Étendre un système de gestion de provenance : ProvSQL

Belkis Djeffal

République Algérienne Démocratique et Populaire
الجمهورية الجزائرية الديمقراطية الشعبية
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
وزارة التعليم العالي و البحث العلمي

Université des Sciences et de la Technologie Houari Boumediene

Faculté d'Informatique

Département des Systèmes Informatiques

Mémoire de Master

Filière : Informatique

Spécialité : Ingénierie des Logiciels

Étendre un système de gestion de provenance : ProvSQL

Sujet Proposé par :
Pr. SENELLART Pierre (ENS)

Présenté par :
DJEFFAL Belkis

Soutenu le 02 Juillet 2023, Devant le jury composé de :

Mme. ABDAT Nadia :	Présidente
Mme. BOUIBEDE Karima :	Membre

PFE $N°IL\_023$

Promotion : 2022/2023

Remerciements

Tout d'abord, je remercie Allah le tout puissant de m'avoir donné le courage et la patience nécessaires à mener ce travail à son terme.

Je tiens tout d'abord à exprimer ma gratitude envers mon encadrant de stage, Monsieur **Pierre SENELLART**, qui m'a offert l'opportunité de travailler au sein de son équipe. Sa grande expertise, sa bienveillance et son soutien constant ont été d'une valeur inestimable.

En deuxième lieu, je souhaite remercier chaleureusement l'ensemble du personnel du département informatique de l'**École Normale Supérieure**. Cela a été un grand honneur pour moi de rejoindre une équipe où l'expertise se marie parfaitement avec une ambiance conviviale.

Je tiens également à exprimer ma profonde reconnaissance envers tous les professeurs qui ont accompli leur travail avec rigueur. Leur enseignement de qualité à l'**Université des Sciences et de la Technologie Houari Boumediene** a été une expérience formatrice, bien que parfois exigeante.

Je remercie également tous les membres du jury pour leur lecture attentive de ce mémoire et les remarques précieuses et constructives qu'ils madresseront lors de la soutenance.

Enfin, je souhaite exprimer ma gratitude envers ma famille, mes amis et mes proches qui m'ont soutenue tout au long de ce stage. Leur soutien indéfectible, leurs encouragements et leurs précieux conseils ont été d'une importance capitale dans la réalisation de ce projet.

Je tiens à adresser mes plus sincères remerciements à toutes ces personnes qui ont contribué à mon parcours académique et professionnel. Leur générosité, leur soutien inconditionnel et leurs encouragements constants ont été une source de motivation et de gratification immense. Je leur suis profondément reconnaissante pour leur confiance et leur présence tout au long de cette expérience enrichissante.

Résumé

Dans un contexte où l'utilisation croissante des données pose des défis majeurs en termes de fiabilité et de traçabilité, la gestion de la provenance des données est devenue cruciale. La provenance des données fait référence à l'historique et aux informations associées à leur collecte, leur transformation et leur stockage, ce qui permet d'évaluer leur validité et de garantir leur intégrité.

Ce projet se concentre sur l'intégration de la gestion de la provenance des données dans les bases de données temporelles. Les bases de données temporelles sont spécialement conçues pour gérer des données évoluant dans le temps, offrant ainsi la possibilité de contextualiser la validité des informations collectées et stockées.

Notre contribution consiste en une solution innovante reposant sur l'utilisation de semi-anneaux pour calculer les temps de validité des données dans les bases de données temporelles. Cette solution sera intégrée à ProvSQL, une extension dédiée à la gestion de la provenance et de la probabilité dans PostgreSQL. L'objectif principal est d'améliorer la compréhension de l'historique associé aux données temporelles, assurant ainsi leur traçabilité et leur fiabilité.

En outre, nous mettons en place une démarche d'optimisation et d'évaluation des performances au sein de ProvSQL. Nous cherchons à réduire le temps de calcul de la provenance et à améliorer l'efficacité globale du système.

En résumé, ce projet propose une intégration de la gestion de la provenance des données dans les bases de données temporelles en étendant ProvSQL avec une solution basée sur les semi-anneaux. Cette approche renforce la traçabilité et la fiabilité des données tout en optimisant les performances de ProvSQL.

Mots clés : Prevenance de données, Bases de données temporelles, temps de validité, semi-anneau.

Abstract

In a context where the increasing use of data poses major challenges in terms of reliability and traceability, managing data provenance has become crucial. Data provenance refers to the history and associated information of data collection, transformation, and storage, allowing for the assessment of their validity and ensuring their integrity.

This project focuses on integrating data provenance management into temporal databases. Temporal databases are specifically designed to handle data evolving over time, thus providing the ability to contextualize the validity of collected and stored information.

Our contribution consists of an innovative solution based on the use of semirings to calculate data validity times in temporal databases. This solution will be integrated into ProvSQL, an extension dedicated to managing provenance and probability in PostgreSQL. The main goal is to enhance the understanding of the history associated with temporal data, thereby ensuring their traceability and reliability.

Furthermore, we are implementing an optimization and performance evaluation approach within ProvSQL. We aim to reduce the computation time for provenance and improve the overall efficiency of the system.

In summary, this project proposes the integration of data provenance management into temporal databases by extending ProvSQL with a semiring-based solution. This approach strengthens data traceability and reliability while optimizing the performance of ProvSQL.

Keywords : Data provennace, temporal database, valid time, semiring.

Table des matières

Table des figures

Liste des tableaux

Liste des algorithmes

Liste des sigles et acronymes

SGBD	*Système de Gestion de Base de Données*
UDF	*User Defined Function*
UUID	*Universally Unique Identifiers*
TPC-BiH	*Temporal Performance Council Benchmark for Interval-based Histories*
TPC	*Temporal Performance Council*
WSL	Windows Subsystem for Linux
SQL	Structured Query Language

Introduction générale

Contexte et problématique

À l'ère numérique d'aujourd'hui, la circulation et l'accès aux données ont été révolutionnés par la croissance rapide du World Wide Web. Alors que nous naviguons à travers la vaste quantité d'informations disponibles en ligne, il devient de plus en plus important de remettre en question l'origine et la fiabilité des données que nous rencontrons. Cela est particulièrement crucial pour les scientifiques et les chercheurs qui se basent sur des données précises et actuelles dans leurs recherches et leurs analyses. Comprendre la provenance des données, c'est-à-dire leur historique de dérivation et le processus par lequel elles sont arrivées dans une base de données, joue un rôle fondamental pour garantir la qualité et la confiance des données.

De nombreux domaines dépendent aujourd'hui des technologies modernes de bases de données pour gérer de vastes quantités de données. Ces bases de données peuvent contenir une variété d'informations provenant de différentes sources et subissant diverses transformations. Cependant, la nature interconnectée de ces bases de données, associée à la possibilité de transformation et de modification des données en cours de route, pose des défis pour déterminer l'origine spécifique d'une donnée. La provenance des données devient donc essentielle pour fournir une description complète de l'origine des données et des processus impliqués dans leur arrivée dans une base de données.

Le besoin de provenance des données s'étend bien au-delà des bases de données scientifiques. À mesure que nous utilisons Internet à diverses fins, telles que trouver les meilleurs achats, accéder aux actualités ou vérifier les évaluations des films, nous sommes devenus prudents quant à l'exactitude des informations extraites du Web. Toutefois, lorsque nous abordons des domaines scientifiques et universitaires, il devient primordial de pouvoir faire confiance à l'exactitude, à la fiabilité et à l'actualité des données.

En plus de la provenance des données, il y a une demande croissante pour prendre en compte le contexte temporel des données. Avec la complexité croissante de la gestion des données et l'avènement des applications de big data, comprendre quand les données collectées étaient valides devient crucial pour une analyse précise et pour obtenir des informations précieuses. Les métadonnées associées aux données, y compris les informations de provenance, peuvent aider à répondre aux questions liées à la validité des données au fil du temps.

En réponse au besoin de provenance des données et de contexte temporel dans la gestion des bases de données, plusieurs efforts de recherche ont émergé. L'un de ces efforts

est ProvSQL, un module open-source développé par l'équipe Valda de l'Inria qui est basée à l'École Normale Supérieure - PSL. ProvSQL étend le système de gestion de base de données PostgreSQL pour prendre en charge le calcul de la provenance et des probabilités des résultats des requêtes. Il offre une prise en charge de divers formalismes de provenance et permet l'évaluation probabiliste des requêtes grâce à l'utilisation d'outils de compilation des connaissances.

Cependant, malgré l'importance croissante d'intégrer la provenance et le contexte temporel dans les systèmes de gestion de bases de données, le support officiel des données temporelles dans PostgreSQL fait encore défaut. Dans ce projet, nous proposons une solution reposant sur l'utilisation de semi-anneaux pour combler cette lacune en tirant parti de ProvSQL et en introduisant un moyen d'ajouter le support des données temporelles à PostgreSQL. L'objectif est d'améliorer la compréhension de l'historique associé aux données temporelles, garantissant ainsi leur traçabilité et leur fiabilité.

En outre, nous mettons en place une démarche d'optimisation et d'évaluation des performances au sein de ProvSQL. Nous cherchons à réduire le temps de calcul de la provenance et à améliorer l'efficacité globale du système.

Ce projet contribue à l'avancement des connaissances dans le domaine de la gestion de la provenance des données et des bases de données temporelles. En intégrant la gestion de la provenance des données dans les bases de données temporelles avec notre solution basée sur les semi-anneaux, nous renforçons la traçabilité et la fiabilité des données tout en optimisant les performances de ProvSQL.

Dans la suite de ce rapport, nous présenterons les concepts fondamentaux liés à la provenance des données et aux bases de données temporelles. Nous décrirons également notre démarche et nos contributions dans l'intégration de la gestion de la provenance des données dans les bases de données temporelles. Enfin, nous présenterons les résultats obtenus et les perspectives d'amélioration pour ce projet.

Objectifs initiaux

Le projet a pour mission principale l'ajout du support de bases de données temporelles afin de manipuler efficacement les données dans un contexte temporel. Pour cela, plusieurs objectifs ont été identifiés :

- Effectuer une revue approfondie des travaux de recherche existants concernant la provenance dans les bases de données temporelles. Cette analyse permettra de comprendre les avancées et les défis actuels dans ce domaine.
- Implémenter un semi-anneau qui permettra de manipuler les données temporelles de manière précise et cohérente. Ce semi-anneau offrira les fonctionnalités nécessaires pour interroger les tables en fonction de leur état pendant une période de temps spécifique et explorer les temps de validité des données obtenues dans les résultats des requêtes.
- Mettre en place des simulations sur des bases de données temporelles en utilisant le semi-anneau développé. Cette simulation permettra de tester et de valider les

fonctionnalités du système, en reproduisant des scénarios réels et en évaluant sa performance.

- Comparer les performances du semi-anneau dans ProvSQL avec d'autres systèmes de gestion de provenance existants. Cette évaluation comparative permettra de mesurer l'efficacité et l'efficience de ProvSQL par rapport à d'autres solutions similaires.
- Optimiser le code de ProvSQL afin d'améliorer ses performances et son utilisation des ressources. identifier et résoudre les éventuels goulots d'étranglement, garantissant ainsi une exécution rapide et efficace des requêtes.
- Définir un formalisme pour la réalisation de tests sur ProvSQL en créant un benchmark spécifique pour les requêtes temporelles. Ce benchmark permettra d'évaluer les performances de ProvSQL dans des scénarios représentatifs et de comparer les résultats avec d'autres systèmes similaires.

Ainsi, nous avons pour ambition d'étendre les capacités de ProvSQL dans la manipulation des données temporelles, tout en cherchant à améliorer ses performances.

Organisation du mémoire

La suite de ce mémoire est structurée en quatre chapitres regroupés dans deux parties.

La première partie du mémoire est consacrée à la revue de la littérature, où deux chapitres sont regroupés. Le premier chapitre explore la provenance dans les bases de données relationnelles, mettant en évidence les travaux existants dans ce domaine. Le deuxième chapitre se concentre sur les bases de données temporelles, en examinant les recherches pertinentes dans ce domaine spécifique.

La seconde partie du mémoire présente les contributions apportées à ProvSQL. Elle est subdivisée en deux chapitres distincts. Le chapitre 03 expose la conception des solutions développées pour étendre les capacités de ProvSQL dans la manipulation des données temporelles. Il met en évidence les approches et les techniques utilisées pour l'amélioration et l'optimisation des performances de de ProvSQL. Le chapitre 04 se concentre sur l'implémentation concrète de ces solutions, en détaillant les aspects techniques, les tests réalisés et les résultats obtenus lors de leur évaluation.

partie I

État de l'art

Chapitre 1

La provenance de données

1.1 Introduction

Dans ce chapitre, nous explorons le domaine de la provenance des données dans les bases de données relationnelles. La provenance joue un rôle essentiel dans la gestion et l'analyse des données, offrant des informations sur l'origine, l'historique, la validité des données et bien d'autres aspects. Cela permet aux utilisateurs de mieux comprendre et interpréter les résultats de leurs requêtes.

Nous débutons en présentant la provenance et sa représentation dans les bases de données relationnelles. Nous examinons ensuite les différentes applications de la provenance, telles que l'audit et la gestion de la qualité des données. Par la suite, nous expliquons les différents types de provenance, qui capturent divers aspects d'information sur les données.

En examinant les systèmes existants de gestion de provenance, nous soulignons leurs limites et les défis auxquels ils font face. Cela nous conduit à présenter ProvSQL, un système novateur qui se distingue par sa capacité à gérer la provenance de manière flexible et efficace au sein des bases de données relationnelles.

Ce chapitre permet aux lecteurs de se familiariser avec le concept de provenance des données, de comprendre ses diverses applications et d'apprécier l'importance de cette notion. De plus, il offre un aperçu des avancées apportées par le système ProvSQL dans ce domaine.

En fournissant une base solide de connaissances, ce chapitre prépare le terrain pour explorer en détail les contributions de ce projet à ProvSQL et qui seront présentées dans la partie suivante de ce rapport.

1.2 Définitions et contexte

Anciennement appelée «lignée des données» *(data lineage)* (Ikeda et al. 2009), la provenance fait référence, dans un sens général, aux informations relatives à l'origine et à l'historique des objets. Elle englobe des détails sur leur création, leur évolution et les différentes transformations qu'ils ont subies au fil du temps. Elle désigne également l'histoire de la propriété d'un objet de valeur, d'une œuvre d'art ou de littérature (Merriam-

Webster 2023).

Dans le contexte des bases de données, la provenance est définie de différentes manières par divers auteurs. Selon Buneman et al. (Buneman et al. 2001), elle consiste en la description des origines des données et du processus par lequel elles ont été intégrées dans la base de données. Cheney et al. (Cheney et al. 2009) la définissent comme les informations décrivant les origines et l'historique des données tout au long de leur cycle de vie. Rani et al. (Rani et al. 2016) proposent une définition plus concise, considérant la provenance des données comme l'historique qui y est associé.

Ces différentes définitions convergent vers l'idée que la provenance permet de retracer le parcours des données depuis leur source initiale jusqu'à leur état actuel. Elle fournit des informations sur les étapes de traitement appliquées aux données, les requêtes exécutées pour les obtenir, ainsi que les mises à jour qui ont modifié leur contenu.

En capturant et en conservant ces informations de provenance, les bases de données offrent une traçabilité complète, permettant de vérifier l'origine et l'intégrité des données (Buneman et al. 2000a). Cela facilite la gestion de la qualité des données, la détection des erreurs et des incohérences, ainsi que la résolution de problèmes liés à la fiabilité et à la confiance dans les données.

1.3 Représentation de la provenance

La provenance est généralement représentée à l'aide d'annotations dites «*de provenance*» (Wang et al. 1990). Une annotation est une information supplémentaire ou une métadonnée qui est associée à une entité de données dans le but de fournir des détails contextuels ou des informations complémentaires sur cette entité (Kostylev et al. 2012). Elle peut inclure des informations telles que des descriptions, des tags, des notes, des dates de création ou de modification, des sources, des auteurs, etc.

Dans le contexte des bases de données relationnelle, les annotations de provenance peuvent prendre différentes formes. Elles peuvent être stockées sous forme de métadonnées associées à chaque élément de données, selon le niveau de granularité choisi, de la base de données (Bettini et al. 1997). Ces métadonnées peuvent inclure des informations telles que l'identifiant de la source des données, les estampilles des opérations effectuées sur les données, les identifiants des processus ou des opérations appliquées, et d'autres détails pertinents.

Nous avons introduit dans le paragraphe précédent la notion de **granularité** (Bettini et al. 1997), il s'agit du niveau de détail avec lequel les informations de provenance sont enregistrées et associées aux éléments de données. Dans certains cas, la provenance peut être enregistrée au niveau des tuples ou des attributs (Cheney et al. 2009), ce qui permet de suivre l'origine et l'historique des données jusqu'à un niveau très fin.

Le choix de la granularité de la provenance dépend des besoins et des objectifs spécifiques d'un domaine d'application (Vansummeren et al. 2007). Une granularité plus fine peut offrir une traçabilité plus détaillée et une compréhension approfondie des données, mais elle peut également entraîner une augmentation des coûts de stockage et de gestion. Une granularité plus grossière, en revanche, peut être suffisante pour certains

cas d'utilisation, mais elle peut ne pas fournir autant de détails sur l'historique et les transformations des données (Buneman et al. 2000b).

Dans le cadre spécifique de notre projet (Senellart et al. 2018), nous adoptons une approche où les annotations de provenance sont représentées sous forme d'attributs de tuple dans une base de données relationnelle. Cela signifie que chaque enregistrement ou tuple de la base de données est étendu pour inclure des attributs supplémentaires qui servent à stocker les informations de provenance.

Pour mettre en œuvre des bases de données annotées, des extensions aux systèmes de gestion de bases de données relationnelles (SGBDR) traditionnels sont souvent utilisées. Ces extensions peuvent inclure des modèles de données spécialisés, des langages de requête ou des techniques d'indexation pour prendre en charge le stockage, la récupération et l'interrogation de données annotées.

1.4 Applications de la provenance

Les systèmes de provenance peuvent être utilisés dans différents domaines. Goble (Goble 2002) résume plusieurs applications de l'information de provenance de la manière suivante :

- **Qualité des données :** La provenance peut être utilisée pour estimer la qualité et la fiabilité des données en se basant sur les données sources et les transformations effectuées (Jagadish et al. 2004). Elle peut également fournir des déclarations de preuve sur la dérivation des données (Jagadish et al. 2004).

- **Piste d'audit :** La provenance peut être utilisée pour retracer la piste d'audit des données (Miles et al. 2006), déterminer l'utilisation des ressources (Greenwood et al. 2003) et détecter les erreurs lors de la génération des données (Galhardas et al. 2001).

- **Attribution :** La provenance aide à établir les droits d'auteur et la propriété des données, permettre leur citation (Jagadish et al. 2004) et déterminer la responsabilité en cas de données erronées.

- **Informations :** Une utilisation générique de la provenance est de rechercher des données en se basant sur les métadonnées de provenance pour la découverte des données. Elle peut également être consultée pour fournir un contexte d'interprétation des données.

L'importance de la provenance dans la gestion des données est démontrée par ces différentes applications. Elle joue un rôle essentiel dans l'évaluation de la qualité des données, le suivi de leur traçabilité, la facilitation de la réplication et de la maintenance des données, l'établissement de la propriété intellectuelle et de la responsabilité, ainsi que la fourniture d'un contexte d'interprétation des données.

1.5 Types de provenance

La provenance dans les bases de données peut être classifiée en différents types en fonction des informations recherchées ainsi que des questions posées (Senellart 2017).

Nous nous appuyons sur le travail de Cheney et al. dans (Cheney et al. 2009), où ils ont réalisé une revue approfondie des trois principaux types de provenance : la *why-provenance*, la *how-provenance* et la *where-provenance*. Chacun de ces types de provenance offre une perspective unique sur les données et fournit des informations complémentaires.

1.5.1 Why-provenance *(Provenance «pourquoi»)*

La *why-provenance*, également connue sous le nom de provenance de causalité ou d'explication, se concentre sur la réponse aux questions liées à la raison pour laquelle un certain résultat ou une certaine issue a été obtenu (Buneman et al. 2001) . Elle fournit des informations sur les dépendances et les relations de causalité entre les données d'entrée et le résultat final. La *why-provenance* explique les raisons ou les facteurs qui ont contribué à la génération d'une sortie particulière. Elle aide à comprendre les causes et les influences derrière un élément de données ou un résultat de calcul.

Exemple 1.1
Considérons une base de données avec les notes des étudiants. La why-provenance expliquerait pourquoi un étudiant a obtenu la note «A», mettant en évidence les devoirs, les scores et les critères qui ont contribué à la note finale.

1.5.2 How-provenance *(Provenance «comment»)*

La *how-provenance* enrichit la *why-provenance* en fournissant des informations sur la manière dont les tuples d'entrée sont utilisés pour créer un tuple de sortie (Glavic et al. 2013). Elle se concentre sur les étapes de calcul ou les opérations appliquées pour obtenir le résultat d'une requête à partir des données d'entrée. Elle répond à la question : «Comment ce tuple a-t-il été calculé à partir des tuples d'entrée ?». La *how-provenance* fournit des informations détaillées sur les opérations, les transformations et les règles utilisées dans le processus de calcul.

Exemple 1.2
Prenons par exemple une requête qui calcule le total des ventes pour chaque produit dans une base de données de ventes. La how-provenance dans ce cas indiquerait les opérations d'agrégation réalisées, telles que la somme des montants de vente, ainsi que les tuples d'entrée associés à chaque agrégat.

1.5.3 Where-provenance *(Provenance «où»)*

La *where-provenance* se concentre sur l'emplacement spatial ou temporel des données d'origine. Elle répond à la question : «d'où les données ont-elles été collectées ou enregistrées ?». La *where-provenance* est particulièrement utile dans les cas où les données sont

distribuées ou lorsque des opérations de fusion ou de partitionnement sont effectuées sur les données.

Exemple 1.3
Par exemple, si des données sont collectées à partir de différents capteurs répartis géographiquement, la where-provenance indiquerait l'emplacement spatial de chaque capteur et permettrait de retracer la provenance des données jusqu'à leur source géographique.

1.5.4 Provenance basée sur les semi-anneaux

Initialement introduite par Green et al. (Green et al. 2007), la provenance basée sur les semi-anneaux est une forme généralisée de provenance qui étend les notions traditionnelles de la *why-provenance* et la *how-provenance*. Cette approche a été motivée par l'observation de similitudes dans les méthodes de traitement des requêtes dans les bases de données annotées telles que les bases de données probabilistes, les bases de données incomplètes et les entrepôts de données.

Ces similarités ont incité les auteurs à proposer une structure algébrique permettant d'étendre l'algèbre relationnelle afin de prendre en charge des opérations sur ces annotations. Ainsi, l'utilisation des semi-anneaux s'est avérée être une solution adéquate pour répondre à ce besoin.

Cette approche basée sur les semi-anneaux offre une représentation symbolique des calculs effectués sur les annotations, ce qui permet de capturer, documenter et suivre la provenance des requêtes de l'entrée à la sortie.

Formellement, un semi-anneau est défini comme suit :

Définition 1.1
Un semi-anneau (Gazek 2002) est une structure algébrique $(A, \oplus, \otimes, 0, 1)$ *où* A *est un ensemble non vide et les opérations binaires* $\oplus$ *(addition) et* $\otimes$ *(multiplication), définies sur l'ensemble* A *satisfont les propriétés suivantes :*

1. $(A, \oplus, 0)$ *forme un monoïde (SAGE 2018) commutatif, c'est-à-dire que pour tout* $a, b, c \in A$*, on a* $a \oplus (b \oplus c) = (a \oplus b) \oplus c$.
2. $(A, \otimes, 1)$ *forme un monoïde (SAGE 2018).*
3. *La multiplication* $(\otimes)$ *est distributive par rapport à l'addition* $(\oplus)$*, c'est-à-dire que pour tout* $a, b, c \in A$*, on a* $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ *et* $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$.
4. 0 *est absorbant pour le produit, c'est à dire que pour tout* $a \in A$*, on a* $a \otimes 0 = a$ *et* $a \otimes 0 = 0$.

Green et al. ont ensuite considéré dans (Green et al. 2007) que les semi-anneaux commutatifs constituent la structure algébrique appropriée, car ils garantissent la cohérence des opérations sur les annotations. Un semi-anneau commutatif est défini comme suit :

Définition 1.2
Soit les semi-anneau noté $(A, \oplus, \otimes, 0, 1)$*, on dit que le semi-anneau est commutatif si*

$(A, \otimes, 1)$ *est* ***commutatif****, c'est-à-dire que pour tout* $a, b, c \in A$*, on a* $a \otimes (b \otimes c) = (a \otimes b) \otimes c$ *(*Kostylev *et al. 2012).*

La propriété de commutativité est requise lorsque l'opérateur de produit cartésien de l'algèbre relationnelle est considéré comme commutatif. Les semi-anneaux commutatifs sont ainsi adaptés pour représenter les annotations et permettre des calculs cohérents.

Dans le contexte de la provenance, un tuple $(A, \oplus, \otimes, 0, 1)$, où A est un ensemble représentant les annotations de provenance, $\oplus$ et $\otimes$ sont des opérations binaires définies sur A représentant l'union et la jointure, respectivement, 0 est l'élément d'identité pour $\oplus$, et 1 est l'élément d'identité pour $\otimes$.

Exemple 1.4
Considérons l'exemple illustré dans la figure 1.5, nous proposons la requête Q qui sélectionne les noms des étudiants, les matières qu'ils ont étudiées et leurs notes à partir des tables `Etudiant` *1.1 et* `Notes` *1.2. Elle effectue une jointure entre les deux tables en utilisant la condition de correspondance entre les identifiants d'étudiant.*

Penchons-nous sur le résultat de la requête Q et les annotations affichées dans la troisième colonne du tableau 1.4. De manière intuitive, ces annotations abstraites décrivent comment chaque tuple de sortie est produit dans le résultat de Q, en utilisant les annotations abstraites (ou identifiants) des tuples sources. Par exemple, l'annotation $t1 \otimes t2$ *indique que le premier tuple de sortie (John, Math, A) est obtenu en joignant* $(\oplus)$ *t1 avec t3. De même, l'annotation* $t3 \otimes t4$ *décrit que le quatrième tuple de sortie (Mike, Physics, B) est créé en joignant t3 avec t4. Green et al. (*Green *et al. 2007) ont observé que ces annotations abstraites décrivant comment les tuples sources sont combinés (par union et/ou jointure) pour produire un tuple de sortie correspondent en réalité à des polynômes dans un semi-anneau commutatif* $(K, \oplus, \otimes, 0, 1)$*.*

1.6 Systèmes de gestion de provenance dans les bases de données relationnelles

La gestion de provenance dans les bases de données relationnelles est un domaine de recherche en pleine expansion. De nombreux systèmes ont été développés pour répondre au besoin croissant de capturer, stocker et interroger la provenance des données. Ces systèmes offrent des fonctionnalités avancées pour comprendre l'origine, la transformation et l'historique des données, ce qui est essentiel dans de nombreux domaines tels que la conformité réglementaire, la gestion des données scientifiques et la gestion des données d'entreprise.

