DBKDA 2026

The Eighteenth International Conference on Advances in Databases, Knowledge, and Data Applications

ISBN: 978-1-68558-359-0

March 8th –12th, 2026

Valencia, Spain

DBKDA 2026 Editors

Andreas Schmidt, Institute for Automation and Applied Informatics, Karlsruhe Institute of Technology, Karlsruhe, Germany

DBKDA 2026

Foreword

The Eighteenth International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA 2026), held between March 8 - 12, 2026, continued a series of international events covering a large spectrum of topics related to advances in fundamentals on databases, evolution of relation between databases and other domains, data base technologies and content processing, as well as specifics in applications domains databases.

Advances in different technologies and domains related to databases triggered substantial improvements for content processing, information indexing, and data, process and knowledge mining. The push came from Web services, artificial intelligence, and agent technologies, as well as from the generalization of the XML adoption.

High-speed communications and computations, large storage capacities, and load-balancing for distributed databases access allow new approaches for content processing with incomplete patterns, advanced ranking algorithms and advanced indexing methods.

Evolution on e-business, ehealth and telemedicine, bioinformatics, finance and marketing, geographical positioning systems put pressure on database communities to push the 'de facto' methods to support new requirements in terms of scalability, privacy, performance, indexing, and heterogeneity of both content and technology.

We take here the opportunity to warmly thank all the members of the DBKDA 2026 Technical Program Committee, as well as the numerous reviewers. The creation of such a high quality conference program would not have been possible without their involvement. We also kindly thank all the authors who dedicated much of their time and efforts to contribute to DBKDA 2026. We truly believe that, thanks to all these efforts, the final conference program consisted of top quality contributions.

Also, this event could not have been a reality without the support of many individuals, organizations, and sponsors. We are grateful to the members of the DBKDA 2026 organizing committee for their help in handling the logistics and for their work to make this professional meeting a success.

We hope that DBKDA 2026 was a successful international forum for the exchange of ideas and results between academia and industry and for the promotion of progress in the fields of databases, knowledge and data applications.

We are convinced that the participants found the event useful and communications very open. We also hope that Valencia provided a pleasant environment during the conference and everyone saved some time for exploring this beautiful city

DBKDA 2026 Chairs:

DBKDA 2026 Steering Committee
Fritz Laux, Reutlingen University, Germany
Andreas Schmidt, Karlsruhe Institute of Technology / University of Applied Sciences
Erik Hoel, Esri, USA
Peter Kieseberg, St. Pölten University of Applied Sciences, Austria
Constantine Kotropoulos, Aristotle University of Thessaloniki, Greece

DBKDA 2026 Publicity Chairs
Francisco Javier Díaz Blasco, Universitat Politècnica de València, Spain

Ali Ahmad, Universitat Politècnica de València, Spain
Sandra Viciano Tudela, Universitat Politecnica de Valencia, Spain
José Miguel Jiménez, Universitat Politecnica de Valencia, Spain

DBKDA 2026

Committee

DBKDA 2026 Steering Committee
Fritz Laux, Reutlingen University, Germany
Andreas Schmidt, Karlsruhe Institute of Technology / University of Applied Sciences
Erik Hoel, Esri, USA
Peter Kieseberg, St. Pölten University of Applied Sciences, Austria
Constantine Kotropoulos, Aristotle University of Thessaloniki, Greece

DBKDA 2026 Publicity Chairs
Francisco Javier Díaz Blasco, Universitat Politècnica de València, Spain
Ali Ahmad, Universitat Politècnica de València, Spain
Sandra Viciano Tudela, Universitat Politecnica de Valencia, Spain
José Miguel Jiménez, Universitat Politecnica de Valencia, Spain

DBKDA 2026 Technical Program Committee

Taher Omran Ahmed, University of Technology and Applied Sciences, Ibri, Oman / Alzintan University, Libya
Paritosh Aggarwal, Snowflake, USA
Julien Aligon, Institut de Recherche en Informatique de Toulouse (IRIT) | Université Toulouse 1 Capitole, France
Alaa Alomoush, University Malaysia Pahang, Malaysia
Emmanuel Andres, Hôpitaux Universitaires de Strasbourg, France
Vincenzo Arceri, Università degli Studi di Parma, Italy
Zeyar Aung, Masdar Institute of Science and Technology, UAE
Qiushi Bai, Microsoft, USA
Aruna Bansal, Thomson Reuters International Services, India
Nelly Barret, INSA Lyon & LIRIS lab, France
Christian Beecks, University of Hagen, Germany
Giacomo Bergami, Newcastle University, UK
Flavio Bertini, University of Parma, Italy
Vincenzo Bonnici, University of Parma, Italy
Savong Bou, University of Tsukuba, Japan
Ali Boukehila, University of Annaba, Algeria
Zouhaier Brahmia, University of Sfax, Tunisia
Martine Cadot, LORIA, Nancy, France
Alessandro Castelnovo, Intesa Sanpaolo S.P.A / University of Milano Bicocca, Italy
Basabi Chakraborty, Iwate Prefectural University, Japan / Madanapalle Institute of Technology and Science, India
Sanjay Chaudhary, Ahmedabad University, India
Yung Chang Chi, National Cheng Kung University, Taiwan
Richard Chbeir, Université de Pau et des Pays de l'Adour (UPPA), France
Jong Choi, Oak Ridge National Laboratory, USA

Table of Contents

Scribe Verification in Chinese Manuscripts Using Siamese, Triplet, and Vision Transformer Neural Networks 1
Dimitrios-Chrysovalantis Liakopoulos, Yanbo Zhang, Chongsheng Zhang, and Constantine Kotropoulos

Property Graph Techniques in Relational Databases 7
Malcolm Crowe and Fritz Laux

QuaQue: Design and SQL Implementation of Condensed Algebra for Concurrent Versioning of Knowledge Graphs 13
Jey Puget Gil, Emmanuel Coquery, John Samuel, and Gilles Gesquiere

On UNIQUE KEYS in SQL/JSON Implementations - Experimenting with JSON Support in DBMS Products 24
Martti Laiho and Fritz Laux

Scribe Verification in Chinese manuscripts using Siamese, Triplet, and Vision Transformer Neural Networks

Dimitrios-Chrysovalantis Liakopoulos*,Yanbo Zhang†, Chongsheng Zhang† and Constantine Kotropoulos*
* *Department of Informatics, Aristotle University of Thessaloniki*
Thessaloniki 54124, Greece
email: {dliako, costas}@csd.auth.gr
† *School of Computer and Information Engineering, Henan University*
Kaifeng, China
email: {zhangyanbo, cszhang}@henu.edu.cn

***Abstract*—The paper examines deep learning models for scribe verification in Chinese manuscripts. That is, to automatically determine whether two manuscript fragments were written by the same scribe using deep metric learning methods. Two datasets were used: the Tsinghua Bamboo Slips Dataset and a selected subset of the Multi-Attribute Chinese Calligraphy Dataset, focusing on the calligraphers with a large number of samples. Siamese and Triplet neural network architectures are implemented, including convolutional and Transformer-based models. The experimental results show that the MobileNetV3+ Custom Siamese model trained with contrastive loss achieves either the best or the second-best overall accuracy and area under the Receiver Operating Characteristic Curve on both datasets.**

***Keywords*-Deep learning models; Siamese Networks; Triplet networks; Vision Transformers.**

I. Introduction

Ancient script images exhibit unique characteristics that pose significant challenges to conventional recognition methods. These challenges stem from limited sample availability and image noise arising from the historical and environmental conditions under which ancient scripts were created and preserved. Many deep learning- and computer vision-based methods for ancient script recognition have emerged, aiming to perform classification, recognition, and interpretation at larger scales and achieve higher accuracy [1].

A modified Convolutional Neural Network (CNN) with an attention mechanism for historical Chinese character recognition is described in [2]. A batch normalization layer is incorporated into a modified GoogLeNet architecture, and a feature-based attention mechanism is integrated into the network. Feature vectors are normalized to unit ℓ_2 norm before the softmax layer, thereby improving generalization.

A CNN is also used to encode a text line image. At the same time, a Bidirectional Long Short-Term Memory network followed by a fully connected neural network serves as the decoder for predicting a sequence of characters in ancient Greek manuscripts [3]. A transformer-based character recognition model incorporating multi-scale attention and Multilayer Perceptron modules is tested on Tsinghua bamboo slips [4]. The motivation was to replace the traditional convolution to improve recognition accuracy.

A comparative analysis between various structural and statistical features and classifiers against deep neural networks is performed in [5]. The incorporation of the Discrete Cosine Transform with the Vision Transformer (ViT) is shown to achieve the top F_1 score.

Scribe verification focuses on determining whether two manuscript fragments were written by the same individual. In the field of digital humanities, this task plays a significant role in the preservation, classification, and authentication of ancient texts, enabling researchers to group manuscript fragments by writing style and origin.

The main objective of the paper is to develop a system capable of automatically determining whether two manuscript images were produced by the same scribe. To this end, we explore a series of neural architectures that learn discriminative embeddings of handwriting images, enabling the measurement of similarity between fragments using distance metrics. The experimental setup involves two datasets: the Tsinghua Bamboo Slips Dataset [6], which consists of digitized images of ancient Chinese manuscripts, and a curated subset of the Multi-Attribute Chinese Calligraphy Dataset (MCCD) [7], which contains samples from different calligraphers and provides abundant image data. These datasets offer a diverse range of handwriting styles, providing a foundation for training and evaluating models.

Several neural architectures have been implemented and compared, including Siamese networks trained with contrastive loss, Triplet networks optimized with triplet loss [8], and ViT-based models [9] for capturing both local and global handwriting features. Modern CNN backbones such as Visual Geometry Group (VGG19) [10], Residual Network (ResNet) [11], and MobileNetV3 [12], as well as recent ViT [9], have been applied to handwriting analysis.

Overall, the paper establishes a unified PyTorch-based training and evaluation framework for scribe verification that can handle both ancient and modern datasets. It adopts a dynamic pair- and triplet-sampling mechanism, implements consistent preprocessing and normalization pipelines, and standardizes evaluation using fixed-pair generation and Receiver Operating Characteristic (ROC)-based analysis. The paper's main contribution is a unified comparison of Siamese, Triplet, and ViT models for scribe verification across Chinese handwriting datasets. Unlike prior studies that typically focus on a single architecture or dataset, this work presents a unified and

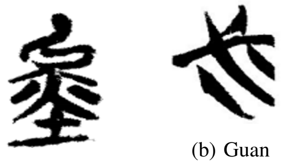


Figure 1. Representative examples from the Tsinghua Bamboo Slips dataset.

reproducible benchmarking framework for scribe verification across both ancient and modern Chinese handwriting datasets. While the individual model components and loss functions are well established, their systematic evaluation under identical preprocessing, dynamic sampling, and ROC-based evaluation protocols provides practical insights into architectural trade-offs for metric learning-based scribe verification.

The remainder of this paper is organized as follows. Section II describes the datasets, model architectures, and training methodology. Section III presents the experimental results and comparative evaluation. Section IV concludes the paper and outlines directions for future research.

II. Methodology

A. Datasets

The Tsinghua Bamboo Slips dataset [6] consists of digitized images of ancient Chinese manuscripts discovered in 2008 at Tsinghua University. Bamboo slips were an early writing medium in ancient China, preceding the widespread adoption of paper. Text was written vertically on narrow bamboo strips using a brush and ink, and manuscripts were formed by binding multiple slips together. Individual scribes exhibit characteristic stroke patterns, spacing, and ink usage, making scribe verification a valuable task for manuscript reconstruction, authentication, and historical analysis. Each fragment corresponds to a section of bamboo slip text written in classical Chinese, attributed to specific scribes. Two such examples are depicted in Figure 1. The dataset serves as a historically significant benchmark for studying scribe handwriting patterns and presents challenges, including faded ink, uneven texture, and varying stroke thickness. These characteristics make it an ideal testbed for evaluating neural networks' ability to extract robust visual features from degraded manuscript data. The distribution of images per scribe is listed in Table I.

TABLE I. DISTRIBUTION OF IMAGES PER SCRIBE IN THE TSINGHUA BAMBOO SLIPS DATASET.

Scribe Class	Training Images	Test Images
Bao Xun	164	40
Bie Gua	32	8
Cheng Wu	214	53
Chu Ju	445	111
Feng Xu Zhi Ming	161	40
Guan Zhong	1508	377
Huang Men	4282	1071
Liang Chen	285	71
Ming Xun	422	107
Shi Fa	1670	420
Yin Zhi	3412	852
Total	**12595**	**3050**

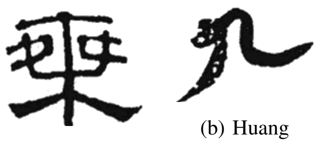


Figure 2. Representative examples from the MCCD dataset.

The MCCD [7] is a large-scale modern collection of handwritten Chinese character images created by multiple professional calligraphers. Two representative examples are shown in Figure 2. To adapt this dataset for the scribe verification task, a curated subset was created by selecting only the Calligraphers folder and including the scribes with the largest number of samples. This selection ensures a more balanced, representative distribution of handwriting styles, thereby avoiding extreme class imbalance. Each subfolder corresponds to a single calligrapher, and the images were used to form positive and negative training examples for Siamese and Triplet learning. The distribution of images per scribe is listed in Table II.

TABLE II. DISTRIBUTION OF IMAGES PER SCRIBE IN THE MCCD SUBSET DATASET.

Scribe Class	Training Images	Test Images
He Shaoji	1048	448
Shan Xiaotian	1114	447
Wu Rui	991	424
Wu Dacheng	1242	532
Mi Fu	1526	654
Deng Shiru	923	395
Yan Zhenqing	1655	708
Huang Tingjian	1353	579
Total	**9852**	**4187**

B. Model Architectures

Each model is designed to learn an embedding representation of handwriting fragments such that samples from the same scribe are mapped close together in the embedding space. In contrast, samples from different scribes are widely separated. Three main architectural families are explored: convolutional Siamese networks, a Triplet network variant, and a ViT Siamese model.

1) Siamese CNN-based Models: The first set of experiments employed convolutional Siamese architectures trained with the contrastive loss function. A Siamese network consists of two identical branches that share weights and process two input images in parallel. Each branch extracts features from an input image and outputs a low-dimensional embedding vector. The distance between these embeddings, typically measured using the Euclidean distance, is minimized for samples belonging to the same scribe and maximized for samples from different scribes.

Several pretrained convolutional backbones were used as feature extractors, including MobileNetV3 [12], MobileNetV3+ Custom [6], VGG19 [10], and ResNet18/34/50

[11]. The final layers of these networks were modified to produce a 10-dimensional embedding vector, consistent with the dimensionality used in the paper on scribe verification [6]. These embeddings were learned by fine-tuning pretrained ImageNet weights, enabling the models to adapt to the specific texture and style features of handwritten Chinese characters.

Among the CNN-based Siamese models, the MobileNetV3+ Custom architecture achieved the best performance on the Tsinghua Bamboo Slips dataset when trained with the contrastive loss. Its lightweight structure, combined with efficient depthwise separable convolutions, allowed it to learn discriminative handwriting representations without overfitting, even with a limited number of samples per scribe.

2) Triplet Network: The second model was based on the Triplet learning paradigm [8], a metric-learning approach designed to learn embeddings in which examples from the same class lie closer together than examples from different classes. Unlike Siamese networks, which operate on image pairs, triplet networks process three inputs simultaneously: an anchor sample, a positive sample (from the same scribe as the anchor), and a negative sample (from a different scribe).

Triplet learning is beneficial for problems where the goal is not classification but verification or similarity comparison, such as writer identification. By explicitly modeling relative similarity rather than absolute labels, the network learns fine-grained variations within each scribe's writing style while maximizing separation between different scribes.

The Triplet architecture used in the paper was derived from the MobileNetV3+ Custom backbone. The shared embedding network outputs a 10-dimensional feature vector for each input, and the loss is computed according to the triplet constraint. Although this approach improves the separation among different scribe classes, it requires significantly more computational time and careful sampling of informative triplets.

3) ViT Siamese Model: To investigate the effectiveness of self-attention mechanisms in scribe verification, a ViT [9] was implemented within the Siamese framework. Unlike convolutional networks, which operate on local receptive fields, ViT processes images as sequences of non-overlapping patches and captures global contextual relationships through multi-head self-attention layers. This property enables the model to understand better the overall structure and spatial relationships in handwritten text.

The ViT-B/16 architecture, pretrained on ImageNet, served as the backbone, with its final classification head replaced by a fully connected layer projecting to a 10-dimensional embedding space. Both branches of the ViT Siamese model share weights, and training is performed using the same contrastive loss formulation as in the CNN-based Siamese models.

C. Loss Functions

Deep metric learning aims to project input images into an embedding space where distances correspond to semantic similarity. Two main loss functions were tested to guide the learning process: the Contrastive Loss and the Triplet Loss. Both are designed to minimize the distance between embeddings of samples from the same scribe while maximizing the distance between samples from different scribes.

