Proceedings of the VLDB Endowment

Volume 19, No. 1 – September 2025

Editors in Chief:

Yanlei Diao and Xiaokui Xiao

Associate Editors:

Alkis Polyzotis, Alkis Simitsis, Andrew Pavlo, Angela Bonifati, Anh Dinh, Antonis Deligiannakis, Ashraf Aboulnaga, Badrish Chandramouli, Bin Cui, Bogdan Cautis, Ce Zhang, Elena Ferrari, Evaggelia Pitoura, Fatemeh Nargesian, Guoliang Li, Hong Cheng, Immanuel Trummer, Jana Giceva, Jennie Rogers, Jian Pei, Jianliang Xu, Karima Echihabi, Katja Hose, Khuzaima Daudjee, Kunsoo Park, Kyuseok Shim, Laks Lakshmanan, Lei Cao, Lei Chen, Li Xiong, Mauro Sozio, Meihui Zhang, Melanie Herschel, Michael Böhlen, Nan Tang, Niv Dayan, Norman May, Raul Castro Fernandez, Raymond Chi-Wing Wong, Reynold Cheng, Ruben Mayer, Semih Salihoglu, Shimin Chen, Sibo Wang, Sourav S Bhowmick, Spyros Blanas, Steven Whang, Sudipto Das, Umar Farooq Minhas, Wei Wang, Wook-Shin Han, Yannis Velegrakis, Yanyan Shen, Yi Chen, Yuncheng Wu, Zhifeng Bao, Zi Huang

Publication Editors:

Andrea Mauri, Giovanni Simonini, Siqiang Luo, Stefan Halfpap, Subhadeep Sarkar

PVLDB – Proceedings of the VLDB Endowment

Volume 19, No. 1, September 2025.

All papers published in this issue will be presented at the 52nd International Conference on Very Large Data Bases, Boston, MA, USA, 2026.

Copyright 2025 VLDB Endowment

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.

Volume 19, Number 1, September 2025

Pages i – viii and 1 - 83

ISSN 2150-8097

Available at: http://www.pvldb.org and https://dl.acm.org/journal/pvldb

TABLE OF CONTENTS

Front Matter

Copyright Notice i
Table of Contents ii
PVLDB Organization and Review Board – Vol. 19 iii

Research Papers

FlatStor: An Efficient Embedded-Index Based Columnar Data Layout for Multimodal Data Workloads 1
Chi Zhang, Shihao Zhang, Yunfei Gu, Chentao Wu, Jie Li, Qin Zhang, Xusheng Chen, Jie Meng

TIMEST: Temporal Information Motif Estimator Using Sampling Trees 15
Yunjie Pan, Omkar Bhalerao, C. Seshadhri, Nishil Talati

ConANN: Conformal Approximate Nearest Neighbor Search 29
Sonia Horchidan, Fabian Zeiher, Henrik Boström, Paris Carbone

RayDB: Building Databases with Ray Tracing Cores 43
Xuri Shi, Kai Zhang, X. Sean Wang, Xiaodong Zhang, Rubao Lee

NeutronCloud: Resource-Aware Distributed GNN Training in Fluctuating Cloud Environments 56
Mingyi Cao, Chunyu Cao, Yanfeng Zhang, Zhenbo Fu, Xin Ai, Qiange Wang, Yu Gu, Ge Yu

CLaP - State Detection from Time Series 70
Arik Ermshaus, Patrick Schäfer, Ulf Leser

PVLDB ORGANIZATION AND REVIEW BOARD - Vol. 19

Editors in Chief of PVLDB
Yanlei Diao (Ecole Polytechnique and Amazon Web Services)
Xiaokui Xiao (National University of Singapore)

Associate Editors of PVLDB
Alkis Polyzotis (Databricks)
Alkis Simitsis (Athena Research Center)
Andrew Pavlo (Carnegie Mellon University)
Angela Bonifati (Univ. of Lyon)
Anh Dinh (Deakin University)
Antonis Deligiannakis (Technical University of Crete)
Ashraf Aboulnaga (University of Texas at Arlington)
Badrish Chandramouli (Microsoft Research)
Bin Cui (Peking University)
Bogdan Cautis (University of Paris-Saclay)
Ce Zhang (University of Chicago)
Elena Ferrari (University of Insubria, Varese)
Evaggelia Pitoura (Univ. of Ioannina)
Fatemeh Nargesian (University of Rochester)
Guoliang Li (Tsinghua University)
Hong Cheng (Chinese University of Hong Kong)
Immanuel Trummer (Cornell University)
Jana Giceva (TU Munich)
Jennie Rogers (Northwestern University)
Jian Pei (Duke University)
Jianliang Xu (Hong Kong Baptist University)
Karima Echihabi (Mohammed VI Polytechnic University)
Katja Hose (TU Wien)
Khuzaima Daudjee (University of Waterloo)
Kunsoo Park (Seoul National University)
Kyuseok Shim (Seoul National University)
Laks Lakshmanan (The University of British Columbia)
Lei Cao (University of Arizona)
Lei Chen (Hong Kong University of Science and Technology)
Li Xiong (Emory University)
Mauro Sozio (Luiss University of Rome)
Meihui Zhang (Beijing Institute of Technology)
Melanie Herschel (Nanyang Technological University)
Michael Böhlen (University of Zurich)
Nan Tang (HKUST (GZ))
Niv Dayan (University of Toronto)
Norman May (SAP SE)
Raul Castro Fernandez (The University of Chicago)
Raymond Chi-Wing Wong (Hong Kong University of Science and Technology)
Reynold Cheng (The University of Hong Kong, China)
Ruben Mayer (University of Bayreuth)
Semih Salihoglu (University of Waterloo)
Shimin Chen (Chinese Academy of Sciences)
Sibo Wang (The Chinese University of Hong Kong)
Sourav S Bhowmick (Nanyang Technological University)
Spyros Blanas (The Ohio State University)
Steven Whang (KAIST)
Sudipto Das (Amazon Web Services)
Umar Farooq Minhas (Apple)
Wei Wang (ByteDance)
Wook-Shin Han (POSTECH)
Yannis Velegrakis (Utrecht University)
Yanyan Shen (Shanghai Jiao Tong University)
Yi Chen (NJIT)
Yuncheng Wu (Renmin University of China)
Zhifeng Bao (The University of Queensland)
Zi Huang (University of Queensland)

