLException {
1814     deleteDb("postgres");
1815     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1816     Statement stat = conn.createStatement();
1817
1818     assertResult("FALSE", stat, "select '[1,3]':numrange >> '[-1,2]':numrange");
1819     assertResult("FALSE", stat, "select '[1,3]':numrange >> '[-1,1]':numrange");
1820     assertResult("TRUE", stat, "select '[1,3]':numrange >> '[-1,0]':numrange");
1821     assertResult("FALSE", stat, "select '[1,3]':numrange >> '[-1,2]':numrange");
1822     assertResult("TRUE", stat, "select '[1,3]':numrange >> '[-1,1]':numrange");
1823     assertResult("TRUE", stat, "select '[1,3]':numrange >> '[-1,0]':numrange");
1824     assertResult("FALSE", stat, "select '(1,3]':numrange >> '[-1,2]':numrange");

```

```

1823     assertResult("TRUE", stat, "select '(1,3)::numrange >> '[-1,1)::numrange");
1824     assertResult("TRUE", stat, "select '(1,3)::numrange >> '[-1,0)::numrange");
1825
1826     conn.close();
1827 }
1828
1829 public void testStrictRightWithEmpty() throws SQLException {
1830     deleteDb("postgres");
1831     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1832     Statement stat = conn.createStatement();
1833
1834     assertResult("FALSE", stat, "select 'empty'::int4range >> '[-1,1)::int4range");
1835     assertResult("FALSE", stat, "select '[1,3)::int4range >> 'empty'::int4range");
1836
1837     conn.close();
1838 }
1839
1840 public void testStrictRightWithInfinity() throws SQLException {
1841     deleteDb("postgres");
1842     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1843     Statement stat = conn.createStatement();
1844
1845     assertResult("FALSE", stat, "select '(,)::int4range >> '[-1,1)::int4range");
1846     assertResult("FALSE", stat, "select '[1,3)::int4range >> '(,)::int4range");
1847
1848     conn.close();
1849 }
1850
1851 public void testStrictRightIdentifierFunction() throws SQLException {
1852     deleteDb("postgres");
1853     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1854     Statement stat = conn.createStatement();
1855
1856     assertResult("FALSE", stat, "select int4range(1,3) >> '[-1,2)::int4range");
1857     assertResult("TRUE", stat, "select int4range(1,3) >> '[-1,1)::int4range");
1858
1859     conn.close();
1860 }
1861
1862 public void testStrictRightIdentifierDot() throws SQLException {
1863     deleteDb("postgres");
1864     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1865     Statement stat = conn.createStatement();
1866
1867     stat.execute("create table testTb (testKey integer primary key, testCol int4range)");
1868     stat.execute("insert into testTb values (1, '[1,3)::int4range)");
1869     assertResult("FALSE", stat, "select testSq.testCol >> '[-1,2)::int4range from (select
1870     tb.testCol from testTb tb where tb.testKey = 1) as testSq");
1871     assertResult("TRUE", stat, "select testSq.testCol >> '[-1,1)::int4range from (select
1872     tb.testCol from testTb tb where tb.testKey = 1) as testSq");
1873
1874     conn.close();
1875 }
1876
1877 public void testStrictRightIdentifierSubquery() throws SQLException {
1878     deleteDb("postgres");
1879     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1880     Statement stat = conn.createStatement();

```

```

1880     assertResult("FALSE", stat, "select testSq >> '[-1,2)::int4range from (select
1881 '[1,3)::int4range as testSq)");
1882     assertResult("TRUE", stat, "select testSq >> '[-1,1)::int4range from (select
1883 '[1,3)::int4range as testSq)");
1884     conn.close();
1885 }
1886 public void testStrictRightNamelessSubquery() throws SQLException {
1887     deleteDb("postgres");
1888     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1889     Statement stat = conn.createStatement();
1890
1891     assertResult("FALSE", stat, "select (select '[1,3)::int4range) >> '[-1,2)::int4range");
1892     assertResult("TRUE", stat, "select (select '[1,3)::int4range) >> '[-1,1)::int4range");
1893
1894     conn.close();
1895 }
1896
1897 public void testNotExtendRight() throws SQLException {
1898     testNotExtendRightLiteralCanonicalBorders();
1899     testNotExtendRightLiteralNotCanonicalBorders();
1900     testNotExtendRightWithEmpty();
1901     testNotExtendRightWithInfinity();
1902     testNotExtendRightIdentifierFunction();
1903     testNotExtendRightIdentifierDot();
1904     testNotExtendRightIdentifierSubquery();
1905     testNotExtendRightNamelessSubquery();
1906 }
1907
1908 public void testNotExtendRightLiteralCanonicalBorders() throws SQLException {
1909     deleteDb("postgres");
1910     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1911     Statement stat = conn.createStatement();
1912
1913     assertResult("TRUE", stat, "select '[1,3)::int4range &< '[1,4)::int4range");
1914     assertResult("TRUE", stat, "select '[1,3)::int4range &< '[1,3)::int4range");
1915     assertResult("FALSE", stat, "select '[1,3)::int4range &< '[1,2)::int4range");
1916
1917     conn.close();
1918 }
1919
1920 public void testNotExtendRightLiteralNotCanonicalBorders() throws SQLException {
1921     deleteDb("postgres");
1922     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1923     Statement stat = conn.createStatement();
1924
1925     assertResult("TRUE", stat, "select '[1,3)::numrange &< '[1,3)::numrange");
1926     assertResult("TRUE", stat, "select '[1,3)::numrange &< '[1,3)::numrange");
1927     assertResult("FALSE", stat, "select '[1,3)::numrange &< '[1,3)::numrange");
1928
1929     conn.close();
1930 }
1931
1932 public void testNotExtendRightWithEmpty() throws SQLException {
1933     deleteDb("postgres");
1934     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1935     Statement stat = conn.createStatement();
1936

```

```

1937     assertResult("FALSE", stat, "select 'empty'::int4range &< '[1,3]':int4range");
1938     assertResult("FALSE", stat, "select '[1,3]':int4range &< 'empty'::int4range");
1939
1940     conn.close();
1941 }
1942
1943 public void testNotExtendRightWithInfinity() throws SQLException {
1944     deleteDb("postgres");
1945     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1946     Statement stat = conn.createStatement();
1947
1948     assertResult("FALSE", stat, "select '(,)'::int4range &< '[1,3]':int4range");
1949     assertResult("TRUE", stat, "select '[1,3]':int4range &< '(,)'::int4range");
1950
1951     conn.close();
1952 }
1953
1954 public void testNotExtendRightIdentifierFunction() throws SQLException {
1955     deleteDb("postgres");
1956     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1957     Statement stat = conn.createStatement();
1958
1959     assertResult("FALSE", stat, "select int4range(1,3) &< '[1,2]':int4range");
1960     assertResult("TRUE", stat, "select int4range(1,3) &< '[1,3]':int4range");
1961
1962     conn.close();
1963 }
1964
1965 public void testNotExtendRightIdentifierDot() throws SQLException {
1966     deleteDb("postgres");
1967     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1968     Statement stat = conn.createStatement();
1969
1970     stat.execute("create table testTb (testKey integer primary key, testCol int4range)");
1971     stat.execute("insert into testTb values (1, '[1,3]':int4range)");
1972     assertResult("FALSE", stat, "select testSq.testCol &< '[1,2]':int4range from (select
tb.testCol from testTb tb where tb.testKey = 1) as testSq");
1973     assertResult("TRUE", stat, "select testSq.testCol &< '[1,3]':int4range from (select
tb.testCol from testTb tb where tb.testKey = 1) as testSq");
1974
1975     conn.close();
1976 }
1977
1978 public void testNotExtendRightIdentifierSubquery() throws SQLException {
1979     deleteDb("postgres");
1980     Connection conn = getConnection("postgres;MODE=PostgreSQL");
1981     Statement stat = conn.createStatement();
1982
1983     assertResult("FALSE", stat, "select testSq &< '[1,2]':int4range from (select
'[1,3]':int4range as testSq)");
1984     assertResult("TRUE", stat, "select testSq &< '[1,3]':int4range from (select
'[1,3]':int4range as testSq)");
1985
1986     conn.close();
1987 }
1988
1989 public void testNotExtendRightNamelessSubquery() throws SQLException {
1990     deleteDb("postgres");
1991     Connection conn = getConnection("postgres;MODE=PostgreSQL");

```