1) Contrastive Loss: The contrastive loss function [6] was used to train the Siamese and Vision Transformer models. Given a pair of images (x_1, x_2) with label $y \in \{0,1\}$, where $y = 1$ indicates that both images belong to the same scribe and $y = 0$ indicates different scribes, the Euclidean distance between their embeddings is defined as $D(x_1, x_2) = \|f(x_1) - f(x_2)\|_2$. The loss is given by:

$$L_{contrastive}(x_1, x_2) = \frac{1}{2}\, y\, D^2(x_1, x_2) + \frac{1}{2}(1-y) \cdot \\ \cdot \max\left(0, m - D(x_1, x_2)\right)^2, \quad (1)$$

where m is a predefined margin that controls the separation between positive and negative pairs. The first term in (1) penalizes large distances for positive pairs (same scribe), while the second term penalizes small distances for negative pairs that lie within the margin. This formulation encourages embeddings of the same scribe to cluster tightly together while maintaining a minimum distance between different scribes.

2) Triplet Loss: The triplet loss was applied to the Triplet Network [8], which processes three images simultaneously: an anchor x_a, a positive x_p (same scribe), and a negative x_n (different scribe). The goal is to enforce that the distance between the anchor and the positive sample is smaller than the distance between the anchor and the negative sample by at least a margin m. Formally, the loss is expressed as:

$$L_{triplet}(x_a, x_p, x_n) = \max\left(0, D(x_a, x_p) - D(x_a, x_n) + m\right), \quad (2)$$

where $D(x_a, x_p)$ and $D(x_a, x_n)$ denote the Euclidean distances between the corresponding embeddings. The objective (2) helps the model to create a structured embedding space in which clusters of samples belonging to the same scribe are well separated from those of different scribes.

D. Data Preprocessing and Sampling Strategy

All manuscript images underwent standardized preprocessing and sampling to ensure consistency across datasets and to maximize intra-class variability during training. Each image was first converted to a three-channel RGB format, regardless of its original grayscale input, to maintain compatibility with pretrained ImageNet backbones. All images were then resized to 224×224 pixels, following the configuration used in the Tsinghua Bamboo Slip experiments in [6]. Normalization was applied using the ImageNet mean $\mu = (0.485, 0.456, 0.406)$ and standard deviation $\sigma = (0.229, 0.224, 0.225)$. These preprocessing steps were implemented within the `PyTorch` transformation pipeline to ensure consistent input scaling across all architectures.

To improve generalization, several lightweight augmentations were applied randomly during training: 1) Horizontal flipping to simulate mirrored character structures; Random grayscale inversion (gray-flip) for minority classes, following the data balancing strategy described in the Tsinghua framework; 3) Random brightness and contrast adjustments within a small range. This combination increased variability in stroke density and ink appearance, which is important for robust learning of scribe representations.

Training samples were generated dynamically at runtime, such as Positive pairs: two images from the same scribe or calligrapher folder; Negative pairs: two images from different scribes; and Triplets: an anchor, a positive from the same scribe, and a negative from a different scribe. Positive and negative pairs were balanced using a $1:1$ ratio to prevent bias toward a particular class type. For triplet training, each batch contained randomly sampled anchor–positive–negative combinations generated on the fly. This dynamic sampling approach eliminates the need for storing predefined pairs and increases the diversity of training examples.

During dataset loading, corrupted or unreadable image files were automatically detected and replaced with a resampled, valid image from the same class. If repeated failures occurred, a blank placeholder image was substituted to maintain batch consistency and prevent runtime interruptions. In practice, image loading failures were rare (occurring in fewer than 0.5% of samples) and were primarily caused by corrupted files. The placeholder mechanism was therefore included as a safeguard to prevent training interruptions rather than as a frequently used operation.

For evaluation, fixed image pairs were generated using a separate script, yielding balanced sets of positive and negative examples. Each pair was labeled as “same” or “different” depending on whether both images originated from the same scribe. This ensured reproducible and unbiased comparisons across different architectures and training configurations.

E. Training Setup

All models were implemented and trained using the `PyTorch` deep learning framework. The experiments were conducted in a controlled environment to ensure reproducibility and consistency across architectures and datasets.

Training was performed exclusively on an NVIDIA GeForce RTX 4060 Laptop GPU with 8 GB of VRAM, supported by an Intel Core i7-13620H processor and 32 GB of system memory. All computations were executed on the GPU using CUDA acceleration, significantly reducing training time and enabling parallelized batch processing for both Siamese and Triplet configurations.

The experiments were executed within a dedicated `Python` virtual environment (`.venvsiamese`) using `Python 3.10` and `PyTorch 2.1.0` with `TorchVision 0.16.0`. Supporting libraries included `NumPy`, `Pillow`, `tqdm`, and `PyYAML` for data handling, progress monitoring, and configuration management. All training parameters were stored in `YAML` configuration files, which defined dataset paths, model settings, and optimizer hyperparameters.

All models were trained using the Adam optimizer with an initial learning rate of 1×10^{-3} and no weight decay. A batch size of 32 was employed, and each training session ran for 30 epochs. A constant learning rate was maintained throughout, as this setting yielded stable convergence across architectures without requiring a scheduler.

The margin hyperparameter for the contrastive loss was fixed at $m = 0.6$, following the configuration of the Tsinghua Bamboo Slip framework. For the triplet loss, the margin was set to $m = 1.0$. All models produced a 10-dimensional embedding vector at the output layer, consistent with the reference design.

At each iteration, positive and negative pairs (or triplets) were generated dynamically to ensure exposure to new sample combinations. The Euclidean distance between embeddings was computed, and the appropriate loss function—contrastive or triplet—was applied. Model weights were updated via backpropagation, and checkpoints were saved after every epoch in the `checkpoints directory` using the format `model_e{epoch}.pt`. CUDA computation enabled efficient mini-batch parallelization and faster convergence across all model variants.

A fixed random seed (42) was applied globally to dataset loading, pair sampling, and model initialization to guarantee reproducibility of training outcomes.

While the main evaluation was conducted after full training, intermediate validation losses were monitored to verify stable convergence and to detect potential overfitting. The best-performing model checkpoint, based on validation stability, was selected for the final evaluation on the fixed test-pair set. The code implementing the scribe verification methods is available in [13]. Neither the Tsinghua Bamboo Slips Dataset nor the MCCD dataset is publicly available. Still, they can be accessed upon reasonable request for research purposes [6], [7]. Only publicly released or author-approved samples were used in the paper, and no restricted or unpublished manuscript materials were distributed.

F. Evaluation Procedure

After training, each model was evaluated on a fixed test set composed of predefined image pairs. These pairs were generated using a dedicated Python script (`make_test_pairs.py`) that created a balanced dataset of positive (same scribe) and negative (different scribe) examples. The resulting list was stored in a CSV file named `test_pairs.csv`, containing three columns: the file paths of the two images (`path1` and `path2`) and the binary label (`label`), where 1 denotes a same-scribe pair and 0 denotes a different-scribe pair. The `CSV` file ensures that all models are evaluated on the same test samples and labels, thereby enabling reproducible, unbiased comparisons across architectures.

During evaluation, each input image was passed through the trained network to obtain its corresponding embedding vector $f(x) \in \mathbb{R}^{10}$. For Siamese and Vision Transformer models, embeddings were generated for both images in each pair, and the similarity between them was computed using the Euclidean distance. Smaller distances indicate higher similarity between the two fragments, suggesting that the same scribe likely wrote them.

A decision threshold τ was applied to classify whether two fragments x_1 and x_2 were written by the same or different scribes:

$$\text{prediction} = \begin{cases} \text{same scribe}, & D(x_1, x_2) < \tau, \\ \text{different scribes}, & D(x_1, x_2) \geq \tau. \end{cases} \quad (3)$$

The optimal threshold was determined by scanning possible values and selecting the one that maximized overall classification accuracy. This approach aligns with the evaluation methodology presented in the Tsinghua Bamboo Slip framework [6].

To assess the discriminative power of the learned embeddings, several quantitative metrics were computed: 1) *ROC curve*, showing the trade-off between true and false acceptance rates. In the context of scribe verification, the ROC curve provides insight into how well the model separates same-scribe pairs from different-scribe pairs across all possible decision thresholds. 2) *Area Under the ROC Curve (AUC)*, summarizing ROC performance by a single scalar value. The AUC offers a threshold-independent measure of discriminative ability. 3) *Accuracy (ACC)*, representing the proportion of correctly classified pairs. 4) *False Acceptance Rate (FAR)*, quantifying the ratio of negative pairs incorrectly predicted as positive ones (i.e., same). 5) *False Rejection Rate (FRR) or False Negative Rate*, quantifying the ratio of positive pairs incorrectly predicted as negative (i.e., different). These metrics were computed using `NumPy` and `scikit-learn` functions implemented in the evaluation pipeline. The ROC curves were plotted using `Matplotlib` and saved as visual summaries for model comparison.

The evaluation script (`evaluate.py` or `evaluate_triplet.py`) automatically loads the trained checkpoint from `checkpoints directory`, reads the `test_pairs.csv` file, extracts embeddings for each pair, computes the Euclidean distances, and evaluates the metrics described above. The results are printed to the console and visualized as ROC curves, while detailed statistics, including AUC, accuracy, and the optimal decision threshold, are recorded. This standardized evaluation procedure ensures that all models are compared fairly and under identical test conditions.

III. Experiments and Results

Experiments were conducted on both the Tsinghua Bamboo Slips dataset and the MCCD subset, following the methodology described in Section II.

Table III summarizes the quantitative performance of all trained architectures on the Tsinghua and MCCD datasets. The MobileNetV3+ Custom Siamese model trained with contrastive loss achieved the best overall results, obtaining an AUC of 0.958 and an accuracy of 89.19% in the Tsinghua Bamboo Slips Dataset. In contrast, the ResNet34 model trained with contrastive loss achieved the best performance on the MCCD Dataset, achieving an AUC of 0.952 and an accuracy of 89.06%. Unlike the standard MobileNetV3, which ends with a large classification head, the MobileNetV3 Custom version replaces this component with a reduced 160-channel bottleneck and a final 10-dimensional embedding layer, making it better suited for distance-based metric learning in scribe verification.

TABLE III. QUANTITATIVE PERFORMANCE COMPARISON OF ALL TRAINED ARCHITECTURES ON THE EVALUATED DATASETS.

Model	**Dataset**	**AUC**	**ACC** (%)	**FAR** (%)	**FRR** (%)
MobileNetV3 (Siamese)	Tsinghua	0.897	82.27	13.28	22.19
MobileNetV3+ Custom (Siamese)	Tsinghua	**0.958**	**89.19**	**5.63**	**15.99**
ResNet18 (Siamese)	Tsinghua	0.921	84.65	16.56	14.15
ResNet34 (Siamese)	Tsinghua	0.912	82.56	19.5	15.37
ResNet50 (Siamese)	Tsinghua	0.884	80.4	21.5	17.69
VGG19 (Siamese)	Tsinghua	0.857	82.07	14.95	20.91
ViT-B/16 (Siamese)	Tsinghua	0.829	77	25.59	20.41
MobileNetV3+ Custom (Triplet)	Tsinghua	0.945	88.16	13.59	10.15
MobileNetV3 (Siamese)	MCCD	0.949	88.99	11.44	10.59
MobileNetV3+ Custom (Siamese)	MCCD	0.949	88.5	12.77	10.23
ResNet18 (Siamese)	MCCD	0.948	88.24	13.93	9.58
ResNet34 (Siamese)	**MCCD**	**0.952**	**89.06**	**14.08**	**7.79**
ResNet50 (Siamese)	MCCD	0.931	86.93	17.91	8.23
VGG19 (Siamese)	MCCD	0.9	82.93	25.05	9.10
ViT-B/16 (Siamese)	MCCD	0.89	82.06	24.41	11.47
MobileNetV3+ Custom (Triplet)	MCCD	0.931	87.63	16.49	8.66

Figures 3 and 4 depict the ROC curves of representative models.

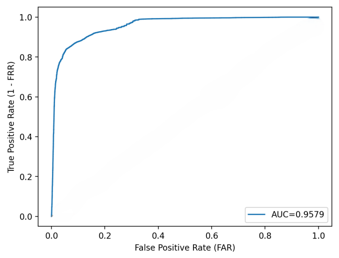


Figure 3. ROC curve of the MobileNetV3+ Custom Siamese model on the Tsinghua Bamboo Slips dataset.

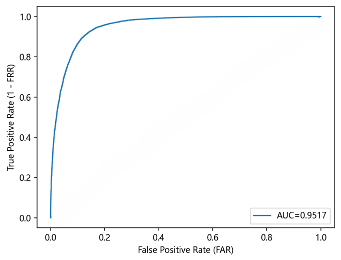


Figure 4. ROC curve of the ResNet34 Custom Siamese model on the MCCD dataset.

The experimental results demonstrate that deep metric approaches are effective for scribe verification across both ancient and modern Chinese handwriting datasets. However, the optimal network configuration varies depending on the characteristics and quality of the dataset. Overall, convolutional Siamese architectures produced the most stable and discriminative embeddings, while Triplet and Transformer-based variants provided additional insights but did not surpass the baseline models.

On the Tsinghua dataset, the MobileNetV3+ Custom achieved the highest overall performance among all tested architectures, as shown in Table III. This success can be attributed to its ability to balance depth and generalization, effectively capturing mid-level texture features crucial for distinguishing writing patterns on aged bamboo fragments. That is, MobileNetV3+ Custom exhibits a strong inductive bias toward local texture and stroke-level features, which are critical in degraded manuscript images. The model's moderate complexity enabled it to handle noise, surface irregularities, and faded ink without overfitting, a challenge observed in deeper networks such as ResNet-50 and VGG-19. Medium-depth convolutional architectures balance representational capacity and generalization, enabling them to capture discriminative handwriting patterns without overfitting to noise. In contrast, deeper CNNs and transformer-based models may be more sensitive to background artifacts and require substantially larger datasets to learn robust representations.

On the modern MCCD dataset, the ResNet34 Siamese model achieved the best results, achieving the highest AUC and accuracy among all tested configurations. The ResNet 34 Siamese model also performed exceptionally well on the Tsinghua Bamboo Slips dataset. Its structure and the efficient feature extraction were advantageous for the cleaner, high-resolution images of modern calligraphers. The strong performance of MobileNetV3+ on the MCCD dataset suggests that efficient CNNs may be sufficient for capturing stylistic differences in consistent, well-preserved handwriting. Future work will include evaluating the ResNet18 backbone on the MCCD dataset to validate this hypothesis.

The Triplet Network based on the MobileNetV3+ Custom backbone did not outperform its Siamese counterpart. Despite theoretically promoting a more structured embedding space, the triplet configuration required considerably longer training and yielded slightly lower accuracy and AUC. This indicates that, for the datasets and sampling strategies used in this study, the contrastive loss function yielded more stable optimization and better generalization.

Direct comparison with previously published results is challenging due to differences in dataset splits, pair generation strategies, and the limited public availability of the evaluated datasets. Nevertheless, the observed performance trends are consistent with prior findings reported in [6], which showed that medium-depth convolutional Siamese networks outperform deeper CNN architectures on degraded ancient manuscript images.

IV. Conclusion and Future Work

The contrastive Siamese networks have been highly effective for this task, achieving robust performance across both datasets. The MobileNetV3+ Custom Siamese model has achieved the top performance on the Tsinghua Bamboo Slips dataset despite challenging image degradation. On the modern MCCD dataset, the ResNet34 Siamese model also achieved the highest overall accuracy and AUC, confirming the suitability of lightweight architectures for high-quality calligraphy data.

Future research will focus on several directions: 1) Exploring hybrid CNN–Transformer architectures that combine efficient local feature extraction with global attention mechanisms; 2) Incorporating hard positive and hard negative mining strategies to improve triplet-based training efficiency; 3) Expanding the dataset with additional manuscript sources to enable large-scale pretraining and improve generalization; 4) Investigating few-shot and one-shot learning approaches to scribe verification with minimal labeled data; and 5) Including multi-seed experiments to quantify performance variability and further assess the robustness of metric learning methods that are sensitive to initialization and sampling strategies.

Acknowledgment

We want to thank Professor Lianwen Jin from the School of Electronic and Information Engineering at South China University of Technology, Guangzhou, China, for sharing the MCCD dataset with us.

References

[1] D. Xiaolei *et al.*, "Ancient script image recognition and processing: A review," 2025, [retrieved: January 31, 2026]. [Online]. Available: https://arxiv.org/abs/2506.19208

[2] H. Qin and L. Peng, "Convolutional neural network with attention mechanism for historical chinese character recognition," in *Proceedings of the 4th International Workshop on Historical Document Imaging and Processing*. Kyoto, Japan: ACM, 2017, pp. 42–47.

[3] K. Markou *et al.*, "A convolutional recurrent neural network for the handwritten text recognition of historical greek manuscripts," in *Pattern Recognition. ICPR International Workshops and Challenges: Virtual Event, January 10-15, 2021, Proceedings, Part VII*. Berlin, Heidelberg: Springer-Verlag, 2021, pp. 249—262.