Publication Editors
Andrea Mauri (Lyon 1 University)
Giovanni Simonini (University of Modena and Reggio Emilia)
Siqiang Luo (Nanyang Technological University)
Stefan Halfpap (TU Berlin)
Subhadeep Sarkar (Brandeis University)

PVLDB Managing Editor
Jun Yang (Duke University)

PVLDB Advisory Board
Alexandra Meliou (University of Massachusetts Amherst)
Divesh Srivastava (AT&T Labs - Research)
Fatma Ozcan (Google)
Felix Naumann (HPI)
Katja Hose (Technische Universität Wien)
Lei Chen (Hong Kong University of Science and Technology)
Matthias Boehm (Technische Universität Berlin)
Meihui Zhang (Beijing Institute of Technology)
Nesime Tatbul (Intel Labs and MIT)
Sourav S. Bhowmick (Nanyang Technological University)
Themis Palpanas (Universite Paris Cite)
Torsten Grust (University of Tuebingen)
Vanessa Braganholo (Universidade Federal Fluminense)
Wolfgang Lehner (TU Dresden)
Xiaokui Xiao (National University of Singapore)
Xin Luna Dong (Meta)
Yanlei Diao (Ecole Polytechnique, Amazon Web Services)

Review Board
Abhishek Roy (Microsoft)
Ahmed Aly (Google)
Ahmed Eldawy (University of California, Riverside)
Ahmed Mahmood (Google)
Alberto Lerner (Computing Flows GmbH)
Alberto Sonnino (MystenLabs & University College London)
Alexander Artikis (University of Piraeus)
Alexander Böhm (SAP SE)
Alexander Krause (TU Dresden)
Amarnadh Sai Eluri (Google)
Amelie Marian (Rutgers University)
Amine Mhedhbi (Polytechnique Montréal)
Amir Shaikhha (University of Edinburgh)
Amr Magdy (University of California Riverside)
Andrea Mauri (Lyon 1 University)
Andreas Züfle (Emory University)
Antonio Boffa (EPFL)
Arash Termehchy (Oregon State University)
Arijit Khan (Bowling Green State University)
Aristides Gionis (KTH Royal Institute of Technology)
Arkaprava Saha (University of Tsukuba)
Arnab Phani (TU Berlin)
Arnd Christian König (Microsoft)
Avinash Kumar (Google)
Baihua Zheng (Singapore Management University)
Berthold Reinwald (IBM Research - Almaden)
Bin Yang (Aalborg University)
Bo Tang (Southern University of Science and Technology)
Bo Zhao (Aalto University)
Bobbi W Yogatama (NVIDIA)
Bojan Karlaš (Harvard University)
Boris Glavic (University of Illinois Chicago)
Brian Kroth (Microsoft)
Brit Youngmann (Technion)
Calisto Zuzarte (IBM)
Carlo Curino (Microsoft)
Chang Ge (University of Minnesota)
Chao Zhang (University of Waterloo)
Chen Zhang (The Hong Kong Polytechnic University)
Cheng Long (Nanyang Technological University)
Chenghao Lyu (Amazon)
Chengkai Li (The University of Texas at Arlington)
Chengliang Chai (Beijing Institute of Technology)
Chenhao Ma (The Chinese University of Hong Kong, Shenzhen)
Christan Grant (University of Florida)
Christos Doulkeridis (University of Piraeus)
Christos Koutras (New York University)
Chrysanthi Kosyfaki (The University of Hong Kong)
Chuan Lei (Oracle)
Chunwei Liu (MIT CSAIL)
Cindy Chen (University of Massachusetts Lowell)
Cong Yue (National University of Singapore)
Daniel Kang (UIUC)
Daniel Ritter (SAP)
Davood Rafiei (University of Alberta)
Demetris Zeinalipour (University of Cyprus)
Dimitrios Skoutas (Athena Research Center)
Dixin Tang (The University of Texas, Austin)
Domagoj Vrgoč (PUC)
Dominik Durner (CedarDB)
Donatella Firmani (Sapienza University)
Donato Tiano (Università degli Studi di Modena e Reggio Emilia)
Dong Deng (Alibaba Cloud Computing)
Dong Xie (Penn State University)
Dumitrel Loghin (OKX)
Eduard Dragut (Temple Univ.)
Egemen Tanin (University of Melbourne)
El Kindi Rezig (University of Utah)
Eleni Tzirita Zacharatou (Hasso Plattner Institute (HPI))
Eliana Pastor (Politecnico di Torino)
Elisa Bertino (Purdue University)
Enzo Veltri (Università della Basilicata)
Ergute Bao (Mohamed bin Zayed University of Artificial Intelligence)
Eser Kandogan (Megagon Labs)
Essam Mansour (Concordia University)
Farnoush Banaei-Kashani (University of Colorado Denver)
Felix M Schuhknecht (Johannes Gutenberg University Mainz)
Florian Hahn (University of Twente)
Florin Rusu (UC Merced)
Francesco Gullo (University of L'Aquila)
Fuad Jamour (Amazon Web Services)
Gao Cong (Nanyang Technological University)
Garima Gaur (Inria Saclay, Ecole Polytechnique)
Gengrui Zhang (Concordia University)
George Christodoulou (TU Delft)
George Papadakis (University of Athens)
George Papastefanatos (ATHENA Research Center)
Georgios Siachamis (Inria)
Georgios R Theodorakis (Nvidia)
Giorgio Vinciguerra (Università di Pisa)
Giuseppina Andresini (University of Bari Aldo Moro)
Goce Trajcevski (Iowa State University)
Grigorios Loukides (King's College London)
Guido Moerkotte (University of Mannheim)
Guillaume Lachaud (Ecole Polytechnique)
Haibo Hu (Hong Kong Polytechnic University)
Haipeng Dai (Nanjing University)
Hannes Voigt (Neo4j)
Hantian Zhang (Google)
Hanzhi Wang (University of Copenhagen)
Haralampos Gavriilidis (BIFOLD & Technische Universität Berlin)
Haridimos Kondylakis (FORTH-ICS & Computer Science Department, University of Crete)
Harry Kai-Ho Chan (The University of Sheffield)
Hazar Harmouch (University of Amsterdam)
Herald Kllapi (Google Inc)
Hongzhi Wang (Harbin Institute of Technology)
Hua Lu (Aalborg University)
Huanchen Zhang (Tsinghua University)
Huiping Chen (University of Birmingham)
Hyungsoo Jung (Seoul National University)
Hyunjoon Kim (Hanyang University)
Ibrahim Sabek (University of Southern California)