```

1992     Statement stat = conn.createStatement();
1993
1994     assertResult("FALSE", stat, "select (select '[1,3]':int4range) &< '[1,2]':int4range");
1995     assertResult("TRUE", stat, "select (select '[1,3]':int4range) &< '[1,3]':int4range");
1996
1997     conn.close();
1998 }
1999
2000 public void testNotExtendLeft() throws SQLException {
2001     testNotExtendLeftLiteralCanonicalBorders();
2002     testNotExtendLeftLiteralNotCanonicalBorders();
2003     testNotExtendLeftWithEmpty();
2004     testNotExtendLeftWithInfinity();
2005     testNotExtendLeftIdentifierFunction();
2006     testNotExtendLeftIdentifierDot();
2007     testNotExtendLeftIdentifierSubquery();
2008     testNotExtendLeftNamelessSubquery();
2009 }
2010
2011 public void testNotExtendLeftLiteralCanonicalBorders() throws SQLException {
2012     deleteDb("postgres");
2013     Connection conn = getConnection("postgres;MODE=PostgreSQL");
2014     Statement stat = conn.createStatement();
2015
2016     assertResult("TRUE", stat, "select '[1,3]':int4range &> '[0,5]':int4range");
2017     assertResult("TRUE", stat, "select '[1,3]':int4range &> '[1,5]':int4range");
2018     assertResult("FALSE", stat, "select '[1,3]':int4range &> '[2,5]':int4range");
2019
2020     conn.close();
2021 }
2022
2023 public void testNotExtendLeftLiteralNotCanonicalBorders() throws SQLException {
2024     deleteDb("postgres");
2025     Connection conn = getConnection("postgres;MODE=PostgreSQL");
2026     Statement stat = conn.createStatement();
2027
2028     assertResult("FALSE", stat, "select '[1,3]':numrange &> '(1,5]':numrange");
2029     assertResult("TRUE", stat, "select '(1,3]':numrange &> '[1,5]':numrange");
2030     assertResult("TRUE", stat, "select '(1,3]':numrange &> '(1,5]':numrange");
2031
2032     conn.close();
2033 }
2034
2035 public void testNotExtendLeftWithEmpty() throws SQLException {
2036     deleteDb("postgres");
2037     Connection conn = getConnection("postgres;MODE=PostgreSQL");
2038     Statement stat = conn.createStatement();
2039
2040     assertResult("FALSE", stat, "select 'empty':int4range &> '[1,3]':int4range");
2041     assertResult("FALSE", stat, "select '[1,3]':int4range &> 'empty':int4range");
2042
2043     conn.close();
2044 }
2045
2046 public void testNotExtendLeftWithInfinity() throws SQLException {
2047     deleteDb("postgres");
2048     Connection conn = getConnection("postgres;MODE=PostgreSQL");
2049     Statement stat = conn.createStatement();
2050

```

```

2051     assertResult("FALSE", stat, "select '(,)'::int4range &> '[1,3]'::int4range");
2052     assertResult("TRUE", stat, "select '[1,3]'::int4range &> '(,)'::int4range");
2053
2054     conn.close();
2055 }
2056
2057 public void testNotExtendLeftIdentifierFunction() throws SQLException {
2058     deleteDb("postgres");
2059     Connection conn = getConnection("postgres;MODE=PostgreSQL");
2060     Statement stat = conn.createStatement();
2061
2062     assertResult("FALSE", stat, "select int4range(1,3) &> '[2,5]'::int4range");
2063     assertResult("TRUE", stat, "select int4range(1,3) &> '[1,5]'::int4range");
2064
2065     conn.close();
2066 }
2067
2068 public void testNotExtendLeftIdentifierDot() throws SQLException {
2069     deleteDb("postgres");
2070     Connection conn = getConnection("postgres;MODE=PostgreSQL");
2071     Statement stat = conn.createStatement();
2072
2073     stat.execute("create table testTb (testKey integer primary key, testCol int4range)");
2074     stat.execute("insert into testTb values (1, '[1,3]'::int4range)");
2075     assertResult("FALSE", stat, "select testSq.testCol &> '[2,5]'::int4range from (select
2076     tb.testCol from testTb tb where tb.testKey = 1) as testSq");
2077     assertResult("TRUE", stat, "select testSq.testCol &> '[1,5]'::int4range from (select
2078     tb.testCol from testTb tb where tb.testKey = 1) as testSq");
2079
2080     conn.close();
2081 }
2082
2083 public void testNotExtendLeftIdentifierSubquery() throws SQLException {
2084     deleteDb("postgres");
2085     Connection conn = getConnection("postgres;MODE=PostgreSQL");
2086     Statement stat = conn.createStatement();
2087
2088     assertResult("FALSE", stat, "select testSq &> '[2,5]'::int4range from (select
2089     '[1,3]'::int4range as testSq)");
2090     assertResult("TRUE", stat, "select testSq &> '[1,5]'::int4range from (select
2091     '[1,3]'::int4range as testSq)");
2092
2093     conn.close();
2094 }
2095
2096 public void testNotExtendLeftNamelessSubquery() throws SQLException {
2097     deleteDb("postgres");
2098     Connection conn = getConnection("postgres;MODE=PostgreSQL");
2099     Statement stat = conn.createStatement();
2100
2101     assertResult("FALSE", stat, "select (select '[1,3]'::int4range) &> '[2,5]'::int4range");
2102     assertResult("TRUE", stat, "select (select '[1,3]'::int4range) &> '[1,5]'::int4range");
2103
2104     conn.close();
2105 }
2106
2107 public void testSumPriorityUnion() throws SQLException {
2108     deleteDb("postgres");
2109     Connection conn = getConnection("postgres;MODE=PostgreSQL");

```