[4] L. Wu, C. Zhang, M. Xu, and M. Wu, "Ancient chinese recognition method based on attention mechanism," in *Proceedings of the 7th IEEE International Conference on Network Intelligence and Digital Content (IC-NIDC)*, Beijing, China, 2021, pp. 309–313.

[5] R. Sivan, P. B. Pati, and M. W. A. Kesiman, "Image quality determination of palm leaf heritage documents using integrated discrete cosine transform features with vision transformer," *International Journal on Document Analysis and Recognition*, vol. 28, no. 1, pp. 41—57, July 2025.

[6] H. Wang *et al.*, "Tsinghua bamboo slip scribe verification using Siamese networks," *Heritage Science*, 2025.

[7] Y. Zhao, Y. Zhang, and L. Jin, "MCCD: A multi-attribute Chinese calligraphy character dataset annotated with script styles, dynasties, and calligraphers," 2025, [retrieved: January 31, 2026]. [Online]. Available: https://arxiv.org/abs/2507.06948

[8] E. Hoffer and N. Ailon, "Deep metric learning using triplet network," 2014, [retrieved: January 31, 2026]. [Online]. Available: https://arxiv.org/abs/1412.6622

[9] A. Dosovitskiy *et al.*, "An image is worth 16×16 words: Transformers for image recognition at scale," 2020, [retrieved: January 31, 2026]. [Online]. Available: https://arxiv.org/abs/2010.11929

[10] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, [retrieved: January 31, 2026]. [Online]. Available: https://arxiv.org/abs/1409.1556

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Las Vegas, NV, USA, 2016, pp. 770–778.

[12] A. Howard *et al.*, "Searching for MobileNetV3," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, Seoul, South Korea, 2019, pp. 1314–1324.

[13] "Scribe Verification Using Siamese, Triplet, and Vision Transformer Networks," [retrieved: January 30, 2025]. [Online]. Available: https://github.com/dimiliakop/ScribeVerification.git

Property Graph Techniques in Relational Databases

Malcolm Crowe
University of the West of Scotland (retired)
Paisley, Scotland
e-mail: Malcolm.Crowe@uws.ac.uk

Fritz Laux
Reutlingen University (retired)
Reutlingen, Germany
e-mail: Friedrich.Laux@Reutlingen-University.de

Abstract— **Current developments in standardization activities for database management systems include the development of the next versions of the Structured Query Language (SQL), and related standards for labelled property graphs. This short paper explores mechanisms for unified implementation of record-based relational and property graph databases: its contribution is that modeling SQL foreign keys as reference keys facilitates the graph insert and advanced pattern matching mechanisms of Property Graph Database Management Systems (PGDMS) and allows the construction of multiple type graph models on a single relational database.**

Keywords- Property graphs; type graph model; linked data; data exchange.

I. INTRODUCTION

Use cases where queries traverse many links have been shown to be inefficient in relational Database Management Systems (DBMS), because every link requires construction of a join [1]. Examples where queries must follow many links between data items include the investigation of financial fraud [2], supply chain and logistics analysis [3], protein-protein interactions in the cell [4], and social networks [5]. Efficient computation in such cases requires the construction of a suitable graph model, selecting the record types to form the nodes of a property graph, and then defining edges between these nodes [6]. Then queries can collect and analyze information resulting from traversals or pattern matching in the graph model [7].

Many products are already available that implement property graph database technology: The Graph Query Language (GQL) standard [8] v.1 emerged in 2024 [9], and the next version is expected in 2027. Most of these products stand alone, but Oracle's Structured Query Language/Property Graph Query (SQL/PGQ) offers a graph view over a relational database.

In this paper, we consider only record-based DBMS, where all durable data begins as a record laid down at a certain time (a persisted entity or object instance): it has an identity in addition to its values/attributes, may be referred to and possibly updated, and it becomes inaccessible if deleted. Many commercial DBMS including Oracle and DB2 have this model in the physical database, but it is not usually exposed to application programs. This paper presents a mechanism for high-performance graph management and pattern matching algorithms to be provided as an extension of any such relational DBMS, using the standard SQL reference type as a starting point.

The key contribution of this paper is that in moving from one node in a graph pattern to the next during a match statement we follow a reference or its reverse; and during a graph insert process, we build node references. We will show that reference values can replace foreign key indexes in implementations of both relational and graph databases.

Section II briefly reviews the reference type concept from SQL as a way of implementing foreign keys. It is shown that for a record-based DBMS, all foreign keys can be implemented using reference keys.

In Section III, we show that it is very straightforward to extend an ordinary relational DBMS to perform the "insert graph" and "match" operations that are key to property graph technology.

Section IV considers how alterative graph models can be supported by a relational database.

Section V presents the conclusions of this research, and a reference to an open-source implementation of these ideas.

II. SQL AND REFERENCE VALUES

The SQL standard [9] offers few clues as to how the REF keyword should be implemented, and it is striking that the implementations in ADO.NET and Oracle are quite different. In ADO.NET we see that an entity reference has the form REF(expression) "where expression is any valid expression that yields an instance of an entity type". The entity reference consists of the entity key and an entity set name. In Oracle REF(correlation variable) "takes as its argument a correlation variable (table alias) associated with a row of an object table or view" and returns a REF value "for the object instance that is bound to the variable or row".

In both cases, it is easiest to think of reference values as belonging to a domain specific to an entity set or object table/view T. For convenience, let us call this domain REF T. It does not matter how the DBMS implements this, but it may be useful to think of it as a position in a real or conceptual log file. If the reference is a physical address a foreign key is replaced by an address (pointer) to the referenced record. This eliminates the need for a search (lookup) and leads directly to the referenced data record. It is easy to see that joins can benefit from such a substantial speed-up for large tables or multiple joins.

In the relational model, a foreign key in a relation R is a mapping from a set of one or more columns of R to the primary key of another table T. The primary key uniquely determines a record in T, and conversely if we know the

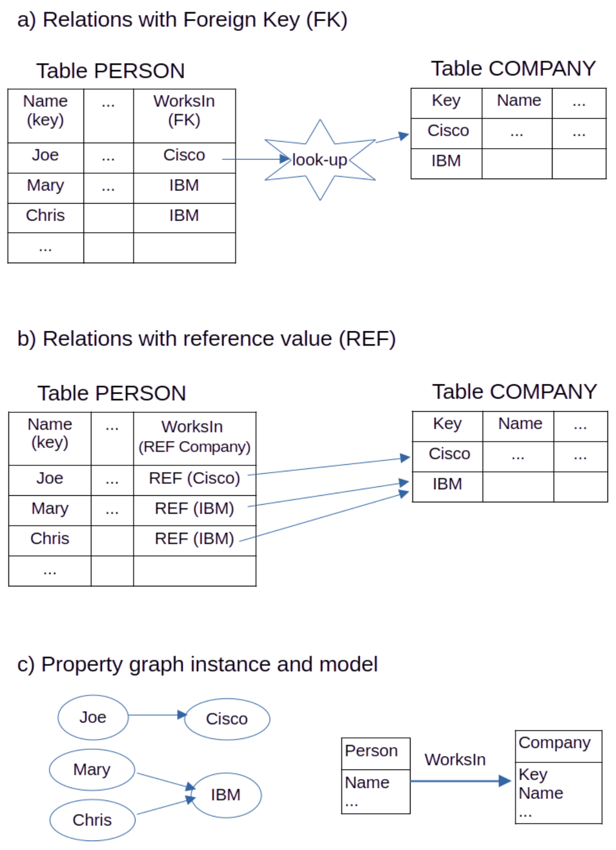


Figure 1. REF and foreign key values

reference to this record, we can read off the value of any of its columns, so that the foreign key can be equivalently represented as a mapping to REF T values. We can store this reference in the referencing table instead of storing the foreign key values themselves.

For example, if we have a table called `Person` whose primary key column is name, a foreign key column `Worksin` might identify a record in a `Company` table.

```
Person (Name primary key,  …, WorksIn fkey(cid))
Company (cid primary key, Business, Town, … )
```

The Person table is a relation with one foreign key. This is a many-to-one relationship. Many-to-many relationships such as `Likes` need an extra table to represent them in SQL.

Such foreign keys correspond in graph databases to directional edges linking nodes representing rows in the `Person` table to rows in the `Company` table. In this example nodes would represent persons and companies, and directed edges would implement the `WorksIn` relationship.

But the SQL version of this graph model would have a separate table for `WorksIn` (the lookup table), so that each edge identifies a person and a company:

```
WorksIn (sid fkey (Name), .., cid fkey (Company))
```

Generally, edge relations have two references as here. In the ANSI/SPARC design method for databases, it was always the case that alternative data models could be used with a single database, and we return to this point later.

In this paper, let us notate by T(n) a reference value to the record in table T identified by the expression n, or REF(n) if the relevant table is clear from the context: where n might be a primary key. Figure 1 illustrates the above example where part a) shows the known relational case, where the foreign key value `'Cisco'` is mapped to a row in `Company`, part b) uses REF values, so that the `WorksIn` column contains a physical pointer to (or row uid of) a row in `Company`, and part c) shows the corresponding property graph model, where each instance of the `WorksIn` relation connects a `Person` node to a `Company` node. In this way all foreign keys can be implemented by reference columns.

Reference columns can also be used to access tables that do not have a primary key. The reference value will be known at the time the referenced row is created and can be used in referring locations or provided by some use-case specific computation, such as MATCH.

Multi-column keys then naturally may be subject to constraints (e.g., “this group of one or more columns in A gives a unique reference value for table A”), and, if keys are used, a foreign key in table B referencing A will have a corresponding group of columns in B.

Many property graph DBMS allow edges to have properties, and in that case an implementation will probably have a table for each edge type to give the properties of each edge and two reference values for the source and destination of the edge. Many SQL tables have several foreign keys, while each edge has only two. Often it will make sense to select different pairs of keys to construct alternative edge models.

The direction of the edge is a property of the edge type rather than its rows (source column TO destination columns), so that if we want to specify this direction in the database, we need the DBMS to support a suitable metadata annotation.

Property graph systems generally use the terminology “node type”, “edge type” both when edges can have their own properties and when they do not. If the details of nodes and edges are saved in a relational database, we should observe that tables and user-defined types are different sorts of database objects in SQL. The values of user-defined types tend to be volatile, and all attributes are usually required to be present and non-null (as in POINT(x,y)). Persistent values of user-defined types can best be placed in a table, and for simplicity in this paper we assume this is happening. The keywords UNDER and ONLY in SQL suggest the possibility of subtype and supertype relationships, and user-defined types can be used in a sort of object-oriented programming.

III. Ease of Use

Immediate agreement on a common syntax unifying SQL and GQL along these lines is unlikely. Our research supports both, while adding metadata extensions in both SQL and GQL. Noting that the GQL standard does not support an automatic index for a property called ID, we can add SQL-like PRIMARY KEY metadata to element type specifications. To SQL we can add GQL-like connecting metadata to table (and user defined type) creation syntax. To combine named properties with arrow syntax we can add direction to SQL and arrow names to GQL. Let us call such a unified relational system DBR.

Then assuming the current user is allowed to make changes to the schema, consider the effect of a statement such as:

```
INSERT (:Person{name:'Fred'})-worksin
        ->(:Company{name:'AWS'})                      (1)
```

This is not a GQL insert statement as the worksin identifier is just naming the arrow. It could create a `Person` record (with a record identity say f) and a `Company` record (say c), creating new tables (node types) for them, and a reference property `worksin` in Person whose value is the AWS record in Company:

```
 PERSON f { Name:Fred, worksin: Company_c}            (2)
 COMPANY c {Name:AWS}
```

On the other hand, the GQL insert syntax

```
INSERT (:Person{name:'Fred'})-[:WorksIn]
->(:Company{name:'AWS'}))                             (3)
```

mandates a separate edge type `WorksIn` that connects `PERSON` to `COMPANY` (which in GQL is the more natural model). This makes more sense if the edge has its own properties, as in a different example:

```
INSERT (Person{Name:'Mary'})-[:Married '2025-07-24']
      ->(:Person{name:'Joe'})                         (4)
```

Ignore syntax (3) for now and consider implementing (4). If Person Joe is already defined, in GQL (which does not yet have a `MERGE` statement) we would need to retrieve a reference for him first, and write something like

```
MATCH (j:Person{Name:'Joe'}) INSERT
   (:Person{name:'Mary')                              (5)
   -[:Married {Date:2025-07-24}]->(j)
```

Here `Joe` is already constructed: we want to construct a new node for Mary and connect a `Married` edge to `Joe`. The `Match` statement binds j to the `Person` node whose name is `Joe`, so that the insert statement can construct an edge to the new Person node Mary. Such an edge type is declared in GQL by the element specification

```
Directed edge MARRIED {date :: Date} connecting (PERSON
-> PERSON)                                            (6)
```

The SQL equivalent syntax (also supported in DBR) creates a user defined type with the edge connection information (from/to/with and named arrows) as metadata

```
Create type MARRIED as (date Date) edge type (from
bride=PERSON to groom=PERSON)                         (7)
```

The resulting DBR database so far has three tables:

```
Company: (Name string)
Person: (Name string, Worksin ref Company)            (8)
Married: (Bride ref Person, Groom ref Person,
Date date}
```

If `Likes` is implemented as in (1) we might expect the `Person` table to have a `Likes` column of type `set ref Person`.

The key to implementing `MATCH` and `INSERT` is a system index that contains all references (as a mapping of referenced record to property uid and referencing record) on the understanding that each record type maintains an index of its records.

IV. SUPPORTING GRAPH INSERT AND MATCH OPERATIONS

Graph insertion is almost trivial once the identity of any referenced rows is established, and many products offer syntax similar to (1) above. The simple syntax of (1) above can be easily extended to allow comma-separated lists, node references allowing a node or edge constructed in one part of the insert statement to be referred to later.

For pattern matching, the syntax tends to be more complex, but the basic idea can be seen from the example (supposing many records of marriages) :

```
MATCH (p)-[:Married]->(:person{name:'Joe'})         (9)
```

This should return a table of results showing all of the `Persons` P that married a `Person` named `Joe`. In more complex cases, there will be several columns of results and ways of aggregating results (count, sum etc). To implement `MATCH` it is best to work along the `MATCH` expressions identifying what possibilities there are at every stage and backtracking to consider alternatives. The programming technique of continuations can be used for complex patterns where groups of edges can be repeated [14].

Reverse references are best managed by indexes constructed by the DBMS when records are added, and then such indexes can simply use the identity of the new record instead of its values. This infrastructure can be added to any relational database and then used to consider what the graph looks like: we consider such choices in the next section.

Graph models are useful to indicate chains of links (such as transfers of funds, supply chains, etc), and the same business database should be able to support different graph models to allow the study of different sorts of chains. Some graph database management systems implement `MATCH` statements to traverse such chains without constructing joins of tables. A unified implementation of SQL and GQL would allow graph insert and match query by graph patterns, which in some use cases results in large efficiency gains.

V. USE ROLE METADATA TO CREATE TYPE GRAPH MODELS

In many use cases, the data in graph models comes from a relational database. Commonly, the node types and edge types in the database have a simple relationship with entities and tables in the relational database. As long as this relationship continues, we expect that execution of a property graph data-modifying statement will make appropriate changes in the underlying relational tables.

In many cases, the graph model comes first (as in (1) above), and the tables in the relational database directly support correspondingly named graph element types (as in

(2)). Single reference properties, such as `worksin` implement simple edges without properties.

It is also common for graph models to be built from tables already in a relational database, and in these cases, choices are available. Any relational table can be treated as a node type, and some or all of its reference columns as simple edges. Any relational table with two reference columns can be treated as an edge type, and there is a choice of direction. If a relational table has more than two reference columns then any two columns can become the source and destination columns to make it an edge type. These choices can be saved in graph metadata such as `EDGETYPE(bride=person TO groom=person)`.

Let us briefly consider the `Worksin` and `Married` examples in this context. Suppose the database has been constructed with the three tables as above. The obvious graph model over this seems to have two node types and one edge type. But let us specify that DBR will support a model that also treats Worksin fully as an edge type even though it does not have its own records.

Equally, a graph model that had no interest in dates of marriages could in DBR treat `Married` as a simple column similar to `Worksin`.

Many SQL implementations support the use of roles to identify aliases and privileges for particular classes of users, so it is natural to expect that alternative graph models can be defined for different roles. Many real relational databases are very large, and such a role definition would help to simplify the database for its users by hiding tables and columns that are not relevant to the model, and/or limiting the ability of the role to modify data. While a graph model is being developed, it would be very useful for such role definitions to be automatically generated by inferring the metadata from graph insert patterns such as (1) or (5).

We propose to go further and implement union types and subtypes in our new implementation DBR. Subtype inheritance is already under discussion in the GQL community: multiple node labels (the GQL notion of label sets) can be implemented with union types.

For example, in the GDC Financial benchmark [10] we see edges with union connectors:

```
directed edge Apply {createTime::timestamp,
organization::string} connecting (Person|Company to
Loan)                                            (10)
```

DBR implements this by allowing connector references in column lists in SQL insert statements:

```
   insert into apply (from person, loan, createTime,
organization)
values('57978','4612532092624966603','2020-01-04
22:30:59.239','OneMain Financial')                (11)
```

This mechanism is used when loading the data in the Financial Benchmark from spreadsheet files. Once the data is loaded, the use of rowids instead of key values promises gains in performance.