Dans cette section, nous présenterons quelques systèmes de gestion de provenance dans les bases de données relationnelles. Chaque système propose des approches et des fonctionnalités spécifiques pour capturer, stocker et exploiter la provenance des données. Nous explorerons brièvement ces systèmes pour avoir un aperçu des différentes solutions disponibles.

student_id	name	major	
1	John	Math	$t1$
2	Lisa	Anglais	$t2$
3	Mike	Physique	$t3$

FIG. 1.1 : La table `Student`

id_etudiant	matiere	note	
1	Math	A	$t4$
1	Anglais	B	$t5$
2	Anglais	A	$t6$
3	Physique	B	$t7$
3	Math	C	$t8$

FIG. 1.2 : La table `Notes`

```
SELECT s.nom, g.matiere, g.note
FROM Students s, Notes g
WHERE s.id_etudiant = g.id_etudiant
```

FIG. 1.3 : La requête Q

name	subject	grade	
John	Math	A	$t1 \otimes t4$
John	Anglais	B	$t1 \otimes t5$
Lisa	Anglais	A	$t2 \otimes t5$
Mike	Physique	B	$t3 \otimes t7$
Mike	Math	C	$t3 \otimes t4$

FIG. 1.4 : Résultat de la requête Q

FIG. 1.5 : Exemple de requête expliquant la représentation de la provenance par un semi-anneau.

Chacun de ces systèmes propose une approche unique pour répondre aux défis de la gestion de provenance dans les bases de données relationnelles. Leur compréhension nous permettra d'appréhender les différentes techniques et fonctionnalités disponibles, ainsi que les avantages et les limites de chaque approche.

1.6.1 Whips (WareHousing Information Project at Stanford)

Le projet WHIPS (WIENER et al. 1997) , a été lancé en 1997 par l'université de Stanford. L'objectif principal du projet est de développer une architecture et des technologies pour l'intégration efficace de données provenant de sources hétérogènes dans un entrepôt de données.

L'architecture de WHIPS repose sur plusieurs modules interconnectés qui travaillent ensemble pour collecter, transformer et maintenir les vues matérialisées de l'entrepôt. Ces modules sont conçus pour être modulaires et extensibles, ce qui permet d'ajouter et de supprimer dynamiquement des sources de données et des vues de l'entrepôt.

Un élément clé de l'architecture de WHIPS est le composant de détection des changements. Ce composant est chargé de détecter les modifications survenant dans les sources de données et de les signaler au système. Cela permet à l'entrepôt de données d'être continuellement mis à jour en fonction des changements survenus dans les sources, sans nécessiter d'interruption du service.

Un autre aspect important de WHIPS est le module de transformation des données. Ce module est responsable de la transformation des données provenant des sources hétérogènes en un format compatible avec l'entrepôt de données. Il effectue des opérations

telles que la conversion des types de données, la normalisation des schémas et la résolution des conflits.

Le projet WHIPS a réalisé un prototype fonctionnel de son architecture et l'a démontré lors de la conférence SIGMOD (Special Interest Group on Management of Data). Le prototype a mis en évidence les fonctionnalités clés de l'architecture, notamment la capacité à intégrer de manière dynamique des sources de données, la détection efficace des changements et la mise à jour continue de l'entrepôt de données.

1.6.2 Trio

Trio (Widom 2004) fait référence à "un système dans lequel les données, l'incertitude des données et la provenance des données sont toutes des citoyens de premier ordre dans un modèle relationnel étendu et un langage d'interrogation basé sur SQL" (Agrawal et al. 2006).

Le projet Trio vise à développer un modèle de données appelé "Trio Data Model" (TDM) et un langage de requête appelé « *Trio Query Language* » (TriQL). Le TDM étend le modèle relationnel traditionnel en ajoutant des mécanismes pour représenter l'exactitude des valeurs attributaires, la confiance des tuples et la couverture des relations. Il introduit également des informations de provenance qui suivent les dépendances et les transformations des données.

TriQL est une extension du langage SQL qui permet d'effectuer des requêtes et des modifications sur les données du modèle Trio. Il étend la sémantique des requêtes SQL pour prendre en compte l'exactitude et la provenance des données, et ajoute de nouvelles fonctionnalités pour gérer explicitement ces aspects.

L'objectif principal de Trio est de permettre la gestion et l'exploitation efficaces des données qui sont susceptibles d'être approximatives ou incertaines, tout en conservant des informations sur leur provenance et leur fiabilité. Le système Trio vise également à intégrer l'exactitude et la provenance de manière transparente dans les opérations de modification des données, en permettant la propagation automatique des mises à jour d'exactitude le long des dépendances et des transformations des données.

1.6.3 DBNotes

DBNotes, développé par Chiticariu et al. (Chiticariu et al. 2005), est un système conçu pour les bases de données relationnelles basé sur les annotations d'attribut, ç à d, qui associe des annotations à chaque valeur d'attribut.

Dans DBNotes, un attribut d'annotation est ajouté pour chaque attribut de la relation, et les valeurs d'annotation sont stockées sous forme textuelle brute dans ces attributs d'annotation. La propagation des annotations se fait en fonction de la provenance sous-jacente. Ces schémas déterminent comment les annotations sont propagées dans les données lors des transformations.

Contrairement à Trio, où un post-traitement des résultats est nécessaire pour générer la représentation finale, DBNotes optimise ce processus en réécrivant une requête psql

en une seule requête. Cette approche simplifie la récupération et la représentation des résultats tout en maintenant l'intégrité des annotations (Mohammadi et al. 2022).

DBNotes propose les fonctionnalités suivantes :

- Schémas de propagation des annotations : DBNotes introduit pSQL, une extension de SQL, permettant aux utilisateurs de spécifier comment les annotations doivent se propager à travers une requête SQL. Il prend en charge trois types de schémas de propagation : par défaut, par défaut pour tous et personnalisé, offrant ainsi une flexibilité dans la définition du comportement de propagation. Ces schémas déterminent comment les annotations sont propagées dans les données lors des transformations.

- Interrogation des données et des annotations : Les utilisateurs peuvent interroger à la fois les données et les annotations dans DBNotes en utilisant pSQL. Le mot-clé `ANNOT(nom-de-l'attribut)` permet aux utilisateurs de récupérer des tuples basés sur des annotations spécifiques ou d'obtenir des informations sur les sources des données.

- Explication de la provenance des données : DBNotes offre une approche déclarative pour expliquer la provenance des annotations, des valeurs d'attributs ou des tuples dans les résultats des requêtes. Il génère des preuves qui démontrent comment les transformations ont produit les sorties souhaitées, y compris les liens avec les tuples et les annotations sources.

- Visualisation de la provenance et du flux : DBNotes propose des représentations visuelles de la provenance des données et du flux à travers les bases de données et les étapes de transformation. Les utilisateurs peuvent explorer le parcours d'une donnée, comprendre sa dérivation et ses dépendances dans d'autres bases de données, et obtenir des explications détaillées de la production des valeurs.

1.6.4 MMS (*Metadata Management System*)

MMS (Metadata Management System) est un système qui a été introduit en 2004 (Talha 2004). Il a été conçu dans le but de stocker, gérer et interroger différents types de métadonnées. Basé sur le modèle relationnel, MMS offre une approche uniforme pour modéliser les métadonnées et les associer aux données.

Les caractéristiques clés et les contributions de MMS comprennent :

- **Stockage et gestion de métadonnées variées :** MMS permet de stocker et gérer différents types de métadonnées, tels que des schémas, des contraintes d'intégrité, des commentaires sur les données, des ontologies, des paramètres de qualité, des annotations, des informations de provenance, des politiques de sécurité, etc.

- **Utilisation de requêtes comme valeurs de données :** Au lieu de se limiter aux valeurs atomiques, MMS utilise des requêtes comme valeurs de données, appelées valeurs de type q ou simplement $q - valeurs$. Cette approche permet d'associer les données avec les métadonnées en décrivant des ensembles de données et en effectuant

des opérations de jointure étendues entre les requêtes et les tuples de la base de données.

- **Mécanisme de jointure étendu :** MMS propose un mécanisme de jointure augmenté qui permet de mettre en correspondance des relations de résultats de requêtes, ouvrant ainsi la possibilité de requêter de manière uniforme les données et les métadonnées.

- **Enregistrement des métadonnées sur les blocs de valeurs :** MMS permet d'associer des métadonnées non seulement à des valeurs individuelles, mais également à des ensembles de valeurs.

- **Définition de métadonnées sur les métadonnées :** MMS permet de définir des métadonnées supplémentaires sur les métadonnées existantes, offrant ainsi une grande flexibilité dans la modélisation des métadonnées complexes.

- **Interrogation indépendante des données et des métadonnées :** Contrairement à de nombreux outils de gestion de métadonnées, MMS permet d'interroger et de récupérer les métadonnées de manière indépendante des données associées. Cela offre une plus grande souplesse dans l'exploration et l'analyse des métadonnées.

- **Enregistrement des processus de transformation des données :** MMS permet d'enregistrer automatiquement les informations sur les processus de transformation des données, ce qui facilite la compréhension et la traçabilité des données transformées.

Ces fonctionnalités avancées permettent une modélisation flexible des métadonnées et offrent une grande expressivité dans l'exploration et l'analyse des métadonnées. Cependant, il est important de noter que MMS n'est plus maintenue et n'est plus disponible pour des essais.

1.6.5 Perm (Provenance Extension of the Relational Model)

Perm (Glavic et al. 2013) est un SGBD (Système de Gestion de Base de Données) proposé par Glavic et al. en 2010 qui se concentre sur la provenance des données, c'est-à-dire les informations sur l'origine et le processus de création des données. Son objectif est de générer et de gérer les informations de provenance pour des requêtes SQL complexes, en prenant en charge différents types de provenance.

Principales fonctionnalités de Perm :

- Support de différents types de provenance : Perm prend en charge divers types de provenance, notamment la *why-provenance* en utilisant la sémantique de contribution d'influence Perm-Influence (PI-CS), la *where-provenance* basée sur le travail de Buneman et al., la *how-provenance* en utilisant des polynômes de provenance. (Glavic et al. 2013).

- Génération de provenance pour des requêtes SQL complexes : Perm peut générer la provenance pour des requêtes SQL complexes, y compris les requêtes de sélection-projection-jointure (SPJ), les requêtes avec agrégation, les opérations ensemblistes, les sous-requêtes imbriquées et corrélées, et les requêtes impliquant des fonctions définies par l'utilisateur.

- Représentation relationnelle de la provenance : Perm représente les informations de provenance sous forme de relations, où chaque tuple dans la relation correspond à un tuple du résultat de la requête d'origine augmenté des informations de provenance. Les attributs de provenance sont ajoutés aux attributs d'origine, et la représentation permet la duplication de tuples lorsqu'ils ont plusieurs sources de provenance.

- Extension du langage SQL (SQL-PLE) : Perm étend SQL avec des mots-clés supplémentaires pour demander et contrôler la génération de provenance. Le mot-clé PROVENANCE est utilisé dans la clause SELECT pour calculer la provenance, et le modificateur ON CONTRIBUTION peut être utilisé pour choisir le type de provenance. Le SQL-PLE permet également de limiter la portée de la génération de provenance à l'aide du mot-clé BASERELATION.

- Techniques de réécriture de requêtes : Perm utilise des techniques de réécriture de requêtes pour transformer une requête en une requête générant de la provenance. Ces règles de réécriture sont dérivées de la définition déclarative de chaque type de provenance et peuvent être traduites en réécritures SQL.

- Implémentation : Perm est implémenté sous la forme d'un moteur PostgreSQL modifié, étendant son dialecte SQL avec des fonctionnalités de provenance. La génération de provenance est légère et paresseuse, ce qui signifie qu'aucune provenance n'est générée à moins d'être explicitement demandée.

Malheureusement, le projet Perm a été abandonné et n'est plus poursuivi activement.

1.6.6 GProM (Middleware Générique de Provenance)

GProM (Arab et al. 2018) est un système middleware proposé par Glavic et al. en 2018 qui offre une prise en charge de la provenance pour les opérations de base de données, notamment les requêtes SQL, les mises à jour et les transactions. Il modélise la provenance comme des annotations sur les données et calcule la provenance des sorties des opérations en propageant ces annotations.

Les caractéristiques clés et les contributions de GProM comprennent :

- Modèle de Provenance pour les *Mises à Jour Transactionnelles* : GProM introduit un modèle de provenance et un mécanisme de capture spécifiquement conçus pour les mises à jour transactionnelles. Cela permet de suivre la provenance des modifications de données au sein des transactions.

- Interopérabilité avec d'autres Systèmes de Provenance : GProM prend en charge l'importation et l'exportation de la provenance représentée sous forme de PROV-JSON, permettant une interopérabilité avec d'autres systèmes de provenance utilisant cette représentation standard.
- Optimiseur Basé sur les Coûts : GProM optimise le calcul de la provenance en intégrant un mécanisme basé sur les coûts pour les pipelines d'instrumentation de provenance.
- Middleware de Provenance Générique : GProM sert de middleware générique pouvant être intégré à plusieurs langages frontend (tels que SQL et Datalog) et à des bases de données backend. Il fournit une interface unifiée pour la capture et la gestion de la provenance dans différents systèmes de bases de données : PostgreSQL, Oracle, SQLite, MonetDB.

L'architecture de GProM repose sur la compilation et l'instrumentation des requêtes et des transactions pour capturer la provenance. Il traduit les requêtes de l'utilisateur en une représentation graphique de l'algèbre relationnelle et applique des règles de réécriture de provenance pour propager les annotations. Il prend également en charge le calcul rétroactif [1] de la provenance en utilisant les journaux d'audit et la fonctionnalité de voyage dans le temps du système de base de données sous-jacent.

Nous concluons que, malgré les efforts constants et les contributions dans le domaine de la provenance, certains de ces systèmes de gestion de provenance connaissent une évolution particulièrement lente, voire stagnante, tandis que d'autres ont été complètement abandonnés.

1.7 ProvSQL

Le projet ProvSQL a été lancé en 2018 au sein de l'équipe Valda (Valda Team Accessed 2023), qui fait partie de l'Inria *Institut national de recherche en informatique et en automatique* (Inria Accessed 2023) en France. L'équipe Valda se spécialise dans les aspects fondamentaux et systèmes de la gestion de données. ProvSQL a été développé en tant que module open-source (Senellart Accessed 2023) qui étend le système de gestion de bases de données PostgreSQL (PostgreSQL Global Development Group Accessed 2023) pour prendre en charge la capture, la gestion et l'analyse de la provenance des données, ainsi que le calcul de probabilités dans les bases de données probabilistes. Depuis son lancement, ProvSQL a continué à être développé et testé pour prendre en charge différentes versions de PostgreSQL et s'intégrer de manière transparente dans les installations existantes.

ProvSQL est manipulé à travers le client de PostgreSQL, également connu sous le nom de `PostgreSQL shell` ou `psql`. Ce client est utilisé pour interagir avec la base de données en exécutant des requêtes SQL, en créant des tables, en insérant des données, et en effectuant toutes les opérations nécessaires à la manipulation et à la gestion de la

[1] Le calcul rétroactif de provenance est une fonctionnalité qui permet de déterminer la provenance d'une donnée à partir de sa valeur actuelle et de remonter jusqu'à ses sources d'origine.

provenance des données. Grâce à son interface en ligne de commande conviviale, le client PostgreSQL offre une expérience intuitive et efficace pour travailler avec ProvSQL.

Dans la suite de cette section, nous nous baserons sur les informations présentées en détail dans le papier (Senellart et al. 2018) pour décrire les principales fondations de ProvSQL ainsi que les détails de son implémentation.

1.7.1 Principales fondations de ProvSQL

Support de provenance en semi-anneau et semi-anneau avec monus

Dans ProvSQL, une partie du support de provenance est basée sur les semi-anneaux et les semi-anneaux avec monus. La notion de semi-anneau a été introduite précédemment dans la section 1.5.4, et nous allons maintenant voir comment elle est utilisée dans le contexte de ProvSQL.

Un semi-anneau permet de représenter les opérations d'agrégation et de jointure, ainsi que les relations entre les résultats et les sources de données. Cependant, il ne permet pas de faire des requêtes non-monotones (Hernich 2010) à savoir la différence ensembliste. C'est là que les semi-anneaux avec monus (Geerts et al. 2010) entrent en jeu.

La provenance en semi-anneau avec monus étend les capacités de la provenance en semi-anneau en incorporant des opérations de monus, notées $\ominus$, qui permettent la soustraction d'informations de provenance. Un semi-anneau avec monus est une structure algébrique définie comme un tuple $(S, \oplus, \ominus, \otimes, 0, 1)$, où S, $\oplus$ $\otimes$, 0 et 1 ont les mêmes définitions que dans la provenance en semi-anneau. L'opérateur monus $\ominus$ est défini comme suit : pour tout élément a et b dans le semi-anneau, $a \ominus b = a \wedge \neg b$ pour les fonctions booléennes (Senellart et al. 2018).

La provenance en semi-anneau avec monus fournit un cadre plus complet pour gérer la provenance dans des scénarios d'opérations non-monotones.

Support de la (*Where-provenance*)

Senellart et al. mentionnent dans (Senellart et al. 2018) que la *where-provenance* (Buneman et al. 2001) est une forme de provenance qui utilise un graphe biparti pour relier les valeurs de sortie d'une requête aux valeurs d'entrée correspondantes, indiquant ainsi leur provenance. Cependant, contrairement à la provenance basée sur les semi-anneaux, Cheney et al. ont démontré dans (Cheney et al. 2009) qu'il n'est pas possible de représenter et reconstruire la provenance *où* à l'aide du formalisme du semi-anneau de provenance.

ProvSQL gère la provenance *où* en utilisant une approche basée sur l'algèbre des termes de provenance, qui sera définie dans la sous-section suivante, et ceci en lui introduisant des opérateurs spéciaux. Ces opérateurs permettent de capturer les informations indiquant où une valeur spécifique dans la sortie d'une requête peut provenir de l'entrée.

Circuit d'algèbre de termes *Term Algebra Circuit*

ProvSQL introduit le concept d'un circuit d'algèbre des termes de provenance pour calculer différents types de provenance et de probabilités associés à une requête. Notée «l'algèbre des termes de provenance» (Senellart et al. 2018) et basée sur l'algèbre des termes (Baader et al. 1998), est une généralisation du semi-anneau universel (Green et al. 2007) et du semi-anneau avec monus universel (Senellart et al. 2018). Dans cette généralisation, les opérations effectuées pour obtenir le résultat d'une requête sont représentées sous forme de termes libres utilisant les opérateurs tels que le $\otimes$ la jointure, le $\oplus$ pour l'union, et le monus $\ominus$ pour la différence ensembliste, π pour la projection et $=$ pour l'égalité de sélection entre colonnes. Les autres opérateurs de requête qui n'affectent pas la provenance sont exclus. Au lieu d'utiliser des formules, ProvSQL représente ces termes sous forme de circuits arithmétiques, qui sont souvent plus compacts. En maintenant un circuit d'algèbre des termes de provenance, ProvSQL peut effectuer toutes les opérations nécessaires pour calculer la provenance et les probabilités d'une requête donnée.

Dans le circuit, les opérations sont représentées par des nœuds, et les dépendances de provenance sont représentées par des arêtes entre les nœuds. Chaque nœud correspond à une opération spécifique ou à un élément de données, tandis que les arêtes capturent le flux d'informations de provenance entre les nuds. Le circuit d'algèbre de termes permet la manipulation et le calcul directs de la provenance, y compris des opérations telles que la jointure et la différence ensembliste.

Bases de données Probabilistes et Calcul des Probabilités

Dans les bases de données probabilistes (Suciu 2020), chaque tuple de la base de données est associé à une probabilité indépendante d'être vrai. La probabilité qu'un tuple se trouve dans le résultat d'une requête est calculée en faisant la somme des probabilités de toutes les sous-bases de données qui satisfont la requête.

Pour calculer ces probabilités, ProvSQL utilise des annotations de provenance représentées sous forme de fonctions booléennes dans le semi-anneau avec monus des fonctions booléennes. Différentes approches peuvent être utilisées, telles que l'énumération de tous les mondes possibles ou l'échantillonnage de Monte-Carlo. De plus, ProvSQL exploite des techniques générales de compilation de connaissances pour transformer les fonctions booléennes en une forme plus facilement traitable. Cette approche permet le calcul des probabilités sur des formes normales de négation déterministes décomposables (d-DNNF), ce qui permet des calculs de probabilités efficaces.

Dans le cadre de ce projet, nous ne mettons pas l'accent sur l'aspect des bases de données probabilistes et le calcul de probabilités.

1.7.2 Description de la mise en œuvre du système ProvSQL

ProvSQL est implémenté en tant qu'extension PostgreSQL. Il s'intègre de manière transparente à PostgreSQL, assurant une compatibilité avec différentes versions. ProvSQL se compose de deux composants principaux : un module de réécriture de requêtes et des

fonctions définies par l'utilisateur.

Module de réécriture des requêtes

Le module de réécriture de requêtes est responsable du calcul de la provenance des résultats de requêtes sous forme de portes dans un circuit de provenance. Il utilise des «*hook*» fournis par PostgreSQL, notamment le «*planner hook*», pour effectuer la réécriture des requêtes après leur analyse mais avant leur envoi au planificateur de requêtes. Ce module fonctionne uniquement sur les requêtes qui font référence à des tables compatibles avec ProvSQL. Lors du processus de réécriture, les références directes à la colonne `provsql`, qui n'est pas destinée à être directement manipulée par les utilisateurs, sont ignorées. À la place, une nouvelle colonne `provsql` est générée pour le résultat de la requête, contenant des jetons de provenance qui identifient les portes du circuit de provenance représentant les opérations effectuées pour produire le résultat, selon l'algèbre des termes de provenance.

Les types de requêtes SQL pris en charge par ProvSQL comprennent les requêtes SELECT...FROM...WHERE simples, les requêtes JOIN, les requêtes SELECT avec des sous-requêtes imbriquées, les requêtes GROUP BY sans agrégation, les requêtes SELECT DISTINCT, les requêtes UNION et UNION ALL, ainsi que les requêtes EXCEPT.

Fonctions définies par l'utilisateur *User Defined Functions*

ProvSQL fournit également des fonctions définies par l'utilisateur (UDFs) qui introduisent des annotations de provenance dans les tables existantes et permettent le calcul de différentes formes de provenance et de probabilités à partir du circuit de provenance. Ces UDFs sont définies dans le schéma (*Schemas* s. d.) `provsql` et incluent des fonctions permettant l'ajout de la provenance à une table, la récupération du jeton de provenance de la requête en cours, la création de mappings de provenance, la visualisation du circuit de provenance, l'évaluation du circuit dans différents semi-anneaux, la récupération de la *where-provenance* et le calcul des probabilités en utilisant différentes méthodes d'évaluation. Ces fonctions fournissent une interface pratique et intuitive pour interagir avec les informations de provenance.

1.7.3 Avantages et contributions

ProvSQL offre plusieurs avantages significatifs par rapport aux autres systèmes disponibles dans le domaine de la provenance des données. Les principales différences entre ces systèmes et ProvSQL sont les suivantes :

- **Calcul dans des semi-anneaux définis par l'utilisateur** : Contrairement à ces systèmes, ProvSQL permet le calcul dans des semi-anneaux définis par l'utilisateur. Cela offre une flexibilité et une expressivité supérieures lors de l'analyse et de la gestion de la provenance des données.

- **Prise en charge des semi-anneaux avec monus** : ProvSQL est le seul système qui prend en charge les sémantiques des semi-anneaux avec monus. Cela permet de capturer des comportements non-monotones dans la provenance des données, offrant ainsi une représentation plus complète et précise.
- **Distinction entre sémantiques ensemblistes et multi-ensemblistes** : Contrairement à ces systèmes, ProvSQL fait une distinction claire entre les sémantiques des requêtes ensemblistes (DISTINCT) et multi-ensemblistes. Cela permet de manipuler et d'analyser efficacement les duplications de données dans la provenance.
- **Formalisme de provenance compact** : Alors que ces systèmes utilisent un formalisme de provenance natif qui stocke la provenance à l'aide d'attributs et de lignes supplémentaires dans la table de résultats, ProvSQL utilise un circuit de provenance. Ce formalisme est beaucoup plus compact, ce qui facilite le stockage, la manipulation et l'analyse de la provenance des données.
- **Maintenance continue et support des versions modernes de PostgreSQL** : ProvSQL bénéficie d'une maintenance active, avec des recherches en cours et des améliorations constantes. Il est également compatible avec les versions modernes de PostgreSQL.

Grâce à ces avantages, ProvSQL se distingue en offrant une flexibilité, une expressivité et une efficacité supérieures dans la gestion et l'analyse de la provenance des données au sein des bases de données.

1.8 Conclusion

Dans ce chapitre, nous avons introduit les bases nécessaires pour comprendre le concept de provenance dans les bases de données relationnelles et avons mis en évidence l'innovation apportée par ProvSQL.

Dans le chapitre suivant, nous approfondirons notre exploration des bases de données relationnelles en abordant un autre aspect essentiel : la prise en charge des données temporelles. Cette exploration approfondie nous permettra de mieux comprendre les défis et les opportunités liés à la gestion des données temporelles, et de déterminer comment ces aspects se combinent avec la gestion de la provenance.

Chapitre 2

Les bases de données temporelles

2.1 Introduction

Les bases de données temporelles (Liu et al. 2018) constituent une branche essentielle de la gestion de données, offrant des fonctionnalités spécifiques pour la gestion des informations qui évoluent dans le temps.

Dans toute la suite de ce projet, nous nous focaliserons sur l'utilisation du modèle relationnel (Date et al. 2014) pour la gestion des données temporelles. Cette décision est motivée par plusieurs facteurs, notamment le support exclusif de notre système de gestion de provenance ProvSQL (Senellart 2017) par PostgreSQL, qui est un système de gestion de base de données relationnel. De plus, le modèle relationnel est largement utilisé dans l'industrie en raison de sa popularité et de sa robustesse, bénéficiant du soutien de nombreux systèmes de gestion de bases de données.