Igor Zablotchi (Mysten Labs)
Ilia Petrov (Reutlingen University)
Ilias Azizi (Mohammed VI Polytechnic University)
Ilie Sarpe (KTH Royal Institute of Technology)
Indrakshi Ray (Colorado State University)
Ioannis Demertzis (UCSC)
Jaeyoung Do (Seoul National University)
Jan Hidders (Birkbeck, University of London)
Jeeta Ann Chacko (Technical University of Munich)
Jeffrey Xu Yu (The Hong Kong University of Science and Technology (Guangzhou))
Jelle Hellings (McMaster University)
Jesús Camacho-Rodríguez (Microsoft)
Jia Li (Hong Kong University of Science and Technology)
Jia Zou (Arizona State University)
Jialin Ding (Princeton University)
Jianguo Wang (Purdue University)
Jiannan Wang (Tsinghua University)
Jianxin Li (Edith Cowan University)
Jiawei Jiang (Wuhan University)
Jieming Shi (The Hong Kong Polytechnic University)
Jignesh Patel (Carnegie Mellon University)
Jin Wang (Arizona State University)
Jiwon Seo (Seoul National University)
Johannes Pietrzyk (TU Dresden)
Johes Bater (Tufts University)
John Paparrizos (The Ohio State University, USA & Aristotle University of Thessaloniki, Greece)
Jonathan Fürst (ZHAW Zurich University of Applied Sciences)
Jonghyeok Park (Korea University)
Joseph Near (University of Vermont)
Ju Fan (Renmin University of China)
Junghoon Kim (UNIST)
Junhao Gan (University of Melbourne)
K. Selçuk Candan (Arizona State University)
Kaisong Huang (University of Calgary)
Kai-Uwe Sattler (TU Ilmenau)
Kanchan Chowdhury (Marquette University)
Karim Benouaret (Univ Lyon)
Katsiaryna Mirylenka (Zalando SE)
Kaustubh Beedkar (Indian Institute of Technology Delhi)
Keke HUANG (The University of British Columbia)
Kenneth A Ross (Columbia University)
KI HYUN TAE (Samsung Research)
Kostas Stefanidis (Tampere University)
Kurt Stockinger (Zurich University of Applied Sciences)
Kwanghyun Park (Yonsei University)
Kyoungmin Kim (EPFL)
Larissa Capobianco Shimomura (Hasselt University)
Le Gruenwald (The University of Oklahoma)
Lefteris Kokoris Kogias (MystenLabs)
Lei Zou (Peking University)
Leong Hou U (University of Macau)
Leopoldo Bertossi (Carleton University)
Liang Liang (EPFL)
Lingyang Chu (McMaster University)
Loredana Caruccio (University of Salerno)
Lu Chen (Zhejiang University)
Luca Gagliardelli (eCampus University)
Lucas Braun (Oracle Labs)
Lukasz Golab (University of Waterloo)
Madhulika Mohanty (Inria & Institut Polytechnique de Paris)
Magnus Mueller (AWS)
Mahdi Esmailoghli (HU Berlin)
Makoto Onizuka (Osaka University)
Man Lung Yiu (Hong Kong Polytechnic University)
Manuel Rigger (National University of Singapore)
Marco Patella (University of Bologna)
Markos Markakis (Massachusetts Institute of Technology)
Martin Prammer (Carnegie Mellon University)
Matteo Ceccarello (University of Padova)
Matteo Interlandi (Microsoft)
Matthaios Olma (MongoDB)
Matthew J Perron (Microsoft)
Matthias Boehm (Technische Universität Berlin)
Matthias Weidlich (Humboldt-Universität zu Berlin)
Maximilian E Schüle (University of Bamberg)
Meichun Hsu (Oracle America)
Miao Qiao (The University of Auckland)
Miao Yu (Meta)
Michael J Cahill (University of Sydney)
Michael Grossniklaus (University of Konstanz)
Michael Gubanov (Florida State University)
Michael J Mior (Rochester Institute of Technology)
Mohamed Eltabakh (Qatar Foundation)
Mohamed Mokbel (University of Minnesota - Twin Cities)
Mohammad Sadoghi (University of California, Davis)
Mohammad Javad Amiri (Stony Brook University)
Mohammed J Saeed (Apple)
Mostafa Milani (The University of Western Ontario)
Nick Koudas (University of Toronto)
Nikolay Yakovets (TU Eindhoven)
Oktie Hassanzadeh (IBM Research)
Oliver A Kennedy (University at Buffalo, SUNY)
Oscar Romero (Universitat Politècnica de Catalunya)
Panagiotis Bouros (Johannes Gutenberg University Mainz)
Panagiotis Karras (University of Copenhagen)
Paolo Garza (Politecnico di Torino)
Paolo Merialdo (Università degli Studi Roma Tre)
Paolo Missier (University of Birmingham)
Paolo Papotti (EURECOM)
Patrick Damme (Technische Universität Berlin)
Patrick Schäfer (Humboldt-Universität zu Berlin)
Paul Boniol (Inria, Ecole normale supérieure)
Peizhi Wu (University of Pennsylvania)
Peng Peng (Hunan University)
Periklis Chrysogelos (Oracle)
Peter M. Fischer (University of Augsburg)
Pierangela Samarati (Universita degli Studi di Milano)
Prashant Pandey (Northeastern University)
Protiva Rahman (University of Florida)
Qingqing Ye (Hong Kong Polytechnic University)
Qitong Wang (Harvard University)
Qizhen Zhang (University of Toronto)
Rada Chirkova (NC State University)
Rainer Gemulla (Universität Mannheim)