```

2106     Statement stat = conn.createStatement();
2107
2108     assertResult("FALSE", stat, "select '[1,2]::int4range -|-[3,5)::int4range +
'[4,6)::int4range");
2109     assertResult("FALSE", stat, "select '[1,2]::int4range && [3,5)::int4range +
'[4,6)::int4range");
2110     assertResult("FALSE", stat, "select '[1,2]::int4range @> [3,5)::int4range +
'[4,6)::int4range");
2111     assertResult("FALSE", stat, "select '[1,2]::int4range <@ [3,5)::int4range +
'[4,6)::int4range");
2112     assertResult("TRUE", stat, "select '[1,2]::int4range << [3,5)::int4range +
'[4,6)::int4range");
2113     assertResult("FALSE", stat, "select '[1,2]::int4range >> [3,5)::int4range +
'[4,6)::int4range");
2114     assertResult("TRUE", stat, "select '[1,2]::int4range &< [3,5)::int4range +
'[4,6)::int4range");
2115     assertResult("FALSE", stat, "select '[1,2]::int4range &> [3,5)::int4range +
'[4,6)::int4range");
2116
2117     conn.close();
2118 }
2119
2120 public void testSumPriorityDifference() throws SQLException {
2121     deleteDb("postgres");
2122     Connection conn = getConnection("postgres;MODE=PostgreSQL");
2123     Statement stat = conn.createStatement();
2124
2125     assertResult("FALSE", stat, "select '[1,2]::int4range -|-[3,5)::int4range -
'[4,6)::int4range");
2126     assertResult("FALSE", stat, "select '[1,2]::int4range && [3,5)::int4range -
'[4,6)::int4range");
2127     assertResult("FALSE", stat, "select '[1,2]::int4range @> [3,5)::int4range -
'[4,6)::int4range");
2128     assertResult("FALSE", stat, "select '[1,2]::int4range <@ [3,5)::int4range -
'[4,6)::int4range");
2129     assertResult("TRUE", stat, "select '[1,2]::int4range << [3,5)::int4range -
'[4,6)::int4range");
2130     assertResult("FALSE", stat, "select '[1,2]::int4range >> [3,5)::int4range -
'[4,6)::int4range");
2131     assertResult("TRUE", stat, "select '[1,2]::int4range &< [3,5)::int4range -
'[4,6)::int4range");
2132     assertResult("FALSE", stat, "select '[1,2]::int4range &> [3,5)::int4range -
'[4,6)::int4range");
2133
2134     conn.close();
2135 }
2136
2137 public void testSumChain() throws SQLException {
2138     deleteDb("postgres");
2139     Connection conn = getConnection("postgres;MODE=PostgreSQL");
2140     Statement stat = conn.createStatement();
2141
2142     assertResult("[1,6)", stat, "select '[1,3]::int4range + [2,5)::int4range +
'[4,6)::int4range");
2143     assertResult("[4,7)", stat, "select '[3,8]::int4range - [1,4)::int4range - [7,9)::int4range");
2144     assertResult("[4,6)", stat, "select '[3,5)::int4range - [1,4)::int4range +
'[5,6)::int4range");
2145     assertResult("[4,6)", stat, "select '[3,5)::int4range + [5,6)::int4range -
'[1,4)::int4range");

```

```

2146     conn.close();
2147 }
2148
2149 public void testFactorPriority() throws SQLException {
2150     deleteDb("postgres");
2151     Connection conn = getConnection("postgres;MODE=PostgreSQL");
2152     Statement stat = conn.createStatement();
2153
2154     assertResult("[3,5]", stat, "select '[3,4]::int4range + '[1,5]::int4range *
2155 '[4,6]::int4range");
2156     assertResult("[1,3]", stat, "select '[1,4]::int4range - '[2,5]::int4range * '[3,6]::int4range");
2157
2158     conn.close();
2159 }
2160
2161 public void testFactorChain() throws SQLException {
2162     deleteDb("postgres");
2163     Connection conn = getConnection("postgres;MODE=PostgreSQL");
2164     Statement stat = conn.createStatement();
2165
2166     assertResult("[3,5]", stat, "select '[2,6]::int4range * '[3,7]::int4range * '[1,5]::int4range");
2167
2168     conn.close();
2169 }
2170
2171 public void testLower() throws SQLException {
2172     deleteDb("postgres");
2173     Connection conn = getConnection("postgres;MODE=PostgreSQL");
2174     Statement stat = conn.createStatement();
2175
2176     assertResult("1", stat, "select lower('[1,3]::int4range)");
2177     assertResult(null, stat, "select lower('(,3]::int4range)");
2178     assertResult(null, stat, "select lower('empty'::int4range)");
2179
2180     conn.close();
2181 }
2182
2183 public void testLowerReturnPropagation() throws SQLException {
2184     deleteDb("postgres");
2185     Connection conn = getConnection("postgres;MODE=PostgreSQL");
2186     Statement stat = conn.createStatement();
2187
2188     assertResult("2.0", stat, "select sqrt(lower('[4,6]::int4range)");
2189
2190     conn.close();
2191 }
2192
2193 public void testUpper() throws SQLException {
2194     deleteDb("postgres");
2195     Connection conn = getConnection("postgres;MODE=PostgreSQL");
2196     Statement stat = conn.createStatement();
2197
2198     assertResult("3", stat, "select upper('[1,3]::int4range)");
2199     assertResult(null, stat, "select upper('[1,]::int4range)");
2200     assertResult(null, stat, "select upper('empty'::int4range)");
2201
2202     conn.close();
2203 }

```



```
2204
2205 public void testUpperReturnPropagation() throws SQLException {
2206     deleteDb("postgres");
2207     Connection conn = getConnection("postgres;MODE=PostgreSQL");
2208     Statement stat = conn.createStatement();
2209
2210     assertResult("2.0", stat, "select sqrt(upper('[1,4]':int4range))");
2211
2212     conn.close();
2213 }
2214
2215 public void testIsEmpty() throws SQLException {
2216     deleteDb("postgres");
2217     Connection conn = getConnection("postgres;MODE=PostgreSQL");
2218     Statement stat = conn.createStatement();
2219
2220     assertResult("FALSE", stat, "select isempty('[1,3]':int4range)");
2221     assertResult("FALSE", stat, "select isempty('[,]':int4range)");
2222     assertResult("TRUE", stat, "select isempty('empty':int4range)");
2223
2224     conn.close();
2225 }
2226
2227 public void testLowerInc() throws SQLException {
2228     deleteDb("postgres");
2229     Connection conn = getConnection("postgres;MODE=PostgreSQL");
2230     Statement stat = conn.createStatement();
2231
2232     assertResult("TRUE", stat, "select lower_inc('[1,3]':numrange)");
2233     assertResult("FALSE", stat, "select lower_inc('(1,3)':numrange)");
2234     assertResult("FALSE", stat, "select lower_inc('empty':numrange)");
2235
2236     conn.close();
2237 }
2238
2239 public void testUpperInc() throws SQLException {
2240     deleteDb("postgres");
2241     Connection conn = getConnection("postgres;MODE=PostgreSQL");
2242     Statement stat = conn.createStatement();
2243
2244     assertResult("FALSE", stat, "select upper_inc('[1,3]':numrange)");
2245     assertResult("TRUE", stat, "select upper_inc('[1,3]':numrange)");
2246     assertResult("FALSE", stat, "select upper_inc('empty':numrange)");
2247
2248     conn.close();
2249 }
2250
2251 public void testLowerInf() throws SQLException {
2252     deleteDb("postgres");
2253     Connection conn = getConnection("postgres;MODE=PostgreSQL");
2254     Statement stat = conn.createStatement();
2255
2256     assertResult("FALSE", stat, "select lower_inf('[1,3]':int4range)");
2257     assertResult("TRUE", stat, "select lower_inf('(,3)':int4range)");
2258     assertResult("FALSE", stat, "select lower_inf('empty':int4range)");
2259
2260     conn.close();
2261 }
2262
```

```

2263 public void testUpperInf() throws SQLException {
2264     deleteDb("postgres");
2265     Connection conn = getConnection("postgres;MODE=PostgreSQL");
2266     Statement stat = conn.createStatement();
2267
2268     assertResult("FALSE", stat, "select upper_inf('[1,3]':int4range)");
2269     assertResult("TRUE", stat, "select upper_inf('[1,]':int4range)");
2270     assertResult("FALSE", stat, "select upper_inf('empty':int4range)");
2271
2272     conn.close();
2273 }
2274
2275 public void testRangeMerge() throws SQLException {
2276     deleteDb("postgres");
2277     Connection conn = getConnection("postgres;MODE=PostgreSQL");
2278     Statement stat = conn.createStatement();
2279
2280     assertResult("[1,5]", stat, "select range_merge('[1,2]':int4range, '[4,5]':int4range)");
2281     assertResult("[1,5]", stat, "select range_merge('(1,5)':numrange, '[1,2]':numrange)");
2282     assertResult("[1,5]", stat, "select range_merge('[4,5]':numrange, '[1,5]':numrange)");
2283
2284     conn.close();
2285 }
2286
2287 }

```

h2/src/test/org/h2/test/db/compatibility/postgresql/TestRangeTypeRecovery.java