Dans ce chapitre consacré aux bases de données temporelles, nous aborderons plusieurs aspects clés. Nous commencerons par explorer les différents modèles (C. S. Jensen et al. 2009 ; Böhlen et al. 1998) de données temporelles, tels que les modèles basés sur les intervalles et les points. Nous détaillerons également les différents types de bases de données temporelles. Ensuite, nous présenterons certains des principaux systèmes de gestion de bases de données temporelles disponibles sur le marché. Enfin, nous nous pencherons sur les bancs d'essai *(benchmarks)* (Dempsey et al. 2018) de bases de données temporelles existants, qui sont utilisés pour évaluer les performances des systèmes et les comparer. Nous examinerons les critères d'évaluation, les métriques et les méthodologies utilisées dans ces bancs d'essai pour mesurer les performances des bases de données temporelles.

L'objectif de ce chapitre est de fournir une compréhension approfondie du concept des bases de données temporelles dans le contexte des bases de données relationnelles. Cette connaissance approfondie nous permettra d'introduire, dans le chapitre suivant, la provenance dans les bases de données temporelles.

2.2 Définition

Une base de données temporelle est une collection de données référencées dans le temps. (Liu et al. 2018), ce qui signifie qu'elles permettent de stocker des informations liées à une date ou une période spécifique.

Bien que les bases de données conventionnelles considèrent que les données qui y sont stockées sont valables à l'instant présent, les bases de données temporelles ajoutent un aspect temporel aux données qui y sont contenues, ce qui permet de mieux comprendre leurs évolution dans le temps.

2.3 Modèles de données temporelles

Une modèle de données est un ensemble de concepts qui peuvent être utilisés pour décrire la structure d'une base de données (Elmasri et al. 1994). Il permet de créer une représentation visuelle des données et d'illustrer comment différents éléments sont liés les uns aux autres. L'utilisation des modèles de données conventionnels tels quels pour la gestion des données temporelles peut être une activité excessivement complexe et propice aux erreurs. La première étape de la prise en charge de la gestion des données temporelles consiste à étendre les structures de base de données du modèle de données pris en charge par le SGBD (C. S. Jensen et al. 1999).

2.4 Aspects temporels des données

Une base de données modélise et enregistre des informations sur une partie de la réalité (C. S. Jensen et al. 1999). Pour représenter la dimension de temps dans une base de données relationnelle, on associe à chaque tuple dans les relations, qui stockent des informations variant dans le temps, les deux notions, qu'on définit selon (C. S. Jensen et al. 1997) :

Temps de validité (*Valid time*) : Le temps de validité désigne la période de temps pendant laquelle un fait est considéré comme vrai par rapport au monde réel. Tout sous-ensemble du domaine temporel peut être associé à un fait en tant que son temps de validité. Ainsi, les estampilles (Dyreson et al. 1993) valides peuvent être des ensembles d'instants temporels et d'intervalles temporels, les instants uniques et les intervalles étant des cas spéciaux importants. Les valeurs de temps de validité sont généralement fournies par les utilisateurs.

Temps de transaction (*Transaction time)* : Le temps de transaction d'un fait d'une base de données correspond au moment où le fait est présent dans la base de données et peut être récupéré. Par conséquent, les temps de transaction ne sont généralement pas des instants temporels mais ont une durée. Les temps de transaction sont liés à l'ordre dans lequel les transactions sont exécutées et sérialisées dans la base de données. Ainsi, ils ne peuvent pas représenter des moments qui se situent dans le futur par rapport au moment présent. De plus, comme il est

impossible de changer le passé, ces mesures ne peuvent pas être modifiées. Les valeurs sont générées et fournies par le système.

2.5 Représentation du temps dans les bases de données temporelles

Les données temporelles peuvent être représentées de différentes manières selon le contexte et les besoins spécifiques de la base de données. Dans la suite de cette section, nous détaillons les quatre principales approches de représentation des données temporelles.

À titre d'illustration, nous considérons la relation `Employé` qui comprend les attributs suivants : `Id`, `Nom` et `Département`. Cette relation illustre la représentation du temps de validité dans les bases de données temporelles.

1. **Estampillage instantané des tuples** (Ariav 1986) : Chaque tuple de la base de données est associé à un moment précis dans le temps. Cela peut être réalisé en ajoutant un attribut temporel qui enregistre la date et l'heure exactes d'insertion ou de modification du tuple (voir le tableau 2.1).

Id	Nom	Département	Estampille instantanée
1	Smith	RH	2022-01-15 09:30:00
2	Johnson	Ventes	2022-03-02 14:15:00
3	Williams	IT	2022-02-18 11:45:00
4	Brown	RH	2022-04-10 16:20:00

Tab. 2.1 : Table `Employé` avec estampille instantanée du temps

2. **Estampillage par intervalle des tuples** (Navathe et al. 1989 ; Sarda 1990 ; Snodgrass et al. 1986) : Les périodes pendant lesquelles les données sont valides sont représentées à l'aide d'intervalles. Chaque tuple est associé à un intervalle de temps défini par une date de début et une date de fin. Cela permet de capturer les changements des données dans le temps et de traiter les requêtes portant sur des périodes spécifiques (voir le tableau 2.2.

Id	Nom	Département	période de validité
1	Smith	RH	[2017-01-15, 2023-05-17]
2	Johnson	Ventes	[2012-03-02, 2020-02-17]
3	Williams	IT	[2022-02-18, présent]
4	Brown	RH	[2022-04-10, présent]

Tab. 2.2 : Table `Employé` avec estampillage par intervalle des tuples

3. **Estampillage instantané des attributs** (Clifford et al. 1985) : Chaque attribut d'un tuple est associé à un instant précis dans le temps. Ainsi, chaque valeur

Id	Nom	Département
1	Smith	RH, 2022-01-15 09:30:00
2	Johnson	Ventes, 2022-03-02 14:15:00
3	Williams	IT, 2022-02-18 11:45:00
4	Brown	RH, 2022-04-10 16:20:00

TAB. 2.3 : Table `Employé` avec estampillage instantané de l'attribut *Département*

d'attribut est estampillée avec la date et l'heure à laquelle elle est devenue valide (voir le tableau 2.3).

4. **Estampillage par intervalle des attributs** (CLIFFORD et al. 1985) : Cette approche est similaire à l'estampillage d'instant, mais au lieu de représenter un instant précis, elle utilise des intervalles de temps pour estampiller les valeurs des attributs. Chaque valeur d'attribut est associée à un intervalle de temps indiquant la période pendant laquelle cette valeur est valide (voir tableau 2.4).

Id	Nom	Département
1	Smith	RH, [2017-01-15, 2023-05-17]
2	Johnson	Ventes, [2012-03-02, 2020-02-17]
3	Williams	IT, [2022-02-18, présent]
4	Brown	RH, [2022-04-10, présent]

TAB. 2.4 : Table `Employé` avec estampillage par intervalle de l'attribut *Département*

2.6 Types de bases de données temporelles

L'univers des bases de données temporelles est vaste et diversifié. D'après (SNODGRASS et al. 1986), elles peuvent être classées en quatre types, que nous allons présenter dans la suite de cette section, en fonction de leur capacité à prendre en charge les concepts temporels (cités dans la section précédente) et le traitement des informations temporelles.

Dans le cadre du modèle relationnel (DATE et al. 2014), une base de données est organisée en relations, qui sont représentées sous forme de tables bi-dimensionnelles. Chaque table dans la base de données correspond à une entité ou à un concept spécifique, et chaque ligne de la table représente un enregistrement individuel de cette entité. Les colonnes de la table représentent les attributs ou les caractéristiques de l'entité.

Dans cette section, nous utiliserons un exemple simple pour illustrer nos concepts. Considérons une table `Citoyen` (voir le tableau 2.5) avec deux colonnes : `nom` et `adresse`. Cette table représente une liste de personnes et leurs adresses respectives.

1. **Bases de données instantanées** (Snapshot Databases) : Elles enregistrent l'état d'une base de données à un moment donné (SNODGRASS et al. 1986), et conservent la version la plus récente des données disponibles. Cependant, ces bases de données

id_citoyen	nom	adresse
1	Anis	13 Rue Senac de Meilhan 13001 Marseille
2	Salima	60 Rue la Fayette 75009 Paris
3	Djihane	34 Rue Daubenton 75005 Paris

TAB. 2.5 : La relation *Citoyen*

ne conservent pas d'informations sur les modifications passées ni les moments auxquels elles ont été effectuées (MITSA 2010). Par conséquent, il n'est pas possible de consulter l'état de la base de données dans le passé ni de retracer les erreurs corrigées.

Par exemple, si Anis déménage à Lyon, le tuple correspondant sera mis à jour en remplaçant l'ancienne adresse par la nouvelle adresse (voir le tableau 2.6). Toutefois, il n'y aura aucune information conservée sur le fait qu'Anis habitait auparavant à Marseille. De même, si Anis n'est plus résident en France ou s'il est décédé, le tuple correspondant sera simplement supprimé de la table, sans garder de trace de ces changements passés. Ce type de bases de données ne supporte ni les requêtes du

id_citoyen	nom	adresse
1	Anis	13 Rue Senac de Meilhan 13001 Lyon
2	Salima	60 Rue la Fayette 75009 Paris
3	Djihane	34 Rue Daubenton 75005 Paris

TAB. 2.6 : La relation *Citoyen* dans une base de données instantanées

genre :

- Où habitait Djihane il y a 9 ans ?
- Salima a-t-elle changé son adresse au cours des deux dernières années ?

Ni les événements du type :

- Anis a changé d'adresse l'année dernière.
- Salima va déménager à Nice le prochain mois

2. **Bases de données de restauration** (Rollback Databases) : comme leur nom l'indique, ce type de bases de données permet d'effectuer des requêtes pour connaître l'état de la base de données à un moment donné dans le passé. Elles enregistrent l'état d'une base de données à différents moments dans le temps et conservent l'historique complet des modifications effectuées sur les données. Les opérations de mise à jour sont réalisées en insérant une nouvelle ligne dans la table. Chaque mise à jour effectuée par une transaction dans une base de données rollback est marquée avec la même estampille de transaction, qui est un attribut supplémentaire dans la relation (LOMET et al. 1992). Dans les bases de données rollback, le concept de temps de transaction est utilisé pour déterminer quand la transaction a été effectuée. Par exemple, dans le tableau 2.7, nous pouvons voir l'événement «Anis a déménagé

id_citoyen	nom	adresse	transaction time
1	Anis	13 Rue Senac de Meilhan 13001 Marseille	2017-01-08
2	Salima	60 Rue la Fayette 75009 Paris	2013-11-23
3	Djihane	34 Rue Daubenton 75005 Paris	2018-02-13
4	Anis	35 Cami De Buret 64170 Lyon	2022-05-01

Tab. 2.7 : La relation *Citoyen* dans une base de données de restauration

de Marseille à Lyon le 1er mai 2022» représenté par l'insertion d'une nouvelle ligne dans la table avec le temps de transaction correspondant.

3. **Bases de données historiques** (Historical Databases) : Elles enregistrent à la fois les données et leur période de validité. Elles stockent un seul état historique pour chaque relation, ce qui se traduit par la modification directe de la base de données pour corriger une erreur en modifiant le temps de validité associé à chaque tuple. Cependant, cela signifie qu'il n'est pas possible de visualiser la base de données telle qu'elle était dans le passé et qu'il n'y a pas de trace des erreurs corrigées(Snodgrass et al. 1986).

   Dans les bases de données historiques, le concept de temps valide est utilisé pour déterminer la période pendant laquelle une relation ou un tuple est considéré comme valide. Cela permet de définir la validité des données dans le contexte de la base de données modélisée en fonction de leur période de validité spécifique. Ainsi, en utilisant le temps de validité, il est possible de déterminer quelles informations étaient vraies à un moment donné dans le passé, ce qui ouvre la possibilité d'effectuer des requêtes ou des analyses basées sur cette information temporelle.

   À titre d'exemple, prenons le cas où Anis déménage de Marseille à Lyon le 1er mai 2022. Pour refléter cet événement dans la base de données, nous mettons à jour l'intervalle de validité du tuple correspondant à Anis et insérons un nouveau tuple avec la nouvelle adresse à Lyon (voir le tableau 2.8).

id_citoyen	nom	adresse	valide time
1	Anis	13 Rue Senac de Meilhan 13001 Marseille	[2017-01-08 ; 2022-05-01]
2	Salima	60 Rue la Fayette 75009 Paris	[2013-11-23 ; -]
3	Djihane	34 Rue Daubenton 75005 Paris	[2018-02-13 ; -]
4	Anis	35 Cami De Buret 64170 Lyon	[2022-05-01 ; -]

Tab. 2.8 : La relation *Citoyen* dans une base de données historique

4. **Bases de données temporelles** (Temporal Databases) : aussi connues sous le nom de «bases de données bi-temporelles», elles prennent en charge à la fois le temps valide (valid time) et le temps de transaction (transaction time). Le terme «bi-temporelles» fait référence au fait qu'elles supportent deux aspects temporels distincts. Le temps valide représente la période pendant laquelle un tuple est considéré comme valide ou vrai dans le contexte de la base de données modélisée (Snodgrass et al. 1986), tandis que le temps de transaction indique le moment où le tuple était

présent dans la base de données (Clifford et al. 1994). En intégrant ces deux dimensions, les bases de données temporelles offrent un cadre complet pour la gestion des informations temporelles. Les utilisateurs peuvent ainsi analyser et interroger les données en tenant compte de leur validité à différents moments, tout en suivant l'historique des changements et des transactions.

Maintenant que nous avons une définition approfondie des quatre types de bases de données temporelles, nous présentons dans le tableau 2.9 une comparaison visant à mettre en évidence leurs différences et leurs avantages respectifs. Pour les besoins de ce projet

Caractéristiques	**Instantanée**	**Restauration**	**Historique**	**Temporelle**
Temps supporté	-	Temps de transaction	Temps de validité	Temps de validité et temps de transaction
Mécanismes de manipulation des relations temporelles	Insertion, suppression, mise à jour	Insertion seulement	Mise à jour, Insertion	Insertion, suppression, mise à jour
Conservation des états passés	Non	Oui	Non	Oui
Suivi des modifications passées	Non	Oui	Non	Oui
Visibilité des erreurs corrigées	Non	Oui	Non	Oui
Support de l'aspect de validité de l'information	Aucun	Tant qu'un tuple est présent dans la base de données il est considéré effectif	Un tuple peut être présent dans une table même s'il n'est pas considéré comme valide	Oui

Tab. 2.9 : Comparaison des types de bases de données temporelles

et dans un souci de simplification, nous nous concentrons spécifiquement sur les bases de données temporelles qui ne prennent en charge que le temps de validité des données, tout en les désignant sous le terme générique de *bases de données temporelles*. Dans les prochains chapitres, nous explorerons les différentes techniques et approches permettant de gérer la provenance dans un contexte temporel.

2.7 Systèmes de gestion de bases de données temporelles

Dans cette section, nous examinerons les systèmes de gestion de bases de données qui ont intégré la gestion de l'évolution des données dans le temps.

2.7.1 IBM

Selon la documentation d'IBM sur le travail avec les tables temporelles à période système (IBM 2023), la gestion des données temporelles implique la capacité de suivre et d'analyser les changements des données au fil du temps, en conservant les versions historiques des lignes dans les tables. Avec les solutions d'IBM, telles que DB2 (IBM s. d.), la gestion des données temporelles est réalisée grâce à l'utilisation de tables temporelles à période système et de leurs tables d'historique associées. Ces tables captent les versions actuelles et passées des données, permettant un stockage, une récupération et une analyse efficaces des données à différents moments.

1. **Les tables temporelles à période système** (system-period temporal tables) : Elles sont organisées de manière à conserver l'historique des lignes modifiées au fil du temps. Chaque ligne de la table temporelle contient plusieurs colonnes spéciales :

   - Colonne `row-begin` : Cette colonne enregistre le moment où les données de la ligne sont devenues actuelles, c'est-à-dire le moment où la ligne a été insérée ou mise à jour pour la première fois.
   - Colonne `row-end` : Cette colonne enregistre le moment où les données de la ligne ne sont plus actuelles, soit parce qu'elles ont été mises à jour ou supprimées ultérieurement.
   - Colonne `transaction start-ID` : Cette colonne capture le moment de la première opération de modification des données dans une transaction. Elle permet de suivre les modifications apportées à une ligne spécifique lors d'une transaction donnée.
   - Colonne `SYSTEM_TIME period` : indique la période de validité de la version d'une ligne.

   Ces colonnes spéciales permettent de déterminer la période pendant laquelle chaque version d'une ligne est valide. Lorsqu'une ligne est modifiée, la base de données insère une copie de l'ancienne version de la ligne dans la table d'historique associée, en mettant à jour la colonne de fin de ligne avec le moment de la modification. Ainsi, l'historique des versions précédentes des lignes est préservé et peut être consulté ultérieurement.

2. **Les tables d'historique** (History tables) : La table d'historique est et similaire en terme de structure et à la table temporelle à période système (system-period temporal table). Lorsqu'une ligne est modifiée ou supprimée dans la table temporelle à période système, une copie de l'ancienne version de la ligne est insérée dans la table

d'historique associée, avec un timestamp de fin de ligne mis à jour. Cela permet de conserver les données précédentes de la table temporelle à période système, ce qui permet de récupérer des données à des moments antérieurs.

2.7.2 Oracle Database

Oracle Database (*Oracle Database 19c Documentation : Application Development, Features, and Security* s. d.) est un Système de Gestion de Base de Données Relationnelle (SGBDR) puissant et évolutif qui permet de stocker des données structurées. Il offre des fonctionnalités avancées telles que l'indexation, la compression, le contrôle de la concurrence et la réplication des données pour assurer l'intégrité, la disponibilité et les performances optimales des données. Il prend en charge le langage SQL pour interroger et manipuler les données de manière efficace.

Le support de la validité temporelle a été ajouté à Oracle Database avec la sortie d'Oracle Database 12c en juin 2013 (Emrah Mete 2020). Il permet d'associer une ou plusieurs dimensions de temps de validité à une table, déterminant la visibilité des données en fonction de leur temps de validité (*Oracle Database 19c Documentation : Design Basics* s. d.).

Pour ajouter l'aspect temporel à une table dans Oracle, il faut définir une dimension de temps de validité pour une table en utilisant la clause `PERIOD FOR`. Cela implique de spécifier des colonnes de début et de fin de validité qui représentent la période de validité de chaque enregistrement. Ces colonnes peuvent être ajoutées manuellement à la table ou créées automatiquement par la base de données.

Le support des bases de données temporelles s'intègre à la technologie *Oracle Flashback*, ce qui permet d'effectuer des requêtes à l'aide des clauses `AS OF` et `VERSIONS BETWEEN`. La clause `AS OF` permet de récupérer les données telles qu'elles étaient à un moment précis, tandis que la clause `VERSIONS BETWEEN` permet de récupérer différentes versions des enregistrements dans une plage de temps spécifiée.

De plus, Oracle fournit la procédure `DBMS_FLASHBACK_ARCHIVE.ENABLE_AT_VALID_TIME` (Jean-Francois Verrier and Dominique Jeunot s. d.), qui permet de contrôler la visibilité des données de la table au niveau de la session. Il est aussi possible de choisir d'afficher toutes les données de la table, les données valides à un moment spécifique ou les données actuellement valides dans la période de validité temporelle.

2.7.3 SAP HANA

SAP HANA (SAP s. d.[b]) est un système de gestion de base de données en mémoire *in-memory* [1] développé par la société allemande SAP SE (SAP s. d.[a]). Il est conçu pour gérer et traiter de grands volumes de données en temps réel. L'acronyme «HANA» signifie «High-Performance Analytic Appliance».

SAP HANA prend en charge le concept de tables temporelles, qui permettent la gestion

[1] Un système de gestion de base de données *en mémoire* stocke et traite les données principalement en mémoire vive (RAM) plutôt que sur des disques traditionnels.

des données historiques et le suivi des modifications dans le temps. Il existe deux types de tables temporelles dans SAP HANA : les tables versionnées *(System-Versioned Tables)* par le système et les tables basées sur une période spécifique à l'application *(Application-Time Period Tables)*.

Les tables versionnées par le système, qui font partie de la norme SQL, suivent les modifications apportées aux tables de stockage en colonnes en capturant la période de validité de chaque enregistrement. Dans SAP HANA, les tables versionnées par le système sont composées d'une table principale pour les enregistrements actuellement valides et d'une table d'historique correspondante pour les enregistrements archivés. Le système maintient automatiquement les valeurs de début de validité *(Valid from)* et de fin de validité *(Valid to)* pour chaque enregistrement. Lors de l'exécution des commandes d'insertion, de suppression ou de mise à jour sur une table versionnée par le système, les enregistrements supprimés ou mis à jour sont archivés dans la table d'historique grâce à des déclencheurs internes.

Les tables d'historique *(History Tables)* dans SAP HANA ont une structure similaire à celle de la table principale, mais elles ne possèdent pas de contrainte de clé primaire et les colonnes Valid from et Valid to sont définies sans clause de génération.

Pour interroger les données historiques, des valeurs d'estampille de temps peuvent être utilisées pour faire référence aux enregistrements archivés dans la table d'historique. Des opérateurs temporels tels que `AS OF`, `FROM - TO` ou `BETWEEN - AND` peuvent être utilisés pour sélectionner des données basées sur des plages de temps spécifiques ou des points dans le temps.

En plus des tables versionnées par le système, SAP HANA a également introduit les tables basées sur une période spécifique à l'application. Ces tables permettent de gérer et de manipuler les données historiques de l'entreprise en fonction de périodes spécifiques à l'application (Kaufmann et al. 2013b). Les tables basées sur une période spécifique à l'application capturent la validité des enregistrements dans le monde de l'entreprise plutôt que sur la base du moment où ils ont été saisis dans la base de données. Ces tables ne sont disponibles que dans le stockage en colonnes et peuvent être combinées avec les tables versionnées par le système pour créer des tables bi-temporelles (Johnston 2014) pour les scénarios nécessitant à la fois le temps de validité et le temps de transaction.

Différentes vues système, telles que `SYS.TABLES` et `SYS.TEMPORAL_TABLES`, fournissent des informations sur les tables temporelles, notamment leur type (versionnées par le système, basées sur une période spécifique à l'application ou historiques) et les tables d'historique associées.

2.8 Bancs d'essai de bases de données temporelles

Un banc d'essai *benchmark* est un ensemble qui comprend généralement : des données, un schéma pour ces données et des charges de travail (*workloads*) qui effectuent des calculs sur ces données. (Dempsey et al. 2018). Ces outils, permettent de comparer de manière objective des systèmes logiciels ou matériels concurrents, des architectures, des algorithmes ou d'autres technologies, tout en évaluant une technologie spécifique dans

différents scénarios. (S. W. Thomas et al. 2014)

Un banc d'essai de base de données temporelle est utilisé pour évaluer et comparer les performances des systèmes de bases de données qui gèrent des données temporelles. Dans la suite, nous présentons certains bancs d'essai temporels qui comprennent des bases de données spécifiques, des schémas adaptés à ces bases de données, ainsi que des charges de travail, dans le but d'évaluer les bases de données qui prennent en charge les données temporelles.

2.8.1 TSQL (The Temporal SQL Benchmark)

Le benchmark Temporel de SQL (TSQL) comprend 150 requêtes conçues pour tester les spécificités des langages de requête temporels. Il a été créé par un groupe de chercheurs et présenté lors de l'atelier international ARPA/NSF sur l'infrastructure des bases de données temporelles en 1993. Le banc d'essai vise à évaluer la convivialité *(user-friendliness)* des bases de données temporelles et couvre un large éventail d'opérations de requête. Cependant, il ne garantit pas une couverture complète de tous les aspects des requêtes temporelles. Le banc d'essai n'inclut pas les requêtes récursives, les changements de schéma, les insertions, les suppressions, les mises à jour ou les requêtes sur des données futures. Le schéma utilisé dans le banc d'essai est maintenu simple et comprend trois ensembles d'entités : employés, compétences et départements. Les données proposées pour le banc d'essai sont également simplifiées, avec des recommandations pour les salaires et les dates de transaction. Les requêtes du banc d'essai sont catégorisées en fonction d'une taxonomie qui prend en compte le type de sortie, tel que les valeurs d'attribut explicites, les composants de temps valide ou les deux. Les requêtes du banc d'essai sont présentées en anglais simple pour s'adapter à différentes implémentations de schéma et de systèmes de gestion de base de données (C. Jensen et al. 1993).

2.8.2 τBench

τBench (S. W. Thomas et al. 2014) est un cadre d'évaluation *(benchmark framework)* [2] dérivé du benchmark XBench (Yao et al. 2004). Il est conçu pour évaluer les performances des bases de données temporelles et des langages de requête temporels. Le framework comprend une suite de bancs d'essai couvrant à la fois des aspects temporels et non temporels couvrant diverses technologies, notamment XML, XML Schema (World Wide Web Consortium 2004), XQuery (World Wide Web Consortium 2017), τXSchema (S. Thomas 2009 ; Snodgrass et al. 2008 ; Currim et al. 2009), τXQuery (Gao et al. 2003), PSM (Melton 1998), and τPSM (Gao 2011). La Figure 2.1 représente la relation entre les références.

Le cadre d'évaluation [2] τBench propose un ensemble d'outils et de méthodologies pour générer, valider et exécuter des bancs d'essai temporels. Il offre une approche pour tester et comparer l'efficacité et l'efficience des systèmes de bases de données temporelles et des langages de requête temporels comme τXQuery (Gao et al. 2003). Les bancs d'essai de τBench simulent des scénarios temporels du monde réel et évaluent les performances de

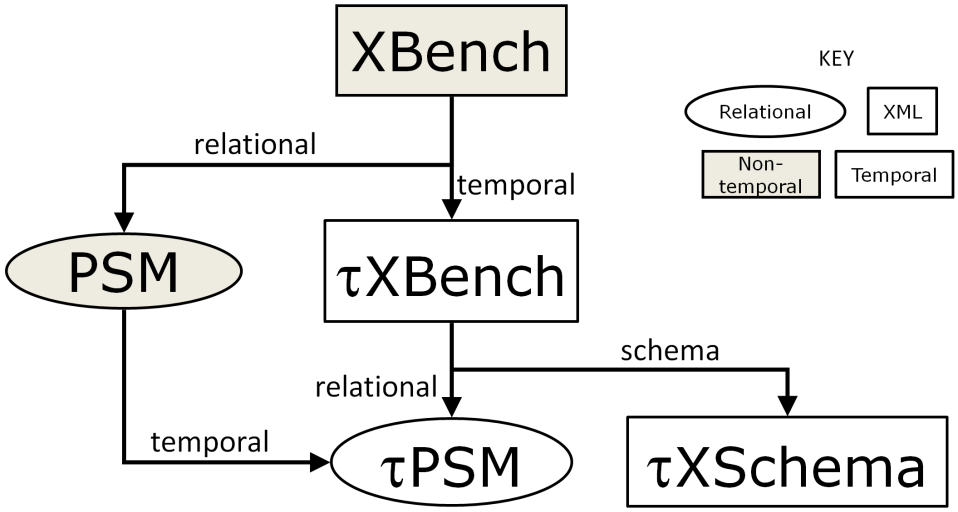


Fig. 2.1 : Vue d'ensemble de la suite de bancs d'essai (*τBench : Benchmark Suite* 2023).

la gestion des données temporelles et du traitement des requêtes (Kuleshova 2011).