Rana Alotaibi (King Abdulaziz City for Science and Technology (KACST))
Rana Shahout (Harvard)
Renata Borovica-Gajic (University of Melbourne)
Renchi Yang (Hong Kong Baptist University)
Ria Borromeo (University of Philippines)
Riccardo Torlone (Roma Tre University)
Rihan Hai (TU Delft)
Ritesh Sarkhel (Amazon)
Roee Shraga (WPI)
Romila Pradhan (Purdue University)
Ruiyuan Li (Chongqing University)
Ryan Marcus (University of Pennsylvania)
Ryan McKenna (University of Massachusetts Amherst)
Sai Wu (Zhejiang University)
Sara Foresti (Universita' degli Studi di Milano)
Sarah Kleest-Mei§ner (Hasselt University)
Sebastian Link (University of Auckland)
Sebastian Michel (RPTU Kaiserslautern-Landau)
Sepanta Zeighami (University of California Berkeley)
Shantanu Sharma (New Jersey Institute of Technology)
Shaoxu Song (Tsinghua University)
Sharad Mehrotra (U.C. Irvine)
Shen Liang (Université Paris Cité)
Shihabur Chowdhury (Apple)
Shimin Di (The Hong Kong University of Science and Technology)
Silviu Maniu (Université Grenoble Alpes)
Siqiang Luo (Nanyang Technological University)
Søren Kejser Jensen (Aalborg University)
Soror Sahri (Université Paris Cité)
Stavros Maroulis (ATHENA Research Center)
Stefania Dumbrava (ENSIIE, INRIA, IRIF, Télécom-SudParis)
Stefano Marchesin (Università di Padova)
Stefano Paraboschi (Universita' degli Studi di Bergamo)
Steffen Zeuch (TU Berlin)
Surajit Chaudhuri (Microsoft)
Suyash Gupta (University of Oregon)
Sven Groppe (Universität zu Lübeck/Institute of Information Systems (IFIS))
Tanzima Hashem (Bangladesh University of Engineering and Technology)
Tarique Ashraf Siddiqui (Microsoft Corporation)
Tenindra Abeywickrama (RIKEN Center for Computational Science)
Theodoros Chondrogiannis (Norwegian University of Science and Technology (NTNU))
Thomas Neumann (TUM)
Tianyi Li (Aalborg University)
Tianzheng Wang (Simon Fraser University)
Tieying Zhang (Bytedance)
Ting Yu (MBZUAI)
Tingjian Ge (University of Massachusetts, Lowell)
Tong Chen (The University of Queensland)
Torben Bach Pedersen (Aalborg University)
Tristan Allard (Univ Rennes, CNRS, IRISA)
Tsz Nam Chan (Shenzhen University)
Ulf Leser (Humboldt-Universität zu Berlin)
Utku Sirin (Harvard University)
Venkata Vamsikrishna Meduri (IBM Research-Almaden)
Verena Kantere (University of Ottawa)
Vikram Nathan (Amazon, Inc.)
Viktor Leis (Technische Universität München)
Vivek Narasayya (Microsoft)
Vivek Shah (Samsung)
Wang-Chien Lee (Pennsylvania State University, USA)
Wei Dong (Nanyang Technological University)
WEI LU (Renmin University of China)
Weiguo Zheng (Fudan University)
Wenqi FAN (The Hong Kong Polytechnic University)
Wentao Wu (Microsoft Research)
Wentao Zhang (Peking University)
William Schultz (MongoDB)
Xiang Lian (Kent State University)
Xiangyao Yu (University of Wisconsin-Madison)
Xiangyu Ke (Zhejiang University)
Xiao Qin (Snowflake)
Xiao Yan (Wuhan University)
Xiaofei Zhang (University of Memphis)
Xiaohui Yu (York University)
Xiaoou Ding (Harbin Institute of Technology)
Xiaoying Wang (Microsoft)
Xike Xie (University of Science and Technology of China)
Xin Huang (Hong Kong Baptist University)
Xuhao Chen (MSU)
Xun Jian (Northwestern Polytechnical University)
Xupeng Miao (Purdue University)
Yael Amsterdamer (Bar-Ilan University)
Yan Huang (University of North Texas)
Yang Cao (Institute of Science Tokyo)
Yang Wang (The Ohio State University)
YAO LU (NUS)
Yesdaulet Izenov (Nazarbayev University)
Yeye He (Microsoft Research)
Yifan Wang (University of Hawaii)
Yihao Ang (National University of Singapore)
Yin Yang (Hamad bin Khalifa University)
Yinan Li (Microsoft Research)
Yinghui Wu (Case Western Reserve University)
Yingtai Xiao (TikTok)
Yiwen Zhu (Microsoft)
Yixiang Fang (School of Data Science, The Chinese University of Hong Kong, Shenzhen)
Yongjoo Park (UIUC)
Yongluan Zhou (University of Copenhagen)
Yongxin Tong (Beihang University)
Yu Yang (City University of Hong Kong)
Yuan Qiu (Southeast University)
Yujie Hui (Google)
Yuke Wang (Rice University)
Yuyu Luo (HKUST (GZ))
Zeyu Ding (Binghamton University)
Zhao Cao (Remin University of China)
ZHAOJING LUO (Beijing Institute of Technology)
Zhengjie Miao (Simon Fraser University)
Zhewei Wei (Renmin University of China)
ZHIWEI FAN (Meta)
Zhongle Xie (Zhejiang University)
Zhuoyue Zhao (University at Buffalo)
Zoi Kaoudi (IT University of Copenhagen)