VI. An experimental implementation of DBR

An implementation of the above concept is underway, as a research project extending the Pyrrho DBMS [11]. It already has been used to construct the Graph Data Council Financial Benchmark [10]. Current progress already indicates that this implementation behaves as it should when given ordinary relational tasks, atomic, consistent, isolated, and durable (ACID) transactions, procedures, methods, triggers etc.) and offers property graph syntax compatible with the new GQL standard [9].

With the above approach, a referencing property/column can be (a) implied by `FOREIGN KEY` metadata based on key values or (b) defined with `REFERENCES` metadata specifying that its value will identify a row or set of rows in a given target. In both cases the column defines a directed or undirected (possibly set-valued) edge connector. The referent(s) may be of a single table or type, or of a join or disjoint sum of tables/types specified by label set(s) (as in (10) above).

Noting that both SQL and GQL specifications tolerate implementation-specific extensions, the conclusion of the research is that a single implementation of both languages is indeed possible and provides a backward-compatible upgrade path for all existing implementations should they choose to follow it.

Performance and memory usage in the experimental version of Pyrrho DBMS is similar to previous versions: database size on disk is somewhat reduced, use of computer memory slightly increased. It is for the beholder to judge matters such as conceptual clarity and elegance: we do claim that in addition to providing graph facilities to SQL and relational storage to GQL, the proposals here provide a way of clarifying the relationship between the design of a database and the construction of a graph model. Our approach allows a given relational database to support different graph models, offering a choice depending on different use cases.

VII. Conclusions and Future Work

Even without node and edge types, it is convenient to be able to use graphical insert and MATCH when working with relational databases, to specify MATCH for table constraints, and to define subtypes using MATCH. Importantly, tests include verification that database objects and metadata can be added or dropped during transactional operation.

The accompanying conference presentation will contain examples additional to those above, based on the GDC Financial Benchmark [10], and a tutorial session at DBKDA is planned.

It is hoped to have the implementation of the combined system by the conference date. A tutorial will include hands-on demonstrations.

The Appendix gives a brief test of the current state of implementation, showing the construction of the financial benchmark [10], using the smallest scale factor sf001. The execution time was 49 seconds on a desktop PC with Ultra i5 procesor.

APPENDIX

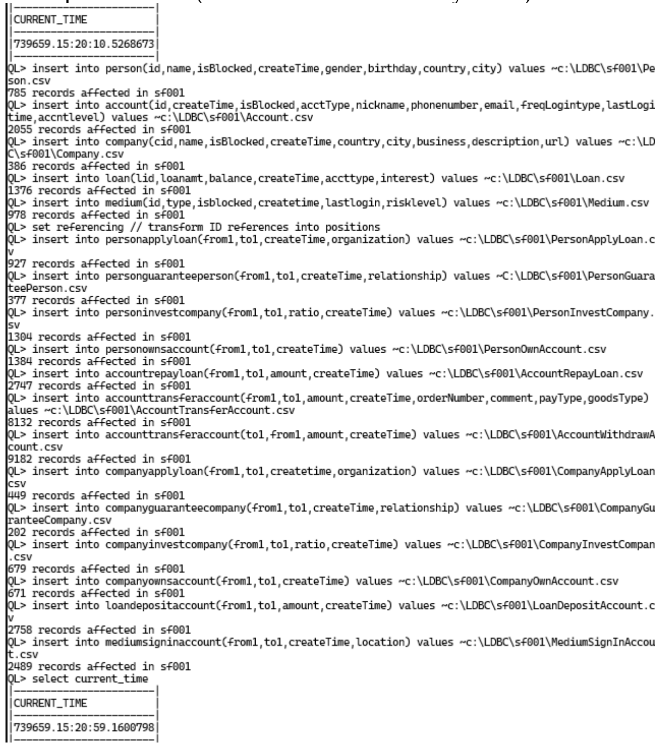
1. The GQL syntax description of the financial benchmark as specified in [10]. This text can be processed by the current version of DBR.

```
create schema /ldbc
[create graph type /ldbc/finBenchMark  {
node Person {id::int,name::string,isBlocked::boolean,
createTime::timestamp,gender::string,birthday:: date,country::
string,city::string},
node Account {id::int,createTime::timestamp,isBlocked::boolean,
type::string,nickname::string,phoneNumber::string,email::string,
freqLoginType::string,lastLoginTime::timestamp,accountLevel:: string},
node Medium {id::int,type::string,isBlocked::boolean,
createTime::timestamp,lastLoginTime::timestamp,riskLevel::string},
node Company{id::int,name::string,isBlocked::boolean,
createTime::timestamp,country::string,city::string,
business::string,description::string, url::string},
node Loan {id::int,loanAmount::float64,balance::float64,
createTime::timestamp,usage::string,interestRate::float32},
directed edge Transfer {amount::float64,createTime::timestamp,
ordernumber::string,comment::string,payType::string,
goodsType::string} connecting (Account to Account),
directed edge Withdraw {createTime::timestamp,amount::float64}
connecting (Account to Account),
directed edge Repay {createTime::timestamp,amount::float64}
connecting (Account to Loan),
directed edge Deposit {createTime::timestamp,amount::float64}
connecting (Loan to Account),
directed edge SignIn {createTime::timestamp,location::string}
connecting (Medium to Account),
directed edge Invest {createTime::timestamp,ratio::float64}
connecting (Person|Company to Company),
directed edge Apply {createTime::timestamp,organization::string}
connecting (Person|Company to Loan),
directed edge Guarantee {createTime::timestamp,relationship::string}
connecting (Person|Company to Person|Company),
directed edge Own {createTime::timestamp}
connecting (Person|Company to Account)}]
```

2. The DBR syntax description of the financial benchmark modified to use inheritance.

```
create type legalentity as(name string,isBlocked boolean,createtime
timestamp,country string,city string) nodetype
create type person under legalentity as (id int primary key,gender
string,birthday timestamp)
create type company under legalentity as(cid int primary key,business
string,description string,url string)
create type actbase as(createtime timestamp,accttype string) nodetype
alter table actbase add balance float64 default 0.0
create type account under actbase as(id int primary key,isBlocked
boolean,nickname string,phonenumber string,email string,freqlogintype
string,lastlogintime timestamp,accntlevel string)
create type loan under actbase as (lid int primary key,loanAmt
float64,interest float32)
create type medium as(id int primary key,type string,isblocked
boolean,createtime timestamp,lastlogin timestamp,risklevel string)
nodetype
create type apply as (createTime timestamp,organization string)
edgetype(legalentity,loan)
create type personapplyloan under apply edgetype(person,loan)
create type companyapplyloan under apply edgetype(company,loan)
create type guarantee as (createTime timestamp,relationship string)
edgetype (legalentity,legalentity)
create type personguaranteeperson under guarantee
edgetype(person,person)
create type invest as (ratio float64,createTime timestamp)
edgetype(legalentity,legalentity)
create type personinvestcompany under invest edgetype(person,company)
create type owns as (createTime timestamp)
edgetype(legalentity,actbase)
create type personownsaccount under owns edgetype(person,account)
create type activatedfor as (createtime timestamp,location string)
edgetype (medium,account)
create type transfer as(amount float64,createTime
timestamp,orderNumber int,comment string,payType string,goodsType
string) edgetype(from actbase with activatedfor optional to actbase)
create type accountrepayloan under transfer edgetype(from account with
activatedfor optional to loan)
create type accounttransferaccount under transfer edgetype(from
account with activatedfor optional to account)
create type companyguaranteecompany under guarantee
edgetype(company,company)
create type companyinvestcompany under invest
edgetype(company,company)
create type companyownsaccount under owns edgetype(company,account)
create type loandepositaccount under transfer edgetype(from loan with
activatedfor optional to account)
create type mediumsigninaccount as (createtime timestamp, location
string) edgetype(medium,account)
```

3. The construction of sf001 with the above DBR specification (recorded on 13 February 2026).



```
|----------------------|
|CURRENT_TIME          |
|----------------------|
|739659.15:20:10.5268673|
|----------------------|
QL> insert into person(id,name,isBlocked,createTime,gender,birthday,country,city) values ~c:\LDBC\sf001\Pe
son.csv
785 records affected in sf001
QL> insert into account(id,createTime,isBlocked,acctType,nickname,phonenumber,email,freqLogintype,lastLogi
time,accntlevel) values ~c:\LDBC\sf001\Account.csv
2055 records affected in sf001
QL> insert into company(cid,name,isBlocked,createTime,country,city,business,description,url) values ~c:\LD
C\sf001\Company.csv
386 records affected in sf001
QL> insert into loan(lid,loanamt,balance,createTime,accttype,interest) values ~c:\LDBC\sf001\Loan.csv
1376 records affected in sf001
QL> insert into medium(id,type,isblocked,createtime,lastlogin,risklevel) values ~c:\LDBC\sf001\Medium.csv
978 records affected in sf001
QL> set referencing // transform ID references into positions
QL> insert into personapplyloan(from1,to1,createTime,organization) values ~c:\LDBC\sf001\PersonApplyLoan.c
v
927 records affected in sf001
QL> insert into personguaranteeperson(from1,to1,createTime,relationship) values ~C:\LDBC\sf001\PersonGuara
teePerson.csv
377 records affected in sf001
QL> insert into personinvestcompany(from1,to1,ratio,createTime) values ~c:\LDBC\sf001\PersonInvestCompany.
sv
1304 records affected in sf001
QL> insert into personownsaccount(from1,to1,createTime) values ~c:\LDBC\sf001\PersonOwnAccount.csv
1384 records affected in sf001
QL> insert into accountrepayloan(from1,to1,amount,createTime) values ~c:\LDBC\sf001\AccountRepayLoan.csv
2747 records affected in sf001
QL> insert into accounttransferaccount(from1,to1,amount,createTime,orderNumber,comment,payType,goodsType)
alues ~c:\LDBC\sf001\AccountTransferAccount.csv
8132 records affected in sf001
QL> insert into accounttransferaccount(to1,from1,amount,createTime) values ~c:\LDBC\sf001\AccountWithdrawA
count.csv
9182 records affected in sf001
QL> insert into companyapplyloan(from1,to1,createtime,organization) values ~c:\LDBC\sf001\CompanyApplyLoan
csv
449 records affected in sf001
QL> insert into companyguaranteecompany(from1,to1,createTime,relationship) values ~c:\LDBC\sf001\CompanyGu
ranteeCompany.csv
202 records affected in sf001
QL> insert into companyinvestcompany(from1,to1,ratio,createTime) values ~c:\LDBC\sf001\CompanyInvestCompan
.csv
679 records affected in sf001
QL> insert into companyownsaccount(from1,to1,createTime) values ~c:\LDBC\sf001\CompanyOwnAccount.csv
671 records affected in sf001
QL> insert into loandepositaccount(from1,to1,amount,createTime) values ~c:\LDBC\sf001\LoanDepositAccount.c
v
2758 records affected in sf001
QL> insert into mediumsigninaccount(from1,to1,createTime,location) values ~c:\LDBC\sf001\MediumSignInAccou
t.csv
2489 records affected in sf001
QL> select current_time
|----------------------|
|CURRENT_TIME          |
|----------------------|
|739659.15:20:59.1600798|
|----------------------|
```

REFERENCES

[1] S. Sakr and G. Al-Naymat, "Efficient relational techniques for processing graph queries." *Journal of Computer Science and Technology* 25.6 (2010): 1237-1255.

[2] D. Wang et al., "A semi-supervised graph attentive network for financial fraud detection." In *2019 IEEE international conference on data mining (ICDM)*, pp. 598-607. IEEE, 2019.

[3] Y-C. Hong, and J. Chen, "Graph database to enhance supply chain resilience for industry 4.0." *International Journal of Information Systems and Supply Chain Management* (IJISSCM) 15.1 (2022): 1-19.

[4] P. Grindrod, and M. Kibble, "Review of uses of network and graph theory concepts within proteomics." *Expert review of proteomics* 1.2 (2004): 229-238.

[5] D. F. Nettleton, "Data mining of social networks represented as graphs." *Computer Science Review 7* (2013): 1-34.

[6] E. A. Hobson et al., "A guide to choosing and implementing reference models for social network analysis". *Biological Reviews*, *96*(6), 2716-2734.

[7] M. A. Rodriguez, "The gremlin graph traversal machine and language" (invited talk). In *Proceedings of the 15th symposium on database programming languages* (pp. 1-10).

[8] S. Plantikow, "Towards an International Standard for the GQL Graph Query Language." *W3C workshop in Berlin on graph data management standards*. Vol. 84. 2019.
https://www.w3.org/Data/events/data-ws-2019/assets/position/Stefan%20Plantikow.pdf

[9] ISO/IEC 39075:2024, Information technology — Database languages — GQL, Published (Edition 1, 2024).

[10] Graph Data Council (formerly Linked Data Benchmark Council), Financial Benchmark https://ldbcouncil.org/benchmarks/finbench/

[11] See https://pyrrhodb.blogspot.com for an introduction.

QuaQue: Design and SQL Implementation of Condensed Algebra for Concurrent Versioning of Knowledge Graphs

Jey Puget Gil, Emmanuel Coquery, John Samuel, Gilles Gesquière
Universite Claude Bernard Lyon 1, CNRS, INSA Lyon, Université Lumière Lyon 2,
Ecole Centrale de Lyon, CPE Lyon, LIRIS, UMR 5205
Villeurbanne, France
e-mail: {jey.puget-gil | emmanuel.coquery | john.samuel | gilles.gesquiere}@liris.cnrs.fr

***Abstract*—The management of versioned knowledge graphs presents significant challenges, particularly in querying data across multiple versions efficiently. This paper introduces QuaQue, a key component of the ConVer-G system, which addresses this challenge by translating SPARQL (SPARQL Protocol and RDF Query Language) queries into SQL (Structured Query Language). QuaQue leverages a novel condensed algebra to operate on a relational model where versioning information is compactly stored using bitstrings. This approach allows for efficient querying of concurrent versions of knowledge graphs within a standard relational database system. We present the key concepts of our condensed algebra, detail the translation process from SPARQL algebra to SQL, and provide a comparative benchmark against a native RDF (Resource Description Framework) triple store, demonstrating the viability and performance benefits of our approach.**

Keywords-SQL; algebra; translation; concurrent versioning; relational.

I. Introduction

The representation of complex, evolving information has driven the widespread adoption of Knowledge Graphs (KGs) in both industry and academia [1]. However, as KGs move from static repositories to dynamic assets, there is a critical need for robust *concurrent versioning systems*. In domains, such as urban planning—where datasets undergo parallel modifications by multiple stakeholders—a linear history is insufficient. Systems must support branching, merging, and the analysis of concurrent states without excessive data redundancy.

While the Resource Description Framework and SPARQL are the de facto standards for KGs, they were primarily designed for static or monotonically increasing datasets. Native support for versioning remains a challenge. Standard approaches often rely on Named Graphs [2] to isolate versions. While accessing a single version remains efficient, this strategy leads to significant data duplication and increased query latency when querying across multiple versions. Consequently, efficiently querying across multiple versions remains an open problem in database research.

The ConVer-G project [3] addresses this by bridging the gap between graph versioning requirements and the mature optimization capabilities of Relational Database Management Systems (RDBMSs). At the core of ConVer-G is **QuaQue**, a system that translates SPARQL queries into SQL that exploits efficient bitwise operations for version filtering.

QuaQue leverages a *condensed relational model*. In RDF, a *quad* is a tuple that extends the standard triple with a graph identifier. We associate every quad with a *bitstring*, where each bit represents the validity of the quad in a specific version. This allows us to push version-filtering logic down to the RDBMS engine using efficient bitwise operations, significantly reducing the I/O overhead typically associated with multi-version queries.

This paper details the design and implementation of QuaQue. We introduce a *condensed algebra*—an extension of relational algebra tailored for bitstring-annotated relations—and define its translation to SQL. This enables the execution of complex graph pattern matching on standard PostgreSQL instances.

Our specific contributions are:

- A novel **condensed relational model** utilizing bitstrings for the storage of concurrent KG versions.
- The implementation of a **Condensed Algebra** and the QuaQue translator which maps SPARQL algebra to SQL.
- A comparative benchmark demonstrating that QuaQue outperforms a native RDF triple store (Apache Jena) in multi-version query scenarios.
- A fully reproducible approach, with the ConVer-G tool and the benchmark framework publicly available as open-source software.

The remainder of this paper is organized as follows. Section 2 presents the state of the art. Section 3 details the design of the QuaQue system and the condensed relational model. Section 4 presents the experimental evaluation and benchmark results. Finally, Section 5 concludes the paper and outlines future work.