2.8.3 TPC-BiH

TPC-BiH est une extension proposée du benchmark TPC-H (Council 2022) conçue pour évaluer les performances et la fonctionnalité des bases de données bi-temporelles (Snodgrass et al. 1986). Il décrit principalement les clients, les commandes et les articles d'une entreprise fictive dont le schéma est illustré dans la figure 2.2. TPC-BiH une évaluation complète de ces bases de données en intégrant une variété de types de requêtes et d'opérations couvrant divers aspects de la manipulation et de la récupération de données temporelles (Kaufmann et al. 2013c) :

1. Requêtes de voyages temporels synthétiques *(Synthetic Time Travel)* : Ce groupe de requêtes permet d'accéder à des tranches spécifiques dans le temps pour établir l'état des tables. Les requêtes testent différentes combinaisons d'opérations temporelles et explorent l'impact des modèles de mise à jour des données. Elles fournissent des informations sur les performances et l'optimisation des opérations de voyage temporel dans des scénarios variés.

2. Voyage temporel pour les requêtes d'application *(Time Travel for Application Queries)* : testent les performances des requêtes combinant le voyage temporel dans différentes dimensions temporelles. Les requêtes en tranche de temps étudient les charges de travail spécifiques aux applications et évaluent les performances du voyage temporel synthétique dans des requêtes analytiques complexes. Elles permettent de mesurer les coûts d'accès aux données actuelles par rapport à des tables non-temporelles équivalentes.

[2]Traduit de l'anglais «benchmark framework», un cadre d'évaluation est un écosystème de bancs d'essai étroitement liés, ainsi que des outils et des techniques partagés qui génèrent et valident des résultats. Il permet de créer et d'évaluer des bancs d'essai à partir de ceux qui existent déjà.

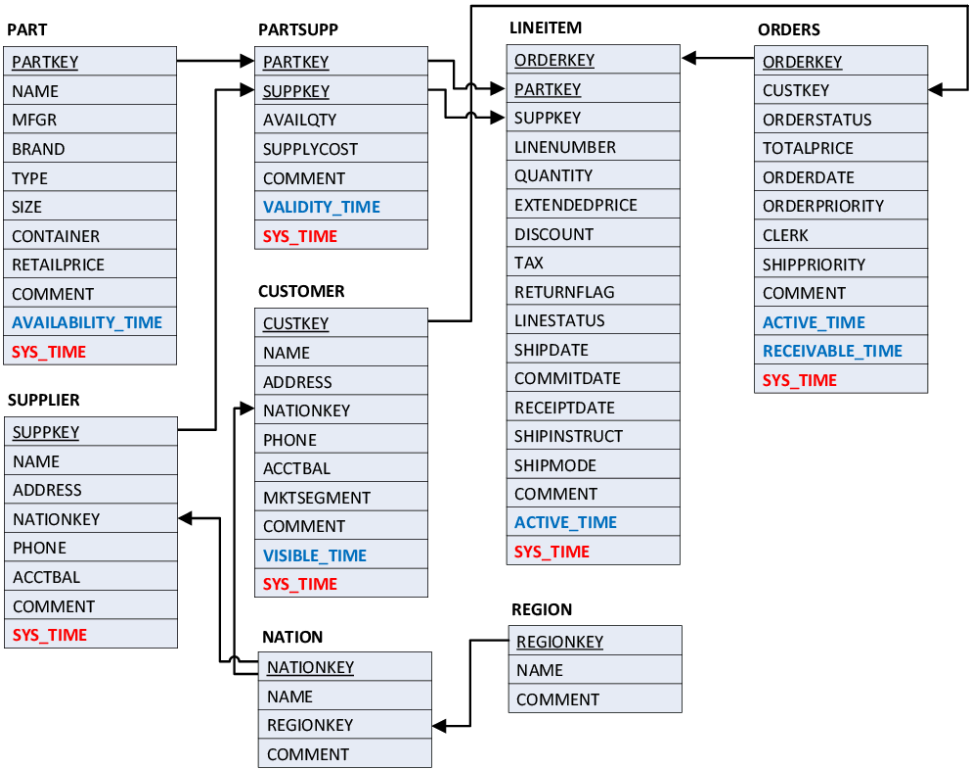


FIG. 2.2 : Schéma de données du benchmark TPC-BiH (KAUFMANN et al. 2013a)

3. Requêtes d'Audit *(Pure-Key Queries)* : étudient l'évolution des tuples au fil du temps pour des tâches d'audit et de détection de tendances. Elles utilisent des sélections basées sur les clés, les plages temporelles, les contraintes de version et les prédicats de valeur pour optimiser l'accès aux données temporelles et comprendre l'organisation du stockage.

4. Requêtes de plage temporelle *(Range-Timeslice Queries)* : traitent des requêtes complexes impliquant des contraintes sur les valeurs et les aspects temporels. Elles couvrent la modélisation des changements d'état, la durée des états, l'agrégation temporelle et les jointures temporelles, nécessitant des opérations avancées pour obtenir les résultats.

5. Requêtes bi-temporelles *(Bitemporal Queries)* : exploitent les deux dimensions temporelles de manière distincte, couvrant différentes combinaisons d'utilisation. Elles permettent d'obtenir des résultats spécifiques à chaque dimension temporelle et offrent une compréhension approfondie des opérations impliquant les deux dimensions.

L'inclusion d'une large gamme de types de requêtes et d'opérations, offre une évaluation complète des capacités de ces bases de données dans la gestion des données temporelles. TPC-BiH constitue un outil essentiel pour les chercheurs et les praticiens, favorisant les avancées dans les systèmes de bases de données bi-temporelles (JOHNSTON 2014) et améliorant la gestion des données temporelles dans diverses applications.

Un benchmark vise donc à fournir une référence ou un standard pour évaluer et comparer les performances et les capacités des systèmes.

2.9 La provenance dans les bases de données temporelles

Avec l'essor constant de l'industrie des données et la prolifération des activités de collecte, de traitement et d'exploitation des données, il est devenu essentiel de prendre en compte le contexte temporel des données ainsi que leur période de validité. En effet, ces données jouent un rôle crucial dans la prise de décisions éclairées et l'amélioration des performances commerciales. Dans cette optique, de nombreuses initiatives ont été lancées pour intégrer la gestion des données temporelles dans les systèmes de gestion de bases de données, tels qu'Oracle, IBM et SAP Hana. Cette évolution a suscité un vif intérêt de la part des chercheurs qui se sont penchés sur la question de la provenance dans les bases de données temporelles. Dans cette section, nous examinerons de près certaines des contributions notables qui ont été apportées à ce domaine.

Une contribution importante dans ce domaine est le travail de Smith et al. (Chen et al. 2012) qui ont introduit une nouvelle méthode de représentation temporelle pour les graphes de provenance. La méthode proposée utilise l'algorithme Logical Clock-P et partitionne les graphes en fonction du temps, capturant les aspects temporels de l'exécution du workflow et permettant l'analyse des motifs temporels. L'efficacité de cette représentation est évaluée en appliquant des techniques de fouille de données pour extraire des informations pertinentes des graphes de provenance. La détermination du type de workflow a généré un graphe de provenance. Des algorithmes de regroupement non supervisés, tels que K-means, sont appliqués à la représentation temporelle à la fois dans le domaine du temps et dans le domaine de la fréquence. Les performances de l'algorithme de regroupement sont évaluées à l'aide de mesures telles que la somme des carrés intra-cluster (WCSS) et la pureté. La fouille de règles d'association, en particulier l'algorithme Apriori, est utilisée pour découvrir des règles pertinentes au sein des clusters. L'étude expérimentale menée sur une base de données de provenance de 10 Go démontre l'efficacité de la méthode de représentation temporelle proposée et des techniques de fouille de données. L'étude montre que la représentation temporelle prend en charge les tâches de regroupement et la classification précise des types de workflow. Elle révèle également la découverte de règles d'association significatives qui mettent en évidence les variantes dans l'exécution du workflow.

Un autre travail (Kuleshova 2011) mené par Kuleshova a exploré l'extension de l'algèbre relationnelle pour inclure des informations de provenance dans les bases de données temporelles. L'objectif de ce travail était de développer des techniques de provenance pour les bases de données temporelles. En exploitant la propriété de réductibilité instantanée de ces bases de données, elle a proposé une méthode de traçabilité ponctuelle pour les données temporelles. Par la suite, elle a présenté un modèle basé sur des intervalles, permettant de regrouper les points temporels ayant la même traçabilité tout en préservant cette traçabilité. Ces concepts ont été illustrés à l'aide d'exemples concrets.

Kuleshova (Kuleshova 2011) a également défini une algèbre relationnelle positive temporelle basée sur des ensembles, permettant ainsi la propagation automatique de la provenance pour capturer ces informations. Cependant, il a été constaté que cette algèbre généralisée, qui opère sur les relations annotées, était uniquement définie pour l'algèbre relationnelle positive, c'est-à-dire pour les opérateurs de sélection, projection, union et jointure naturelle. Kuleshova a donc souligné qu'à ce stade, il n'était pas possible de remplacer la trace de la provenance par une algèbre temporelle basée sur des ensembles qui permettrait de propager la provenance.

Enfin, elle a suggéré d'explorer d'autres structures mathématiques afin de rendre possible la propagation de la provenance pour tous les opérateurs de l'algèbre relationnelle, ouvrant ainsi de nouvelles perspectives pour des études futures dans ce domaine.

Nous prenons en compte le travail réalisé par Kuleshova dans les contributions de ProvSQL, qui seront discutées dans les chapitres suivants.

2.10 Conclusion

Dans ce chapitre, nous avons réalisé une revue approfondie de la littérature portant sur les bases de données temporelles. Nous avons exploré différents aspects de ce domaine, notamment la représentation du temps, les types de données temporelles, les systèmes de gestion de bases de données temporelles, les applications et les travaux de recherche sur la provenance temporelle.

Nous avons examiné les approches de représentation du temps utilisées dans les bases de données temporelles, en mettant en évidence des modèles de données tels que les modèles basés sur les instants et les modèles basés sur les intervalles. De plus, nous avons identifié les types de données temporelles couramment collectées, tels que les informations sur les périodes de validité des données et de présence dans les bases de données.

En étudiant les systèmes de gestion de bases de données temporelles existants, nous avons constaté l'existence de solutions qui répondent aux besoins de gestion et d'analyse des données temporelles. Cependant, nous avons également identifié que ces solutions sont encore loin de répondre aux questions de provenance des données temporelles.

De plus, nous avons exploré les travaux de recherche sur la provenance dans les bases de données temporelles, mettant en évidence des contributions telles que la représentation temporelle des graphes de provenance et l'extension de l'algèbre relationnelle pour inclure des informations de provenance temporelle.

En résumé, ce chapitre, en combinaison avec le premier chapitre, nous a fourni une base solide de connaissances pour mener à bien nos contributions dans ce projet. Nous avons acquis une compréhension approfondie des bases de données temporelles, de leurs fonctionnalités, de leurs applications et des travaux de recherche liés à la gestion de la provenance dans ces bases de données. Cette connaissance approfondie nous permettra d'aborder le développement de notre contribution dans ProvSQL de manière éclairée et pertinente.

partie II

Contribution

Chapitre 3

Conception

3.1 Introduction et motivation

La principale particularité des bases de données temporelles réside dans la prise en compte de laspect temporel des données présentes dans les tables dites *dynamiques*. Ces tables sont souvent sujettes à des mises à jour fréquentes et leurs données peuvent varier au fil du temps. Pour gérer cette évolution dans les bases de données relationnelles, les relations sont étendues pour contenir des informations relatives aux temps de transactions et aux temps de validité des données, en fonction de la nature des tables : tables de restauration, tables historiques ou tables bi-temporelles.

Les informations temporelles peuvent être associées soit aux tuples eux-mêmes, soit aux attributs des tables. Dans le premier cas, chaque tuple est doté d'un ou deux attributs supplémentaires dédiés à l'aspect temporel : le temps de validité ou le temps de transaction (Snodgrass et al. 1986). Le temps de validité spécifie la période pendant laquelle les informations contenues dans le tuple sont considérées comme valides, tandis que le temps de transaction représente le moment précis où le tuple a été inséré, mis à jour ou supprimé dans la base de données.

Dans le second cas, chaque attribut peut également être enrichi avec des informations temporelles. Ainsi, en plus de l'information qu'il stocke, chaque attribut dispose d'une ou deux informations supplémentaires pour représenter l'aspect temporel. Le temps de validité indique la période de validité de la valeur de l'attribut, tandis que le temps de transaction enregistre le moment précis de la modification de cette valeur.

Lorsqu'une requête `SELECT` incluant des opérations combinant des tuples, telles que la jointure, la différence ou l'agrégation, est effectuée sur ces tables, l'une des informations que l'on cherche à obtenir en plus des résultats de la requête est le temps de validité (dont nous nous intéressons spécifiquement dans ce projet) associé à chaque tuple renvoyé. Cependant, la plupart des systèmes de gestion de base de données (SGBD) ne fournissent pas de mécanismes directs pour interroger ou déduire cette information.

Afin de combler cette lacune dans PostgreSQL, nous avons développé une méthode basée sur les semi-anneaux qui étend les fonctionnalités de ProvSQL (Senellart et al. 2018), permettant ainsi d'interroger et de déduire les temps de validité des données renvoyées par les résultats des requêtes.

Dans la suite de ce chapitre, nous allons présenter les fondements de notre solution basée sur les semi-anneaux de provenance. Nous aborderons les principes théoriques qui sous-tendent notre approche et expliquerons les différentes étapes de sa conception.

3.2 Ajout du support aux bases de données temporelles

3.2.1 Description de la solution

L'information de provenance que nous cherchons à extraire dans le cadre de ce projet est le temps de validité des résultats des requêtes. En d'autres termes, lorsque nous exécutons une requête impliquant des opérations telles que l'agrégation, l'union ou la jointure, nous souhaitons connaître la période pendant laquelle les tuples renvoyés étaient vrais dans la base de données. Cette information n'est généralement pas disponible dans les systèmes de gestion de bases de données traditionnels, car son calcul nécessite un travail supplémentaire que nous proposons d'implémenter dans ProvSQL (Senellart et al. 2018).

Notre solution consiste à définir un semi-anneau spécifique pour modéliser les temps de validité des données dans les requêtes SQL. Un semi-anneau est une structure algébrique qui possède deux opérations, généralement appelées «addition» et «multiplication», qui satisfont certaines propriétés mathématiques.

Dans notre cas, nous définissons un semi-anneau d'unions d'intervalles de temps pour représenter les temps de validité. Chaque union d'intervalles de temps correspond à une période pendant laquelle les données sont valides dans la base de données. Nous utilisons les types de données "intervalle de valeurs multiples" disponibles dans PostgreSQL, tels que `datemultirange`, `tsmultirange` ou `tstzmultirange`, en fonction des besoins temporels spécifiques.

L'addition dans le semi-anneau d'unions d'intervalles de temps est définie comme l'union des intervalles, ce qui permet de combiner plusieurs périodes de validité en une seule. Par exemple, si nous avons deux unions d'intervalles de temps $[A, B] \cup [C, D]$ et $[E, F] \cup [G, H]$, leur addition sera l'union des unions d'intervalles $[A, B] \cup [C, D] \cup [E, F] \cup [G, H]$, qui représente la période de validité combinée des deux unions d'intervalles d'origine.

La multiplication dans le semi-anneau des unions d'intervalles de temps est définie comme l'intersection des unions d'intervalles, ce qui permet de trouver la période de validité commune à deux unions d'intervalles. Par exemple, si nous avons deux unions d'intervalles de temps $[A, B] \cup [C, D]$ et $[E, F] \cup [G, H]$, leur multiplication sera l'intersection des unions d'intervalles $([A, B] \cup [C, D]) \cap ([E, F] \cup [G, H])$, qui représente la période de validité commune aux deux unions d'intervalles d'origine.

En étendant ProvSQL, nous ajoutons des fonctions utilisateur (*User-Defined Functions*) qui permettent de manipuler les unions d'intervalles de temps et d'effectuer les opérations d'addition et de multiplication dans le semi-anneau. Ces fonctions permettent d'interroger et de déduire les temps de validité des données renvoyées par les requêtes SQL.

Ainsi, notre solution utilise un semi-anneau d'unions d'intervalles de temps pour modéliser les temps de validité des données. En ajoutant des UDF à ProvSQL, nous permettons aux utilisateurs d'interroger et de déduire les temps de validité des données dans les résultats des requêtes SQL. Les utilisateurs peuvent spécifier des unions d'intervalles de temps pour les colonnes de validité dans leurs tables, et exécuter des requêtes qui renvoient non seulement les tuples correspondants, mais également les unions d'intervalles de temps indiquant la validité de ces résultats dans la base de données.

3.2.2 Choix de type de données et granularité

Depuis la version 14 de PostgreSQL, des types de données ont été introduits pour représenter des intervalles de valeurs (*range*) d'un certain type d'élément (appelé sous-type de l'intervalle) (PostgreSQL Global Development Group s. d.). Par exemple, des intervalles de `timestamp` (estampille de temps ou horodatage) pourraient être utilisés pour représenter les intervalles de temps durant lesquels un malade était hospitalisé dans un hôpital. De plus, étant donné que les informations peuvent être valides sur des périodes de temps non contiguës, chaque type *intervalle de valeurs* dispose d'un type *intervalle de valeurs multiples* (*multirange*) correspondant. Un *multirange* est une liste triée d'intervalle de valeurs non contiguës, non vides et non NULL.

Pourquoi *«union» d'intervalles de temps* ?

Dans un contexte temporel, les types *intervalle de valeurs* sont utiles parce qu'ils ils permettent de représenter les périodes de temps pendant lesquelles une information est valide. Ils sont particulièrement adaptés pour représenter des données avec des périodes de validité sous forme d'unions d'intervalles, et que des concepts comme le chevauchement d'intervalles peuvent être exprimés clairement.

PostgreSQL dispose des types d'unions d'intervalles de temps suivants :

- `tsmultirange` : Il s'agit d'un type de données représentant des unions d'intervalles de temps sans fuseau horaire. Il peut être utilisé pour modéliser des unions d'intervalles de dates et d'heures spécifiques.
- `tstzmultirange` : Il s'agit d'un type de données représentant des unions d'intervalles de temps avec prise en compte des fuseaux horaires.
- `datemultirange` : Il s'agit d'un type de données représentant des unions d'intervalles de dates uniquement, sans heure ni fuseau horaire.

Granularité des informations de provenance

Nous considérons une granularité au niveau du tuple, c'est-à-dire que les relations comportent une colonne contenant des informations sur la période de validité de chaque ligne, représentée par un intervalle de temps.

3.2.3 Définition du semi-anneau

Une *union d'intervalles de temps* est un ensemble fini d'intervalles, deux à deux disjoints ; on va représenter la période de validité d'un tuple par une telle union d'intervalles de temps.

Soit I l'ensemble des unions d'intervalles de temps qui représentent les périodes de validité des données, muni de deux lois de composition interne $\cup$ et $\cap$ définies sur I. L'opération d'addition binaire du semi-anneau est définie comme l'union d'union d'intervalles, ce qui signifie que pour deux unions d'intervalles a et b, l'opération d'addition $a \oplus b$ représente l'union des intervalles contenus dans a et b (sans répétition des parties communes à a et b) où l'élément neutre est l'ensemble vide $\emptyset$. Cet ensemble vide ne contient aucun intervalle temporel. De même, l'opération de multiplication binaire est définie comme l'intersection d'union d'intervalles, ce qui signifie que pour deux intervalles a et b, l'opération de multiplication $a \otimes b$ représente l'intersection des deux unions d'intervalles a et b, c'est-à-dire, l'ensemble des sous-intervalles de temps qui sont communs à a et à b. L'élément neutre pour cette opération est l'union d'intervalles formé du simple intervalle universel, également appelé intervalle total. Cet intervalle s'étend sur toute la durée possible, de la borne inférieure la plus petite à la borne supérieure la plus grande, et contient tous les instants temporels possibles.

Plus formellement, nous avons :

Théorème 3.1
Soit U l'ensemble des unions d'intervalles de temps qui représentent les périodes de validité des données. On définit deux opérations binaires sur U : l'union $\cup$ et l'intersection $\cap$. Alors $(U, \cup, \cap, \emptyset, \{]-\infty, +\infty[\})$ forme un semi-anneau avec l'élément neutre $\emptyset$ pour l'union et l'élément neutre $\{]-\infty, +\infty[\}$ pour l'intersection.

Preuve 3.1
Pour procéder à la preuve du théorème, nous suivons les étapes suivantes :

1. ***Commutativité de l'union ($\cup$) :*** *Pour toute union d'intervalles A et B dans U, nous devons montrer que $A \cup B = B \cup A$. Soit x un élément appartenant à $A \cup B$. Cela signifie que x appartient à l'un des intervalles de l'union A ou de l'union B. Dans les deux cas, x appartient également à l'union $B \cup A$, ce qui montre que $A \cup B \subseteq B \cup A$. De manière similaire, en appliquant le même raisonnement, nous pouvons montrer que $B \cup A \subseteq A \cup B$. Par conséquent, $A \cup B = B \cup A$, et la commutativité de l'union est démontrée.*

2. ***Distributivité de l'intersection ($\cap$) par rapport à l'union ($\cup$) :*** *Soit A, B, C des unions d'intervalles dans U. Nous devons montrer que $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ et $(A \cup B) \cap C = (A \cap C) \cup (B \cap C)$.*

   - *Pour la première équation, soit $x \in A \cap (B \cup C)$. Cela signifie que x appartient à la fois à l'union A et à l'union $B \cup C$. Par définition de l'union $B \cup C$, cela signifie que x appartient à l'union A et x appartient soit à l'union B soit à l'union C. Ainsi, x appartient à l'union $(A \cap B) \cup (A \cap C)$. Donc, $A \cap (B \cup C) \subseteq (A \cap B) \cup (A \cap C)$.*

- *Pour la deuxième équation, soit $x \in (A \cup B) \cap C$. Cela signifie que x appartient à la fois à l'union $A \cup B$ et à l'union C. Par définition de l'union $A \cup B$, cela signifie que x appartient soit à l'union A soit à l'union B, et il appartient également à l'union C. Ainsi, x appartient à l'union $(A \cap C) \cup (B \cap C)$. Donc, $(A \cup B) \cap C \subseteq (A \cap C) \cup (B \cap C)$.*

En combinant les deux inclusions, nous avons montré la distributivité de l'intersection par rapport à l'union.

3. ***Propriété de l'élément absorbant $\emptyset$ pour l'intersection :*** *Pour montrer que l'ensemble vide $\emptyset$ est absorbant pour l'intersection des unions d'intervalles, nous devons démontrer que pour toute union d'intervalles A, nous avons $\emptyset \cap A = \emptyset$.*

   Supposons que $\emptyset \cap A$ soit non vide, ce qui signifie qu'il existe un élément x tel que $x \in \emptyset \cap A$. Cependant, par la définition de l'intersection, cela implique que $x \in \emptyset$ et $x \in A$.

   Puisque l'ensemble vide $\emptyset$ ne contient aucun élément, cela est une contradiction. Donc, l'ensemble $\emptyset \cap A$ ne peut pas contenir d'éléments.

   Par conséquent, nous pouvons conclure que $\emptyset \cap A = \emptyset$ pour toute union d'intervalles A, ce qui démontre que l'ensemble vide $\emptyset$ est absorbant pour l'intersection des unions d'intervalles.

   Ainsi, nous avons montré que $(U, \cup, \cap)$ forme un semi-anneau avec les éléments neutres $\emptyset$ et $]-\infty, +\infty[$.

Le présent semi-anneau a d'autres propriétés dont on distingues :

Théorème 3.2
Soit $(I, \cup, \cap, \emptyset,]-\infty, +\infty[)$ un semi-anneau. Pour tout union d'intervalle a et b dans I, nous avons : l'opération d'intersection ($\cap$) est commutative, c'est-à-dire que $a \cap b = b \cap a$.

Preuve 3.2
Soit x un élément appartenant à $(a \cup b) \cap (b \cup a)$. Cela signifie que x appartient à la fois à $a \cup b$ et à $b \cup a$.

Par définition de l'union, cela implique que x appartient soit à a soit à b, et il appartient également soit à b soit à a.

Il y a deux cas possibles à considérer :

- *Si $x \in a$ et $x \in b$, alors x appartient à la fois à a et à b. Donc, $x \in a \cap b$.*
- *Si $x \in b$ et $x \in a$, alors x appartient à la fois à b et à a. Donc, $x \in b \cap a$.*

Dans les deux cas, x appartient à la fois à $a \cap b$ et à $b \cap a$. Donc, $(a \cup b) \cap (b \cup a) \subseteq (b \cup a) \cap (a \cup b)$.

En inversant les rôles de a et b, nous obtenons également $(b \cup a) \cap (a \cup b) \subseteq (a \cup b) \cap (b \cup a)$.

Par conséquent, $(a \cup b) \cap (b \cup a) = (b \cup a) \cap (a \cup b)$.

Ainsi, l'intersection des unions d'intervalles est commutative.

3.2.4 Le semi-anneau d'union d'intervalles de temps avec monus

Nous pouvons étendre ce semi-anneau commutatif en ajoutant la nouvelle opération binaire (définie dans 1.7.1), appelée différence, dénotée par le symbole $\backslash$. L'opération de différence entre deux unions d'intervalles A et B, notée $A\backslash B$, est définie comme l'union des intervalles de A qui ne se chevauchent pas avec les intervalles de B, ce qui donne l'intervalle des instants temporels présents dans A mais absents de B. Formellement, nous avons :

$$A\backslash B = \bigcup_{a \in A} (a \cap \overline{B})$$

où $\overline{B}$ représente le complément de l'union des intervalles de B. C'est-à-dire, $\overline{B} = I\backslash B$, où I est l'ensemble total des instants temporels.

Il est important de noter que l'opération de différence n'est pas commutative, c'est-à-dire que $A\backslash B \neq B\backslash A$ en général. La différence dépend de l'ordre des unions d'intervalles et donne des résultats différents selon cet ordre.

En ajoutant l'opération de différence, nous obtenons un nouvel ensemble $(I, \cup, \cap, \backslash, \emptyset,]-\infty, +\infty[)$ qui forme un semi-anneau avec monus qui permet de manipuler les unions d'intervalles de temps de manière plus complète en incluant l'opération de différence.

3.2.5 Modélisation du semi-anneau d'union d'intervalles dans ProvSQL

Nous modélisons la structure algébrique du semi-anneau d'union d'intervalles dans ProvSQL (Senellart et al. 2018) en définissant plusieurs UDF ('*User-Defined Functions*) spécifiques. Ces fonctions permettent d'effectuer des opérations sur les unions d'intervalles et de manipuler la provenance associée.