Light Load Review Board
Abdulhakim Qahtan (Utrecht University)
Abolfazl Asudeh (University of Illinois Chicago)
Ahmed El-Roby (Carleton University)
Alberto Abelló (Universitat Politècnica de Catalunya)
Alex Conway (Cornell Tech)
Alexander Erben (Technical University of Munich)
Alexandros Labrinidis (University of Pittsburgh)
Anupam Sanghi (TU Darmstadt)
Ariful Azad (Texas A&M University)
Babak Salimi (University of California at San Diego)
Baotong Lu (Microsoft Research)
Bettina Kemme (McGill University)
C. Seshadhri (UCSC)
Curtis Dyreson (Utah State University)
Danfeng Zhang (Duke University)
Daniel Kocher (University of Salzburg)
Davide Mottin (Aarhus University)
Fabian Panse (University of Augsburg)
Felix Naumann (Hasso Plattner Institute, University of Potsdam)
Haitao Yuan (Nanyang Technological University)
Hangdong Zhao (Microsoft)
Hideyuki Kawashima (Keio University)
Hongyan Chang (NUS)
Hongzhi Yin (The University of Queensland)
Ingo Müller (Google)
Jiaheng Lu (University of Helsinki)
Jinho Lee (Seoul National University)
Juan Sequeda (data.world)
Katia Papakonstantinopoulou (Athens University of Economics and Business)
Kexin Rong (Georgia Institute of Technology)
Konstantinos Mamouras (Rice University)
Kyriakos Mouratidis (Singapore Management University)
Lipeng Wan (Georgia State University)
Manolis Terrovitis (Athena Research Center)
Matteo Brucato (Microsoft Research)
Mayuresh Kunjir (Amazon AWS)
Michele Linardi (CYU)
Mo Sha (Alibaba Group)
Murat Demirbas (MongoDB Research)
Myeongjae Jeon (POSTECH)
Nikos Giatrakos (Technical University of Crete)
Odysseas Papapetrou (TU Eindhoven)
Panagiotis Liakos (Athens University of Economics and Business)
Panagiotis Liakos (University of Athens)
Panos Vassiliadis (University of Ioannina)
Paris Carbone (KTH Royal Institute of Technology)
Paul Groth (University of Amsterdam)
Pierre Bourhis (CNRS)
Quang-Trung Ta (National University of Singapore)
Rathijit Sen (Microsoft)
Remy Wang (University of California, Los Angeles)
Renzo Angles (Universidad de Talca)
Seokki Lee (University of Cincinnati)
Sergi Nadal (Universitat Politècnica de Catalunya)
Sivaprasad Sudhir (MIT)
Song Jiang (University of Texas, Arlington)
Sonia Bergamaschi (Università di Modena e Reggio Emilia)
Subarna Chatterjee (Datastax)
Subhadeep Sarkar (Brandeis University)
Sujaya Maiyya (University of Waterloo)
Supawit Chockchowwat (University of Illinois Urbana-Champaign)
Theodore Johnson (AT&T Labs - Research)
Thomas Fahringer (University of Innsbruck)
Thorsten Papenbrock (Philipps University of Marburg)
Vassilis J. Tsotras (UC Riverside)
Venkatesh Emani (Microsoft)
Wenjia He (University of Michigan)
Wolfgang Gatterbauer (Northeastern University)
Xingguang Chen (National University of Singapore)
Yogesh Simmhan (Indian Institute of Science (IISc), Bangalore)
Yuxin Tang (Rice University)
Zhe Jiang (University of Florida)
Ziawasch Abedjan (BIFOLD/TU Berlin)

LETTER FROM THE EDITORS IN CHIEF

We are pleased to introduce the first issue of Volume 19 of PVLDB – the Proceedings of the VLDB Endowment. PVLDB continues to publish original, high-quality research advancing the broad field of data and information management, processing, and analytics. Our scope encompasses foundational theory, novel data models, forward-looking system architectures, new techniques and applications, and comprehensive assessment of real-world deployments at scale. In the research track, PVLDB welcomes contributions in four equally important categories: (a) regular research papers, (b) scalable data science (SDS), (c) experiment, analysis and benchmark (EA&B), and (d) vision papers.

This year, we have introduced several extensions. First, to support submissions from emerging research areas while ensuring that PVLDB can properly evaluate their contributions, we have added a "self-assessment of relevance" item to the submission form. If a paper does not clearly fall within traditional core database topics, authors are invited to explain its relevance to the PVLDB community by identifying (i) the key data management challenges addressed, and (ii) the principled data management ideas or contributions offered. Authors may also point reviewers to specific sections that substantiate this self-assessment. This information assists the Review Board in assessing relevance in rapidly expanding or interdisciplinary topic areas.

We have also strengthened the PVLDB review process. PVLDB remains committed to a rigorous, fair, and constructive review pipeline. Every submission is screened for plagiarism and potential conflicts of interest before being assigned to reviewers. Each paper is evaluated by at least three experienced reviewers under the coordination of an Associate Editor, and a fourth reviewer may be added when specialized domain expertise is required. A structured three-week discussion phase follows the initial reviews, culminating in a consensus documented in a meta-review. Some submissions may proceed to a revision phase, in which authors are given up to 2.5 months to address feedback. Accepted papers are published in PVLDB and will also be presented at the upcoming VLDB Conference. To support scale and review quality, PVLDB employs CMT for workflow management, TPMS for reviewer identification, CLOSET for conflict-of-interest detection, and iThenticate for plagiarism screening.

In addition, we have improved on how we recruit and deploy reviewing capacity. We significantly expanded the PVLDB Review Board to improve topical coverage across the community. Furthermore, to ensure access to domain expertise in emerging areas, each submission is required to nominate one qualified reviewer (PhD and at least two prior publications in SIGMOD, VLDB, or ICDE). When appropriate, nominated authors may be invited to serve as Light Load Reviewers, who may be assigned at most one review per month and up to three reviews in total. This mechanism enables PVLDB to secure authoritative assessments precisely in those domains where specialized expertise is needed.

PVLDB also continues to encourage artifact availability to support transparency and reproducibility. Associate Editors assess artifact accessibility during the meta-review, and accepted papers whose artifacts satisfy established availability requirements receive the official ACM badge.

Finally, beginning in Volume 19, PVLDB is introducing a Shadow PC program. The Shadow PC is educational and wholly independent of the official review process. Authors may opt in to have their submissions included in the Shadow PC pool, and selected participants gain hands-on reviewing experience by evaluating real submissions on a schedule that mirrors the official review cycles. Shadow PC reviews remain confidential and do not influence any PVLDB decisions. This initiative aims to help develop the next generation of reviewers in the data management community.

This first issue of Volume 19 includes six papers, covering novel database architectures exploiting emerging hardware, efficient search in vector databases, data management support for machine learning workloads, data mining and analytics over graph data, and distributed training in cloud environments.

We extend our sincere appreciation to our Review Board, Associate Editors, reviewers, and Proceedings Chairs, whose efforts are instrumental to the operation and success of PVLDB.