II. State of the Art

The challenge of querying versioned data has been a long-standing topic in database research [4]. The relational model [5], introduced by Codd, provides a foundation with relational algebra and calculus as powerful query languages. The expressive power of these languages has been a central theme of research, with extensions proposed to handle more complex queries, such as those involving recursion [6][7] and aggregate functions [8]. A key milestone in bridging the gap between high-level query languages and efficient execution is the work of Ullman [9], which systematically addresses the translation of relational algebra and calculus queries into implementations, laying the groundwork for query processing and optimization. This connection is important, as the development of new algebras or extensions—such as those needed for versioned or condensed data—must ultimately be supported by practical translation and execution strategies. Recent surveys, such as Hofer et al. [10], Jiang et al. [11][12], and Ji et al. [13] provide a comprehensive overview of the current state and challenges in knowledge graph construction, highlighting the increasing complexity of managing evolving and versioned knowledge graphs. They emphasize the need for scalable and efficient methods for both constructing and maintaining knowledge graphs, especially as these graphs become larger and more dynamic. This underscores the importance of advanced versioning and querying mechanisms to support the evolving requirements of knowledge graph applications. A notable example of leveraging versioned knowledge graphs in practice is the work by Gonzalez-Hevia and Gayo-Avello [14], who utilize Wikidata's edit history for knowledge graph refinement tasks. Their approach demonstrates the value of exploiting historical edit information to improve the quality and reliability of knowledge graphs, further motivating the need for efficient storage and querying of versioned data.

Recent work by Zhong et al. [15] further illustrates the importance of knowledge graphs in real-world applications, specifically in the domain of intelligent audit. Their study discusses both the opportunities and challenges of applying knowledge graphs to extract insights from complex, evolving data sources. This highlights the direct link between the need for advanced versioning and querying mechanisms—such as those discussed in this section—and the practical requirements of domains where data evolution and efficient cross-version analysis are critical for generating reliable insights.

A. Extending Relational Algebra

Relational algebra has been extended with techniques, such as rewriting queries with arbitrary aggregation functions using views, as discussed by Cohen et al. [16]. The extensibility of relational algebra has also been a subject of research, particularly in the context of supporting new data types and operations. Haas et al. [17] proposed an extensible query processor architecture that allows the integration of user-defined types and functions into the relational algebra framework. This extensibility adapts relational systems to various application domains, enabling the seamless incorporation of specialized operators and data structures without sacrificing the benefits of a declarative query language.

B. Relational Algebra in Query Languages

The principles of relational algebra have been foundational to the design of numerous query languages. SQL, the de facto standard for relational databases, is based on a tuple relational calculus, which is equivalent in expressive power to relational algebra [18]. The translation of SPARQL [19], the standard query language for RDF, to SQL has been a topic of interest for leveraging the performance and scalability of relational databases for semantic web data. Several systems, such as Ontop [20], have explored this translation, often relying on mappings between RDF and relational schemas.

Relational algebras are also applied outside the context of traditional databases. In qualitative spatial and temporal reasoning [21][22], relation algebras serve as formal tools for modeling and inferring relationships between spatial or temporal entities. For instance, Allen's interval algebra [23] provides a calculus for reasoning about temporal intervals, while the Region Connection Calculus (RCC) [24] is used for spatial reasoning. These algebras provide a formal, equational framework for deriving new knowledge from a set of base relations.

C. Semantic Versioned Querying: The Fundamentals

A contribution to the field of versioned querying for knowledge graphs is presented by Taelman et al. [25]. Their work systematically investigates the requirements and challenges of querying versioned semantic data, focusing on the formalization of versioned query semantics and the practical implications for query processing.

Taelman et al. introduce a formal framework for semantic versioned querying, distinguishing between different types of versioned queries, such as snapshot queries (retrieving data as it existed at a specific version), longitudinal queries (tracking the evolution of data across versions), and difference queries (identifying changes between versions). They emphasize the importance of clearly defined semantics for each query type, as ambiguity can lead to inconsistent or unintuitive results.

A key insight from their work is the need for query languages and systems to natively support version-aware operations, rather than treating versioning as an afterthought or external feature. They propose extensions to SPARQL that allow users to specify version constraints directly within queries, enabling more expressive and precise retrieval of historical or evolving data.

Overall, the work of Taelman et al. provides a foundational perspective on the semantics of querying versioned knowledge graphs. Their formalization of query types has been influential in our work, serving as a guideline for the capabilities our system aims to support.

D. Versioning Models for Evolving Data

Foundational models for data evolution stem from Software Configuration Management (SCM) [26], which distinguishes between sequential revisions and parallel variants. This distinction applies to knowledge graph versioning, where evolution involves both temporal changes and concurrent viewpoints [27]. SCM also differentiates between state-based (snapshots) and change-based (deltas) models [26]. In Model-Driven Engineering (MDE), these concepts extend to graph structures, where revisions are defined as sequences of atomic graph modifications [28]. Formalizing changes as graph operations enables structured reasoning about conflicts and merging [29], providing a theoretical basis for versioned query algebras.

Several strategies for storing versioned RDF data have emerged, balancing storage efficiency and query performance:

1) Independent Copies (IC): The IC or snapshot approach stores each version as a full copy [30]. While efficient for single-version querying (Version Materialization), it suffers from high storage redundancy.

2) Change-Based (CB): CB or delta-based versioning stores a base version and subsequent changes. This is space-efficient but requires costly reconstruction for querying. Tools, such as R43ples [29], R&WBase [31], and Stardog, follow this paradigm.

3) Timestamp-Based (TB): TB approaches annotate triples with validity intervals, facilitating "time-travel" queries. While suited for linear evolution, supporting branching adds complexity. Tools adopting this strategy include ConVer-G [32], Drydra [33], RDF-TX [34], v-RDFCSA [35], and x-RDF-3X [36], often drawing on temporal database concepts [37].

4) Fragment-Based (FB): FB or hybrid approaches partition the graph into independently versioned fragments, balancing storage and query performance. However, managing dependencies between fragments is complex. QuitStore [38] implements this by versioning modified files in a Git repository.

5) Graph Compression: Graph compression techniques, such as HDT (Header, Dictionary, Triples) [39], offer another perspective on efficient RDF storage. HDT is a binary format that achieves high compression ratios by utilizing dictionary encoding for RDF terms and a compact bitmap-based structure for the graph topology. Crucially, HDT files are designed to be queryable directly without decompression, providing excellent read performance for static datasets. However, the primary limitation of HDT and similar compression-centric approaches in the context of versioning is their static nature. They are optimized for read-only scenarios; modifying the data typically requires a computationally expensive reconstruction of the entire file. While one could theoretically store each version as a separate HDT file (effectively an optimized IC approach), this does not inherently solve the redundancy problem for small incremental changes, nor does it facilitate efficient cross-version querying or branching/merging operations. Therefore, while compression is a valuable component of storage optimization, it does not by itself constitute a complete versioning strategy for dynamic, evolving knowledge graphs.

6) Limitation: Git-based solutions, such as R&WBase [31] and QuitStore, require explicit checkout to query a version, hindering concurrent cross-version analysis [3]. Existing strategies struggle to balance storage efficiency and query performance for such analysis.

E. Summary: The Case for a Condensed TB Representation and Algebra

Current versioning strategies face a trade-off between storage and query efficiency, often lacking support for concurrent cross-version analysis [40]. A **condensed TB representation** addresses this by storing unique quads annotated with version validity, as seen in ConVer-G [3]. To exploit this model, a condensed algebra is needed to define operators on version-annotated structures, similar to specialized algebras, such as Knowledgebra [41]. Finally, an algebra-to-SQL translator (QuaQue) bridges the gap to efficient execution on relational systems [27].

III. Design

The QuaQue component of the ConVer-G system is a SPARQL-to-SQL translator designed to query a condensed relational model of versioned RDF data. Our approach is motivated by the need for efficient cross-version queries, which are cumbersome and inefficient with traditional triple-store-based versioning methods that replicate data for each version.

A. Condensed Relational Model

Our condensed model represents versioned RDF data in a relational database management system. We chose PostgreSQL for our implementation because it provides native support for bit string data types and efficient bitwise operations. This representation avoids storing duplicate quads that exist across multiple versions. The interaction between the SPARQL translation and this model is depicted in Figure 2. Versioned quad table, which stores quads (subject, predicate, object, named graph) along with a validity bitstring. Each bit in the validity string corresponds to a specific version, and a '1' at a given position indicates that the quad is present in that version. The structure of our condensed relational model is illustrated in Figure 1 and described as follows:

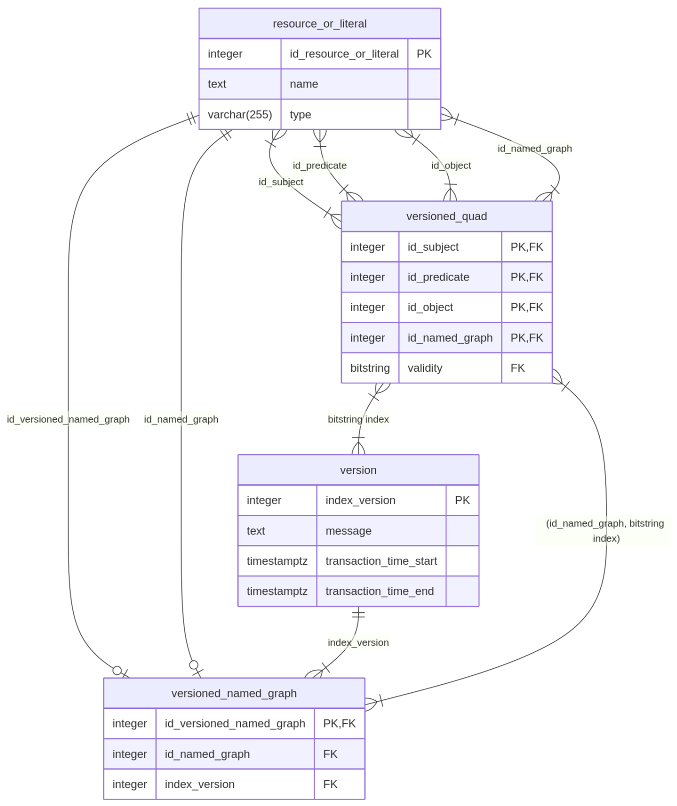


Figure 1. Relational model of the condensed representation for versioned RDF data.

The schema comprises the following relations:

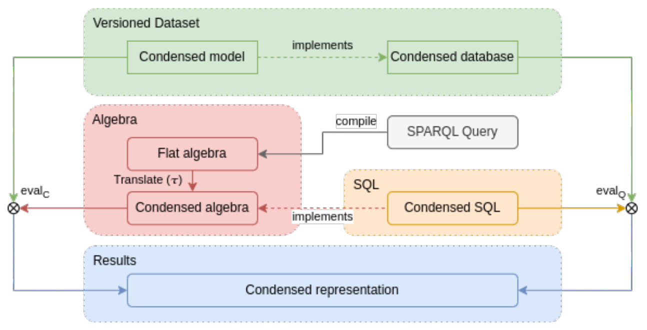


Figure 2. Overview of the SPARQL to SQL translation process in QuaQue.

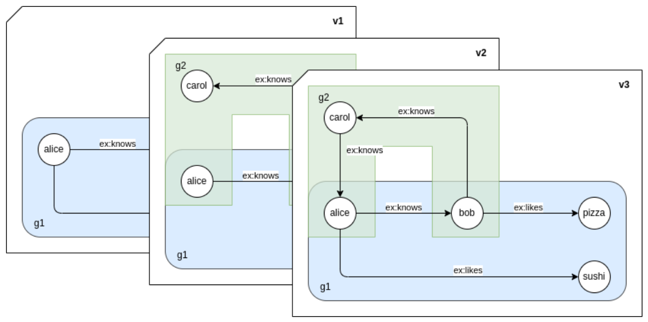


Figure 3. Graphical representation of the sample versioned RDF dataset.

- `versioned_quad`: This is the central table, storing unique quads (subject, predicate, object, named graph) as integer identifiers. Each entry includes a validity bitstring that indicates the versions in which the quad is present. This design minimizes redundancy by storing each unique quad only once, regardless of how many versions it appears in. An example is provided in Table III.
- `resource_or_literal`: This table acts as a dictionary, mapping RDF terms (URIs and literals) to the integer identifiers used in other tables. This practice, known as dictionary encoding, optimizes storage and join performance by replacing long strings with compact integers. See Table XI in the Appendix for an example.
- `version`: This table holds metadata specific to each version, such as its creation timestamp or a descriptive label. Separating version metadata allows for efficient retrieval of version-specific information without scanning the quad data.
- `versioned_named_graph`: This table links named graphs to the versions they belong to. This separates the association between graphs and versions from the quad data, adhering to database normalization principles to support scenarios where named graphs evolve independently across versions.
- `metadata`: This table stores additional metadata, which can be user-defined, about versions and named graphs. This flexibility accommodates diverse application requirements for tracking contextual information.

This schema design ensures the storage and retrieval of versioned RDF data while maintaining the flexibility required for diverse application scenarios.

IV. Development

A. Condensed Algebra and SQL Translation

The QuaQue component translates SPARQL queries into SQL by first converting the SPARQL query into a SPARQL algebra expression, and then mapping the SPARQL algebra operators to SQL operations on our condensed model, as shown in Figure 2.

The core of our approach lies in how we handle the validity bitstring. For operations that combine quads, such as joins, we use bitwise operations on the validity bitstrings. For example, a join between two quad patterns corresponds to a bitwise AND operation on their validity bitstrings. This allows us to efficiently determine the versions in which the joined pattern is valid.

The translation process can be summarized as follows (see Figure 2):

- **SPARQL to SPARQL Algebra:** The incoming SPARQL query is parsed into a SPARQL algebra expression tree using Apache Jena.
- **SPARQL Algebra to Condensed SQL:** The SPARQL-toSQLTranslator traverses the algebra tree and, for each operator, generates a corresponding SQL query fragment. The QuadPatternSQLOperator handles the base case of translating a quad pattern into a SQL query on the `versioned_quad` table. Other operators, such as JoinSQLOperator and Group-SQLOperator, combine these fragments using bitwise operations and other SQL constructs.
- **Finalization:** The FinalizeSQLOperator combines the generated SQL fragments into a single, executable SQL query.

B. Sample Dataset

To illustrate the QuaQue approach, we use a simple versioned RDF dataset about users, their friendships, and their preferences. The dataset consists of three versions, each representing a snapshot of the data at a different point in time.

TABLE I. Sample RDF Quads Across Versions

Subject	Predicate	Object	Graph	Versions
:alice	ex:knows	:bob	:g1	1,2,3
:bob	ex:likes	"pizza"	:g1	2,3
:alice	ex:likes	"sushi"	:g1	1,3
:carol	ex:knows	:alice	:g2	3
:bob	ex:knows	:carol	:g2	2,3

Figure 3 provides a graphical representation of the dataset, while Table II lists the metadata associated with the versioned graphs.

In the table II, resources usually prefixed with `:vng` (e.g., `:vng1`, `:vng2`) serve as identifiers for "Versioned Named Graphs", explicitly linking a named graph to a specific version.

For three versions, the bitstring has three bits (e.g., `111` for all versions, `010` for version 2 only). We illustrate the condensed representation of versioned RDF data in Table III.

TABLE II. METADATA OF VERSIONED GRAPHS

Subject	Predicate	Object
:vng1	v:in-version	1
:vng1	v:version-of	:g1
:vng2	v:in-version	2
:vng2	v:version-of	:g1
:vng3	v:in-version	3
:vng3	v:version-of	:g1
:vng4	v:in-version	2
:vng4	v:version-of	:g2
:vng5	v:in-version	3
:vng5	v:version-of	:g2

TABLE III. CONDENSED REPRESENTATION OF THE VERSIONED QUADS (VERSIONED_QUAD)

id_subj.	id_pred.	id_obj.	id_n_graph	validity
1	6	2	10	111
2	7	4	10	011
1	7	5	10	101
3	6	1	20	001
2	6	3	20	011

1) Quad Pattern Translation: The translation of a SPARQL quad pattern into SQL in the condensed model is straightforward. Each quad pattern is mapped to a SQL query over the `versioned_quad` table, with conditions on the `subject`, `predicate`, `object`, and `graph` columns as specified by the quad pattern. The validity bitstring is always projected, as it encodes the presence of the quad across versions. Each variable in the quad pattern is represented as a column in the SQL result, prefixed with `v$` for variables, `ng$` for named graph variables and `bs$` for bitstring variables.

For example, the SPARQL quad pattern:

Query 1: Example SPARQL quad pattern.

```
?s <ex:knows> ?o ?g .
```

is translated to the following SQL:

Query 2: SQL translation of the example SPARQL quad pattern.

```
SELECT (t0.validity) as bs$g,
   t0.id_subject as v$s,
   t0.id_object as v$o,
   t0.id_named_graph as ng$g
FROM versioned_quad t0
WHERE bit_count(t0.validity) <> 0 AND t0.
    id_predicate =
   (Subquery to get id of <ex:knows>)
```

Given the dataset from Table III and the quad pattern of Query 1, the result of the query is shown in Table IV.

Variables `v$s`, `v$o`, and `ng$g` correspond to the subject, object, and named graph IDs, and `bs$g` is the validity bitstring for each result.