Voici les principales UDF utilisées pour la modélisation du semi-anneau d'union d'intervalles dans ProvSQL :

- `union_intervalles_monus(union_intervalles1, union_intervalles2)` : Cette fonction prend en entrée deux unions d'intervalles de temps et retourne leur différence. Elle est utilisée pour effectuer l'opération de différence ensembliste entre deux intervalles.

- `union_intervalles_plus_state(state, value)` : Cette fonction prend en entrée un état (`state`) et une valeur (`value`) de type `tsmultirange` et retourne leur union. Elle est utilisée pour effectuer l'opération d'union entre plusieurs unions d'intervalles.

- `union_intervalles_times_state(state, value)` : Cette fonction prend en entrée un état (`state`) et une valeur (`value`) de type `tsmultirange` et retourne leur intersection. Elle est utilisée pour effectuer l'opération d'intersection entre plusieurs unions d'intervalles.

- `union_intervalles_plus(tsmultirange)` : Cette fonction est une agrégation qui effectue une union d'intervalles de temps. Elle utilise la fonction `union_intervalles_plus_state` pour réaliser l'union de toutes les unions d'intervalles.

- `union_intervalles_times(tsmultirange)` : Cette fonction est une agrégation qui effectue une intersection d'union d'intervalles de temps. Elle utilise la fonction `union_intervalles_times_state` pour réaliser l'intersection de toutes les unions d'intervalles.

- `union_intervalles(token, token2value)` : Cette fonction prend en entrée une annotation de provenance et une table de mapping (`token2value`) et retourne l'ensemble des intervalles correspondants. Elle utilise les UDF précédentes pour effectuer les opérations sur les intervalles.

Ces UDF (User-Defined Functions) permettent de passer de la définition algébrique de notre structure de semi-anneau à sa modélisation dans une forme fonctionnelle, offrant ainsi des fonctionnalités adaptées pour la gestion et l'analyse de la provenance des données.

Les types des paramètres des fonctions qui modélisent les opérations binaires du semi-anneau sont liés aux types de données choisis pour représenter les temps de validité des tuples. Ainsi, outre le type de données `tsmultirange`, il est également possible d'utiliser d'autres types pour représenter les temps de validité des tuples.

Nous décrivons dans la sous-section suivante comment obtenir les temps de validité des résultats des requêtes dans la même requête lors de l'interrogation d'une base de données temporelle.

3.2.6 Interrogation du temps de validité

Pour illustrer les étapes de calcul de l'information de provenance, nous utilisons un exemple d'une table `personnel`, présentée dans le tableau 4.1, qui contient des informations sur le personnel d'une entreprise. Nous représentons le temps de validité des tuples par des unions d'intervalles de type `datemultirange` (PostgreSQL Global Development Group s. d.). Nous adoptons la notation de PostgreSQL où un `datemultirange` est représenté sous forme d'accolades ({ et }), contenant zéro ou plusieurs intervalles, séparés par des virgules. Les annotations de provenance sont représentées par les t_i pour simplifier.

Prenons l'exemple d'une requête $Q3$ qui affiche les villes référencées dans la table 4.1 qui accueillent au moins deux employés différents de l'entreprise. Nous détaillons dans ce qui suit les étapes à suivre pour pouvoir obtenir les temps de validité des résultats de la requête Q3,

- Annotation de la table avec la provenance : Avant d'exécuter la requête, la table `personnel` doit être annotée avec les informations de provenance. Cela se fait en appelant la fonction `add_provenance('personnel')`. Elle crée une nouvelle colonne `provsql` dans la table `personnel` et attribue un jeton (annotation)de provenance

id	nom	position	ville	temps_de_validite	
1	John	Directeur	New York	{[2022-01-10,2022-09-16)}	t_1
2	Paul	Concierge	New York	{[2023-03-05,2023-07-21)}	t_2
3	Dave	Analyste	Paris	{[2022-10-01,2023-01-01)}	t_3
4	Ellen	Agent de terrain	Berlin	{2023-06-08,2023-12-21)}	t_4
5	Magdalen	Agent double	Paris	{[2022-07-15,2022-12-01)}	t_5
6	Nancy	RH	Paris	{[2023-01-08,2023-07-01)}	t_6
7	Susan	Analyste	Berlin	{[2022-03-10,2022-10-26)}	t_7

Fig. 3.1 : Tableau `Personnel` pour le personnel d'une entreprise

```
SELECT DISTINCT P1.ville
FROM Personnel P1, Personnel P2
WHERE P1.ville = P2.ville AND P1.id < P2.id;
```

Fig. 3.2 : La requête $Q3$

unique ayant le type `textit` (*Universally Unique Identifiers*), qui sont des identifiants uniques de 128 bits utilisés pour identifier des informations de manière unique (Group 2021) à chaque tuple.

- Création d'une correspondance de provenance : Nous appelons la fonction `create_provenance_mapping` qui crée une table à partir de la table que nous souhaitons interroger. Cette table contient deux colonnes : `value` et `provenance`. Chaque annotation de provenance présente dans la table d'origine est associée à la valeur de la colonne spécifiée (dans notre cas, c'est la colonne `temps de validité`). La fonction est appelée comme suit : `SELECT create_provenance_mapping('personnel_mapping','personnel','temps_de_validite')`.
- Exécution de la requête modifiée : La requête est modifiée pour inclure les intervalles de temps de validité en utilisant la fonction `union_intervalles` dans la clause `SELECT`.

```
SELECT *, union_intervalles(provenance(), 'personnel_mapping')
AS temps_validite
FROM (
  SELECT DISTINCT P1.ville
  FROM personnel P1
  JOIN personnel P2 ON P1.ville = P2.ville
  WHERE P1.id < P2.id
) t;
```

  - La sous-requête `SELECT DISTINCT P1.city FROM Personnel P1 JOIN Personnel P2 ON P1.city = P2.city WHERE P1.id < P2.id` est exécutée pour obtenir une liste de villes distinctes où l'identifiant (`id`) du premier tuple est inférieur à l'identifiant du deuxième tuple.

- Pour chaque ville de cette liste, la fonction `provenance()` est appelée pour récupérer l'annotation de provenance associée à chaque tuple.
- La fonction `union_intervalles(provenance(), 'personnel_level')` est appelée, et elle fait appel à la fonction `provenance_evaluate()` pour calculer les temps de validité associés à chaque jeton de provenance en utilisant la table de correspondance `personnel_level`.
- La fonction `provenance_evaluate()` examine le type de porte associé à l'annotation de provenance. Selon ce type, elle effectue les opérations appropriées pour calculer les temps de validité.
- Les temps de validité calculés sont renvoyés comme colonne `union_intervalles` dans le résultat de la requête principale, qui combine les enregistrements de la table `Personnel` avec les temps de validité correspondants.

3.3 Adaptation d'un benchmark de bases de données temporelles

Pour tester et valider le nouveau semi-anneau d'union d'intervalles, nous avons décidé de tester des requêtes sur des bases de données temporelles avec des volumes de données considérables. Et pour cela, nous avons décidé d'utiliser le benchmark TPC-BiH (Kaufmann et al. 2013c).

Le benchmark TPC-BiH (*Temporal Performance Council Benchmark for Interval-based Histories*) est spécifiquement conçu pour évaluer les performances des bases de données temporelles prenant en charge à la fois le temps de transaction et le temps de validité (Snodgrass et al. 1986). Il est conçu pour mesurer les performances des opérations de requêtes temporelles et des opérations de gestion des données dans un contexte temporel.

Le schéma de TPC-BiH est basé sur le schéma TPC-H (Council 2022) et étendu avec des colonnes temporelles pour représenter les temps système et d'application.

Il convient de noter que le benchmark TPC-BiH n'est pas largement disponible sous la forme d'un package d'installation standard. Cependant, nous avons trouvé un projet de recherche réalisé par Dignös et al. (Dignös et al. 2019) qui utilisait le benchmark et fournissait des données générées. Nous avons extrait les données à partir du conteneur Docker du projet pour les utiliser dans nos expérimentations.

Cependant, le format des données extraites ne correspondait pas exactement à nos besoins pour les expérimentations. Les temps de validité des données étaient représentés dans des colonnes de type `date`, avec des colonnes de début et de fin représentant respectivement le début et la fin de la période de validité des tuples. De plus, les requêtes proposées par Kauffman et al. (Kaufmann et al. 2013c) étaient spécifiquement adaptées au SGBD Oracle.

Pour remédier à ces contraintes, nous avons effectué des pré-traitements sur les tables de données. Pour chaque table, nous avons créé une nouvelle colonne de type `datemultirange` (**range_types**). Le schéma de données adapté est illustré dans la figure 3.3. Nous avons

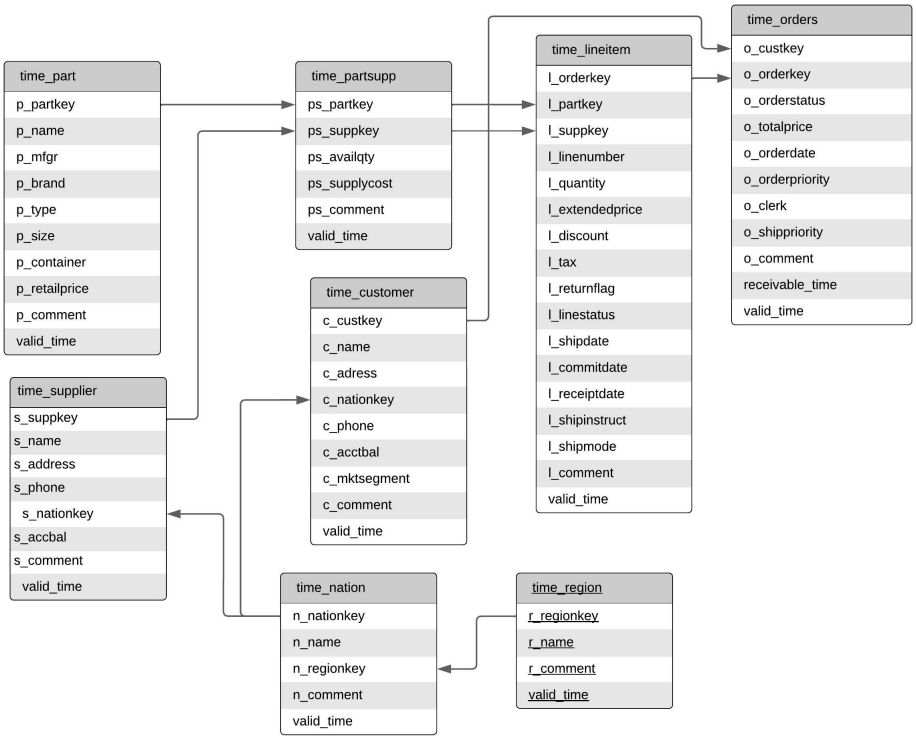


FIG. 3.3 : Schéma de base de données temporelle adapté à partir de TPC-BiH.

ensuite alimenté ces colonnes en construisant des unions d'intervalles de dates à partir des valeurs des colonnes `valid_time_begin` et `valid_time_end`. Nous avons utilisé les constructeurs du type `datemultirange` pour effectuer cette opération. Par exemple, voici la requête permettant d'alimenter la nouvelle colonne `valid_time` de la table `time_customer` :

```
UPDATE time_customer SET valid_time = datemultirange (
daterange(t.visible_time_begin, t.visible_time_end, '[]'))
FROM
(
    SELECT c_custkey, visible_time_begin, visible_time_end
    FROM time_customer
) t
WHERE time_customer.c_custkey = t.c_custkey;
```

Enfin, nous avons également proposé une charge de travail (*workload*) composée de requêtes plus adaptées à PostgreSQL et spécifiquement à nos objectifs de recherche sur ProvSQL. Les requêtes proposées permettaient de tester les performances du semi-anneau d'union d'intervalles dans plusieurs scénarios :

- Requêtes avec une opération de différence.
- Requêtes avec une opération de sélection distincte impliquant une jointure dans la même table.

- Requêtes avec une opération de jointure sur plusieurs tables contenant des volumes de données importants.
- Requêtes avec une opération d'agrégation.

3.4 Optimisation de requêtes dans ProvSQL

Nous avons constaté que les requêtes manipulant des tables annotées prennent effectivement plus de temps à s'exécuter que les requêtes ne manipulant pas de telles tables. L'un des objectifs principaux de ce projet est donc de proposer des solutions d'optimisation du temps d'exécution des requêtes dans ProvSQL afin de réduire significativement les temps d'exécution et d'améliorer l'expérience des utilisateurs.

3.4.1 Analyse approfondie

Lors de l'exécution d'une requête dans PostgreSQL, plusieurs étapes sont suivies pour garantir la bonne exécution et l'obtention des résultats attendus. Ces étapes, qui comprennent l'analyse syntaxique, l'analyse sémantique, la planification de la requête, l'optimisation, l'exécution et le renvoi des résultats, permettent d'assurer l'efficacité et la cohérence des opérations. Les étapes d'exécution d'une requête dans PostgreSQL sont généralement les suivantes :

- Analyse syntaxique : La requête est analysée pour vérifier sa syntaxe et sa structure. Si la requête est invalide, une erreur est renvoyée.
- Analyse sémantique : La requête est analysée pour vérifier les références aux tables, aux colonnes et aux objets de la base de données. Des vérifications supplémentaires sont effectuées pour garantir la cohérence des types de données et des contraintes.
- Planification de la requête : PostgreSQL génère un plan d'exécution optimal pour la requête. Cela implique de déterminer les meilleures stratégies d'accès aux données, les méthodes de jointure, les filtres à appliquer, etc.
- Optimisation de la requête : Le plan d'exécution est optimisé pour minimiser le coût total de la requête. Différentes techniques d'optimisation, telles que la réécriture de requête, la réorganisation des opérations et l'utilisation des index, sont appliquées pour améliorer les performances.
- Exécution de la requête : Le moteur d'exécution de PostgreSQL exécute le plan d'exécution généré. Cela peut impliquer l'accès aux données sur le disque, l'application des filtres et des opérations sur les tuples, les jointures, les agrégations, etc.
- Renvoi des résultats : Une fois que la requête est exécutée, les résultats sont renvoyés au client. Cela peut inclure des tuples de données, des agrégations, des statistiques ou tout autre résultat spécifié dans la requête.

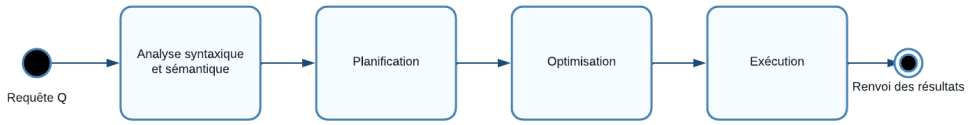


FIG. 3.4 : Processus d'exécution d'une requête sans support de provenance.

Nous illustrons ces étapes dans la figure 3.4.

Le module de réécriture de requête de ProvSQL modifie le comportement des requêtes SQL afin d'intégrer la gestion de la provenance. Son rôle est de réécrire les requêtes qui font référence à des tables annotées en incluant les annotations de provenance dans les résultats de la requête.

Lors de l'exécution d'une requête `SELECT` sur des tables conscientes de ProvSQL (tables annotées), son module de réécriture de requêtes (*Query Rewriter*) intervient dans une phase intermédiaire avant la planification de la requête. Il génère une nouvelle colonne `provsql` dans la clause `SELECT` de la requête. Cette colonne contient des annotations de provenance, qui sont des *UUID (Universally Unique Identifiers)* représentant les portes du circuit de provenance associées à chaque résultat de la requête. Un circuit de provenance est une structure de données maintenue par ProvSQL, qui représente le flux d'opérations et de transformations appliquées aux données, fournissant ainsi un mécanisme pour suivre la provenance des données.

Les *UUID* sont générés de manière à la fois unique et reproductible pour des requêtes identiques, ce qui garantit que les annotations de provenance seront les mêmes pour des requêtes identiques exécutées à différents moments.

Prenons par exemple la requête suivante :

```
SELECT DISTINCT P1.ville
FROM personnel P1
JOIN personnel P2 ON P1.ville = P2.ville
WHERE P1.id < P2.id
```

En utilisant ProvSQL, le module de réécriture de requêtes réécrira automatiquement la requête de la manière suivante :

```
SELECT DISTINCT P1.ville, provenance_plus(array_agg(provenance_times(
P1.provenance(), P2.provenance()))) AS provsql
FROM personnel P1
JOIN personnel P2 ON P1.ville = P2.ville
WHERE P1.id < P2.id
GROUP BY P1.ville
```

Dans la requête réécrite, la colonne `provsql` est ajoutée au résultat de la requête. Les annotations de provenance sont générées en utilisant les fonctions `provenance_plus()` et `provenance_times()`. La fonction `provenance_plus()` génère une porte de type `plus` $\oplus$ représentant une opération d'agrégation dans le circuit de provenance, tandis que la

fonction `provenance_times()` génère une porte de type `fois` $\otimes$ représentant une opération de jointure. Nous illustrons le processus d'exécution d'une requête avec support de ProvSQL dans la figure 3.5.

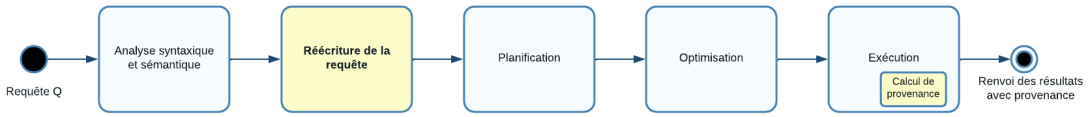


FIG. 3.5 : Processus d'exécution d'une requête avec support de provenance.

Dans le cas des requêtes utilisant le semi-anneau d'unions d'intervalles de temps, la réécriture de la requête suit le même processus expliqué précédemment. Cela signifie que lors de l'exécution de la requête, elle est réécrite pour prendre en compte les annotations de provenances. Considérons à titre d'exemple la requête suivante :

```
SELECT *, union_intervalles(provenance(),'personnel_level')
FROM (
  SELECT DISTINCT P1.city
  FROM Personnel P1 JOIN Personnel P2
  ON P1.city = P2.city
  WHERE P1.id < P2.id
) t;
```

Elle sera réécrite comme suit :

```
SELECT *, union_intervalles(provenance(),'personnel_mapping')
FROM (
  SELECT DISTINCT P1.city, provenance_plus(array_agg(provenance_times(
    P1.provenance(), P2.provenance()))) AS provsql
  FROM Personnel P1 JOIN Personnel P2
  ON P1.city = P2.city
  WHERE P1.id < P2.id
) t;
```

Nous décrivons le processus d'exécution d'une requête avec le semi-anneau d'unions d'intervalles de temps dans la figure 3.6. Une attention particulière a été portée à l'optimisation des

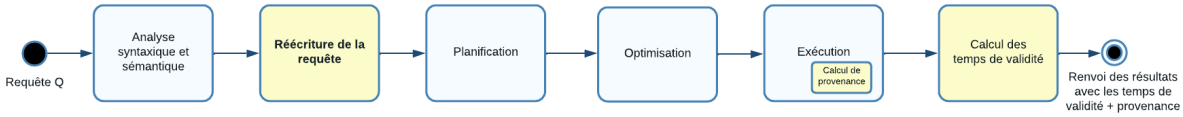


FIG. 3.6 : Processus d'exécution d'une requête avec support du temps de validité.

traitements liés à la génération et à l'exécution des opérations sur les annotations de provenance. Étant donné que ProvSQL intervient pour la première fois dans le processus d'exécution des requêtes SQL lors de l'étape de réécriture automatique, cette caractéristique nous a permis d'avoir une manipulation initiale du code source de ProvSQL et ainsi d'explorer les possibilités d'optimisation.

Dans cette perspective, nous avons entrepris de réécrire manuellement les requêtes afin d'obtenir une première impression des fonctions et des traitements ayant un impact significatif sur les temps d'exécution. Pour ce faire, nous avons choisi d'utiliser la requête $Q3$ de la charge de travail (*workload*) que nous avons proposée, et qui renvoie uniquement les clients se trouvant dans le même pays et le même segment de marché. La figure 3.7 illustre les modifications effectuées, et les détails relatifs à l'environnement et à la méthodologie de test seront exposés en détail dans le chapitre 04 de ce rapport. Il convient

```
SELECT DISTINCT c1.c_name, c1.c_nationkey, c1.c_mktsegment
FROM (SELECT * FROM time_customer c1 ORDER BY c_custkey LIMIT 5000) AS c1
JOIN (SELECT * FROM time_customer c2 ORDER BY c_custkey LIMIT 5000) AS c2
ON c1.c_nationkey = c2.c_nationkey AND c1.c_mktsegment = c2.c_mktsegment
WHERE c1.c_custkey <> c2.c_custkey
AND c1.c_name < c2.c_name;
```

```
SELECT c1.c_name, c1.c_nationkey, c1.c_mktsegment,
provenance_plus(array_agg(provenance_times(c1.provenance, c2.provenance)))
AS provenance
FROM (SELECT * FROM time_customer2 c1 ORDER BY c_custkey LIMIT 5000) AS c1
JOIN (SELECT * FROM time_customer2 c2 ORDER BY c_custkey LIMIT 5000) AS c2
ON c1.c_nationkey = c2.c_nationkey AND c1.c_mktsegment = c2.c_mktsegment
WHERE c1.c_custkey <> c2.c_custkey
AND c1.c_name < c2.c_name
GROUP BY c1.c_name, c1.c_nationkey, c1.c_mktsegment;
```

Fig. 3.7 : Réécriture de la requête $Q3$

de noter que la table `time_customer` contient environ 100 000 tuples. Toutefois, en raison de la limite de ProvSQL concernant le nombre de portes qu'il peut générer dans un circuit de provenance (`provsql_max_nb_gates = 1000000`), augmenter ce nombre nécessite une augmentation de l'espace mémoire dédié à PostgreSQL (`shared_buffers`), ce qui constitue une contrainte pour certaines machines. Ainsi, nous avons choisi de travailler uniquement avec les premières 5000 tuples de cette table pour nos manipulations.

Lors de l'exécution de cette requête, nous avons constaté qu'elle prenait en moyenne 1960 ms, qui est une durée similaire à celle de l'exécution de la requête normale, c'est-à-dire, en laissant la réécriture se faire automatiquement. Cette observation souligne l'importance de rechercher des moyens d'optimiser les performances de ProvSQL pour réduire le temps d'exécution des requêtes. Dans la suite, nous décrivons la démarche suivie et les contributions apportées en termes d'optimisation.

3.4.2 Identification des points problématiques

Dans cette section, nous procédons à l'identification des parties du code de ProvSQL qui impactent négativement les performances globales. En analysant attentivement les résultats obtenus lors des mesures de performance, nous avons pu repérer les traitements spécifiques de ProvSQL qui nécessitent une attention particulière. Nous allons maintenant examiner en détails ces traitements afin de comprendre les raisons de leurs performances sous-optimales et de proposer des solutions d'améliorations adaptées.

La fonction `provenance_times()`

Après nos observations, nous avons remarqué que la fonction `provenance_times()` était responsable d'environ 80% du temps d'exécution de la requête $Q3$ lorsque nous l'exécutions isolément, en excluant les fonctions `provenance_plus()` et `array_agg`.

1. La fonction `provenance_times()` (voir la figure 3.8), écrite en code PL/pgSQL, prend en entrée un tableau variable (`VARIADIC`) de jetons de type UUID.
2. Elle commence par filtrer les jetons du tableau en éliminant ceux qui ont la valeur «1» en utilisant la clause `WHERE t <> gate_one()`. Les jetons de valeur «1» sont considérés comme des éléments neutres dans l'opération de multiplication et n'ont pas d'impact sur le résultat final, il est donc préférable de les exclure pour optimiser le calcul.
3. Ensuite, la fonction vérifie la taille du tableau `filtered_tokens` après le filtrage :
   - Si la taille est de 0, cela signifie qu'il ne reste aucun jeton après le filtrage. Dans ce cas, la fonction attribue au jeton `times_token` la valeur de la porte neutre `gate_one()` qui représente l'opération de multiplication neutre.
   - Si la taille est de 1, cela signifie qu'il reste un seul jeton après le filtrage, et ce jeton est attribué directement à `times_token` sans effectuer d'autres opérations.
   - Si la taille est supérieure à 1, la fonction utilise la fonction `uuid_generate_v5` pour générer un nouveau *UUID* en utilisant l'espace de noms `uuid_ns_provsql()` et en concaténant les UUIDs des jetons filtrés. Ce nouveau jeton représente l'opération de multiplication des jetons filtrés. Ensuite, la fonction appelle `create_gate` qui est une fonction *C* qui permet créer une nouvelle porte avec le jeton `times_token`, le type "times" et les jetons filtrés en tant que valeurs de provenance.
4. Enfin, la fonction retourne le jeton `times_token`, qui représente la valeur de provenance de l'opération de multiplication.

En utilisant la même démarche d'analyse des performances, nous avons identifié trois parties du code qui consomment un temps d'exécution significatif :

1. Le parcours du tableau et la suppression des jetons «1» ont pris environ 200 ms sur un temps total d'exécution de 1890 ms. Cette partie du code est responsable du filtrage des jetons neutres dans le but d'optimiser le calcul de l'opération de multiplication. Cependant, le processus de parcours du tableau et la comparaison avec la valeur «1» ralentissent l'exécution globale de la fonction.
2. La fonction `uuid_generate_v5` a pris environ 70 ms de temps d'exécution. Cette fonction est définie dans le code source de PostgreSQL. Elle utilise une fonction de hachage cryptographique (SHA-1) pour calculer un nouvel *UUID* basé sur l'espace de noms (*namespace*) de `provsql`, ce qui permet de générer un *UUID* unique et déterministe. Cela garantit que pour 2 requêtes identiques, un *UUID* unique est généré.

```
CREATE FUNCTION provenance_times(VARIADIC tokens uuid[])
  RETURNS UUID AS
$$
DECLARE
  times_token uuid;
  filtered_tokens uuid[];
BEGIN

  SELECT array_agg(t) FROM unnest(tokens) t WHERE t <> gate_one() INTO filtered_tokens;

  CASE array_length(filtered_tokens,1)
    WHEN 0 THEN
      times_token:=gate_one();
    WHEN 1 THEN
      times_token:=filtered_tokens[1];
    ELSE
      SELECT uuid_generate_v5(uuid_ns_provsql(),concat('times',uuid_provsql_agg(t)))
      INTO times_token
      FROM unnest(filtered_tokens) t;

      PERFORM create_gate(times_token, 'times', filtered_tokens);
  END CASE;

  RETURN times_token;
END
$$ LANGUAGE plpgsql SET search_path=provsql,pg_temp,public SECURITY DEFINER;
```

FIG. 3.8 : La fonction `provenance_times()`

3. L'appel à la fonction `create_gate()` a pris environ 250 ms sur un temps total d'exécution de 1890 ms.

Ces résultats nous ont incité à explorer en profondeur la fonction `create_gate()` qui fait partie du code *C* contenu dans le fichier (PIERRE SENELLART 2023) afin d'identifier les problèmes potentiels liés à cette fonction et aux fonctions auxquelles elle fait référence.