Yanlei Diao and Xiaokui Xiao
Editors-in-Chief of PVLDB Vol. 19
Program Chairs for VLDB 2026

FlatStor: An Efficient Embedded-Index Based Columnar Data Layout for Multimodal Data Workloads

Chi Zhang
Shanghai Jiao Tong University
C_Zhang1996@sjtu.edu.cn

Shihao Zhang
Shanghai Jiao Tong University
zsh_henry200135@sjtu.edu.cn

Yunfei Gu
Shanghai Jiao Tong University
gu.yunfei@sjtu.edu.cn

Chentao Wu*
Shanghai Jiao Tong University
wuct@sjtu.edu.cn

Jie Li
Shanghai Jiao Tong University
lijiecs@sjtu.edu.cn

Qin Zhang
Huawei Cloud
kevin.zhangqin@huawei.com

Xusheng Chen
Huawei Cloud
chenxusheng6@huawei.com

Jie Meng
Huawei Cloud
mengjie09@huawei.com

ABSTRACT

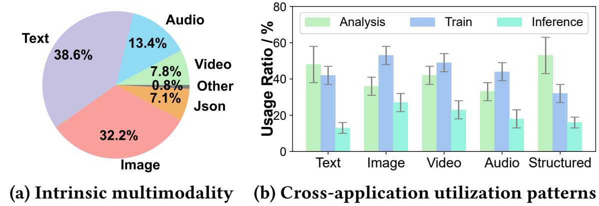
Modern data lakes have become essential for storing, managing, and analyzing massive amounts of heterogeneous data. As production data increasingly exhibits multimodal storage characteristics and multi-purpose access patterns, efficient management of such complexities becomes critical. However, current hybrid storage system-based data lakes face persistent challenges, including synchronization overhead, data correlation disruption, and escalating storage costs due to the involvement of multiple underlying storage systems. While columnar storage, central to data lakes, addresses hybrid-system inefficiencies, it struggles with the complexities of multimodal data storage and multi-purpose access.

To tackle these challenges, we analyze access patterns across various scenarios and assess the issues in storing multimodal data. Based on these insights, we propose FlatStor, a FlatBuffers-based columnar Storage format with embedded indexing. It supports point access through indexing and handles multimodal data by vertically partitioning and treating each modality as a byte stream for storage. It also applies FSST compression, reducing storage overhead significantly. Benchmark evaluations reveal that FlatStor reduces the access latency by 99.6% and the storage overhead by 91.3% compared to Parquet in inference workloads. Furthermore, FlatStor outperforms LanceV2 with a 41.3% latency improvement, maintaining minimal additional overhead.

PVLDB Reference Format:
Chi Zhang, Shihao Zhang, Yunfei Gu, Chentao Wu, Jie Li, Qin Zhang, Xusheng Chen, and Jie Meng. FlatStor: An Efficient Embedded-Index Based Columnar Data Layout for Multimodal Data Workloads. PVLDB, 19(1): 1-14, 2025.
doi:10.14778/3772181.3772182

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/SJTU-Storage-Lab/FlatStor-Columnar-Layout.

*Corresponding Author
This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 1 ISSN 2150-8097.
doi:10.14778/3772181.3772182



(a) Intrinsic multimodality **(b) Cross-application utilization patterns**

Figure 1: Modern production data landscape

1 INTRODUCTION

In the big data era, enterprises increasingly struggle with storing and analyzing massive, heterogeneous data. Data lake technology has emerged as an effective solution [43, 53], providing distributed storage for raw data while integrating modern query engines to enable efficient data management and analysis [24–28, 30, 38]. At the core of this technology lies columnar storage, with mainstream formats like Parquet [7] and ORC [6, 44] becoming de facto industry standards. These formats are widely supported by distributed storage systems (e.g., HDFS [62], S3 [2] and processing frameworks (e.g., Spark [8], Flink [5]), as well as major cloud data warehouses including Redshift [33, 42, 58] and Snowflake [23, 39].

Recent studies underscore inherent limitations of conventional columnar storage in meeting emerging application demands [45, 51, 64]. Figure 1 highlights two key characteristics of modern production data at petabyte scale: the *multimodal nature* at storage and the *multi-purpose access* at the application layer, where identical data serve multiple departments for diverse purposes. Multimodality introduces distinctive storage challenges, including massive aggregate volumes, numerous small items, and predominantly item-level access. Existing solutions mitigate these complexities by integrating specialized storage systems for different modalities alongside columnar architectures. For example, H2O [31] deploys mixed layouts and dynamically adjusts formats at runtime. Nevertheless, this approach has notable drawbacks. First, it incurs considerable synchronization overhead due to frequent format transformations. Second, it disrupts data correlations by dispersing data across systems, requiring intricate metadata management to maintain holistic relationships. The fundamental challenge is achieving efficient multimodal storage while preserving accessibility and integrity.

The data utilization patterns in Figure 1b highlight another challenge: in production, identical data are often reused across multiple organizational units for three main functions—analytical processing by BI teams, model training by ML engineers, and real-time inference in operational systems (e.g., RAG scenarios using internal data). This tripartite usage produces varying access patterns, from batch-oriented analytical queries to latency-sensitive point-access inference, all on shared data. Current implementations rely on department-specific copies, causing substantial storage overhead—each unit maintains independent replicas, often with extra copies for high availability—leading to 3–5× higher raw storage needs. Interconnected workflows exacerbate latency due to continuous synchronization across isolated storage, reducing throughput. These inefficiencies highlight the need for storage architectures that support diverse access patterns while ensuring data integrity, optimizing resources, and reducing operational complexity.

Given the proven success of columnar storage in data lake implementations for analytical and model training workloads [32, 55, 66], it is natural to explore whether its architecture can be extended to natively support multi-purpose multimodal data storage. A unified columnar approach simplifies system design and offers the potential to: (1) eliminate unnecessary data synchronization, (2) preserve critical data correlations, and (3) reduce storage costs by minimizing redundant replication. The key challenge is enabling columnar systems to efficiently support diverse, dynamic access patterns for multimodal data while retaining benefits such as low storage overhead and high query performance. This challenge can be distilled into two core problems: 1) identifying and managing access patterns across applications and 2) analyzing multimodal data layouts to uncover inefficiencies in columnar usage.

To address limitations in existing multimodal storage, we propose FlatStor, a storage format based on FlatBuffers that extends columnar storage with embedded indexing and multimodal-aware partitioning. Unlike prior work treating multimodal data as isolated vectors or using coarse partitioning, FlatStor applies fine-grained vertical partitioning, dividing data into modality-specific columns and treating each data item as a byte stream, enabling unified, efficient storage of heterogeneous data. It also embeds an index for point access, reducing inference latency without sacrificing throughput. Additionally, FlatStor uses FSST for deduplication, balancing storage efficiency with fast decoding and access.