2) Join Operation Translation: The translation of a SPARQL join operation into SQL involves combining the SQL fragments generated for each participating quad pattern. The key aspect of this translation is the handling of the validity bitstrings.

Figure 4: Quad pattern translation to SQL

```
Input: Quad Pattern qp (from current operator)
Output: String $sql\_query$
Function TranslateQuadPattern(qp) is
    $select \leftarrow$ "$id\_subject, id\_predicate, id\_object$";
    $from \leftarrow$ "";
    $where \leftarrow$ "";
    /* Check if the quad pattern
       targets the metadata or a
       versioned graph                    */
    if $qp.graph = $ "$defaultgraph$" then
        $from \leftarrow$ "$metadata$";
    else
        $select \leftarrow$
          $select +$ "$, id\_named\_graph, validity$";
        $from \leftarrow$ "$versioned\_quad$";
        $where \leftarrow$ "$bit\_count(validity) <> 0$";
    end
    foreach $t$ in $[qp.subject, qp.predicate, qp.object]$
      do
        $where \leftarrow where + term\_to\_condition(t)$;
    end
    return "$SELECT$" + $select$ + "$FROM$" +
      $from$ + "$WHERE$" + $where$;
end
```

TABLE IV. RESULT OF THE QUAD PATTERN QUERY

bs$g	v$s	v$o	ng$g
111	1	2	10
001	3	1	20
011	2	3	20

When two quad patterns are joined, their validity bitstrings are combined using a bitwise AND operation. This ensures that the resulting rows only include versions where both patterns are valid.

For example, consider the SPARQL join of two quad patterns on a shared graph name variable:

Query 3: Example SPARQL join of two quad patterns (join on graph name variable).

```
?s <ex:knows> ?o ?g .
?o <ex:likes> ?liked ?g .
```

This join between two quad patterns where the joined variables are in a condensed representation is translated into SQL as follows:

Query 4: SQL translation of the example SPARQL join.

```
SELECT (t0.validity & t1.validity) as bs$g, t0
    .id_subject as v$s, t0.id_named_graph as
    ng$g, t1.id_object as v$liked, t0.
    id_object as v$o
FROM versioned_quad t0, versioned_quad t1
WHERE bit_count(t0.validity & t1.validity) <>
    0 AND
   t0.id_object = t1.id_subject AND
```

Figure 5: Algorithm of the Join translation

```
Input: Join Operator $join$
Output: String $sql\_query$
Function $TranslateJoin(join)$ is
    $left \leftarrow translate(join.left\_op)$;
    $right \leftarrow translate(join.right\_op)$;
    /* Align representations of joined
       variables                              */
    foreach $joined\_var$ in $left.vars \cap right.vars$ do
        $l\_var \leftarrow left.vars.get(joined\_var)$;
        $r\_var \leftarrow right.vars.get(joined\_var)$;
        if $l\_var.repr < r\_var.repr$ then
            $right \leftarrow lower(right, r\_var)$;
        end
        if $r\_var.repr < l\_var.repr$ then
            $left \leftarrow lower(left, l\_var)$;
        end
    end
    $select \leftarrow get\_joined\_select()$;
    $from \leftarrow left + "," + right$;
    $where \leftarrow get\_joined\_where()$;
    return $"SELECT" + select + "FROM" +$
     $from + "WHERE" + where$;
end
```

```
t0.id_named_graph = t1.id_named_graph AND
t0.id_predicate = (Subquery to get id of <
    ex:knows>) AND
t1.id_predicate = (Subquery to get id of <
    ex:likes>)
```

Given the dataset in Table III, the result of the join Query 3 is shown in Table V.

TABLE V. RESULT OF THE JOIN PATTERN QUERY

bs$g	v$s	ng$g	v$liked	v$o
011	1	10	4	2

Here, bs$g is the bitwise AND of the validity bitstrings for the joined quads, indicating the versions in which both patterns are valid.

Consider another example where the join is between a condensed variable and a non-condensed variable.

Query 5: Example SPARQL join of two quad patterns (join between a condensed and a non-condensed variable).

```
?s <ex:knows> ?o ?g .
?g <v:in-version> ?v <ng:Metadata> .
```

In this case, one of the quad patterns includes a variable that is not condensed (i.e., it does not have a validity bitstring associated with it). This scenario requires a different approach to ensure that the join is correctly represented in SQL. In this situation, we need to first flatten the results of the condensed quad pattern to get the capability to join on the non-condensed variable. The associated SQL translation is as follows:

Query 6: SQL translation of the example SPARQL join between a condensed and a non-condensed variable.

```
SELECT left_table.v$s, right_table.v$v,
    left_table.v$g, left_table.v$o
FROM (SELECT flatten_table.v$s, vng.
    id_versioned_named_graph AS v$g,
    flatten_table.v$o FROM (Quad pattern 1)
    flatten_table JOIN versioned_named_graph
    vng ON flatten_table.ng$g = vng.
    id_named_graph AND get_bit(flatten_table.
    bs$g, vng.index_version - 1) = 1)
    left_table
   JOIN (Quad pattern 2) right_table
      ON left_table.v$g = right_table.v$g;
```

Given the dataset from Table III and Table X, the result of the join query 5 is:

TABLE VI. RESULT OF THE JOIN PATTERN QUERY BETWEEN A CONDENSED AND A NON-CONDENSED VARIABLE

v$s	v$o	v$g	v$v
1	6	20	1
1	6	21	2
1	6	22	3
3	1	24	3
2	3	23	2
2	3	24	3

3) Group Operation Translation: Translating a SPARQL group operation into SQL requires aggregating results according to the specified grouping variables. The GroupSQLOperator achieves this by producing SQL with a GROUP BY clause for the grouping variables, along with the necessary aggregate functions for the selected variables. The validity bitstring is incorporated to ensure that the aggregation correctly reflects the versioned nature of the data.

For example, consider the SPARQL query that groups by a variable and counts occurrences:

Query 7: Example SPARQL group operation.

```
SELECT ?o (COUNT(?s) AS ?count)
WHERE {
   ?s <ex:knows> ?o ?g .
}
GROUP BY ?o
```

This translation highlights a concept that has been studied in the context of query rewriting with aggregation. Cohen et al. [16] present methods for rewriting queries with aggregation functions using views, which aligns with our approach [27]. In our translation, the aggregation function counts the number of '1's in the validity bitstring, representing the number of versions in which each object occurs. This group operation is translated into SQL as follows:

Query 8: SQL translation of the example SPARQL group operation.

```
SELECT *, agg0 AS v$count
FROM (SELECT v$o, SUM(bit_count(bs$g)) AS agg0
     FROM (
   Quad Pattern
) gp GROUP BY (v$o)) ext
```

Figure 6: Algorithm of the Group translation

```
Input: Group Operator $gp$
Output: String $sql\_query$
Function TranslateGroup(gp) is
    $subquery \leftarrow translate(gp.sub\_op)$;
    $select \leftarrow$ "";
    $from \leftarrow$ "";
    $group\_by \leftarrow$ "";
    foreach $var$ in $gp.grouped\_vars$ do
        if $var.repr = $ "condensed" then
            $var.repr \leftarrow$ "id";
            $subquery \leftarrow lower(subquery, var)$;
        end
        $select \leftarrow select + var.name$;
    end
    foreach $agg$ in $gp.aggregates$ do
        $select \leftarrow select + translate\_aggregate(agg)$;
    end
    $from \leftarrow$ "(" $+ subquery +$ ")gb";
    $group\_by \leftarrow get\_group\_by(gp.grouped\_vars)$;
    return "SELECT" $+ select +$ "FROM" $+$
      $from +$ "GROUPBY" $+ group\_by$;
end
```

Given the dataset from Table III, the result of the group Query 7 is shown in Table VII.

TABLE VII. RESULT OF THE GROUP OPERATION QUERY

v$o	agg0 = v$count
2	3
1	1
3	2

This result demonstrates that the aggregation correctly counts the total occurrences across all versions, leveraging the bitstring representation to efficiently compute version-aware aggregates.

V. BENCHMARKS

A. Benchmark Setup

To evaluate the performance of QuaQue, we conducted a benchmark comparing our system against Jena, high-performance native RDF triple stores.

The benchmark was conducted on a virtual machine hosted on the PAGODA cloud platform provided by LIRIS [42], offering a stable, high-performance, and controlled environment to ensure the accuracy and reliability of results. We leveraged Docker, a containerization platform, to deploy each component of the benchmark in isolated environments. For evaluation, we used dataset and query workloads from the BEAR benchmarks [43], which supply a diverse range of versioned RDF graphs and queries, allowing assessment of system performance across different data sizes. Utilizing the official BEAR queries ensures our evaluation remains standardized and comparable to previous studies. A fixed memory limit was applied throughout to maintain consistency and comparability of results.

BEAR archives datasets with different versioning policies, including Time-Based (TB) and Change-Based (CB) versioning. For this benchmark, we selected the BEAR-B-day dataset, which employs a Time-Based versioning policy. The BEAR benchmarks focus exclusively on triple pattern and join pattern queries, varying across some index—predicate or predicate and object.

The benchmarking environment used the PAGODA virtual machine provider and Docker as the containerization platform. OpenNebula was used as the cloud management platform to orchestrate the virtual machines and is supervised by KVM (Kernel-based Virtual Machine) hypervisor with a host passthrough configuration. The virtual machine ran Ubuntu 24.04 LTS as the operating system with 500GB of allocated disk space. An AMD EPYC 7443 24-Core Processor (48 threads) @ 2.85 GHz CPUs was utilized for the benchmarking tests. This environment had access to a total of 12 virtual CPU cores and 64GB of RAM.

B. Development

For the relational backend of QuaQue, we utilized **PostgreSQL 15**. To support efficient query answering for any given triple pattern, we implemented a comprehensive indexing strategy inspired by the Hexastore approach [44]. We created composite B-Tree indexes on six permutations of the quad components (Graph, Subject, Predicate, Object). This ensures that the query optimizer can utilize an index-only scan for any combination of bound and unbound variables. The specific index definitions are:

- $(id\_named\_graph, id\_subject, id\_predicate, id\_object)$
- $(id\_named\_graph, id\_subject, id\_object, id\_predicate)$
- $(id\_named\_graph, id\_predicate, id\_object, id\_subject)$
- $(id\_named\_graph, id\_predicate, id\_subject, id\_object)$
- $(id\_named\_graph, id\_object, id\_predicate, id\_subject)$
- $(id\_named\_graph, id\_object, id\_subject, id\_predicate)$

Additionally, there is an index on the `digest` column of the `resource_or_literal` table to speed up lookups of RDF terms.

C. Benchmark Results

The results of the storage consumption and query performance evaluations are presented below and are published in more detail in a Zenodo repository [45].

1) Storage Efficiency: Table VIII compares the disk space usage. Native RDF stores, such as Jena TDB2, are highly optimized for storage, utilizing dictionary encoding to map URIs (Uniform Resource Identifiers) and literals to integers, resulting in a compact footprint (694 MB). QuaQue, despite also employing dictionary encoding, exhibits higher storage consumption (4.7 GB) due to its comprehensive indexing strategy on the relational engine. We also include **QuaQue-flat** in our comparison, which serves as a baseline relational implementation where each quad-version pair is stored separately. This is a known trade-off in relational RDF mapping: trading storage space (via exhaustive indexing) for query flexibility and performance.

TABLE VIII. STORAGE CONSUMPTION COMPARISON.

Dataset	Policy	Tool	Space (MB)
BEAR-B-day	TB	Jena TDB2	**694.39**
BEAR-B-day	TB	QuaQue-flat	6489.91
BEAR-B-day	TB	QuaQue	4707.63

2) Query Execution Time: Table IX reports the execution times for BEAR-B query templates. To ensure accurate and stable measurements, each query was executed 200 times. The first 50 executions were treated as a warm-up phase to mitigate cold-start effects, and the reported results are the average of the final 150 runs. QuaQue demonstrates better performance, performing better than Jena TDB2 in all observed categories (Join, Predicate-Object, and Predicate queries). A Mann-Whitney U-test confirmed the statistical significance of the following results. This suggests that the overhead of the SQL layer is compensated by the efficiency of the PostgreSQL query planner and the availability of covering indexes.

3) Discussion: The experimental results highlight a trade-off between storage efficiency and query performance that warrants critical analysis. As shown in Table VIII, QuaQue's storage footprint is larger than that of Jena TDB2 (4.7 GB vs. 694 MB). This is a direct consequence of our exhaustive indexing strategy, which maintains six B-Tree indexes to cover all possible quad patterns and the index on the digest of the resource values.

However, this investment in storage yields substantial dividends in query execution time.

Table IX shows that QuaQue achieves improvements of approximately 14% for predicate queries, 6% for predicate-object queries, and 14% for join queries compared to Jena TDB2. While these gains are statistically significant, a nearly sevenfold increase in storage for a 10–15% performance improvement represents a trade-off that may not be justified in storage-constrained environments. Figure 7 presents box plots showing the distribution of query execution times. These plots reveal that QuaQue exhibits lower median times and reduced variability compared to Jena TDB2, which may be valuable in latency-sensitive applications where predictability matters.

These results suggest that the primary value of our approach lies not in raw performance gains over highly optimized native RDF stores, but rather in the flexibility of using standard SQL infrastructure and the potential for integration with existing relational ecosystems. The condensed relational model can serve as an effective backend for versioned knowledge graph querying when storage capacity is not a constraint and when interoperability with relational systems is a priority.

VI. CONCLUSION AND FUTURE WORKS

In this paper, we addressed the challenge of efficiently querying versioned Knowledge Graphs. We introduced QuaQue, a system that bridges the gap between Semantic Web standards and Relational Database Management Systems. Our approach relies on a condensed relational model that uses bitstrings to represent the validity of quads across multiple versions, avoiding data redundancy while enabling efficient cross-version querying.

A. Discussions and Future work

1) Benchmark extension: While our current evaluation provides valuable insights, it is limited to basic query patterns. Future work will extend the benchmark to include aggregate queries for a more comprehensive assessment of real-world workloads. Additionally, we plan to compare QuaQue against other versioning policies, such as Change-Based (CB) and Independent Copies (IC), to better understand the trade-offs between storage efficiency and query performance.

2) Extensive implementation: Standard relational algebra lacks support for recursive queries essential for graph analysis, such as path traversal. To address this, we plan to extend our implementation with advanced operators, such as Agrawal's alpha (α) [7]. This extension will enable efficient execution of complex path-finding and recursive queries, significantly enhancing the system's analytical capabilities.

3) Reproducibility: We emphasize that the approach presented in this paper, implemented in the ConVer-G tool, as well as the benchmark used for evaluation, are fully reproducible. The source code, datasets, and experimental scripts are publicly available. To facilitate verification, we provide a containerized environment (Docker) that automates the setup of the database, the loading of datasets, and the execution of the benchmark queries.

In conclusion, QuaQue represents a step towards robust and scalable management of evolving Knowledge Graphs, offering a practical solution for domains requiring complex, concurrent versioning.

ACKNOWLEDGMENTS

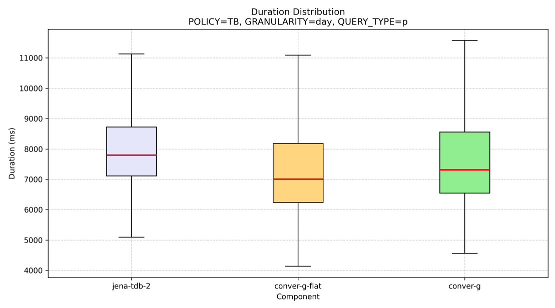
This work, *QuaQue: Design and SQL Implementation of Condensed Algebra for Concurrent Versioning of Knowledge Graphs*, was funded by the Université Claude Bernard Lyon 1 as an ATER position, the IADoc@UDL project, and supported by the LIRIS UMR 5205. We would also like to express our sincere gratitude to the BD team and all members of the Virtual City Project [46] for their insightful feedback, constructive discussions, and continuous support throughout the development of this work. Their expertise and collaboration have been instrumental in shaping the direction and quality of this research.

REFERENCES

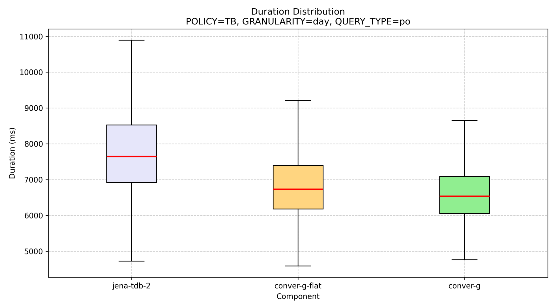
[1] A. e. a. Hogan, "Knowledge graphs", *ACM Computing Surveys*, vol. 54, no. 4, pp. 1–37, 2021.
[2] J. J. Carroll, C. Bizer, P. Hayes, and P. Stickler, "Named graphs, provenance and trust", in *Proceedings of the 14th International Conference on World Wide Web*, 2005, pp. 613–622.
[3] J. P. Gil, E. Coquery, J. Samuel, and G. Gesquière, "Conver-g: Concurrent versioning of knowledge graphs", *arXiv preprint arXiv:2409.04499*, 2024.