La fonction `create_gate()`

La fonction `create_gate()` (PIERRE SENELLART 2023) dans ProvSQL est utilisée pour créer une nouvelle porte dans le circuit de provenance d'une requête.

1. Elle prend en paramètre les informations nécessaires pour créer une porte, telles que le type de porte, la probabilité associée, les informations supplémentaires, etc.
2. La fonction génère un *UUID* à l'aide de la fonction `gen_random_uuid()` pour la nouvelle porte.
3. Ensuite, elle crée une nouvelle instance de la structure `provsqlHashEntry` pour représenter la porte.
4. Les informations spécifiques à la porte sont remplies dans la structure `provsqlHashEntry`.
5. Si la porte a des enfants, la fonction assigne l'indice approprié dans le tableau `provsql_shared_state->wires` à la porte en utilisant la fonction `assign_wire()`.
6. La porte nouvellement créée est insérée dans la table de hachage `provsql_hash` avec l'*UUID* généré et la valeur correspondante (la structure `provsqlHashEntry`) en utilisant la fonction `hash_insert()`.

7. La fonction utilise également la fonction `hash_search()` pour vérifier si la porte existe déjà dans la table de hachage avant de l'insérer. L'appel à `hash_search()` avec l'option `HASH_ENTER` est effectué de la manière suivante :

```
entry = (provsqlHashEntry *)hash_search(
provsql_hash,
token,
HASH_ENTER,
&found);
```

   L'appel à `hash_search()` avec l'option `HASH_ENTER` permet d'insérer la nouvelle porte dans la table de hachage si elle n'existe pas déjà. La fonction renvoie un pointeur vers l'entrée correspondante dans la table de hachage (`provsqlHashEntry`) et définit la variable `found` à `true` si la porte est déjà présente dans la table de hachage.

8. Si la porte n'est pas trouvée (`!found`), ce qui signifie qu'elle n'existait pas encore dans la table de hachage, la fonction procède à son insertion. Sinon, elle supprime la porte nouvellement créée de la table de hachage, libère le verrou (`LWLockRelease()`) et génère une erreur (`elog(ERROR, "Unknown gate")`) pour indiquer qu'une porte inconnue a été trouvée.

9. Enfin, la fonction renvoie l'*UUID* généré pour la nouvelle porte.

Dans ProvSQL, un circuit de provenance est représenté à l'aide de trois structures de données principales : `provsqlHashEntry`, `provsql_hash`, et `provsql_shared_state`.

1. `provsqlHashEntry` :
   - Cette structure représente une porte dans le circuit de provenance.
   - Elle contient les informations spécifiques à une porte, telles que le type de porte, la probabilité associée, les informations supplémentaires, etc.
   - Chaque porte est identifiée par un *UUID* (`pg_uuid_t`), qui est généralement un identifiant unique.
   - Les portes peuvent avoir des enfants, représentés par un indice (`children_idx`) dans le tableau `provsql_shared_state->wires`.
   - La structure `provsqlHashEntry` est utilisée pour stocker les portes dans la table de hachage `provsql_hash`.

2. `provsql_hash` :
   - C'est une table de hachage utilisée pour stocker les portes du circuit de provenance.
   - Chaque entrée de la table `provsql_hash` est composée d'un *UUID* (`pg_uuid_t`) qui identifie la porte, et d'une valeur (`provsqlHashEntry`) qui représente la structure de données de la porte.

- La table de hachage `provsql_hash` est utilisée pour accéder rapidement aux portes du circuit en fonction de leur *UUID*.

3. `provsql_shared_state` :

   - C'est une structure partagée qui contient les données partagées entre les processus.
   - Elle contient des informations sur les fils (*wires*) du circuit de provenance, tels que le nombre total de fils (`nb_wires`) et un tableau d'*UUIDs* (`wires`) représentant les fils du circuit.
   - Le verrou (`lock`) est utilisé pour protéger l'accès concurrentiel aux données partagées.

En utilisant ces structures de données, ProvSQL représente un circuit de provenance en stockant les informations des portes dans la table de hachage `provsql_hash` et les informations des fils dans `provsql_shared_state->wires`. Chaque porte est associée à un *UUID* qui agit comme un identifiant, et les portes peuvent être liées entre elles à travers les indices des fils.

La fonction `create_gate()` crée alors une nouvelle porte dans le circuit de provenance, lui attribue un *UUID*, remplit les informations spécifiques à la porte, et l'insère dans la table de hachage `provsql_hash`. Cela permet d'ajouter de nouvelles portes et de les intégrer au circuit de provenance existant.

En utilisant la même démarche d'analyse des performances, nous avons identifié la fonction de recherche dans la table de hachage `hash_search` utilisée pour rechercher si une porte existe déjà dans la table de hachage `provsql_hash`. Cette fonction peut consommer beaucoup de temps si la table de hachage est grande et si la fonction de hachage est complexe.

3.4.3 Proposition des améliorations

Après avoir identifié les traitements qui affectent les performances de manière significative, nous nous sommes concentrés sur la proposition de solutions d'améliorations. Dans la suite de notre étude, nous détaillons en profondeur ces propositions d'optimisation, en fournissant des recommandations spécifiques et des améliorations potentielles pour chaque aspect à optimiser. Notre objectif est d'optimiser les performances globales de ProvSQL, en réduisant les temps de traitement et en augmentant son efficacité. Après avoir identifié plusieurs problèmes impactant les performances et analysé en détail leurs causes, nous proposons les solutions suivantes :

Filtrage des `gate_one()` dans `provenance_times`

Nous proposons de ne plus filtrer les portes «1». Nous démontrons les résultats de cette proposition dans le chapitre suivant.

La fonction `uuid_generate_v5`

Pour des raisons de performance, nous recommandons de remplacer l'utilisation de la fonction `uuid_generate_v5` par la fonction `uuid_generate_v3` (*PostgreSQL Documentation - uuid-ossp* 2021). La fonction `uuid_generate_v5` crée un UUID de version 5 en utilisant l'algorithme *SHA-1* comme méthode de hachage, ce qui peut prendre plus de temps en raison des considérations de sécurité supplémentaires. En revanche, la fonction `uuid_generate_v3` utilise également un mécanisme de hachage (MD5) et garantit la reproductibilité des résultats. Étant donné que la sécurité n'est pas une préoccupation majeure dans notre cas et que la reproductibilité est importante, nous souhaitons privilégier la performance en utilisant la version 3 plutôt que la version 5.

Fonction de hachage `hash_search()`

Pour de hachage `hash_search`, nous avons décidé d'implémenter la fonction simplifiée `provsql_hash_uuid()`. Cette fonction prend en paramètre un pointeur vers une clé de type `void*` et retourne un entier non signé de 32 bits (`uint32`). Elle convertit le pointeur `key` en un pointeur vers un entier non signé de 32 bits, puis retourne la valeur pointée par ce pointeur.

Suppression des portes du circuit

Cette proposition vise à garantir des tests précis et fiables sur le benchmark de requêtes afin d'évaluer les performances. Pour obtenir des résultats corrects et de partir d'un circuit de provenance vide, nous proposons de vider le circuit à chaque exécution du script de test, pour garantir des mesures précises.

En d'autres termes, en ayant un circuit vide à chaque exécution, nous renforçons la fiabilité des mesures en éliminant les influences indésirables ou les biais potentiels liés à des données précédentes qui pourraient fausser les résultats. L'algorithme permettant la suppression de toutes les portes du circuit de la requête.

Algorithm 1 : clean_circuit

Entrée :
Sortie :

```
1  if le nombre d'entrées dans provsql_hash est égal à 0 then
2  |  return
3  end
4  Acquérir le verrou provsql_shared_state.lock en mode EXCLUSIVE ;
5  foreach entrée dans provsql_hash do
6  |  Supprimer l'entrée de provsql_hash ;
7  end
8  Réinitialiser le tableau provsql_shared_state.wires à zéro ;
9  Réinitialiser le nombre de fils provsql_shared_state.nb_wires à 0;
10 Réinitialiser provsql_hash à NULL ;
11 Relâcher le verrou provsql_shared_state.lock ;
12 return
```

Toutes les solutions que nous avons proposées seront implémentées et évaluées en détail dans le prochain chapitre. Nous décrirons en profondeur l'implémentation de chaque solution, en fournissant des codes sources et des explications détaillées sur les modifications apportées. De plus, nous présenterons les résultats des tests de performance pour chaque solution, afin de quantifier les améliorations obtenues.

Le prochain chapitre sera essentiel pour valider l'efficacité des solutions proposées et pour fournir des justifications solides quant à leur impact sur les performances globales du système. Nous présenterons les mesures de performance, les comparaisons avant et après l'implémentation des solutions, ainsi que les conclusions tirées des résultats obtenus.

3.5 Conclusion

Dans ce chapitre, nous avons détaillé trois parties différentes de notre projet.

La première partie était axée sur la solution apportée pour ajouter le support aux bases de données temporelles à ProvSQL. Nous avons conçu cela en définissant un semi-anneau d'unions d'intervalles de temps. Cette approche permet d'interroger et de manipuler efficacement les périodes de validité des données, en prenant en compte les aspects temporels.

La deuxième partie de ce chapitre a porté sur l'adaptation d'un benchmark de bases de données temporelles pour tester et valider notre solution de semi-anneau d'unions d'intervalles. Nous avons choisi le benchmark TPC-BiH (*Temporal Performance Council Benchmark for Interval-based Histories*) et effectué des pré-traitements sur les données extraites pour les adapter à nos besoins expérimentaux. De plus, nous avons créé un benchmark de requêtes spécifique qui nous a permis de tester différents types de requêtes pris en charge par ProvSQL. Ces requêtes nous ont permis d'évaluer les performances du semi-anneau d'unions d'intervalles dans des scénarios tels que les opérations de différence, les sélections distinctes avec jointure dans la même table, les jointures sur plusieurs tables avec des volumes de données importants et les opérations d'agrégation.

Enfin, la troisième partie de ce chapitre s'est concentrée sur le mécanisme de calcul de la provenance dans ProvSQL. Notre objectif était d'identifier les optimisations possibles afin de minimiser le surcoût par rapport aux requêtes classiques. Nous avons examiné attentivement les traitements appliqués dans le processus de calcul de la provenance et proposé des solutions pour améliorer ses performances, en prenant en compte les spécificités temporelles.

Dans le prochain chapitre, nous présenterons l'implémentation de ces solutions et les résultats obtenus. Nous décrirons en détail comment nous avons mis en uvre les fonctionnalités liées aux bases de données temporelles dans ProvSQL, ainsi que les performances obtenues lors de l'exécution des requêtes temporelles et des opérations de provenance.

Chapitre 4

Réalisation, tests et évaluation

4.1 Introduction

Dans ce chapitre, nous présentons la réalisation, les tests et l'évaluation des contributions apportées à ProvSQL. Nous décrivons les technologies et les outils que nous avons utilisés pour développer nos contributions à ProvSQL, ainsi que les langages de programmation et les outils de gestion de versions. Ensuite, nous nous concentrons sur la validation du semi-anneau d'union d'intervalles en effectuant des tests pour évaluer sa cohérence et son bon fonctionnement. Nous comparons également les performances de ProvSQL avec celles de GProM, un autre outil de gestion de provenance, en exécutant des requêtes spécifiques dans les deux systèmes et en analysant les résultats obtenus. Enfin, nous présentons et évaluons les optimisations apportées à ProvSQL.

4.2 Technologies et outils utilisés

Dans ce projet, nous avons utilisé différentes technologies et outils pour le développement des contributions apportées à ProvSQL (Senellart et al. 2018). Cette section présente les principales technologies et outils que nous avons utilisés.

4.2.1 Environnement de développement

Microsoft WSL

Microsoft WSL (Windows Subsystem for Linux) est un outil qui permet d'exécuter un environnement Linux directement sur un système d'exploitation Windows (Microsoft Corporation 2023b). Étant donné que ProvSQL est principalement utilisé sous Linux ou macOS, nous avons utilisé Microsoft WSL pour créer un environnement Linux sur notre plateforme Windows. Cela nous a permis de travailler avec ProvSQL de manière transparente, en bénéficiant des fonctionnalités et des outils spécifiques à ProvSQL sur une plateforme Windows.

Visual Studio Code

Nous avons utilisé Visual Studio Code, un éditeur de code extensible développé par Microsoft, en utilisant l'extension `Remote Development`. Cette configuration nous a permis d'intégrer pleinement WSL dans notre environnement de développement (Microsoft Corporation 2023a). Nous avons pu exécuter et déboguer le code source de ProvSQL directement depuis Visual Studio Code, profitant ainsi des fonctionnalités et des avantages de WSL tout en bénéficiant de l'interface conviviale et des outils productifs de VS Code. Cette combinaison d'outils nous a offert une expérience de développement fluide et efficace avec ProvSQL.

4.2.2 Langages de programmation

PL/pgSQL

PL/pgSQL (The PostgreSQL Global Development Group 2023) est un langage de programmation procédural utilisé pour écrire des fonctions stockées dans PostgreSQL. Nous avons utilisé PL/pgSQL pour écrire des fonctions spécifiques dans ProvSQL, notamment pour implémenter des fonctionnalités de gestion des données temporelles et de calcul de provenance. Ce langage nous a permis de tirer pleinement parti des fonctionnalités et des performances de PostgreSQL.

C et C++

Nous avons également utilisé les langages de programmation C et C++ pour l'implémentation de certaines fonctionnalités ainsi que certaines parties de ProvSQL nécessitant une optimisation de performance.

4.2.3 Outils de gestion de versions

GitHub

En tant que projet open-source, ProvSQL dispose d'un dépôt de code source sur la plateforme GitHub (Senellart Accessed 2023). Nous avons utilisé GitHub comme outil de contrôle de version pour gérer les modifications apportées au code source de ProvSQL. Cette approche nous a permis de créer des branches distinctes pour travailler sur des fonctionnalités spécifiques, ainsi que pour effectuer des optimisations et des améliorations du code. L'utilisation de branches distinctes nous a offert la flexibilité nécessaire pour expérimenter de nouvelles fonctionnalités sans affecter le code principal, tout en facilitant la fusion des modifications une fois qu'elles étaient testées.

4.2.4 Gestion de bases de données

PostgreSQL

ProvSQL est conçu pour être utilisé avec PostgreSQL, un système de gestion de base de données relationnelle open-source (PostgreSQL Global Development Group Accessed 2023). Nous avons choisi PostgreSQL comme SGBD pour notre projet, car ProvSQL est spécifiquement développé pour fonctionner avec PostgreSQL. En tant que système open-source, PostgreSQL offre une grande flexibilité, une communauté active et un accès à la documentation complète de son code source, ce qui nous a permis de comprendre en détail son fonctionnement interne et d'optimiser nos développements en conséquence.

pgAdmin

pgAdmin (The pgAdmin Development Team 2023) est un outil d'administration de base de données open-source spécialement conçu pour PostgreSQL. Nous avons utilisé pgAdmin comme interface graphique pour interagir avec notre base de données PostgreSQL lors du développement et du test de ProvSQL. Grâce à ses fonctionnalités conviviales, pgAdmin nous a permis de gérer facilement notre base de données, d'exécuter des requêtes SQL, d'explorer les schémas et de surveiller les performances.

Langage SQL

Le langage SQL (Structured Query Language) est un langage de requête standardisé utilisé pour interagir avec les bases de données relationnelles. Nous avons utilisé le langage SQL pour écrire des requêtes, manipuler les données et effectuer des opérations de gestion de base de données dans le cadre de notre travail sur ProvSQL. Le langage SQL nous a permis de définir un benchmark de requêtes temporelles, de les analyser.

4.3 Validation du semi-anneau d'union d'intervalles

Nous avons réalisé des tests pour évaluer la validité et le bon fonctionnement du nouveau semi-anneau d'intervalles de temps. Notre objectif était de tester la cohérence du semi-anneau. Pour ce faire, nous avons commencé par exécuter la requête utilisée dans le chapitre précédent avec le semi-anneau. Cette requête renvoie toutes les villes où il y a au moins deux employés travaillant dans la même ville dans la table `Personnel`.

```
SELECT *, union_intervalles(provenance(),'personnel_level')
FROM (
  SELECT DISTINCT P1.ville
  FROM Personnel P1 JOIN Personnel P2
  ON P1.ville = P2.ville
  WHERE P1.id < P2.id
) t;
```

id	nom	position	ville	temps_de_validite	
1	John	Directeur	New York	{[2022-01-10,2022-09-16)}	t_1
2	Paul	Concierge	New York	{[2023-03-05,2023-07-21)}	t_2
3	Dave	Analyste	Paris	{[2022-10-01,2023-01-01)}	t_3
4	Ellen	Agent de terrain	Berlin	{2023-06-08,2023-12-21)}	t_4
5	Magdalen	Agent double	Paris	{[2022-07-15,2022-12-01)}	t_5
6	Nancy	RH	Paris	{[2023-01-08,2023-07-01)}	t_6
7	Susan	Analyste	Berlin	{[2022-03-10,2022-10-26)}	t_7

FIG. 4.1 : Tableau `Personnel` pour le personnel d'une entreprise

L'exécution de cette requête a produit le résultat affiché dans la figure 4.2. Afin de vérifier sa cohérence, nous avons procédé à la construction manuelle du circuit de provenance afin de calculer le résultat de la requête. Nous avons constaté que les valeurs de temps de validité obtenues manuellement étaient en accord avec celles renvoyées par la requête. La cohérence du semi-anneau, dans ce contexte, signifie que les calculs et les opérations

```
belkis=# SELECT *, union_intervalles(provenance(),'personnel_level')
FROM (
  SELECT DISTINCT P1.ville
  FROM Personnel P1 JOIN Personnel P2
  ON P1.ville = P2.ville
  WHERE P1.id < P2.id
) t;
  ville   |     union_intervalles     |               provsql
----------+---------------------------+--------------------------------------
 Berlin   | {}                        | 38448a45-7b64-57aa-a15a-3bf0a9ac331d
 New York | {}                        | d1c3ca97-ffc0-5f6e-90d2-314af80a7520
 Paris    | {[2022-10-01,2022-12-01)} | eb7b19ef-3f7b-5c93-a576-cbfe03d79344
(3 rows)
```

FIG. 4.2 : Résultat de l'exécution de la requête $Q1$ avec le semi-anneau d'unions d'intervalles

effectués par le semi-anneau d'intervalles de temps sont conformes aux attentes et aux spécifications définies. En d'autres termes, le semi-anneau fonctionne de manière appropriée et produit des résultats valides et cohérents.

Pour aller plus loin dans notre évaluation, nous avons appliqué la même méthode à la requête $Q2$ qui

```
SELECT *, union_intervalles(provenance(),'personnel_level')
FROM (
    SELECT DISTINCT ville FROM Personnel
  EXCEPT
    SELECT DISTINCT P1.ville
    FROM Personnel P1 JOIN Personnel P2
    ON P1.ville = P2.ville
    WHERE P1.ville = P2.ville AND P1.id < P2.id
) t;
```

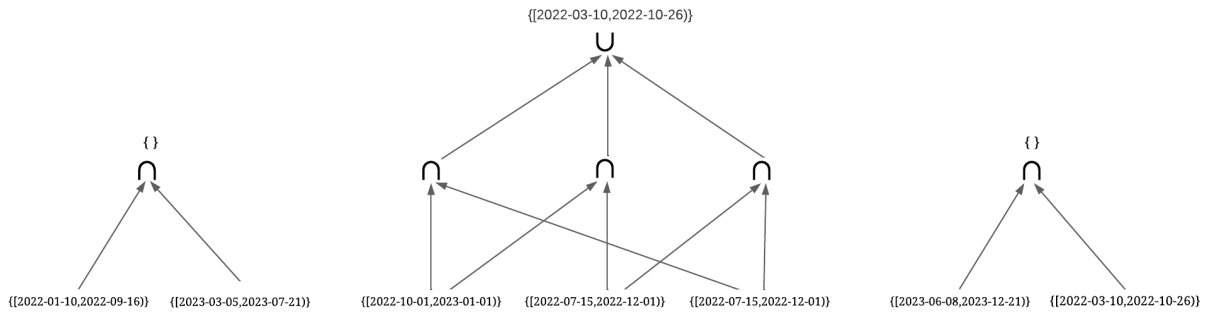


FIG. 4.3 : Circuit de provenance de la requête $Q1$

Après avoir exécuté cette requête, nous avons obtenu le résultat présenté dans la figure 4.4. En construisant manuellement le circuit de provenance, qui est illustré dans la figure , nous avons constaté que les résultats obtenus étaient identiques. Ces tests nous ont

```
belkis=# SELECT *, union_intervalles(provenance(),'personnel_level')
belkis-# FROM (
belkis(#     SELECT DISTINCT ville FROM Personnel
belkis(#   EXCEPT
belkis(#     SELECT DISTINCT P1.ville
belkis(#     FROM Personnel P1 JOIN Personnel P2
belkis(#     ON P1.ville = P2.ville
belkis(#     WHERE P1.ville = P2.ville AND P1.id < P2.id
belkis(# ) t;
  ville   |                          union_intervalles                          |               provsql
----------+-----------------------------------------------------------------------+--------------------------------------
 Paris    | {[2022-07-15,2022-10-01),[2022-12-01,2023-01-01),[2023-01-08,2023-07-01)} | 464db4ac-ea20-5b3b-9d0f-da24f820d1d8
 Berlin   | {[2022-03-10,2022-10-26),[2023-06-08,2023-12-21)}                       | 9ddf7a1b-4fa8-5720-85b4-3eb864520e2a
 New York | {[2022-01-10,2022-09-16),[2023-03-05,2023-07-21)}                       | 496aae55-c7c7-5297-93f7-4b09ee3c823c
(3 rows)
```

FIG. 4.4 : Résultat de l'exécution de la requête $Q2$ avec le semi-anneau d'unions d'intervalles

permis de prouver la validité et la cohérence du semi-anneau d'union d'intervalles dans ses calculs. Les valeurs de temps de validité renvoyées par le semi-anneau correspondent aux résultats attendus, validant ainsi son bon fonctionnement.

4.4 Simulation et comparaison avec GProm dans une base de données temporelle

Afin de tester les performances de ProvSQL, nous avons décidé de le comparer avec GProM (ARAB et al. 2018), un middleware de base de données qui ajoute la prise en charge de la provenance à plusieurs backends de bases de données. Nous nous restreignons sur le backend PostgreSQL pour les besoins de notre projet. GProM implémente son propre compilateur de requêtes avec un optimiseur basé sur des règles heuristiques et des règles de coût pour transformer les requêtes instrumentées en code SQL pour ses différents backends.

Pour cette comparaison, nous avons utilisé la base de données du benchmark TPC-BiH que nous avons générée et adaptée afin d'effectuer des requêtes sur les deux outils de

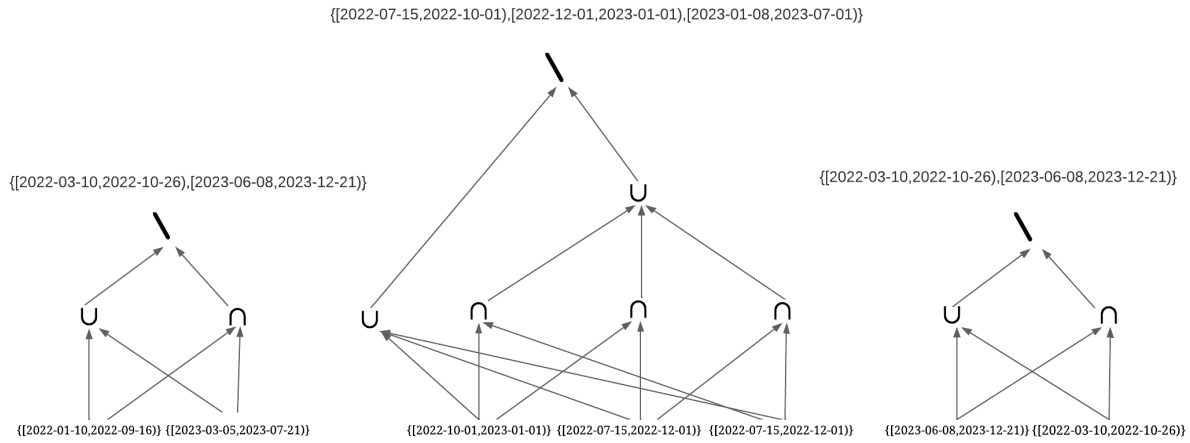


Fig. 4.5 : Circuit de provenance de la requête $Q2$

gestion de provenance.

Nous avons créé un ensemble de requêtes spécifiques pour évaluer les performances des deux systèmes. Pour chaque requête, nous avons effectué les actions suivantes :

1. Exécution de la requête classique : Nous avons exécuté la requête de manière traditionnelle, sans aucune fonctionnalité de gestion de provenance.

2. Exécution de la requête avec ProvSQL sans semi-anneau : Nous avons exécuté la requête en utilisant ProvSQL, mais sans invoquer le calcul de semi-anneau pour la gestion de provenance.

3. Exécution de la requête avec ProvSQL et appel de semi-anneau : Nous avons exécuté la requête en utilisant ProvSQL et en invoquant le calcul de semi-anneau pour la gestion de provenance.

4. Exécution de la requête classique dans GProM (sans calcul de provenance) : Nous avons exécuté la requête dans GProM sans activer le calcul de provenance.

5. Exécution de la requête dans GProM avec calcul de provenance : Nous avons exécuté la requête dans GProM en activant le calcul de provenance.