```

1 package org.h2.test.db.compatibility.postgresql;
2
3 import org.h2.test.TestBase;
4 import org.h2.test.TestDb;
5
6 import java.sql.Connection;
7 import java.sql.SQLException;
8 import java.sql.Statement;
9
10 public class TestRangeTypeRecovery extends TestDb {
11
12     /**
13      * Run just this test.
14      *
15      * @param a ignored
16      */
17     public static void main(String... a) throws Exception {
18         TestBase.createCaller().init().testFromMain();
19     }
20
21     @Override
22     public void test() throws Exception {
23         deleteDb("test");
24         testUnionSumCorrection();
25         testDifferenceSubtractionCorrection();
26         testIntersectionMultiplyCorrection();
27         testUpperCorrection();
28         testLowerCorrection();
29         deleteDb("test");
30         System.out.println("end of tests");
31     }

```

```
32
33     private void testUnionSumCorrection() throws SQLException {
34         deleteDb("test");
35         Connection conn = getConnection("test;MODE=PostgreSQL");
36         Statement stat = conn.createStatement();
37
38         stat.execute("create table T (R integer, S integer)");
39         stat.execute("insert into T values (1, 2)");
40         assertResult("5", stat, "select testSq.S + 3 from (select tb.S from T tb where tb.R = 1)
as testSq");
41
42         conn.close();
43     }
44
45     private void testDifferenceSubtractionCorrection() throws SQLException {
46         deleteDb("test");
47         Connection conn = getConnection("test;MODE=PostgreSQL");
48         Statement stat = conn.createStatement();
49
50         stat.execute("create table T (R integer, S integer)");
51         stat.execute("insert into T values (1, 2)");
52         assertResult("-1", stat, "select testSq.S - 3 from (select tb.S from T tb where tb.R = 1)
as testSq");
53
54         conn.close();
55     }
56
57     private void testIntersectionMultiplyCorrection() throws SQLException {
58         deleteDb("test");
59         Connection conn = getConnection("test;MODE=PostgreSQL");
60         Statement stat = conn.createStatement();
61
62         stat.execute("create table T (R integer, S integer)");
63         stat.execute("insert into T values (1, 2)");
64         assertResult("6", stat, "select testSq.S * 3 from (select tb.S from T tb where tb.R = 1) as
testSq");
65
66         conn.close();
67     }
68
69
70     private void testUpperCorrection() throws SQLException {
71         deleteDb("test");
72         Connection conn = getConnection("test;MODE=PostgreSQL");
73         Statement stat = conn.createStatement();
74
75         assertResult("ABC", stat, "select upper('abc')");
76
77         conn.close();
78     }
79
80     private void testLowerCorrection() throws SQLException {
81         deleteDb("test");
82         Connection conn = getConnection("test;MODE=PostgreSQL");
83         Statement stat = conn.createStatement();
84
85         assertResult("abc", stat, "select lower('ABC')");
86
87         conn.close();
```

```

88     }
89 }

```

h2/src/test/org/h2/test/unit/TestRange.java

```

1  package org.h2.test.unit;
2
3  import org.h2.compatibility.postgresql.ExtTypeInfoRange;
4  import org.h2.compatibility.postgresql.PostgreSQLDataConverter;
5  import org.h2.compatibility.postgresql.RangeOperation2;
6  import org.h2.expression.ValueExpression;
7  import org.h2.test.TestBase;
8  import org.h2.value.TypeInfo;
9  import org.h2.value.Value;
10 import org.h2.value.ValueVarchar;
11
12 public class TestRange extends TestBase {
13
14     /**
15      * Run just this test.
16      *
17      * @param a ignored
18      */
19     public static void main(String... a) throws Exception {
20         TestBase.createCaller().init().testFromMain();
21     }
22
23     @Override
24     public void test() {
25         dataConverter();
26         extTypeInfoRangeConverter();
27         rangeOperationStringBuilder();
28         System.out.println("end of tests");
29     }
30
31     private void dataConverter() {
32         dataConverterUpperInfinity();
33         dataConverterUpperInfinityInclusive();
34         dataConverterLowerInfinity();
35         dataConverterLowerInfinityInclusive();
36         dataConverterFullInfinity();
37         dataConverterFallThrough();
38         dataConverterTypelessUpperInfinity();
39         dataConverterTypelessUpperInfinityInclusive();
40         dataConverterTypelessLowerInfinity();
41         dataConverterTypelessLowerInfinityInclusive();
42         dataConverterTypelessFullInfinity();
43         dataConverterTypelessFallThrough();
44     }
45
46     private void dataConverterUpperInfinity() {
47         PostgreSQLDataConverter converter = new PostgreSQLDataConverter();
48
49         Value result = converter.convert(new ValueVarchar("[1,)",
50 TypeInfo.getTypeInfo(Value.VARCHAR, -1, -1, ExtTypeInfoRange.INT4RANGE));
51
52         assertEquals("[1,)", result.getString());
53         assertEquals(ExtTypeInfoRange.INT4RANGE, result.getType().getExtTypeInfo());
54     }
55 }

```

```
54
55     private void dataConverterUpperInfinityInclusive() {
56         PostgreSQLDataConverter converter = new PostgreSQLDataConverter();
57
58         Value result = converter.convert(new ValueVarchar("[1,]"),
59     TypeInfo.getTypeInfo(Value.VARCHAR, -1, -1, ExtTypeInfoRange.INT4RANGE));
60
61         assertNull(result);
62     }
63
64     private void dataConverterLowerInfinity() {
65         PostgreSQLDataConverter converter = new PostgreSQLDataConverter();
66
67         Value result = converter.convert(new ValueVarchar("(,1)"),
68     TypeInfo.getTypeInfo(Value.VARCHAR, -1, -1, ExtTypeInfoRange.INT4RANGE));
69
70         assertEquals("(,1)", result.getString());
71         assertEquals(ExtTypeInfoRange.INT4RANGE, result.getType().getExtTypeInfo());
72     }
73
74     private void dataConverterLowerInfinityInclusive() {
75         PostgreSQLDataConverter converter = new PostgreSQLDataConverter();
76
77         Value result = converter.convert(new ValueVarchar("[,1)"),
78     TypeInfo.getTypeInfo(Value.VARCHAR, -1, -1, ExtTypeInfoRange.INT4RANGE));
79
80         assertNull(result);
81     }
82
83     private void dataConverterFullInfinity() {
84         PostgreSQLDataConverter converter = new PostgreSQLDataConverter();
85
86         Value result = converter.convert(new ValueVarchar("(,)",
87     TypeInfo.getTypeInfo(Value.VARCHAR, -1, -1, ExtTypeInfoRange.INT4RANGE));
88
89         assertEquals("(,)", result.getString());
90         assertEquals(ExtTypeInfoRange.INT4RANGE, result.getType().getExtTypeInfo());
91     }
92
93     private void dataConverterFallThrough() {
94         PostgreSQLDataConverter converter = new PostgreSQLDataConverter();
95
96         Value result = converter.convert(new ValueVarchar("abc"),
97     TypeInfo.getTypeInfo(Value.VARCHAR, -1, -1, ExtTypeInfoRange.INT4RANGE));
98
99         assertNull(result);
100     }
101
102     private void dataConverterTypelessUpperInfinity() {
103         PostgreSQLDataConverter converter = new PostgreSQLDataConverter();
104
105         Value result = converter.convertTypelessToRangeValue(new ValueVarchar("[1,]"));
106
107         assertEquals("[1,]", result.getString());
108     }
109
110     private void dataConverterTypelessUpperInfinityInclusive() {
111         PostgreSQLDataConverter converter = new PostgreSQLDataConverter();
```