TABLE IX. AVERAGE QUERY EXECUTION TIMES (MS).

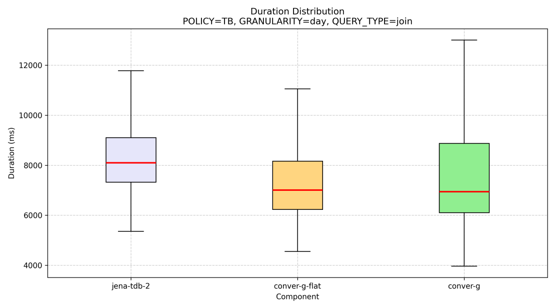
Dataset	Policy	Query type	QuaQue	QuaQue-flat	Jena TDB2
BEAR-B-day	TB	Join	**6936.00**	7006.50	8096.50
BEAR-B-day	TB	P-O	7309.50	**7007.00**	7798.50
BEAR-B-day	TB	P	**6533.50**	6730.50	7644.00



(a) Query Times for predicate index queries



(b) Query Times for predicate-object index queries



(c) Query Times for join queries

Figure 7. Benchmark Results - Query Times

[4] A. e. a. Polleres, “How does knowledge evolve in open knowledge graphs?”, *Transactions on Graph Data and Knowledge*, vol. 1, no. 1, pp. 11–1, 2023.

[5] E. F. Codd, “A relational model of data for large shared data banks”, *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[6] M. A. Roth, H. F. Korth, and A. Silberschatz, “Extended algebra and calculus for nested relational databases”, *ACM Transactions on Database Systems (TODS)*, vol. 13, no. 4, pp. 389–417, 1988.

[7] R. Agrawal, “Alpha: An extension of relational algebra to express a class of recursive queries”, *IEEE Transactions on Software Engineering*, vol. 14, no. 7, pp. 879–885, 2002.

[8] G. Özsoyoğlu, Z. M. Özsoyoğlu, and V. Matos, “Extending relational algebra and relational calculus with set-valued attributes and aggregate functions”, *ACM Transactions on Database Systems (TODS)*, vol. 12, no. 4, pp. 566–592, 1987.

[9] J. D. Ullman, “Implementation of logical query languages for databases”, *ACM Transactions on Database Systems (TODS)*, vol. 10, no. 3, pp. 289–321, 1985.

[10] M. Hofer, D. Obraczka, A. Saeedi, H. Köpcke, and E. Rahm, “Construction of knowledge graphs: State and challenges”, *arXiv preprint arXiv:2302.11509*, 2023.

[11] X. e. a. Jiang, “On the evolution of knowledge graphs: A survey and perspective”, *arXiv preprint arXiv:2310.04835*, 2023.

[12] N. e. a. Abbas, “Knowledge graphs evolution and preservation – a technical report from isws 2019”, *arXiv preprint arXiv:2012.11936*, 2020.

[13] S. Ji, S. Pan, E. Cambria, P. Marttinen, and P. S. Yu, “A survey on knowledge graphs: Representation, acquisition, and applications”, *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 2, pp. 494–514, 2021.

[14] A. Gonzalez-Hevia and D. Gayo-Avello, “Leveraging wikidata’s edit history in knowledge graph refinement tasks”, *arXiv preprint arXiv:2210.15495*, 2022.

[15] H. Zhong, D. Yang, S. Shi, L. Wei, and Y. Wang, “From data to insights: The application and challenges of knowledge graphs in intelligent audit”, *Journal of Cloud Computing*, vol. 13, no. 1, p. 114, 2024.

[16] S. Cohen, W. Nutt, and Y. Sagiv, “Rewriting queries with arbitrary aggregation functions using views”, *ACM Transactions on Database Systems (TODS)*, vol. 31, no. 2, pp. 672–715, 2006.

[17] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh, “Extensible query processing in starburst”, in *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, 1989, pp. 377–388.

[18] E. F. Codd, “Relational completeness of data base sublanguages”, in *Data Base Systems*, Prentice-Hall, 1972, pp. 65–98.

[19] R. Cyganiak, “A relational algebra for sparql”, *Digital Media Systems Laboratory HP Laboratories Bristol, HPL-2005-170*, vol. 35, no. 9, 2005.

[20] D. e. a. Calvanese, “Ontop: Answering sparql queries over relational databases”, *Semantic Web*, vol. 8, no. 3, pp. 471–487, 2016.

[21] I. Düntsch, "Relation algebras and their application in temporal and spatial reasoning", *Artificial Intelligence Review*, vol. 23, no. 4, pp. 315–357, 2005.

[22] Z. e. a. Li, "Temporal knowledge graph reasoning based on evolutional representation learning", in *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2021, pp. 408–417.

[23] J. F. Allen, "Maintaining knowledge about temporal intervals", *Communications of the ACM*, vol. 26, no. 11, pp. 832–843, 1983.

[24] A. G. Cohn, B. Bennett, J. Gooday, and N. M. Gotts, "Qualitative spatial representation and reasoning with the region connection calculus", *GeoInformatica*, vol. 1, no. 3, pp. 275–316, 1997.

[25] R. Taelman, H. Takeda, M. Vander Sande, and R. Verborgh, "The fundamentals of semantic versioned querying", in *SSWS2018, the 12th International Workshop on Scalable Semantic Web Knowledge Base Systems*, 2018, pp. 1–14.

[26] R. Conradi and B. Westfechtel, "Version models for software configuration management", *ACM Computing Surveys (CSUR)*, vol. 30, no. 2, pp. 232–282, 1998.

[27] J. P. Gil, E. Coquery, J. Samuel, and G. Gesquiere, "Condensed representation of rdf and its application on graph versioning", *arXiv preprint arXiv:2506.21203*, 2025.

[28] G. Taentzer, C. Ermel, P. Langer, and M. Wimmer, "A fundamental approach to model versioning based on graph modifications: From theory to implementation", *Software & Systems Modeling*, vol. 13, no. 1, pp. 239–272, 2014.

[29] M. Graube, S. Hensel, and L. Urbas, "R43ples: Revisions for triples", in *Proceedings of the 1st Workshop on Linked Data Quality co-located with 10th International Conference on Semantic Systems (SEMANTiCS 2014)*, 2014.

[30] M. Völkel, W. Winkler, Y. Sure, S. R. Kruk, and M. Synak, "Semversion: A versioning system for rdf and ontologies", in *Proceedings of the 2nd European Semantic Web Conference (ESWC)*, 2005, pp. 193–202.

[31] M. e. a. Vander Sande, "R&wbase: Git for triples", *LDOW*, vol. 996, 2013.

[32] J. P. Gil, E. Coquery, J. Samuel, and G. Gesquiere, *Conver-g: Concurrent versioning of knowledge graphs*, arXiv:2409.04499 [cs.DB], 2024. arXiv: 2409.04499 [cs.DB].

[33] J. Anderson and A. Bendiken, "Transaction-time queries in dydra", *MEPDaW/LDQ@ESWC*, vol. 1585, pp. 11–19, 2016.

[34] S. Gao, J. Gu, and C. Zaniolo, "Rdf-tx: A fast, user-friendly system for querying the history of rdf knowledge bases", in *Proceedings of the 19th International Conference on Extending Database Technology (EDBT)*, 2016, pp. 269–280.

[35] A. Cerdeira-Pena, A. Fariña, J. D. Fernández, and M. A. Martínez-Prieto, "Self-indexing rdf archives", in *2016 Data Compression Conference (DCC)*, 2016, pp. 526–535. DOI: 10.1109/DCC.2016.40

[36] T. Neumann and G. Weikum, "X-rdf-3x: Fast querying, high update rates, and consistency for rdf databases", *Proceedings of the VLDB Endowment*, vol. 3, no. 1–2, pp. 256–263, Sep. 2010. DOI: 10.14778/1920841.1920877

[37] K. Kulkarni and J.-E. Michels, "Temporal features in sql:2011", *SIGMOD Record*, vol. 41, no. 3, pp. 34–43, Oct. 2012, ISSN: 0163-5808. DOI: 10.1145/2380776.2380786

[38] N. Arndt, P. Naumann, N. Radtke, M. Martin, and E. Marx, "Decentralized collaborative knowledge management using git", *Journal of Web Semantics*, vol. 54, pp. 29–47, 2019, Managing the Evolution and Preservation of the Data Web, ISSN: 1570-8268. DOI: https://doi.org/10.1016/j.websem.2018.08.002

[39] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias, "Binary rdf representation for publication and exchange (hdt)", *Journal of Web Semantics*, vol. 19, pp. 22–41, 2013.

[40] I. Cuevas and A. Hogan, "Versioned queries over rdf archives: All you need is sparql?", in *MEPDaW@ ISWC*, 2020, pp. 43–52.

[41] T. Yang, Y. Wang, L. Sha, J. Engelbrecht, and P. Hong, "Knowledgebra: An algebraic learning framework for knowledge graph", *Machine Learning and Knowledge Extraction*, vol. 4, no. 2, pp. 432–445, 2022.

[42] LIRIS, "PAGODA cloud platform", Accessed: 2025.12.01. [Online]. Available: https://projet.liris.cnrs.fr/pagoda/latest/

[43] J. D. Fernández, J. Umbrich, A. Polleres, and M. Knuth, "BEAR – benchmark for rdf archive versioning systems", Accessed: 2025.12.01. [Online]. Available: https://aic.ai.wu.ac.at/qadlod/bear.html

[44] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: Sextuple indexing for semantic web data management", *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 1008–1019, 2008.

[45] J. P. Gil, E. Coquery, J. Samuel, and G. Gesquiere, "Quaque benchmark results", Accessed: 2025.12.01. [Online]. Available: https://zenodo.org/records/17780464

[46] VCity Team, "Virtual city project", Accessed: 2025.12.01. [Online]. Available: https://projet.liris.cnrs.fr/vcity/

Appendix

A. QuaQue: A Queryable Versioned Quad Store

1) Sample Dataset: The metadata about versions and graphs is represented in Table X.

TABLE X. Representation of the metadata (`metadata`)

id_subject	id_predicate	id_object
20	8	1
20	9	10
21	8	2
21	9	10
22	8	3
22	9	10
23	8	2
23	9	11
24	8	3
24	9	11

The following Table XI provides the dictionary mapping for resources and literals.

TABLE XI. Dictionary Representation of the resources or literals (resource_or_literal)

id_resource_or_literal	name	type
1	:alice	resource
2	:bob	resource
3	:carol	resource
4	"pizza"	literal
5	"sushi"	literal
6	ex:knows	resource
7	ex:likes	resource
8	v:in-version	resource
9	v:version-of	resource
10	:g1	resource
11	:g2	resource
20	:vng1	resource
21	:vng2	resource
22	:vng3	resource
23	:vng4	resource
24	:vng5	resource

On UNIQUE KEYS in SQL/JSON Implementations - Experimenting with JSON support in DBMS products

Martti Laiho
DBTechNet
www.dbtechnet.org
formerly at Haaga Helia
email: martti.laiho@gmail.com

Fritz Laux (retired)
Fakultät Informatik
Reutlingen University
D-72762 Reutlingen, Germany
email: fritz.laux@reutlingen-university.de

***Abstract*—The popularity of the JavaScript Object Notation (JSON) data format has led to its early support by major database vendors. Relational database systems added a JSON data type and implemented functions for manipulation and querying JSON data. The ISO/IEC Standardization lagged behind this development and only recently specified the update and maintenance of JSON data. Now we have a Babel of confusion with different syntax for most major relational systems. This complicates and hinders reliable JSON data exchange, SQL portability and knowledge transfer between developers. In the context of relational databases with strong consistency support, JSON requires reliable and deterministic data retrieval behavior. This paper we make the reality check and investigate how popular relational database products (Db2 for Linux/Unix/Windows (LUW), Oracle 23ai, SQL Server, PostgreSQL, and MariaDB) handle JSON data manipulation and how they conform to the ISO/IEC 19075-6:2021 and ISO/IEC 9075-2:2023 standards.**

***Keywords*-ISO SQL/JSON; UNIQUE KEYS; data retrieval; data update;**

I. INTRODUCTION

JSON [1] has rapidly become a standard for data representation and exchange due to its simplicity. Initially designed for data interchange in NoSQL systems like MongoDB, its integration into Relational Database Systems (RDBMS) has grown in popularity. The adoption of JSON allows RDBMSs to handle semi-structured data, expanding their versatility to accommodate more dynamic data models. A JSON document is hierarchically structured consisting of primitive (scalar) value types (number, boolean, string), JSON literals (true, false, null) and structured types (array, JSON object) which may be nested to form a data hierarchy.

The following example is a valid JSON document:

```
{
  "Image": {
        "Width": 800,
        "Height": 600,
        "Title": "View from 15th Floor",
        "Thumbnail": {
            "Url":"http://www.example.com/img4819",
            "size":"125x100"
         },
        "Animated" : false,
        "IDs": [116, 943, 234, 38793]
    }
}
```

It consists of one JSON object "Image" (case sensitive!) and 6 members. Each object is enclosed in curly brackets and consists of a "name" and a "value" separated by a colon. In the IETF RFC 8259 [1] specification the "name" acts as a key for the object. This is why the ISO/IEC standard calls it "key" and we will adopt this convention because our paper deals with databases. The value of the key "Thumbnail" is again a JSON object. The value of "IDs" is an array with 4 numbers.

According to the JSON specification RFC 8259 "The names (keys) within an object SHOULD be unique" and continues ".. When the names (keys) within an object are not unique, the behaviour of software that receives such an object is unpredictable." The RFC does not deny the possibility of non-unique keys of object keys, and we need to remember that JSON structures in general are applied for data exchange between systems. However, the ISO/IEC 19075 SQL/JSON specification [8] for the RDBMS context needs to be stricter than the generic JSON specification of RFC 8259, as we will see later.

However, the possibility of non-unique keys has consequences apart from software being unable to process the JSON content properly: (1) Because the member elements of a JSON object can be found only by key names, it is not possible to identify a single member if two or more members have the same key on the same structural level, for example: { `"key1":"val", "key1":"val"` }. Because the members of a JSON object are unordered, we cannot refer to the "first" or "second" member. However, it is possible to have the same key name on different levels of the JSON object like `{ "key1": { "key1":"val1" } }`. In this case, the value of `"key1"` on the second level is accessed as `"key1"."key1"`. (2) The processing software has to deal with two possibilities, either the access addresses one single member or it receives a collection of elements where the ordering has no meaning. In the latter case, the collection must be processed as a whole in order to yield correct results.

SQL/JSON extends the relational model into a new hybrid model, whose implementation should preserve RDBMS functionality including transaction, strict consistency, and the prevention of "unpredictable behaviour" when processing data.

A. Contribution

We present systematic tests for Db2 for LUW, Oracle 23ai, SQL Server, PostgreSQL, and MariaDB and look how they behave when (1) attempting to insert duplicate keys, and (2) retrieving and updating objects with duplicate keys. To the best of our knowledge, no scientific publications address how SQL/JSON databases behave when there are duplicate JSON keys in a document. The paper provides advice how to prevent unintended behaviour or even inconsistent JSON data processing for the tested database systems.

B. Structure of the Paper

The paper is structured as follows: Section II discusses literature on the JSON update framework for RDBMS. In Section III we present and discuss the behaviour of five databases with reference to retrieval and manipulation of duplicate JSON objects. Section IV discusses the results and Section V concludes with a critique of the situation and recommends a work-around for developers to avoid these problems.

II. Related Work

Most papers on handling SQL/JSON are promotional work about implementations of JSON data type (e.g., JSON Binary (JSONB) for PostgreSQL [2], Optimized JSON (OSON) for Oracle [3]) and model control features (e.g., key uniqueness control for Db2 [4]) or JSON functions and operators.

Petkovic [5] presents a comprehensive description based on the proposed ISO SQL/JSON standard. At that time (2017) only a "light weight" functionality was defined, lacking native JSON data type and update functions for JSON data. In 2020 the big relational database vendors had already implemented their own JSON data types (like BSON, OSON, JSONB) before the ISO standard was ready in 2021. The databases also provided a set of manipulation functions for their JSON data types. The internal representation of the JSON data type is not relevant for our experiments as it affects only the performance, storage space needed, and some JSON function behavior. The JSON standard and its external representation is always text based. Its functionality was investigated by Petkovic [6]. He tested INSERT, DELETE, and REPLACE operations. The INSERT distinguished the cases of inserting an object (key:value pair) and inserting an array element before and after a certain element. The DELETE distinguished two cases: delete an object and delete the n^{th} element of an array. The REPLACE made the same distinction as the DELETE. His observation was that SQL Server, PostgreSQL, and MySQL fully supported the above modification operations but with a very different syntax and different functions. Petkovic's paper does not consider the case of duplicate JSON members.

III. JSON data manipulation tests

In this section, we look at Db2 for LUW[1], Oracle 23ai[2], SQL Server[3], PostgreSQL[4], and MariaDB[5]. Some of the tested products allow duplicate JSON object members and others require unique keys. We place special focus on both situations when testing the SELECT, UPDATE, JSON_VALUE, and JSON_MODIFY functionality.