Nous avons ensuite recueilli les résultats obtenus pour l'exécution de la requête $Q4$ illustrée dans la figure 4.6, qui renvoie uniquement les clients se trouvant dans le même pays et le même segment de marché comme cela a été utilisé dans la section précédente. Nous avons également effectué la requête $Q4.1$ (voir Figure 4.7) qui renvoie tous les clients sauf ceux qui se trouvent dans le même pays et le même segment de marché. Cette requête utilise le semi-anneau d'union d'intervalles avec monus. Nous présentons les résultats dans la figure 4.8 afin de faciliter la comparaison. Nous reconnaissons que provsql n'a pas encore le même niveau de maturité et de complexité que GProm, cependant, les mesures obtenues montrent que provsql reste plus efficace en termes de temps d'exécution pour certains types de requêtes.

```
SELECT DISTINCT c1.c_name, c1.c_nationkey, c1.c_mktsegment
FROM (SELECT * FROM time_customer c1 ORDER BY c_custkey LIMIT 5000) AS c1
JOIN (SELECT * FROM time_customer c2 ORDER BY c_custkey LIMIT 5000) AS c2
ON c1.c_nationkey = c2.c_nationkey AND c1.c_mktsegment = c2.c_mktsegment
WHERE c1.c_custkey <> c2.c_custkey
AND c1.c_name < c2.c_name;
```

FIG. 4.6 : La requête $Q4$

```
SELECT DISTINCT c.c_name, c.c_nationkey, c.c_mktsegment
FROM (SELECT * FROM time_customer ORDER BY c_custkey LIMIT 5000) as c
EXCEPT
    SELECT DISTINCT c1.c_name, c1.c_nationkey, c1.c_mktsegment
    FROM (SELECT * FROM time_customer c1 ORDER BY c_custkey LIMIT 5000) AS c1
    JOIN (SELECT * FROM time_customer c2 ORDER BY c_custkey LIMIT 5000) AS c2
    ON c1.c_nationkey = c2.c_nationkey AND c1.c_mktsegment = c2.c_mktsegment
    WHERE c1.c_custkey <> c2.c_custkey
    AND c1.c_name < c2.c_name;
```

FIG. 4.7 : La requête $Q4.1$

4.5 Optimisation et tests de performances

Dans cette section, nous allons présenter les différentes optimisations que nous avons mises en œuvre dans le cadre de notre projet, ainsi que les résultats obtenus en termes de temps d'exécution. Nous utiliserons l'option `EXPLAIN ANALYSE` qui permet d'obtenir des informations détaillées sur l'exécution d'une requête SQL dans PostgreSQL afin de mesurer les temps d'exécution avant et après chaque optimisation.

Nous utilisons dans la suite de cette section la requête $Q4$ illustrée dans la figure utilisée précédemment pour effectuer nos tests.

Dans la partie conception, nous avons identifié que la fonction `provenance_times` était responsable de la majorité du temps d'exécution, soit 97% du temps d'exécution total des requêtes. Dans la figure 4.9, nous illustrons le plan d'exécution ainsi que le temps d'exécution de la requête sans aucune modification apportée au processus de réécriture.

La première capture d'écran, présentée dans la figure 4.9, illustre l'exécution de la requête normale, sans aucune modification. Nous y observons que la fonction `provenance_times` occupe la majeure partie du temps d'exécution, ce qui confirme son impact significatif sur les performances globales.

Dans la figure 4.10, nous avons exécuté la requête en ne considérant que la fonction `provenance_times`. Ce résultat met en évidence le temps d'exécution spécifique de cette fonction isolée des autres parties de la requête.

En analysant ces résultats, nous avons constaté que la fonction `provenance_times`

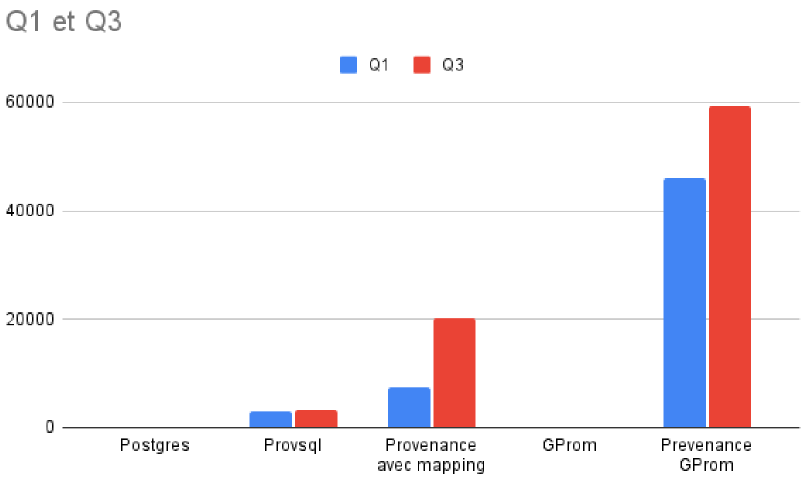
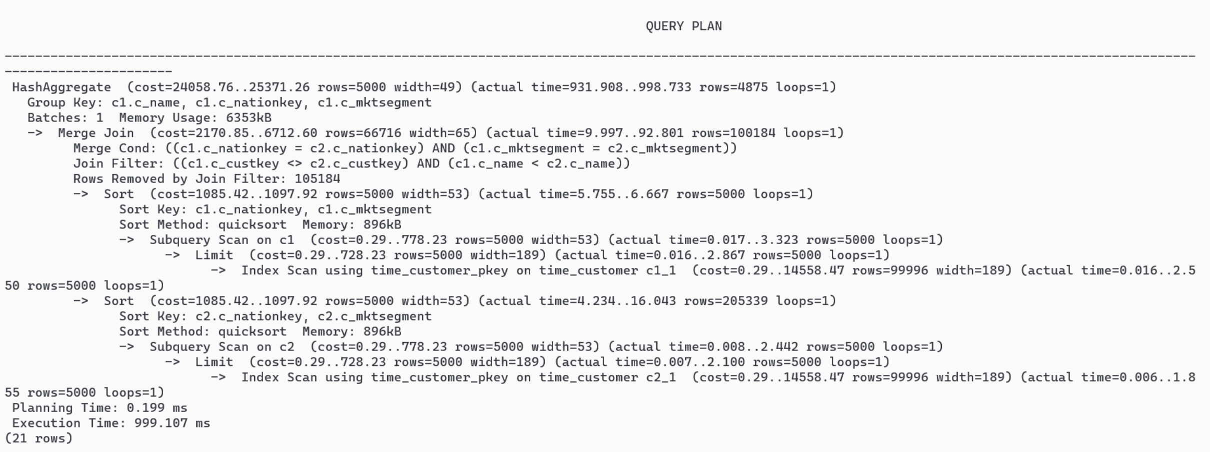
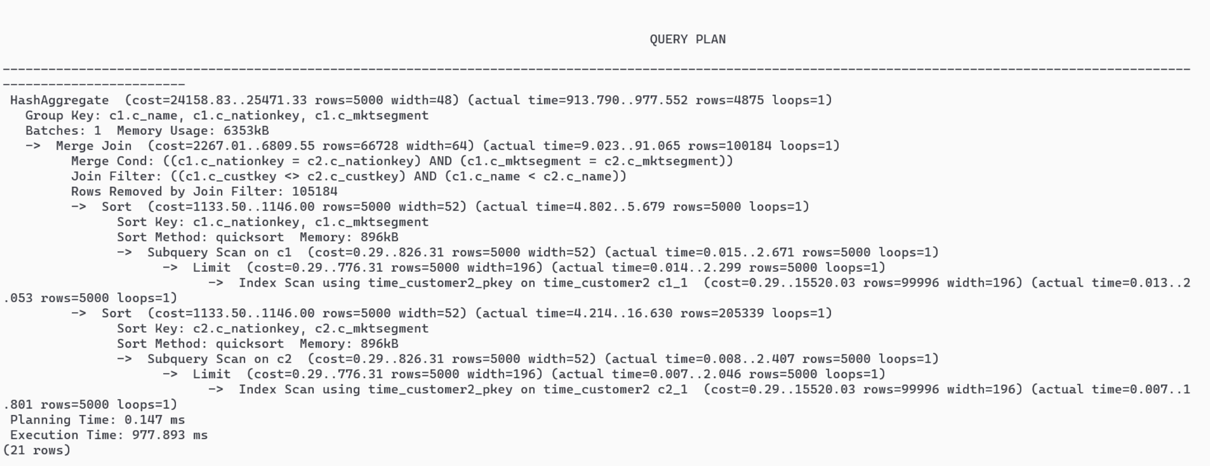


Fig. 4.8 : Temps d'exécution des requêtes $Q1$ et $Q4$



```
                                                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------
 HashAggregate  (cost=24058.76..25371.26 rows=5000 width=49) (actual time=931.908..998.733 rows=4875 loops=1)
   Group Key: c1.c_name, c1.c_nationkey, c1.c_mktsegment
   Batches: 1  Memory Usage: 6353kB
   ->  Merge Join  (cost=2170.85..6712.60 rows=66716 width=65) (actual time=9.997..92.801 rows=100184 loops=1)
         Merge Cond: ((c1.c_nationkey = c2.c_nationkey) AND (c1.c_mktsegment = c2.c_mktsegment))
         Join Filter: ((c1.c_custkey <> c2.c_custkey) AND (c1.c_name < c2.c_name))
         Rows Removed by Join Filter: 105184
         ->  Sort  (cost=1085.42..1097.92 rows=5000 width=53) (actual time=5.755..6.667 rows=5000 loops=1)
               Sort Key: c1.c_nationkey, c1.c_mktsegment
               Sort Method: quicksort  Memory: 896kB
               ->  Subquery Scan on c1  (cost=0.29..778.23 rows=5000 width=53) (actual time=0.017..3.323 rows=5000 loops=1)
                     ->  Limit  (cost=0.29..728.23 rows=5000 width=189) (actual time=0.016..2.867 rows=5000 loops=1)
                           ->  Index Scan using time_customer_pkey on time_customer c1_1  (cost=0.29..14558.47 rows=99996 width=189) (actual time=0.016..2.5
50 rows=5000 loops=1)
         ->  Sort  (cost=1085.42..1097.92 rows=5000 width=53) (actual time=4.234..16.043 rows=205339 loops=1)
               Sort Key: c2.c_nationkey, c2.c_mktsegment
               Sort Method: quicksort  Memory: 896kB
               ->  Subquery Scan on c2  (cost=0.29..778.23 rows=5000 width=53) (actual time=0.008..2.442 rows=5000 loops=1)
                     ->  Limit  (cost=0.29..728.23 rows=5000 width=189) (actual time=0.007..2.100 rows=5000 loops=1)
                           ->  Index Scan using time_customer_pkey on time_customer c2_1  (cost=0.29..14558.47 rows=99996 width=189) (actual time=0.006..1.8
55 rows=5000 loops=1)
 Planning Time: 0.199 ms
 Execution Time: 999.107 ms
(21 rows)
```

Fig. 4.9 : Plan d'exécution de la requête $Q4$ normale



```
                                                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------
 HashAggregate  (cost=24158.83..25471.33 rows=5000 width=48) (actual time=913.790..977.552 rows=4875 loops=1)
   Group Key: c1.c_name, c1.c_nationkey, c1.c_mktsegment
   Batches: 1  Memory Usage: 6353kB
   ->  Merge Join  (cost=2267.01..6809.55 rows=66728 width=64) (actual time=9.023..91.065 rows=100184 loops=1)
         Merge Cond: ((c1.c_nationkey = c2.c_nationkey) AND (c1.c_mktsegment = c2.c_mktsegment))
         Join Filter: ((c1.c_custkey <> c2.c_custkey) AND (c1.c_name < c2.c_name))
         Rows Removed by Join Filter: 105184
         ->  Sort  (cost=1133.50..1146.00 rows=5000 width=52) (actual time=4.802..5.679 rows=5000 loops=1)
               Sort Key: c1.c_nationkey, c1.c_mktsegment
               Sort Method: quicksort  Memory: 896kB
               ->  Subquery Scan on c1  (cost=0.29..826.31 rows=5000 width=52) (actual time=0.015..2.671 rows=5000 loops=1)
                     ->  Limit  (cost=0.29..776.31 rows=5000 width=196) (actual time=0.014..2.299 rows=5000 loops=1)
                           ->  Index Scan using time_customer2_pkey on time_customer2 c1_1  (cost=0.29..15520.03 rows=99996 width=196) (actual time=0.013..2
.053 rows=5000 loops=1)
         ->  Sort  (cost=1133.50..1146.00 rows=5000 width=52) (actual time=4.214..16.630 rows=205339 loops=1)
               Sort Key: c2.c_nationkey, c2.c_mktsegment
               Sort Method: quicksort  Memory: 896kB
               ->  Subquery Scan on c2  (cost=0.29..826.31 rows=5000 width=52) (actual time=0.008..2.407 rows=5000 loops=1)
                     ->  Limit  (cost=0.29..776.31 rows=5000 width=196) (actual time=0.007..2.046 rows=5000 loops=1)
                           ->  Index Scan using time_customer2_pkey on time_customer2 c2_1  (cost=0.29..15520.03 rows=99996 width=196) (actual time=0.007..1
.801 rows=5000 loops=1)
 Planning Time: 0.147 ms
 Execution Time: 977.893 ms
(21 rows)
```

Fig. 4.10 : Plan d'exécution de la requête $Q4$ avec la fonction `provenance_times` seulement.

nécessitait un temps d'exécution important, ce qui justifie les optimisations que nous allons entreprendre dans cette partie.

Filtrage des `gate_one()` dans `provenance_times`

Dans le cadre de notre optimisation, nous avons abordé la problématique du filtrage des portes neutres dans la fonction `provenance_times`. Conformément à notre décision mentionnée dans le chapitre précédent, nous avons choisi de ne pas effectuer le filtrage des portes neutres avant la génération des portes «plus».

La nouvelle version mise à jour de la fonction `provenance_times` est présentée dans la Figure 4.11. Cette version modifiée ne comprend pas l'étape de filtrage des portes neutres, ce qui réduit la complexité du traitement.

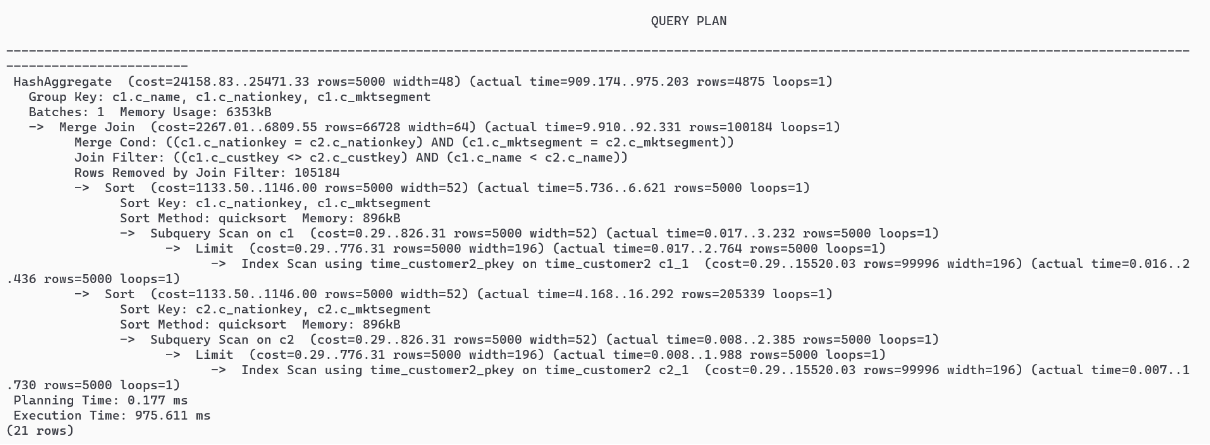
```
CREATE OR REPLACE FUNCTION provenance_times(VARIADIC tokens uuid[])
  RETURNS UUID AS
$$
DECLARE
  times_token uuid;
BEGIN
  CASE array_length(tokens, 1)
    WHEN 0 THEN
      times_token := gate_one();
    WHEN 1 THEN
      times_token := tokens[1];
    ELSE
      times_token := uuid_generate_v5(uuid_ns_provsql(),
        concat('times', tokens));

      PERFORM create_gate(times_token, 'times', tokens);
  END CASE;

  RETURN times_token;
END
$$ LANGUAGE plpgsql SET search_path=provsql,pg_temp,public SECURITY DEFINER;
```

Fig. 4.11 : La fonction `provenance_times` sans filtrage des portes neutres

Pour évaluer les performances de cette optimisation, nous avons exécuté la requête Q4 en utilisant la nouvelle version de la fonction `provenance_times`. Le plan d'exécution de cette requête est illustré dans la Figure 4.12.

La suppression du filtrage des portes neutres a entraîné l'exécution de plus d'opérations de multiplication de portes. Cependant, malgré cet ajout d'opérations supplémentaires, nous avons observé une légère diminution du temps d'exécution de la requête, représentant environ $2,5\%$ du coût total. Cette réduction démontre l'efficacité de notre optimisation.

Il est important de noter que l'ampleur de l'amélioration peut varier en fonction de



```
                                                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------
 HashAggregate  (cost=24158.83..25471.33 rows=5000 width=48) (actual time=909.174..975.203 rows=4875 loops=1)
   Group Key: c1.c_name, c1.c_nationkey, c1.c_mktsegment
   Batches: 1  Memory Usage: 6353kB
   ->  Merge Join  (cost=2267.01..6809.55 rows=66728 width=64) (actual time=9.910..92.331 rows=100184 loops=1)
         Merge Cond: ((c1.c_nationkey = c2.c_nationkey) AND (c1.c_mktsegment = c2.c_mktsegment))
         Join Filter: ((c1.c_custkey <> c2.c_custkey) AND (c1.c_name < c2.c_name))
         Rows Removed by Join Filter: 105184
         ->  Sort  (cost=1133.50..1146.00 rows=5000 width=52) (actual time=5.736..6.621 rows=5000 loops=1)
               Sort Key: c1.c_nationkey, c1.c_mktsegment
               Sort Method: quicksort  Memory: 896kB
               ->  Subquery Scan on c1  (cost=0.29..826.31 rows=5000 width=52) (actual time=0.017..3.232 rows=5000 loops=1)
                     ->  Limit  (cost=0.29..776.31 rows=5000 width=196) (actual time=0.017..2.764 rows=5000 loops=1)
                           ->  Index Scan using time_customer2_pkey on time_customer2 c1_1  (cost=0.29..15520.03 rows=99996 width=196) (actual time=0.016..2
.436 rows=5000 loops=1)
         ->  Sort  (cost=1133.50..1146.00 rows=5000 width=52) (actual time=4.168..16.292 rows=205339 loops=1)
               Sort Key: c2.c_nationkey, c2.c_mktsegment
               Sort Method: quicksort  Memory: 896kB
               ->  Subquery Scan on c2  (cost=0.29..826.31 rows=5000 width=52) (actual time=0.008..2.385 rows=5000 loops=1)
                     ->  Limit  (cost=0.29..776.31 rows=5000 width=196) (actual time=0.008..1.988 rows=5000 loops=1)
                           ->  Index Scan using time_customer2_pkey on time_customer2 c2_1  (cost=0.29..15520.03 rows=99996 width=196) (actual time=0.007..1
.730 rows=5000 loops=1)
 Planning Time: 0.177 ms
 Execution Time: 975.611 ms
(21 rows)
```

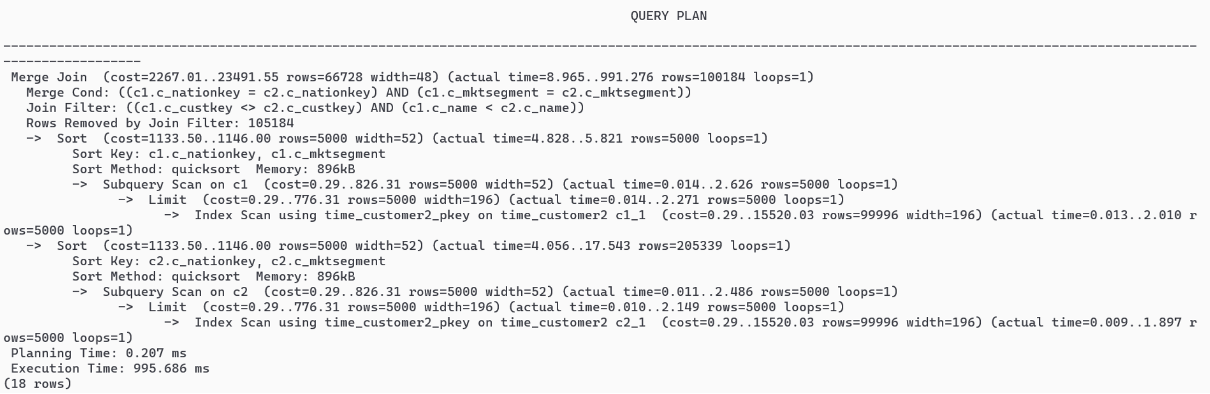
Fig. 4.12 : Plan d'exécution de la requête Q4 avec filtrage des portes neutres

la complexité de la requête, de la taille des données et d'autres facteurs. Dans notre cas d'utilisation, la suppression du filtrage des portes neutres s'est avérée être une optimisation efficace, permettant d'accélérer le traitement des requêtes et d'améliorer les performances globales du système.

La fonction `uuid_generate_v5` dans `provenance_times`

Dans le chapitre «conception», nous avons abordé l'utilisation de la fonction `uuid_generate_v5` pour générer des UUID de version 5. Cependant, dans un souci de performance, nous avons exploré une alternative en remplaçant cette fonction par la fonction `uuid_generate_v3` (*PostgreSQL Documentation - uuid-ossp* 2021).

Nous avons donc procédé à des tests d'exécution en utilisant la fonction `uuid_generate_v3` pour évaluer son impact sur les performances. Le plan d'exécution de la requête $Q4$ est présenté dans la figure 4.13. Cependant, les résultats ont montré que le temps d'exécution avec la fonction `uuid_generate_v3` était similaire à celui avec la fonction `uuid_generate_v5`. Par conséquent, dans notre cas spécifique, le remplacement de la fonction n'a pas entraîné une amélioration significative des performances.



```
                                                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------
 Merge Join  (cost=2267.01..23491.55 rows=66728 width=48) (actual time=8.965..991.276 rows=100184 loops=1)
   Merge Cond: ((c1.c_nationkey = c2.c_nationkey) AND (c1.c_mktsegment = c2.c_mktsegment))
   Join Filter: ((c1.c_custkey <> c2.c_custkey) AND (c1.c_name < c2.c_name))
   Rows Removed by Join Filter: 105184
   ->  Sort  (cost=1133.50..1146.00 rows=5000 width=52) (actual time=4.828..5.821 rows=5000 loops=1)
         Sort Key: c1.c_nationkey, c1.c_mktsegment
         Sort Method: quicksort  Memory: 896kB
         ->  Subquery Scan on c1  (cost=0.29..826.31 rows=5000 width=52) (actual time=0.014..2.626 rows=5000 loops=1)
               ->  Limit  (cost=0.29..776.31 rows=5000 width=196) (actual time=0.014..2.271 rows=5000 loops=1)
                     ->  Index Scan using time_customer2_pkey on time_customer2 c1_1  (cost=0.29..15520.03 rows=99996 width=196) (actual time=0.013..2.010 r
ows=5000 loops=1)
   ->  Sort  (cost=1133.50..1146.00 rows=5000 width=52) (actual time=4.056..17.543 rows=205339 loops=1)
         Sort Key: c2.c_nationkey, c2.c_mktsegment
         Sort Method: quicksort  Memory: 896kB
         ->  Subquery Scan on c2  (cost=0.29..826.31 rows=5000 width=52) (actual time=0.011..2.486 rows=5000 loops=1)
               ->  Limit  (cost=0.29..776.31 rows=5000 width=196) (actual time=0.010..2.149 rows=5000 loops=1)
                     ->  Index Scan using time_customer2_pkey on time_customer2 c2_1  (cost=0.29..15520.03 rows=99996 width=196) (actual time=0.009..1.897 r
ows=5000 loops=1)
 Planning Time: 0.207 ms
 Execution Time: 995.686 ms
(18 rows)
```

Fig. 4.13 : Plan d'exécution de la requête $Q4$ avec `uuid_generate_v3`

Après une analyse plus approfondie, nous avons pris la décision de conserver la fonction `uuid_generate_v5` dans notre implémentation.

La fonction `hash_search()` dans `create_gate()`

Pour évaluer les performances de la fonction `create_gate()`, nous avons utilisé le mode de débogage pour suivre l'exécution du code pas à pas. Nous avons ajouté des points d'arrêt aux endroits clé et en particulier où la recherche dans la table de hachage se produit, plus précisément dans le bloc de code suivant :

```
entry = (provsqlHashEntry *)hash_search_with_hash_value(
    provsql_hash,
    token,
    *(uint32*)token,
    HASH_ENTER,
    &found);
```

En utilisant cette technique, nous avons pu mesurer le temps d'exécution passé dans ce bloc spécifique. Nos observations ont révélé qu'environ 200 millisecondes étaient nécessaires pour effectuer la recherche des portes dans la table de hachage, soit 20% du temps totale

Ainsi, pour optimiser cette partie, nous avons décidé d'implémenter une fonction simplifiée appelée `provsql_hash_uuid()` pour le hachage. Elle prend en paramètre un pointeur vers une clé de type `void*` et retourne un entier non signé de 32 bits (`uint32`). Son rôle est de convertir le pointeur `key` en un pointeur vers un entier non signé de 32 bits, puis de renvoyer la valeur pointée par ce pointeur. Le code source est présenté dans la figure 4.14.

```
uint32 provsql_hash_uuid(void *key) {
    uint32 *ptr = (uint32 *)key;
    return *ptr;
}
```

Fig. 4.14 : Code source de la nouvelle fonction de hachage

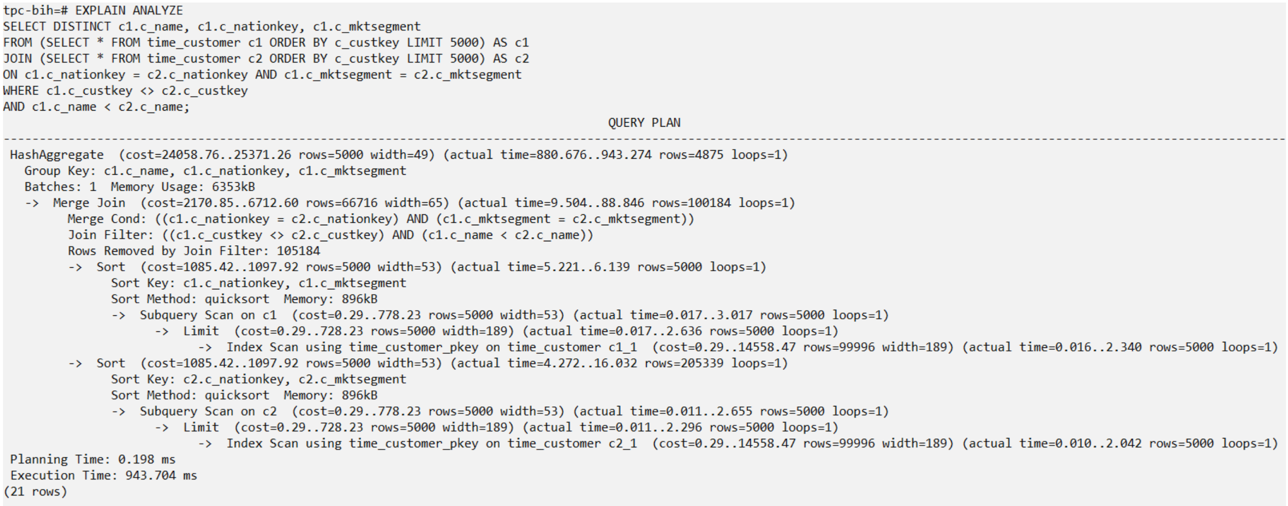
Pour tester les performances de la nouvelle fonction de hachage, nous avons exécuté la requête $Q4$ et constaté qu'elle n'a pas entraîné d'améliorations significatives, comme le montre la Figure 4.15.