```

108     Value result = converter.convertTypelessToRangeValue(new ValueVarchar("[1,]"));
109
110     assertNull(result);
111 }
112
113 private void dataConverterTypelessLowerInfinity() {
114     PostgreSQLDataConverter converter = new PostgreSQLDataConverter();
115
116     Value result = converter.convertTypelessToRangeValue(new ValueVarchar("(,1)"));
117
118     assertEquals("(,1)", result.getString());
119 }
120
121 private void dataConverterTypelessLowerInfinityInclusive() {
122     PostgreSQLDataConverter converter = new PostgreSQLDataConverter();
123
124     Value result = converter.convertTypelessToRangeValue(new ValueVarchar("[,1)"));
125
126     assertNull(result);
127 }
128
129 private void dataConverterTypelessFullInfinity() {
130     PostgreSQLDataConverter converter = new PostgreSQLDataConverter();
131
132     Value result = converter.convertTypelessToRangeValue(new ValueVarchar("(,)"));
133
134     assertEquals("(,)", result.getString());
135 }
136
137 private void dataConverterTypelessFallThrough() {
138     PostgreSQLDataConverter converter = new PostgreSQLDataConverter();
139
140     Value result = converter.convertTypelessToRangeValue(new ValueVarchar("abc"));
141
142     assertNull(result);
143 }
144
145 private void extTypeInfoRangeConverter() {
146     assertEquals(ExtTypeInfoRange.INT4RANGE,
147 ExtTypeInfoRange.getExtTypeBySubtype(TypeInfo.TYPE_INTEGER));
147     assertEquals(ExtTypeInfoRange.INT8RANGE,
148 ExtTypeInfoRange.getExtTypeBySubtype(TypeInfo.TYPE_BIGINT));
148     assertEquals(ExtTypeInfoRange.NUMRANGE,
149 ExtTypeInfoRange.getExtTypeBySubtype(TypeInfo.TYPE_NUMERIC_FLOATING_POINT));
149     assertEquals(ExtTypeInfoRange.DATERANGE,
150 ExtTypeInfoRange.getExtTypeBySubtype(TypeInfo.TYPE_DATE));
150     assertEquals(ExtTypeInfoRange.TSRANGE,
151 ExtTypeInfoRange.getExtTypeBySubtype(TypeInfo.TYPE_TIMESTAMP));
151     assertEquals(ExtTypeInfoRange.TSTZRANGE,
152 ExtTypeInfoRange.getExtTypeBySubtype(TypeInfo.TYPE_TIMESTAMP_TZ));
152     assertNull(ExtTypeInfoRange.getExtTypeBySubtype(TypeInfo.TYPE_BOOLEAN));
153 }
154
155 private void rangeOperationStringBuilder() {
156     ValueExpression a = ValueExpression.get(new ValueVarchar("a"));
157     ValueExpression b = ValueExpression.get(new ValueVarchar("b"));
158
159     assertEquals("'a' + 'b'", new RangeOperation2("+", a, b).getUnenclosedSQL(new
160     StringBuilder(), 0).toString());

```