Our contributions are fourfold: 1. summarizing common multimodal data access patterns across applications; 2. revealing limitations of existing hybrid methods in handling such access; 3. proposing FlatStor, a columnar layout with embedded indexing addressing point access and multimodal storage challenges; 4. implementing and evaluating FlatStor on multiple benchmarks. Experiments show that, in inference scenarios, FlatStor achieves significantly lower access latency than Parquet and LanceV2 [18], with minimal overhead. These results validate FlatStor as a practical, effective solution for diverse multimodal data access, bridging the gap between traditional columnar storage and modern workload demands.

The remainder of this paper is organized as follows: Section 2 presents background and motivation; Sections 3 and 4 describe the design and implementation of FlatStor; Section 5 presents experimental results and analysis; Section 6 discusses related work; and Section 7 concludes.

2 INSIGHTS & MOTIVATIONS

This section analyzes application access patterns, evaluates existing hybrid schemes and their limitations for multi-purpose multimodal data, examines issues in columnar storage with multi-granularity access, and highlights challenges in multimodal data.

2.1 Access Patterns Across Diverse Applications

By analyzing the I/O processes of various application categories, we summarize their key characteristics as follows:

Data Analytics. In big data analytics, I/O is dominated by batch access, where systems sequentially read and process large data volumes for operations like aggregation and filtering, involving continuous high-volume reads with minimal random access.

Model Training. During training, data are accessed via batch-based requests. Each batch retrieves a contiguous subset of data, which is shuffled locally for randomness before GPU training. The I/O pattern follows a periodic batch access pattern, as batches are requested at regular intervals.

Model Inference. Inference tasks like RAG [50] often start by retrieving a few relevant records—facts, memories, or documents—via vector search. These *top-k* results point to scattered data entries, producing random point-access patterns. Such fine-grained, unpredictable access is essential for context-aware, accurate responses. Inference also spans multiple levels, from point access for fact lookup to batch access for complex reasoning [67], showing that multiple I/O granularities coexist in these scenarios.

In summary, despite varied applications, I/O falls into two types: batch access and point access. This classification guides storage system design to balance batch tasks (like analytics) and real-time inference (like RAG [50]).

2.2 Limitations of Existing Approaches

We divide current storage schemes into two types: **Discrete Storage (DS)** and **Aggregated Storage (AS)**. DS involves minimal preprocessing before direct storage, as in file, object, and key-value stores like MinIO [19], Redis [21], and RocksDB [22]. AS aggregates and processes data before storage, including row- and column-based formats such as Parquet [7], ORC [6], Arrow IPC [4], SnowFlake [23], RedShift [33], DuckDB [11], PostgreSQL [20], and Aurora [63]. Based on access pattern analysis in Section 2.1, we evaluate AS and DS using the LAION-Aesthetics dataset [60, 61], following Section 5. Raw data is stored in MinIO (DS) and aggregated into Parquet (AS), accessed via Arrow [3], enabling evaluation of hybrid system performance across multiple schemes.

Overall Performance. Although training workloads are generally constrained by throughput, latency remains a critical factor during cache warm-up phases and in environments with resource contention, both of which affect training stability. As shown in Figure 2, during model training (batch query), AS reduces latency by 46% compared to DS. Conversely, during inference (point query), DS reduces latency by 97% relative to AS. This significant contrast indicates that neither AS nor DS alone can fully satisfy the latency and throughput requirements across different usage scenarios, underscoring the necessity for a unified and flexible storage solution.

Table 1: Comparison between AS and DS

	Discrete Storage	Aggregated Storage
Storage Overhead	High	Low
Metadata Overhead	High	Low
Management Difficulty	Medium	Medium
System Portability	Medium	High

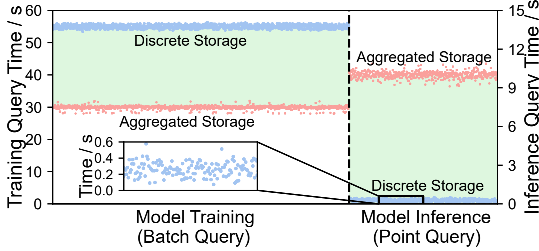


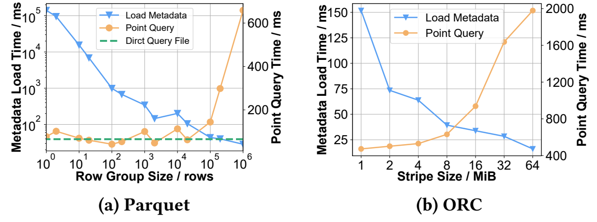
Figure 2: I/O performance of AS vs DS across applications

Data Synchronization Overhead. Data conversion experiments show that converting the dataset from AS to DS takes **4,800 CPU hours**, while the reverse takes **7,400 CPU hours**. Since LAION-Aesthetics is a small subset, overhead grows with scale. This reveals a key flaw in hybrid systems—their inability to adapt to changing needs without heavy resources. The problem worsens when data serves both analysis and inference, as repeated conversions create bottlenecks and complexity. These results highlight the need for a unified storage system that supports diverse access patterns without repeated conversions.

Disruption of Data Correlation. While hybrid storage solutions can meet diverse access needs, they often cause *Loss of Data Correlation*. When data is fragmented across formats, cross-field or cross-row analysis becomes more complex. Frequent migrations also increase overhead and risk data integrity, reducing query efficiency. In contrast, a unified format can preserve data relationships, enabling efficient multi-dimensional analysis and complex queries, offering a more robust solution.

Substantial Storage Overhead. Hybrid multi-storage schemes often require maintaining separate high-availability replicas for different formats to ensure functionality during failures. This redundancy multiplies storage overhead, as the same data must be stored in multiple formats. For example, supporting both analytical and inferential workloads may demand full data copies in both KV and columnar storage, driving up a typical 3-5× increase in storage overhead. A unified storage format eliminates redundant replication, cutting costs and simplifying system design.

Table 1 compares additional metrics, showing that DS incurs higher storage and metadata overhead due to independent data item management, which complicates compression. In contrast, AS offers greater portability with consistent data definitions and standardized interfaces like *Schema* and *SQL*, ensuring adaptability across infrastructures. Based on these findings, AS is the preferred solution. It can be divided into row- and columnar formats. Given the superior efficiency of columnar storage in filtering and tasks essential for analysis and model training via selective access and better compression, it is chosen as the final format.