A. The Standard for SQL/JSON

JSON was first proposed as a data exchange format in 2014, and RFC8259 recommended that the JSON object members SHOULD have unique keys as mentioned in the Introduction. ISO/IEC 21778:2017 [7] and the standard of 2021, ISO/IEC19075-6:2021 [8], are even more relaxed and do not mention unique keys: "The JSON syntax does not impose any restrictions on the strings used as names, does not require that name strings be unique, and does not assign any significance to the ordering of name/value pairs" [7]. This is unfortunate and allows the developers to implement JSON retrieval functions in an incompatible way. The latest ISO standard of 2023 [9][10] introduces the options WITH UNIQUE KEYS and WITHOUT UNIQUE KEYS but does not define a default. On the implementation side, as we will see, database developers mostly decide for the retrieval of JSON data to ignore duplicate keys and return either the first or last matching object without any notice. This makes data exchange problematic because duplicates may be lost. Duplicates can be avoided if the insert statement uses the clause "WITH UNIQUE KEYS".

The following tests are based on the study "JSON Data Maintenance" [11] conducted by the main author of this paper. An extensive study considering the general support of JSON data in relational databases is found in the document "JSON on RDBMS Databases" [12] by the same author.

B. Test set-up

We created the following table for our test set-up, using default values for each RDBMS product tested:

```
CREATE TABLE T1 (
K  INTEGER NOT NULL PRIMARY KEY,
J  <json | clob | blob | jsonb> );
-- depending on the RDBMS
```

We then attempted to insert JSON objects with duplicate key names (here: `"duplica"`). The exact syntax for the JSON column depends on the SQL dialect of the database:

```
-- generic insert statement
INSERT INTO T1 (K, J) VALUES
(3, '{ "duplica":"First", "duplica":"Second",
      "duplica":"Third","duplica":"Last"}');
-- use syntax for specific product
```

C. Db2 for LUW

In the following INSERT statement the function JSON_TO_BSON converts the external, text based JSON document into an IBM specific binary format, called BSON.

```
db2 => INSERT INTO T1 (K, J) VALUES
db2 (cont.) => (3, JSON_TO_BSON('{ "duplica":
 "First", "duplica":"Second", "duplica":
 "Third", "duplica":"Last"}'));
DB21034E  The command was processed as an SQL
 statement because it was not a valid Command
```

[1]Version 12.1.1 Community Edition
[2]Version 23ai Free
[3]2017 Express Edition
[4]Version 17
[5]Version 11.8.2

```
 Line Processor command.  During SQL
 processing it returned:
SQL16406N  JSON data has non-unique keys.
db2 =>
```

Db2 apparently uses WITH UNIQUE KEYS by default. However, the DB2 version for the *IBM I* operating system has now added the clauses WITHOUT UNIQUE KEYS and WITH UNIQUE KEYS to the JSON publishing functions and changed its default to WITHOUT UNIQUE KEYS. This is worrying because the SQL/JSON specification should not be implemented without criticism when there is a risk of breaking the content integrity of Db2. Having WITH UNIQUE KEYS as default would definitely be the better choice.

D. Oracle 23ai

Oracle recommends using the CHECK constraint “IS JSON WITH UNIQUE KEYS” for text-based JSON columns to avoid inconsistent contents. This constraint is automatically activated for columns based on Oracle’s native JSON data type, called OSON, which we are using. So, the effect is the same as using the WITH UNIQUE KEYS clause.

```
SQL> -- Duplicate object members test:
SQL> INSERT INTO T1 (K, J) VALUES
  2* (3, '{ "duplica":"First", "duplica":"Second",
   "duplica":"Third","duplica":"Last"}');
Error starting at line : 1 in command -
INSERT INTO T1 (K, J) VALUES
(3, '{ "duplica":"First", "duplica":"Second",
 "duplica":"Third","duplica":"Last"}')
Error at Command Line : 1 Column : 13
Error report -
SQL Error: ORA-40473: duplicate key names 'duplica'
in JSON object
JZN-00007: Object member key 'duplica' is not unique
Help: https://docs.oracle.com/error-help/db/ora-
40473/40473. 00000 -  "duplicate key names '%s' in
JSON object"
*Cause:    The provided JavaScript Object Notation
(JSON) data had duplicate key names in one object.
```

Oracle reports a verbose error message when trying to insert duplicate JSON members.

E. SQL Server

The SQL Server XE does not recognize the constraint IS JSON WITH UNIQUE KEYS. Let’s test what happens when we enter a JSON document having duplicate member keys:

```
-- Duplicate object members test:
BEGIN TRANSACTION;
INSERT INTO T1 (K, J) VALUES
(3, '{ "duplica":"First", "duplica":"Second",
 "duplica":"Third","duplica":"Last"}');
(1 row affected)
 -- The following LEFT function is only used
 -- to limit the column width
SELECT LEFT(K, 4) AS K, LEFT(JSON_VALUE(J,
'$.duplica'),  10) AS Duplica
FROM T1 WHERE K=3;
K    Duplica
---- ----------
3    First
(1 row affected)

-- But let's see them all
SELECT J FROM T1 WHERE K=3;
J
----------------------------------------
{ "duplica":"First", "duplica":"Second",
  "duplica":"Third","duplica":"Last"}
(1 row affected)

-- Let's remove the 'first' member. There is
-- no real REMOVE, the member is set to NULL
UPDATE T1 SET J = JSON_MODIFY(J, '$.duplica', NULL)
WHERE K=3;
(1 row affected)

SELECT LEFT(K, 4) AS K, JSON_VALUE(J, '$.duplica')
AS Duplica FROM T1 WHERE K=3;
K    Duplica
---- ----------
3    Second
(1 row affected)

-- OK, let's see them all
SELECT J FROM T1 WHERE K=3;
J
-----------------------------------------
{ "duplica":"Second", "duplica":"Third",
  "duplica":"Last"}
(1 row affected)
```

Duplicate keys cannot be prevented in SQL Server, as it would appear that the “WITH UNIQUE KEYS” clause has not been implemented yet.

F. PostgreSQL

Using the following script, we experiment how duplicate object members behave in a PostgreSQL transaction.

```
testdb=> BEGIN;
BEGIN
testdb=> INSERT INTO T1 (K, J) VALUES
(3, '{ "duplica":"First", "duplica":"Second",
  "duplica":"Third","duplica":"Last"}');
INSERT 0 1

-- Next, select the duplica members
testdb=> SELECT J->'duplica' AS Duplica FROM T1
  WHERE K=3;
 duplica
---------
 "Last"
(1 row)
```

After the insert we might expect that PostgreSQL has accepted the duplicates. But, the immediately following query displayed only the last duplica. Listing all members in the following query does not bring up other duplicate members.

```
-- select each member
testdb=> (SELECT (JSONB_EACH(J)).* FROM T1
  WHERE K=3);
   key   | value
---------+--------
 duplica | "Last"
(1 row)

-- This pure SQL query confirms the case
testdb=*> (SELECT J FROM T1 WHERE K=3);
          j
---------------------
 {"duplica": "Last"}
(1 row)
```

Selecting the whole JSON document reveals that only the last duplicate has been stored. All other duplicates have been dropped. As consequence there is nothing left if the object `"duplica":"Last"` gets removed:

```
testdb=> UPDATE T1 SET J = J - 'duplica' WHERE K=3;
UPDATE 1
testdb=>(SELECT(JSONB_EACH(J)).* FROM T1 WHERE K=3);
 key | value
-----+-------
(0 rows)
```

The reason for this is that already during the INSERT of the JSONB typed column the last `"duplica"` wins while others are removed automatically without warnings. The PostgreSQL manual states that in case of duplicate object members only the last one is stored. But, it does not mention that this is done silently and no exception is raised. Is this the service we want? Note that we will silently lose the information in the value parts of those automatically deleted members due to accidentally having the same key names! The current version of PostgreSQL supports the WITH UNIQUE KEYS constraint clause only for the publishing function JSON_OBJECT().

G. MariaDB

MariaDB does not recognize the constraint IS JSON WITH UNIQUE KEYS. Let's test what happens when we enter a JSON document having duplicate member keys:

```
MariaDB [testdb]> -- Duplicate object members test:
MariaDB [testdb]> START TRANSACTION;
Query OK, 0 rows affected (0.000 sec)

MariaDB [testdb]> INSERT INTO T1 (K, J) VALUES
   -> (3, '{ "duplica":"First", "duplica":"Second",
      "duplica":"Third","duplica":"Last"}');
Query OK, 1 row affected (0.000 sec)
```

MariaDB accepts duplicates without complaining because the default for inserts is "WITHOUT UNIQUE KEYS". This can be verified with the following query:

```
MariaDB [testdb]> SELECT J FROM T1 WHERE K=3;
+----------------------------------------------+
| J
+----------------------------------------------+
| { "duplica":"First", "duplica":"Second",
 "duplica":"Third","duplica":"Last"}          |
+----------------------------------------------+
1 row in set (0.000 sec)

-- using the JSON_VALUE function returns only
-- the first member, i.e. a single member
MariaDB [testdb]> SELECT LEFT(K, 4) AS K, LEFT(
 JSON_VALUE(J, '$.duplica'), 10) AS Duplica
    -> FROM T1 WHERE K=3;
+------+---------+
| K    | Duplica |
+------+---------+
| 3    | First   |
+------+---------+
1 row in set (0.000 sec)

MariaDB [testdb]> -- How about others if the
                     "duplica" get removed
MariaDB [testdb]> UPDATE T1 SET J = JSON_
         REMOVE(J, '$.duplica') WHERE K=3;
Query OK, 1 row affected (0.000 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Only the the "first" duplicate member was removed as the following queries reveal.

```
MariaDB [testdb]> SELECT LEFT(K, 4) AS K, LEFT(
  JSON_VALUE(J, '$.duplica'), 10) AS Duplica
    -> FROM T1 WHERE K=3;
+------+---------+
| K    | Duplica |
+------+---------+
| 3    | Second  |
+------+---------+
1 row in set (0.000 sec)

MariaDB [testdb]> SELECT J FROM T1 WHERE K=3;
+----------------------------------------------+
| J                                            |
+----------------------------------------------+
| {"duplica": "Second", "duplica": "Third",
  "duplica": "Last"}                           |
+----------------------------------------------+
1 row in set (0.000 sec)
```

These tests show that the current MariaDB version stores duplicates and the JSON_VALUE and JSON_REMOVE only retrieve resp. remove the first member of duplicate objects.

IV. DISCUSSION

We see quite different behaviour for the five database products. The default configuration of Db2 and Oracle is WITH UNIQUE KEYS making it impossible to insert duplicate keys. This makes it simple to select a particular key and get a unique result. SQL Server, PostgreSQL, and MariaDB support only WITHOUT UNIQUE KEYS and therefore allow duplicates with quite different behaviour. PostgreSQL silently ignores all duplicates except the last one during the insert. It effectively forces the JSON members to have unique KEYS and loses possible duplicates. This is intolerable because users are not aware of dropping all members except the last one. There is a way to fix this flaw in the insert statement. The missing functionality WITH UNIQUE KEYS can be created as a PL/pgSQL function used as CHECK constraint for the JSON column. The CHECK constraint can be activated when the table is created. Example: `CREATE TABLE T (K SERIAL PRIMARY KEY, J JSONB NOT NULL, CONSTRAINT with_unique_keys CHECK (unique_keys(J));`

SQL Server and MariaDB store all members including the duplicates. The problem with duplicates arises when you retrieve duplicate values. Only the first duplicate is retrieved by the JSON_VALUE function ignoring all other duplicates because this function can only return a single member. In fact, the function returns the wrong result. The problem with the implementation of this function has already been mentioned in RFC8259 in 2016 but until now nothing was changed. As consequence the JSON object must be handled outside the database using string manipulation and updated as a whole again. This can be achieved with the JSON_QUERY(expression [, path]) function which retrieves all objects matching the path. The result is a string that can be manipulated and subsequently updated in the database with the JSON_MODIFY function.

TABLE I. BEHAVIOUR OF THE TESTED DATABASES

DBMS	(1) WITH UNIQUE KEYS	(2) WITHOUT UNIQUE KEYS	default	insert duplica	retrieves duplica	comment
Db2 LUW 12.1.1 Community Edition	yes	yes	(1)	no	n/a	
Oracle Version 23ai Free	yes	yes	(1)	no	n/a	
SQL Server 2017 Express Edition	no	yes	(2)	yes	first	other duplicas hidden by JSON_VALUE
PostgreSQL Version 17	n/a	n/a	(2)	no	last	other duplicas silently dropped
MariaDB Version 11.8.2	n/a	n/a	(2)	yes	first	other duplicas hidden by JSON_VALUE

V. CONCLUSION

We have presented test scenarios with special emphasis on unique JSON object keys for five popular DBMS supporting the SQL/JSON extension.

The tests show that these five database products behave quite differently making the data exchange between different products error-prone. The results are summarized in Table I. Great care is required to insist that the database uses the option WITH UNIQUE KEYS if possible.

Our recommendations is to always use the WITH UNIQUE KEYS options when exchanging JSON documents between systems and particularly when storing JSON documents in a database. If this option is not supported as in SQL Server and MariaDB unique keys can be forced with check constraints to protect against duplicate keys in JSON documents.

If the JSON data inevitably contain duplicate keys then the JSON_VALUE function is not useful as it returns only a scalar value even when there are duplicate JSON object keys. We recommend to use the JSON_QUERY function instead to retrieve the duplicates and manipulate then the result outside the database. For example, the duplicate object member keys (e.g. {`"duplica":"first", "duplica":"second", "duplica":"third"`} could be transformed into an JSON object array {`"duplica": ["first", "second", "third"]`}.

PostgreSQL is a special case as it silently drops all duplicates except the last one. This lulls the user into a false sense of security. We hope that the next version of PostgreSQL will address the above issue and provide a standard conform solution.

In a future version of the ISO/IEC SQL/JSON standard we hope to see fuctionality that can correctly handle the situation of duplicate JSON object member keys.

REFERENCES

[1] T. Bray (ed.), “The JavaScript Object Notation (JSON) Data Interchange Format”, [Online]. URL: https://dl.acm.org/doi/pdf/10.17487/RFC8259, Dec. 2017, (Accessed Jan 24, 2026)

[2] G. Turutin and M. Puzevich, “PostgreSQL JSONB-based vs. Typed-column Indexing: A Benchmark for Read Queries”, IEEE Dataport, October 29, 2025, doi:10.21227/fxws-3a11

[3] Z. H. Liu et al., “Native JSON Datatype Support: Maturing SQL and NoSQL convergence in Oracle Database”. Proc. VLDB Endow. 13(12): 3059-3071 (2020)

[4] N.N., IBM, “Uniqueness controls for key names”, Apr. 2023, [Online]. URL: https://www.ibm.com/support/pages/json-uniqueness-controls-key-names, (Accessed Jan 24, 2026)

[5] D. Petkovic, “SQL/JSON Standard: Properties and Deficiencies”, Datenbank-Spektrum, Volume 17, pp 277-287, Oct 24 2017, [Online]. URL: https://link.springer.com/content/pdf/10.1007/s13222-017-0267-4.pdf, (Accessed Jan 24, 2026)

[6] D. Petkovic, “Implementation of JSON Update Framework in RDBMSs”, International Journal of Computer Applications Foundation of Computer Science (FCS), NY, USA, Volume 177 - Number 37, 2020, DOI: 10.5120/ijca2020919881

[7] N. N., ISO/IEC 21778:2017 “Information technology - The JSON data interchange syntax”, [Online]. URL: https://www.iso.org/standard/71616.html , (Accessed Jan 24, 2026)

[8] N. N., ISO/IEC 19075-6:2021 “Information technology — Guidance for the use of database language SQL Part 6: Support for JSON”, Edition 1, [Online]. URL: https://www.iso.org/standard/78937.html, (Accessed Jan 24, 2026)

[9] N.N., ISO/IEC 9075-2:2023 “Information technology — Database languages SQLPart 2: Foundation (SQL/Foundation)”, [Online]. URL: https://www.iso.org/standard/76584.html, Edition 6, 2023, URL: https://www.iso.org/standard/76584.html, (Accessed Jan 24, 2026)

[10] P. Eisentraut, “SQL:2023 is finished: Here is what’s new”, [Online]. URL: https://peter.eisentraut.org/blog/2023/04/04/sql-2023-is-finished-here-is-whats-new, Apr. 4, 2023, (Accessed Jan 24, 2026)

[11] M. Laiho, “JSON Data Maintenance”, Appendix 1, Nov. 2025, [Online]. URL: https://dbtechnet.org/wp-content/uploads/JSON-Data-Maintenance.pdf, (Accessed Jan 27, 2026)

[12] M. Laiho, “JSON on RDBMS Databases”, Sept. 2025, [Online]. URL: https://drive.google.com/file/d/1eoU9KIQrMwI0YkdPdPm6jIgi-0hRffNI/view, (Accessed Jan 24, 2026)