```
160     assertEquals("'a' - 'b'", new RangeOperation2("-", a, b).getUnenclosedSQL(new
StringBuilder(), 0).toString());
161     assertEquals("'a' * 'b'", new RangeOperation2("*", a, b).getUnenclosedSQL(new
StringBuilder(), 0).toString());
162     assertEquals("'a' -|- 'b'", new RangeOperation2("-|-", a, b).getUnenclosedSQL(new
StringBuilder(), 0).toString());
163     assertEquals("'a' && 'b'", new RangeOperation2("&&", a, b).getUnenclosedSQL(new
StringBuilder(), 0).toString());
164     assertEquals("'a' @> 'b'", new RangeOperation2("@>", a, b).getUnenclosedSQL(new
StringBuilder(), 0).toString());
165     assertEquals("'a' <@ 'b'", new RangeOperation2("<@", a, b).getUnenclosedSQL(new
StringBuilder(), 0).toString());
166     assertEquals("'a' << 'b'", new RangeOperation2("<<", a, b).getUnenclosedSQL(new
StringBuilder(), 0).toString());
167     assertEquals("'a' >> 'b'", new RangeOperation2(">>", a, b).getUnenclosedSQL(new
StringBuilder(), 0).toString());
168     assertEquals("'a' &< 'b'", new RangeOperation2("&<", a, b).getUnenclosedSQL(new
StringBuilder(), 0).toString());
169     assertEquals("'a' &> 'b'", new RangeOperation2("&>", a, b).getUnenclosedSQL(new
StringBuilder(), 0).toString());
170     }
171 }
```

APÊNDICE C – Artigo

Extensão de compatibilidade para tipos de dados range e operadores de conjuntos do banco de dados PostgreSQL para H2

Rafael Barbaresco¹, Ronaldo dos Santos Mello¹, Patricia Vilain¹

¹Universidade Federal de Santa Catarina (UFSC)
Campus Universitário Trindade – CP 476 – 88.040-900 – Florianópolis (SC), Brasil
rafaelbrbesc@gmail.com, r.mello@ufsc.br, patricia.vilain@ufsc.br

Abstract. *The practice of software testing has grown considerably in recent years. The continuous integration process requires running tests many times during software development. This process, in some cases, is slowed down due to long test execution times or incompatibilities between test and production environments. Tests involving databases (DB) tend to take a long time due to the access time to secondary memory. To mitigate DB access time during tests, it is common to use a high-performance DB different from the DB used in production, as it has shorter data access time. However, this can cause incompatibilities between test and production environments. H2 is an open source in-memory DB Management System (DBMS) that offers a compatibility mode to simulate the functionalities of other popular DBMSs, including PostgreSQL. H2 does not offer complete compatibility of all functions of other DBMSs, being limited to the SQL standard. The purpose of this work is to implement, in H2, compatibility with the "range" data type, exclusive to PostgreSQL and outside the SQL standard, as well as related functions and operations. The implementation validation is done exploring all identified scenarios that involve the use of the new data type, as well as the application of the extension in a real use case. This work is expected to establish a basis for future extensions of the H2 compatibility mode, beyond "range" compatibility.*

Resumo. *A prática de testes de software têm crescido consideravelmente nos últimos anos. O processo de integração contínua exige executar os testes muitas vezes durante o desenvolvimento de software. Esse processo, em alguns casos, é contraído devido ao grande tempo de execução dos testes ou incompatibilidades entre os ambientes de testes e de produção. Testes que envolvem banco de dados (BD) costumam ser demorados devido ao tempo de acesso à memória secundária. Para mitigar o tempo de acesso ao BD durante os testes, é comum utilizar um BD de alto desempenho diferente do BD utilizado em produção, uma vez que ele possui menor tempo de acesso aos dados. Porém, isso pode causar incompatibilidades entre ambientes de teste e produção. O H2 é um Sistema de Gerência de BD (SGBD) em memória, de código aberto, que oferece um modo de compatibilidade*

para simular o funcionamento de outros SGBDs populares, entre eles o PostgreSQL. O H2 não oferece compatibilidade completa de todas as funções de outros SGBDs, limitando-se ao padrão SQL. A proposta deste trabalho é implementar, no H2, compatibilidade do tipo de dado "range", exclusivo do PostgreSQL e fora do padrão SQL, bem como as funções e operações relacionadas. A avaliação da implementação é feita explorando todos os cenários identificados que envolvem o uso do novo tipo de dado, bem como a aplicação da extensão em um caso de uso real. Espera-se com este trabalho estabelecer uma base para futuras extensões do modo de compatibilidade do H2, além da compatibilidade do tipo "range".

1. Introdução

Testes de software é uma área promissora que cresceu muito nos últimos anos e estima-se que vai continuar crescendo a uma taxa de 6% ao ano até 2026 (Global Market Insights, 2019). O processo de integração contínua, aplicado aos modelos atuais de desenvolvimento de software, implica executar diversas suítes de teste a cada integração de novas funcionalidades no software (Sommerville, 2011, p. 65). Isso exige subir a aplicação e o Banco de Dados (BD) a cada integração, sendo o BD descartado ao final da execução dos testes. Apesar da integração contínua ser uma boa ideia, ela pode ser inviável devido ao tempo de execução dos testes ou incompatibilidades entre o ambiente de teste e o ambiente de produção (Sommerville, 2011, p. 698).

Uma prática comum para testes automatizados que envolvem BD é a utilização de um sistema gerenciador de BD (SGBD) com persistência em memória principal (RAM) para o ambiente de testes, enquanto a aplicação em produção ainda usa um SGBD de persistência em memória secundária (HD ou SSD). Essa prática é realizada devido à natureza rápida e volátil dos BDs em memória (Paris Technologies, 2020). Um BD em memória armazena todos os dados na memória principal, tornando-o mais rápido que BDs tradicionais de persistência em memória secundária.

Porém, a prática de utilizar, durante os testes, um SGBD diferente do SGBD utilizado em produção gera controvérsias (Hauer, 2017). Diversos problemas são derivados da incompatibilidade e singularidades dos SGBDs, como diferentes formas de tratar uma mesma consulta ou a ausência de certos tipos de dados.

Comparando com outras formas de testar o BD durante testes automatizados, o problema de utilizar BD em memória principal está na confiabilidade dos testes. Testes de aplicações que usam um SGBD com persistência estritamente em memória secundária requerem um SGBD alternativo para que o teste seja executado apenas em memória principal. As diferenças no comportamento entre os SGBDs potencialmente comprometem o resultado dos testes. Existem diversos SGBDs disponíveis no mercado e, mesmo adotando o padrão SQL (ISO 9075), existem diferenças entre eles, derivados das ferramentas base para a construção do sistema ou decisões de projeto. Além das diferenças dentro do padrão SQL, muitos SGBDs oferecem funções exclusivas, fora do padrão SQL, que se destacam diante de outros SGBDs.

O SGBD Hypersonic 2 (H2) é um SGBD de persistência em memória principal, com código aberto, que tenta mitigar as incompatibilidades com outros SGBDs oferecendo um modo de compatibilidade. Porém, o modo de compatibilidade do H2 limita-se a somente alguns SGBDs populares e funcionalidades especificadas no padrão SQL.

O SGBD PostgreSQL, por sua vez, é um SGBD relacional tradicional, também com código aberto, e com armazenamento em memória secundária. O PostgreSQL é um SGBD popular e o H2 possui o modo de compatibilidade com ele. Uma singularidade do PostgreSQL é o tipo de dado "range", que define um conjunto de valores contínuos ou discretos (intervalo), bem como funções e operações com esse tipo de dado. Por ser uma singularidade fora do padrão SQL, o SGBD H2 não suporta operações com esse tipo de dado, mesmo em seu modo de compatibilidade.

Este trabalho visa criar uma extensão de compatibilidade do SGBD H2 para que ele reconheça o tipo de dado "range" e as funções e operações relacionadas. Considerando que o H2 possui suporte limitado para adaptações do analisador de código, e que não existem APIs públicas que expandem seu modo de compatibilidade, este trabalho também oferece uma base para futuras expansões do modo de compatibilidade do H2.

2. Tipo de dado *Range*

Range é um tipo de dado exclusivo do PostgreSQL que representa um intervalo de valores de outro tipo conhecido (um "subtipo"). Esse tipo de dado também é acompanhado de diversas operações sobre intervalos. É um tipo de dado útil para casos onde operações são realizadas sob o intervalo entre os valores limites, ao invés dos limites em si.

Esse tipo de dado tem uma estrutura semelhante a uma String ou Varchar. Ele começa com um delimitador inferior (caractere '[' para limite fechado ou '(' para limite aberto), o valor do limite inferior conforme seu subtipo, um separador de limites (caractere vírgula ','), o valor do limite superior conforme seu subtipo, e por fim um delimitador superior (caractere ']' para limite fechado ou ')' para limite aberto). Caso um intervalo de valores não tenha limite superior ou inferior, o seu respectivo valor limite na representação do dado não é preenchido, sendo definido como vazio. Em casos excepcionais, onde um intervalo não possui valores representáveis, o dado é representado apenas com uma String "empty".

Por padrão, o PostgreSQL oferece 6 tipos de dado *range*: *int4range*, *int8range*, *numrange*, *daterange*, *tsrange* e *tstzrange*, representando intervalos de valores *integer*, *bigint*, *numeric*, *date*, *timestamp* sem e com fuso horário respectivamente. Também dispõe de 11 operadores de intervalos: contém (@>), está contido (<@), sobreposição (&&), estritamente à esquerda (<<), estritamente à direita (<<), não se estende à direita (&<), não se estende à esquerda (&>), adjacentes (-|-), união (+), intersecção (*) e diferença (-). Por fim, dispõe também de funções auxiliares para construção e manipulação de intervalos. As funções de construção de intervalos possuem o mesmo nome dos seus respectivos tipos, e as funções auxiliares são *isEmpty*, *lower*, *upper*, *lower_inc*, *upper_inc*, *lower_inf*, *upper_inf* e *range_merge*.

3. H2 Extended

O H2 Extended é uma proposta, definida neste trabalho, de extensão ao SGBD H2 visando expandir o seu modo de compatibilidade. A extensão proposta é a inclusão de um compilador secundário (*CompatibilityParser* ou *Parser secundário*) paralelo ao compilador nativo do H2 (*Parser* ou *Parser primário*). Esse compilador secundário é capaz de interpretar tipos de dados e expressões exclusivas dos outros SGBDs disponíveis no modo de compatibilidade. A Figura 1 ilustra o relacionamento entre os componentes envolvidos e o processo de interpretação e resolução de consultas, com a inclusão do novo componente desenvolvido neste trabalho, destacado.

A intenção deste trabalho é implementar somente o reconhecimento de tipos de dado range e o comportamento das funções e operadores relacionados. Posteriormente, o trabalho pode ser expandido para suportar novas funções, operadores e tipos de dados tanto do PostgreSQL quanto dos demais SGBDs suportados.

Foi avaliado a implementação da extensão com abordagem profunda ou superficial. A abordagem profunda seria o reconhecimento verdadeiro dos novos tipos e implicaria na adaptação de muitos componentes do sistema. Já a abordagem superficial seria a implementação do reconhecimento dos novos tipos de dados como sinônimo (*alias*) de outro tipo de dado já reconhecido, minimizando o impacto das mudanças no sistema.

A solução proposta por este trabalho adota a abordagem superficial. O modo de compatibilidade do H2 possui suporte para traduzir um novo tipo de dado para outro tipo conhecido (*alias*), evitando assim a necessidade de alterações nas partes do sistema que possuam dependência com o tipo de dado em questão. Entretanto, a complexidade inerente ao tipo de dado range vai além do que é atualmente oferecido pelo H2. Portanto, a proposta de solução expande o modo de compatibilidade do H2, permitindo a inclusão de novos tipos complexos como range e eventuais tipos em futuras expansões. Para o reconhecimento dos novos tipos de *range*, estes são tratados como *alias* de *varchar*. Além disso, foi introduzido um conversor de dados, capaz de converter dados no formato *varchar* para *range* em tempo de execução. Esse conversor é necessário porque os novos tipos de dados são salvos em seu formato *alias* (*varchar*). Sempre que um dado de um tipo *range* é lido de uma tabela ele precisa ser convertido de *varchar* novamente para *range*.

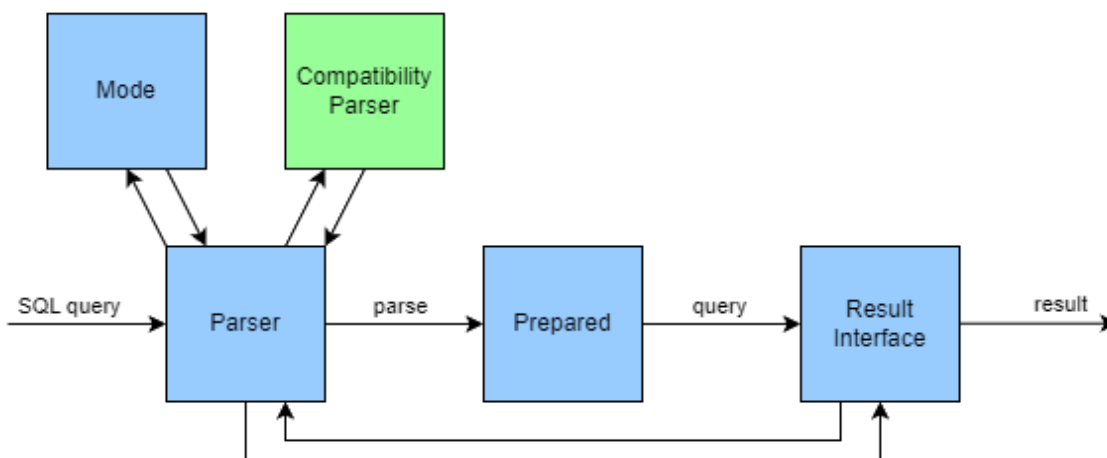


Figura 1. Componentes envolvidos no processamento de consultas no H2.

O processamento de consultas SQL no H2 pode ser dividido em 3 etapas: preparação, otimização e resolução. Na preparação é feita a compilação do código, onde o componente *Parser* transforma a consulta original em formato String em uma árvore de objetos, cuja raiz é denominada *Prepared*. Na otimização são identificados problemas de sintaxe e semântica, bem como a simplificação de elementos na árvore e a vinculação de dependências entre os elementos. Já a etapa de resolução calcula efetivamente o resultado da consulta.

O componente *Mode* possui informações sobre o modo de compatibilidade ativo, bem como um dicionário de nomes de tipos reconhecidos. Este componente foi complementado com os nomes dos novos tipos de dados a serem reconhecidos e também os novos operadores.

O componente *CompatibilityParser* é um compilador secundário e possui uma interface para que possa ser trocado de acordo com o modo de compatibilidade ativo no componente *Mode*. A seleção do parser secundário é feita de acordo com o modo de compatibilidade ativo. Visando expansões futuras, foi implementada uma estrutura para receber um parser específico para cada modo de compatibilidade.

Durante a etapa de preparação, quando uma operação desconhecida é identificada pelo compilador principal *Parser*, este transfere o fluxo de execução para o compilador secundário *CompatibilityParser*. O compilador secundário, por sua vez, interpreta o fragmento da consulta desconhecido pelo compilador principal, constrói um elemento encapsulado e o entrega para o compilador principal, pronto para ser inserido na árvore do objeto *Prepared*. O componente *Parser* se encarrega de vincular esse elemento com os demais elementos da árvore e continua a interpretação da consulta.

Como o escopo deste trabalho abrange apenas o PostgreSQL, foi implementado apenas um parser para compatibilidade com o PostgreSQL. Para o modo de compatibilidade padrão e os demais modos de compatibilidade, foi definido um compilador secundário neutro. Esse componente neutro recebe as chamadas do compilador primário, mas sempre retorna valores que não interferem no fluxo natural nativo do H2.

Foram identificados 4 momentos na interpretação de consultas SQL onde o compilador primário deve transferir o fluxo de execução para o compilador secundário: interpretação de função nomeada, interpretação de operadores lógicos, interpretação de operadores binários com prioridade de soma e subtração, e interpretação de operadores binários com prioridade de multiplicação e divisão. Em cada um desses momentos o compilador principal consulta o compilador secundário para decidir se o contexto deve ser transferido.

4. Resultados

Os resultados foram avaliados de 2 formas. A primeira é a interpretação de um conjunto de várias consultas pequenas e pontuais, abordando cenários específicos de utilização dos tipos de dados range, operadores e funções. E a segunda é a aplicação da extensão em um caso de uso real, com uma consulta que envolve vários elementos introduzidos pelo trabalho ao mesmo tempo.

4.1. Cenários avaliados

O trabalho foi feito com o intuito de integrar as mudanças no ramo principal do H2, como uma forma de contribuição. Portanto, foram obedecidas as normas para submissão de contribuições. Entre elas, a cobertura de testes automatizados, cobrindo pelo menos 90% do código produzido. Os testes automatizados são realizados de ponta-a-ponta e, portanto, são muito próximos de um cenário real de utilização da extensão. Dessa forma, o ambiente de testes do H2 foi utilizado para validar todos os cenários identificados que envolvem os novos elementos reconhecidos. Os cenários foram divididos nas seguintes categorias: construção e declaração de valores, operações com tabela para inserção, recuperação, atualização e remoção de dados, operadores especiais e funções especiais. Os cenários avaliados totalizam mais de 500 consultas pontuais.

A construção de valores range foi quase inteiramente replicada da forma como é feita no PostgreSQL. No H2 a construção dos valores pode ser feita de 3 formas: declaração explícita com o operador "::", conversão de valor com a função "cast", ou utilização das funções exclusivas do PostgreSQL para a construção dos valores, cada uma com mesmo nome do tipo de dado. Os tipos *int4range*, *int8range* e *daterange* possuem forma canônica (fechado no limite inferior e aberto no limite superior), e suas funções de canonização foram corretamente replicadas. A extensão também é capaz de reconhecer os conceitos de intervalo vazio e limites infinitos. Porém, o H2 não é capaz de reconhecer palavras-chave dentro de um varchar. Portanto, palavras como "today" e "infinity", reconhecidas pelo PostgreSQL na declaração explícita e na função "cast", não são reconhecidas pelo H2.

O quadro 1 demonstra exemplos de cenários avaliados para a construção de intervalos contínuos de números inteiros.

Quadro 1. Demonstração de declaração de valores range.