Pour tester les performances de la nouvelle fonction de hachage, nous avons effectué des mesures après avoir apporté les modifications suggérées à la fonction `provenance_times`.

Nous en déduisons que le temps passé dans la recherche dans la table de hachage n'est pas principalement dû à la fonction de hachage elle-même, mais plutôt au grand nombre de portes générées.

4.6 Conclusion

Dans ce chapitre, nous avons présenté les technologies et les outils que nous avons utilisés pour le développement des contributions apportées à ProvSQL.



```
tpc-bih=# EXPLAIN ANALYZE
SELECT DISTINCT c1.c_name, c1.c_nationkey, c1.c_mktsegment
FROM (SELECT * FROM time_customer c1 ORDER BY c_custkey LIMIT 5000) AS c1
JOIN (SELECT * FROM time_customer c2 ORDER BY c_custkey LIMIT 5000) AS c2
ON c1.c_nationkey = c2.c_nationkey AND c1.c_mktsegment = c2.c_mktsegment
WHERE c1.c_custkey <> c2.c_custkey
AND c1.c_name < c2.c_name;
                                                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=24058.76..25371.26 rows=5000 width=49) (actual time=880.676..943.274 rows=4875 loops=1)
   Group Key: c1.c_name, c1.c_nationkey, c1.c_mktsegment
   Batches: 1  Memory Usage: 6353kB
   ->  Merge Join  (cost=2170.85..6712.60 rows=66716 width=65) (actual time=9.504..88.846 rows=100184 loops=1)
         Merge Cond: ((c1.c_nationkey = c2.c_nationkey) AND (c1.c_mktsegment = c2.c_mktsegment))
         Join Filter: ((c1.c_custkey <> c2.c_custkey) AND (c1.c_name < c2.c_name))
         Rows Removed by Join Filter: 105184
         ->  Sort  (cost=1085.42..1097.92 rows=5000 width=53) (actual time=5.221..6.139 rows=5000 loops=1)
               Sort Key: c1.c_nationkey, c1.c_mktsegment
               Sort Method: quicksort  Memory: 896kB
               ->  Subquery Scan on c1  (cost=0.29..778.23 rows=5000 width=53) (actual time=0.017..3.017 rows=5000 loops=1)
                     ->  Limit  (cost=0.29..728.23 rows=5000 width=189) (actual time=0.017..2.636 rows=5000 loops=1)
                           ->  Index Scan using time_customer_pkey on time_customer c1_1  (cost=0.29..14558.47 rows=99996 width=189) (actual time=0.016..2.340 rows=5000 loops=1)
         ->  Sort  (cost=1085.42..1097.92 rows=5000 width=53) (actual time=4.272..16.032 rows=205339 loops=1)
               Sort Key: c2.c_nationkey, c2.c_mktsegment
               Sort Method: quicksort  Memory: 896kB
               ->  Subquery Scan on c2  (cost=0.29..778.23 rows=5000 width=53) (actual time=0.011..2.655 rows=5000 loops=1)
                     ->  Limit  (cost=0.29..728.23 rows=5000 width=189) (actual time=0.011..2.296 rows=5000 loops=1)
                           ->  Index Scan using time_customer_pkey on time_customer c2_1  (cost=0.29..14558.47 rows=99996 width=189) (actual time=0.010..2.042 rows=5000 loops=1)
 Planning Time: 0.198 ms
 Execution Time: 943.704 ms
(21 rows)
```

Fig. 4.15 : Résultats des tests de performance de la nouvelle fonction de hachage

Dans la deuxième partie de ce chapitre, nous avons réalisé des tests pour évaluer la validité et le bon fonctionnement du semi-anneau d'union d'intervalles de ProvSQL. Nous avons exécuté des requêtes et construit manuellement les circuits de provenance pour vérifier la cohérence des résultats obtenus. Les tests ont confirmé que le semi-anneau d'union d'intervalles fonctionne de manière appropriée et produit des résultats valides et cohérents.

Dans la troisième partie, nous avons effectué une comparaison des performances de ProvSQL avec GProM, un middleware de base de données qui prend en charge la gestion de provenance. Nous avons utilisé le benchmark de requêtes que nous avons créé à partir de la base de données du benchmark *TPC-BiH* pour évaluer les performances des deux systèmes.

Enfin, la dernière section était dédiée à l'optimisation et aux tests de performances. Nous avons présenté les modifications que nous avons apportées à ProvSQL et les avons évaluées en effectuant des tests approfondis. L'objectif était d'améliorer les temps d'exécution du système et d'optimiser les parties critiques du code. Malheureusement, certaines propositions d'amélioration n'ont pas été aussi efficaces que prévu. Certaines modifications n'ont pas abouti à des améliorations significatives des temps d'exécution du système.

Conclusion et perspectives

Conclusion générale

ProvSQL est un module open-source développé par l'équipe Valda de l'Inria. Il étend le système de gestion de base de données PostgreSQL en lui ajoutant la capacité de calculer la provenance et les probabilités des résultats des requêtes. La provenance des données fait référence à l'historique de dérivation et au processus par lequel les données sont arrivées dans une base de données. Comprendre la provenance des données joue un rôle crucial dans la garantie de leur qualité et de leur fiabilité.

Dans le cadre de notre projet, nous nous sommes concentrés sur l'extension des fonctionnalités de ProvSQL pour prendre en compte le contexte temporel des données. Les données temporelles sont des données qui évoluent dans le temps, et il est important de pouvoir interroger et analyser les temps de validité des résultats des requêtes sur des bases de données temporelles. Afin de répondre à ce besoin, nous avons conçu, développé et évalué une solution qui permet d'interroger les temps de validité des résultats des requêtes effectuées sur de telles bases de données.

Notre solution repose sur un semi-anneau qui offre les fonctionnalités nécessaires pour manipuler les données temporelles de manière précise et cohérente. Grâce à cette extension, il est possible d'explorer les temps de validité des données obtenues dans les résultats des requêtes et d'effectuer des analyses basées sur le contexte temporel.

Nous avons débuté par une analyse approfondie des travaux de recherche existants sur la provenance et les bases de données temporelles, ce qui nous a permis de mieux comprendre les avancées et les défis actuels dans ces domaines. En tirant parti de ProvSQL, nous avons introduit un moyen d'ajouter le support des données temporelles à PostgreSQL, comblant ainsi une lacune importante dans le système.

L'implémentation d'un semi-anneau nous a permis de manipuler les données temporelles de manière précise et cohérente, en offrant des fonctionnalités pour interroger les tables en fonction de leur état pendant une période de temps spécifique et explorer les temps de validité des données obtenues dans les résultats des requêtes.

Ensuite, nous avons défini un formalisme pour la réalisation de tests sur ProvSQL en créant un benchmark spécifique pour les requêtes temporelles. Ce benchmark a été conçu pour évaluer les performances de ProvSQL dans des scénarios représentatifs et comparer les résultats avec d'autres systèmes similaires.

Nous avons ensuite réalisé des simulations sur des bases de données temporelles en utilisant le semi-anneau développé. Ces simulations ont été essentielles pour tester et

valider les fonctionnalités du système, en reproduisant des scénarios réels et en évaluant sa performance.

Une comparaison des performances du semi-anneau dans ProvSQL avec d'autres systèmes de gestion de provenance existants a été effectuée. Cette évaluation comparative nous a permis de mesurer l'efficacité et l'efficience de ProvSQL par rapport à d'autres solutions similaires.

Dans un souci d'optimisation, nous avons mené une évaluation du code source de ProvSQL afin d'accélérer ses performances et d'optimiser son utilisation des ressources. Nous avons identifié et résolu les éventuels goulots d'étranglement, garantissant ainsi une exécution rapide et efficace des requêtes.

Perspectives

Pour les perspectives futures, nous envisageons les orientations suivantes afin d'améliorer ProvSQL :

- Étendre le sous-ensemble du langage SQL supporté par ProvSQL en un sous-ensemble suffisamment large pour couvrir un grand nombre de cas d'utilisation, en particulier en termes de requêtes avec agrégation.
- Permettre le passage à l'échelle de ProvSQL pour traiter des bases de données de taille arbitrairement grandes, en assurant une performance optimale.
- Optimiser le temps de calcul des requêtes avec provenance et probabilités, afin de minimiser le surcoût par rapport aux requêtes classiques.
- Développer le support des distributions continues de probabilité au sein de ProvSQL, offrant ainsi une plus grande flexibilité dans l'analyse des données probabilistes.
- Faciliter les applications portant sur différents semi-anneaux de provenance, permettant ainsi une gestion avancée de la provenance dans les bases de données.

En intégrant ces améliorations, notre objectif ultime est d'améliorer la qualité des données, de faciliter une analyse précise et de permettre aux chercheurs et universitaires de tirer des informations précieuses de leurs données.

Cela conclut notre projet, qui a été guidé par la volonté de répondre aux besoins croissants de provenance et de contexte temporel dans le domaine de la gestion des bases de données. Nous espérons que ces perspectives ouvriront de nouvelles opportunités pour des avancées futures dans ce domaine passionnant.

Bibliographie

Agrawal, Parag et al. (2006). «Trio : a system for data, uncertainty, and lineage». In : *Very Large Data Bases Conference.*

Arab, Bahareh Sadat et al. (2018). «GProM - A Swiss Army Knife for Your Provenance Needs». In : *IEEE Data Eng. Bull.* 41, p. 51-62.

Ariav, Gad (déc. 1986). «A Temporally Oriented Data Model». In : *ACM Trans. Database Syst.* 11.4, p. 499-527.

Baader, Franz et Tobias Nipkow (1998). *Term Rewriting and All That.* USA : Cambridge University Press.

Bettini, Claudio et al. (1997). «A Glossary of Time Granularity Concepts». In : *Temporal Databases, Dagstuhl.*

Böhlen, Michael, R. Busatto et Christian Jensen (mars 1998). «Point-versus interval-based temporal data models». In : p. 192-200.

Buneman, Peter, Sanjeev Khanna et Wang-chiew Tan (déc. 2000a). «Data Provenance : Some Basic Issues». In : t. 1974, p. 87-93.

Buneman, Peter, Sanjeev Khanna et Tan Wang-Chiew (2001). «Why and Where : A Characterization of Data Provenance». In : *Database Theory — ICDT 2001.* Sous la dir. de Jan Van den Bussche et Victor Vianu. Berlin, Heidelberg : Springer Berlin Heidelberg, p. 316-330.

Buneman, Peter, David Maier et Jennifer Widom (2000b). «Where was your data yesterday, and where will it go tomorrow ? Data Annotation and Provenance for Scientific Applications». In : *Position paper for NSF Workshop on Information and Data Management (IDM 2000) : Research Agenda into the Future, Chicago IL.*

Chen, Peng, Beth Plale et Mehmet S. Aktas (2012). «Temporal representation for scientific data provenance». In : *2012 IEEE 8th International Conference on E-Science*, p. 1-8.

Cheney, James, Laura Chiticariu et Wang-chiew Tan (jan. 2009). «Provenance in Databases : Why, How, and Where». In : *Foundations and Trends in Databases* 1, p. 379-474.

Chiticariu, Laura, Wang-Chiew Tan et Gaurav Vijayvargiya (2005). «DBNotes : A Post-It System for Relational Databases Based on Provenance». In : *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data.* SIGMOD '05. Baltimore, Maryland : Association for Computing Machinery, p. 942-944.

Clifford, James, Albert Croker et Alexander Tuzhilin (mars 1994). «On Completeness of Historical Relational Query Languages». In : *ACM Trans. Database Syst.* 19.1, p. 64-116.

CLIFFORD, James et Abdullah UZ TANSEL (1985). «On an Algebra for Historical Relational Databases : Two Views». In : *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*. SIGMOD '85. Austin, Texas, USA : Association for Computing Machinery, p. 247-265.

COUNCIL, Transaction Processing Performance (2022). «TPC Benchmark H (TPC-H) Specification». In : Available : https://www.tpc.org/tpc_documents_current_versions/tpch_current.pdf.

CURRIM, S. et S. THOMAS (2009). «Efficient Data and Temporal Query Execution in a Benchmark Environment». In : *Technical Report TR-91, School of Computing, Queen's University*.

DATE, C.J., Hugh DARWEN et Nikos LORENTZOS (2014). *Time and Relational Theory, Second Edition : Temporal Databases in the Relational Model and SQL*. 2nd. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc.

DEMPSEY, Jennifer et al. (2018). «Temporal Benchmarks». In : *Encyclopedia of Database Systems*. Sous la dir. de Ling LIU et M. Tamer ÖZSU. New York, NY : Springer New York, p. 3912-3917.

DIGNÖS, Anton et al. (fév. 2019). «Snapshot Semantics for Temporal Multiset Relations». In : *Proc. VLDB Endow.* 12.6, p. 639-652.

DYRESON, Curtis E. et Richard T. SNODGRASS (1993). «Timestamp semantics and representation». In : *Information Systems* 18.3, p. 143-166.

ELMASRI, Ramez et Shamkant B. NAVATHE (1994). *Fundamentals of Database Systems (2nd Ed.)* USA : Benjamin-Cummings Publishing Co., Inc.

GALHARDAS, Helena et al. (juill. 2001). «Improving Data Cleaning Quality using a Data Lineage Facility». In.

GAO, Dengfeng (sept. 2011). «Supporting the Procedural Component of Query Languages over Time-Varying Data». In.

GAO, Dengfeng et Richard T. SNODGRASS (2003). «Temporal Slicing in the Evaluation of XML Queries». In : *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*. VLDB '03. Berlin, Germany : VLDB Endowment, p. 632-643.

GEERTS, Floris et Antonella POGGI (2010). «On database query languages for K-relations». In : *Journal of Applied Logic* 8.2. Selected papers from the Logic in Databases Workshop 2008, p. 173-185.

GLAVIC, Boris, Renée J. MILLER et Gustavo ALONSO (2013). «Using SQL for Efficient Generation and Querying of Provenance Information». In : *In Search of Elegance in the Theory and Practice of Computation : Essays Dedicated to Peter Buneman*. Sous la dir. de Val TANNEN et al. Berlin, Heidelberg : Springer Berlin Heidelberg, p. 291-320.

GAZEK, Kazimierz (2002). «The Algebraic Theory of Semirings». In : *A Guide to the Literature on Semirings and their Applications in Mathematics and Information Sciences : With Complete Bibliography*. Dordrecht : Springer Netherlands, p. 11-23.

GOBLE, Carole (2002). «Position statement : Musings on provenance, workflow and (semantic web) annotations for bioinformatics». In : *Workshop on Data Derivation and Provenance, Chicago*. T. 3.

GREEN, Todd J., Grigoris KARVOUNARAKIS et Val TANNEN (2007). «Provenance Semirings». In : *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '07. Beijing, China : Association for Computing Machinery, p. 31-40.

Greenwood, Mark A. et al. (2003). «Provenance of e-Science Experiments - Experience from Bioinformatics». In.

Group, PostgreSQL Global Development (2021). *PostgreSQL : Documentation - UUID Type*.

Hernich, André (2010). «Answering non-monotonic queries in relational data exchange». In : *International Conference on Database Theory*.

Ikeda, Robert et Jennifer Widom (2009). «Data lineage : A survey». In : *Stanford University Publications. http ://ilpubs. stanford. edu* 8090.918, p. 1.

Jagadish, H. V. et Frank Olken (2004). «Database Management for Life Sciences Research». In : *SIGMOD Rec.* 33.2, p. 15-20.

Jensen, Christian et al. (jan. 1993). *A consensus test suite of temporal database queries*.

Jensen, Christian S. et Richard T. Snodgrass (1999). «Temporal Data Management». In : *IEEE Trans. Knowl. Data Eng.* 11.1, p. 36-44.

— (2009). «Temporal Data Models». In : *Encyclopedia of Database Systems*. Sous la dir. de LING LIU et M. TAMER ÖZSU. Boston, MA : Springer US, p. 2952-2957.

Jensen, Christian S. et al. (1997). «The Consensus Glossary of Temporal Database Concepts - February 1998 Version». In : *Temporal Databases, Dagstuhl*.

Johnston, T. (jan. 2014). «Bitemporal Data : Theory and Practice». In : p. 1-367.

Kaufmann, Martin et al. (2013a). «Benchmarking Databases with History Support». In.

Kaufmann, Martin et al. (août 2013b). «Comprehensive and Interactive Temporal Query Processing with SAP HANA». In : *Proc. VLDB Endow.* 6.12, p. 1210-1213.

Kaufmann, Martin et al. (août 2013c). «TPC-BiH : a benchmark for bitemporal databases». In.

Kostylev, Egor V., Juan L. Reutter et András Z. Salamon (2012). «Classification of Annotation Semirings over Query Containment». In : *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS '12. Scottsdale, Arizona, USA : Association for Computing Machinery, p. 237-248.

Liu, Ling et M. Tamer Özsu, éd. (2018). *Encyclopedia of Database Systems, Second Edition*. Springer.

Lomet, David B et Betty Joan Salzberg (1992). *Rollback Databases*. Citeseer.

Melton, Jim (1998). *Understanding SQL's Stored Procedures : A Complete Guide to SQL/PSM*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc.

Miles, Simon et al. (jan. 2006). «The requirements of recording and using provenance in e-Science experiments». In : *Journal of Grid Computing - GRID*.

Mitsa, Theophano (2010). *Temporal data mining*. CRC press.

Mohammadi, Sareh et Nematollaah Shiri (2022). «ARDBS : Efficient Processing of Provenance Queries Over Annotated Relations». In : *Database and Expert Systems Applications*. Sous la dir. de Christine Strauss et al. Cham : Springer International Publishing, p. 263-269.

Navathe, Shamkant B. et Rafi Ahmed (1989). «A Temporal Relational Model and a Query Language». In : *Inf. Sci.* 49, p. 147-175.

Rani, Asma, Navneet Goyal et Shashi K. Gadia (2016). «Efficient Multi-Depth Querying on Provenance of Relational Queries Using Graph Database». In : *Proceedings of the 9th Annual ACM India Conference*. COMPUTE '16. Gandhinagar, India : Association for Computing Machinery, p. 11-20.

SAGE, Marc (2018). «Groupes, monoïdes (reliquat) - Normale Sup». In : Accessed on June 14, 2023.

Sarda, Nandlal L. (1990). «Algebra and Query Language for A Historical Data Model». In : *Comput. J.* 33, p. 11-18.

Senellart, Pierre (déc. 2017). «Provenance and Probabilities in Relational Databases : From Theory to Practice». In : *ACM SIGMOD Record* 46.

Senellart, Pierre et al. (août 2018). «ProvSQL : Provenance and Probability Management in PostgreSQL». In : *Proc. VLDB Endow.* 11.12, p. 2034-2037.

Snodgrass, Richard T. et Ilsoo Ahn (1986). «Temporal Databases». In : *Computer* 19.9, p. 35-42.

Snodgrass, Richard T. et al. (mai 2008). «Validating Quicksand : Temporal Schema Versioning in XSchema». In : *Data Knowl. Eng.* 65.2, p. 223-242.

Suciu, Dan (2020). «Probabilistic Databases for All». In : *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS'20. Portland, OR, USA : Association for Computing Machinery, p. 19-31.

Talha, Norhaizan (oct. 2004). «Metadata management system (MMS)». In.

Thomas, Stephen (avr. 2009). «Implementation and Evaluation of Temporal Representations in XML». In.

Thomas, Stephen W, Richard Snodgrass et Rui Zhang (sept. 2014). «Benchmark frameworks and Bench». In : *Software : Practice and Experience* 44, p. 1047-1075.

Vansummeren, Stijn et James Cheney (2007). «Recording Provenance for SQL Queries and Updates». In : *IEEE Data Eng. Bull.* 30, p. 29-37.

Wang, Y. Richard (Yng-Yuh Richard) et Stuart E. Madnick (1990). *A polygen model for heterogeneous database systems : the source tagging perspective*. Working papers 3119-90. CIS (Series) (Sl. Massachusetts Institute of Technology (MIT), Sloan School of Management.

Widom, Jennifer (2004). «Trio : A System for Integrated Management of Data, Accuracy, and Lineage». In : *Conference on Innovative Data Systems Research*.

Wiener, J.L. et al. (mai 1997). «The WHIPS prototype for data warehouse creation and maintenance». In : p. 589-.

Yao, B.B., M.T. Ozsu et N. Khandelwal (2004). «XBench benchmark and performance testing of XML DBMSs». In : *Proceedings. 20th International Conference on Data Engineering*, p. 621-632.

Webographie

τBench : Benchmark Suite (2023). https://www2.cs.arizona.edu/projects/tau/tBench/. Accessed on June 9, 2023.

Emrah Mete (juill. 2020). *Implementing Temporal Validity in Oracle*. DZone. Accessed on June 9, 2023. URL : https://dzone.com/articles/implementing-temporal-validity-in-oracle.

IBM (2023). *Working with System-Period and Temporal Tables*. IBM Documentation. Accessed on May 19, 2023. URL : %7Bhttps://www.ibm.com/docs/en/i/7.4?topic=administration-working-system-period-temporal-tables%7D.

— (s. d.). *IBM Db2*. IBM. Accessed on June 10, 2023. URL : https://www.ibm.com/fr-fr/products/db2.

Inria (Accessed 2023). *Inria*. Website. URL : https://www.inria.fr/fr.

Jean-Francois Verrier and Dominique Jeunot (s. d.). *Temporal Validity and Information Lifecycle Management*. Oracle Technology Network. Accessed on June 10, 2023. URL : https://www.oracle.com/webfolder/technetwork/tutorials/obe/db/12c/r1/ilm/temporal/temporal.html.

Kuleshova, Ekaterina (2011). *Provenance in temporal databases : Facharbeit*.

Merriam-Webster (2023). *Provenance*. https://www.merriam-webster.com/dictionary/provenance. Retrieved May 24, 2023.

Microsoft Corporation (2023a). *Developing in WSL using Visual Studio Code*. https://learn.microsoft.com/en-us/windows/wsl/tutorials/wsl-vscode. [Accessed : June 23, 2023].

— (2023b). *Windows Subsystem for Linux*. https://learn.microsoft.com/en-us/windows/wsl/. [Accessed : June 23, 2023].

Oracle Database 19c Documentation : Application Development, Features, and Security (s. d.). Oracle Documentation. Accessed on June 10, 2023. URL : https://docs.oracle.com/en/database/oracle/oracle-database/19/adfns/index.html.

Oracle Database 19c Documentation : Design Basics (s. d.). Oracle Documentation. Accessed on June 10, 2023. URL : https://docs.oracle.com/en/database/oracle/oracle-database/19/adfns/design_basics.html#GUID-53F19441-81F3-4A4D-923C-464CAE0964D3.

Pierre Senellart (2023). *provsql_shmem.c - GitHub*. https://github.com/PierreSenellart/provsql/blob/master/src/provsql_shmem.c. [Accessed : June 23, 2023].

PostgreSQL Documentation - uuid-ossp (2021). https://docs.postgresql.fr/13/uuid-ossp.html. Accessed on June 24, 2023.

PostgreSQL Global Development Group (Accessed 2023). *PostgreSQL*. Website. URL : https://www.postgresql.org.

PostgreSQL Global Development Group (s. d.). *Range Types - PostgreSQL 14 Documentation*. PostgreSQL Documentation. Accessed on June 10, 2023. URL : https://www.postgresql.org/docs/current/rangetypes.html.

SAP (s. d.[a]). *SAP France*. https ://www.sap.com/france/index.html. Accessed : 10 June 2023.

— (s. d.[b]). *What is SAP HANA*. SAP. Accessed on June 10, 2023. URL : https://www.sap.com/france/products/technology-platform/hana/what-is-sap-hana.html.

Schemas (s. d.). PostgreSQL Documentation. https://www.postgresql.org/docs/current/ddl-schemas.html.

Senellart, Pierre (Accessed 2023). *ProvSQL*. GitHub. URL : https://github.com/PierreSenellart/provsql.

The pgAdmin Development Team (2023). *pgAdmin Documentation*. https://www.pgadmin.org/docs/pgadmin4/latest/index.html. [Accessed : June 23, 2023].

The PostgreSQL Global Development Group (2023). *PL/pgSQL - PostgreSQL 14 Documentation*. https://www.postgresql.org/docs/current/plpgsql.html. [Accessed : June 23, 2023].

Valda Team (Accessed 2023). *Présentation - Valda*. Website. URL : https://team.inria.fr/valda/fr/presentation/.

World Wide Web Consortium (2004). *XML Schema Part 1: Structures (Second Edition)*. https://www.w3.org/TR/xmlschema-1/. Accessed on 07/06/2023.

— (2017). *XQuery 3.1: An XML Query Language*. https://www.w3.org/TR/2017/REC-xquery-31-20170321/. Accessed on juin 08, 2023].

Résumé

Dans un contexte où l'utilisation croissante des données pose des défis majeurs en termes de fiabilité et de traçabilité, la gestion de la provenance des données est devenue cruciale. La provenance des données fait référence à l'historique et aux informations associées à leur collecte, leur transformation et leur stockage, ce qui permet d'évaluer leur validité et de garantir leur intégrité.

Ce projet se concentre sur l'intégration de la gestion de la provenance des données dans les bases de données temporelles. Les bases de données temporelles sont spécialement conçues pour gérer des données évoluant dans le temps, offrant ainsi la possibilité de contextualiser la validité des informations collectées et stockées.

Notre contribution consiste en une solution innovante reposant sur l'utilisation de semi-anneaux pour calculer les temps de validité des données dans les bases de données temporelles. Cette solution sera intégrée à ProvSQL, une extension dédiée à la gestion de la provenance et de la probabilité dans PostgreSQL. L'objectif principal est d'améliorer la compréhension de l'historique associé aux données temporelles, assurant ainsi leur traçabilité et leur fiabilité.

En outre, nous mettons en place une démarche d'optimisation et d'évaluation des performances au sein de ProvSQL. Nous cherchons à réduire le temps de calcul de la provenance et à améliorer l'efficacité globale du système.

En résumé, ce projet propose une intégration de la gestion de la provenance des données dans les bases de données temporelles en étendant ProvSQL avec une solution basée sur les semi-anneaux. Cette approche renforce la traçabilité et la fiabilité des données tout en optimisant les performances de ProvSQL.

Mots clés : Prevenance de données, Bases de données temporelles, temps de validité, semi-anneau.