```
-- Construção do intervalo '[1,3)'  
select '[1,3)::int4range;  
select cast('[1,3)' as int4range);  
select int4range(1,3);  
select int4range(1,3,[')');
```

Todas as consultas que interagem com tabelas produzem resultados equivalentes às consultas realizadas no PostgreSQL, no que se refere ao uso de tipos de dados range. Como foi adotada a abordagem superficial, todos os valores de range são reconhecidos, também, como valores varchar. Isso significa que operações como a construção e utilização de índices ou consultas que interagem com as colunas especiais (utilizadas em cláusulas order by, group by, join, etc) interpretam os valores como varchar.

Todos os operadores que lidam com tipos de dado range foram inteiramente replicados. São 11 operadores novos e os operadores "contém" (@>) e "está contido" (<@) podem ser usados de duas formas, para verificar se um elemento pertence a um intervalo ou se um intervalo inteiro pertence a outro intervalo, totalizando 13 novas combinações de operações binárias. Além das operações isoladas, também foi avaliado o enfileiramento de operações de união, diferença e intersecção, a priorização das operações de união e diferença

sobre as operações de condição, e a priorização da operação de intersecção sobre as operações de união e diferença.

O Quadro 2 demonstra exemplos de cenários avaliados para operações com intervalos, bem como enfileiramento e priorização de operações.

Quadro 2. Demonstração de operações binárias, enfileiramento e priorização.

```
-- Operação binária de união
select '[1,3)::int4range + '[2,5)::int4range;

-- Prioridade de intersecção sobre união.
select '[3,4)::int4range + '[1,5)::int4range * '[4,6)::int4range;

-- Prioridade de união sobre operadores condicionais, neste caso sobreposição.
select '[1,2)::int4range && '[3,5)::int4range + '[4,6)::int4range;
```

E as funções especiais incluídas pela extensão foram inteiramente implementadas, exceto a função de criação de novos tipos de range. O PostgreSQL permite criar dinamicamente novos tipos de valores range utilizando o recurso de User Defined Type (UDT). Devido a ausência do recurso UDT no H2, não foi possível replicar a criação dinâmica de novos tipos.

4.2. Caso de uso

A extensão proposta foi experimentada em um caso de uso de um projeto real. O projeto consiste numa aplicação Web para controle de horas trabalhadas pelos colaboradores de uma empresa. A aplicação possui um cadastro de colaboradores, períodos de contratações, com data de início e fim, itinerários, com a carga horária semanal de cada período de contratação, períodos de ausência, onde consta as férias usufruídas pelo colaborador, e outros utilitários. A aplicação também utiliza o PostgreSQL para armazenamento de dados em ambiente de produção e H2 para execução de testes automatizados em ambiente de desenvolvimento.

Dentre as funções relevantes para este trabalho, a aplicação é capaz de calcular a quantidade de dias de férias que o colaborador tem disponível para usufruir. O cálculo de férias é baseado no total de horas trabalhadas, convertido em dias de acordo com a carga horária atual do colaborador. O cálculo é feito inteiramente em uma consulta SQL, que retorna diretamente a quantidade de dias que o colaborador pode usufruir de férias. Essa consulta utiliza as funções exclusivas do PostgreSQL *daterange*, *upper* e *lower*, além de utilizar o operador de intersecção.

A Figura 2 apresenta o diagrama de entidade-relacionamento das entidades envolvidas. Cada colaborador (*Collaborator*) possui diversos períodos de contratação (*Schedule*) ao longo de sua jornada na empresa, devido à renovação de contrato (data de início de contrato *startDate* e fim *endDate*). Cada período de contratação possui definições de horário de trabalho (*ScheduleInterval*) ao longo da semana, que devem somar a quantidade total de horas acordadas no contrato (início do período *startTime* e término *endTime*). As férias já usufruídas pelo colaborador são armazenadas numa tabela de registro de ausências

(*Absence*) com uma classificação de tipo correspondente (*absenceType*) e datas de início e fim (*startDate* e *endDate*).

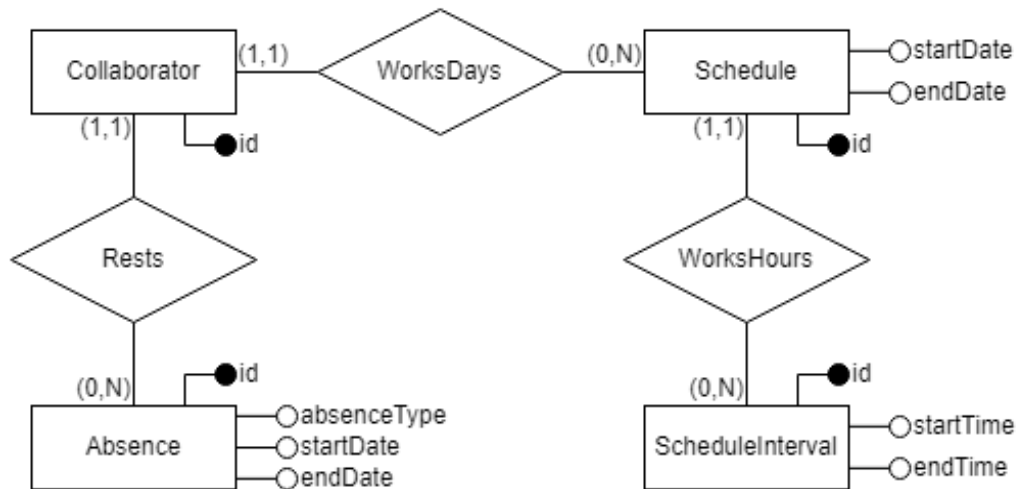


Figura 2. Diagrama de entidade-relacionamento do caso de uso.

A extensão apresentada no trabalho provou-se eficaz para resolver a consulta deste caso de uso em sua íntegra, produzindo o resultado correto.

5. Conclusões e Trabalhos Futuros

Todos os objetivos propostos por este trabalho foram alcançados. A construção de valores range, com seus 6 subtipos padrão, foi possível com a utilização de alias de varchar. A persistência e recuperação de dados em tabelas foi alcançada utilizando tratamentos nativos para varchar, sendo o resultado produzido por consultas no H2 equivalente aos resultados obtidos em consultas no PostgreSQL. E os operadores e funções relacionadas aos tipos de dados range foram inteiramente replicados de forma quase idêntica às suas implementações no PostgreSQL, salvo apenas pelos fluxos de correção para resolução dos casos de ambiguidade.

Existem limitações derivadas da abordagem superficial, todas relacionadas à conversão de valores *varchar* para *range*. Essas limitações poderiam ser resolvidas com a implementação profunda. Porém, considerando que o suporte próprio do H2 para *alias* de tipos de dados já utiliza uma implementação superficial, foi decidido fazer a implementação superficial. O propósito dos modos de compatibilidade são a equivalência das consultas, não necessariamente tendo suas implementações idênticas. A abordagem superficial é suficiente para atingir a equivalência das consultas.

Este trabalho estabeleceu uma base para futuras expansões. Ao longo do desenvolvimento do projeto, novas funcionalidades do PostgreSQL relacionadas aos tipos de dados range foram desenvolvidas. Essas novidades incluem os tipos de dados multirange, que são conjuntos de range capazes de representar lacunas, e, recentemente, a criação de índices especiais que utilizam multirange e são capazes de acelerar consultas utilizando os

operadores condicionais implementados neste trabalho. Este trabalho pode ser expandido para abranger multirange e a criação dos novos índices.

A infraestrutura desta extensão não está limitada ao PostgreSQL, e muito menos ao tipo de dado range. Ela pode servir de base para futuras implementações de comportamentos exclusivos do PostgreSQL e qualquer outro SGBD incluído nos modos de compatibilidade do H2.

Por fim, este trabalho descreve em detalhes a representação de tipos de dados no H2. Ele pode servir de objeto de estudo para a implementação de UDT no H2, uma demanda dentro do padrão SQL.

Referências

Global Market Insights (2019) Software Testing Market Size By Component (Application Testing, [By Type (Functional and Non-Functional)], Services [Professional and Managed], By Application (IT & Telecom, BFSI, Manufacturing, Retail, Healthcare, Transportation & Logistics, Government), Industry Analysis Report, Regional Outlook, Application Potential, Competitive Market Share & Forecast, 2020 – 2026. <https://www.gminsights.com/industry-analysis/software-testing-market>, visitado em 20 de abril de 2021.

Hauer, P. (2017) Don't use In-Memory Databases for Tests. Philipp Hauer's Blog. <https://phauer.com/2017/dont-use-in-memory-databases-tests-h2/>, visitado em 20 de abril de 2021.

Sommerville, I. (2011) Software Engineering. Pearson Addison Wesley. 9ª Edição, 2011.