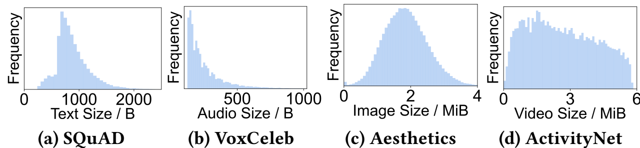
(a) Parquet **(b) ORC**

Figure 3: Point query performance between Parquet and ORC

2.3 Constraints of Columnar Storage

2.3.1 ***Poor point access performance.*** In Section 2.2, we analyze limitations in current hybrid approaches and explain the rationale for choosing columnar storage as the base solution. This section also investigates key factors affecting point access performance. Figure 3 presents performance of mainstream columnar formats under point access and their metadata loading overhead, using the same setup as Figure 2. In Figure 3a, the green dashed line indicates baseline performance when accessing raw data directly from remote object storage such as MinIO. Results show that increasing row group (or ORC stripe) size lowers metadata overhead but significantly worsens access performance. Due to small test case scale, ORC stripe count varies less than Parquet row groups, leading to smaller metadata size and better access performance for ORC. Because a column chunk within a row group is the minimal accessible unit, smaller row groups reduce chunk size and I/O amplification, narrowing the gap with raw data. Based on the connection among chunk size, I/O amplification, and observed results, we conclude that I/O amplification is the primary factor behind degraded performance. This conclusion broadly holds for columnar formats like Parquet, ORC, Arrow IPC, Snowflake, Redshift, and DuckDB, all using the row group concept.

2.3.2 ***Layout-related motivations.*** An intuitive way to improve point query performance is by reducing column chunk size, but this is constrained by the row group configuration, which applies uniformly across the table. Once data is written, changing this setting typically requires reloading and rewriting the table, causing high overhead and leaving row group size mostly fixed. Many analytical applications favor large row groups for sequential loading—Parquet recommends 512 MiB to 1 GiB, while ORC defaults to 64 MiB to match HDFS [62] block sizes. Existing systems also rely on column chunks for metadata like Bloom filters and dictionaries. Smaller chunks increase metadata overhead, explaining the sharp rise in metadata loading seen in Figures 3a and 3b as chunk size shrinks. Additionally, current columnar layouts inherit legacy structures from relational databases, using row groups originally to partition tables for easier management. While this enables partitioning without detailed schema knowledge, it mismatches columnar usage—users seldom access entire tables or row groups but rather specific columns or chunks. This misalignment, combined with tight coupling of row groups to I/O, leads to excessive I/O amplification in point access. These problems highlight the need to decouple data management from access granularity to better support efficient point queries and flexible access, such as inference.



(a) SQuAD **(b) VoxCeleb** **(c) Aesthetics** **(d) ActivityNet**

Figure 4: Multimodal data size distributions across datasets

2.4 Inside Multimodal Dataset

2.4.1 ***Data characteristics.*** Our work focuses on efficient storage and access of multimodal data items such as single images, audio clips, or video segments, while tasks like video segmentation and alignment are performed upstream and lie outside the scope. To analyze data characteristics, we select four representative datasets: SQuAD [59] (text QA), VoxCeleb [56] (speech), LAION Aesthetics (images), and ActivityNet [36] (video). Figure 4 shows item size distributions, with even video rarely exceeding 6 MiB. Total storage ranges from hundreds of GiB to several TiB, implying vast numbers of items. Due to semantic alignment challenges [41], multimodal data is usually processed via single-modality training and late-stage fusion, so most datasets contain no more than three modalities (text, image, audio). In summary, modern datasets exhibit three traits: *small item size, large volume*, and *limited modalities*, posing two key challenges: **storage efficiency** and **multimodal support**.

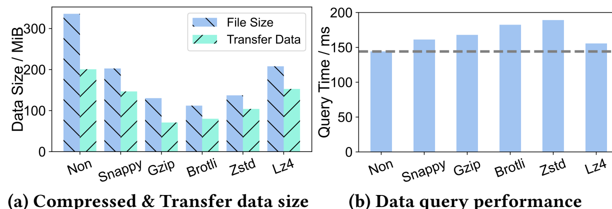
2.4.2 ***Data storage efficiency.*** Data storage efficiency addresses the challenges of large volumes of small data items in current datasets, aiming to minimize storage costs. These costs consist of metadata and data. In Section 2.2, columnar storage is selected as the foundational solution through comparative analysis, with Table 1 demonstrating its lower metadata costs. Consequently, the primary remaining challenge lies in reducing data storage costs.

In columnar storage, data in the same column often share similar types and characteristics, making compression a common and effective way to reduce storage costs. Compression methods generally fall into two types: *dictionary-based* and *statistic-based*. Dictionary-based compression replaces repeated symbols with references from a dictionary. The algorithm builds a dictionary of unique symbols and assigns each an index. Repeated symbols are then replaced with indices, reducing overall size. Decompression restores the original data using the dictionary. Statistic-based compression uses data properties such as frequency and symbol probabilities to assign shorter codes to frequent symbols, minimizing encoded length. An encoding table is created for compression, and decompression uses it to restore the original data.

Both compression algorithms typically process large datasets globally for optimal compression ratios, requiring global decompression to restore the original data. Figure 5 shows the size of Parquet files generated from part of the LAION-Aesthetics dataset using different compression algorithms, along with point access performance for a specified column. While compression reduces data size, it can degrade access performance due to global decompression. Recent studies [49, 64] suggest that as storage costs decrease, compression methods should prioritize decompression efficiency to improve access performance. Emerging algorithms like FSST [35]

Table 2: Challenges and Corresponding Solutions in FlatStor

Challenges	Solutions
Balancing fine-grained access with metadata overhead	Vertical partitioning enables independent access granularity per column and overcomes row group limitations.
Relatively large data storage overhead	Random-accessible compression preserving granularity and access storage efficiency.
Row group explosion due to atomic access requirements for modality data	Multi-level fine-grained metadata design reduces redundant metadata overhead.



(a) Compressed & Transfer data size **(b) Data query performance**

Figure 5: Parquet query performance under different compression algorithms

support random access and partial decompression, but rely on explicit boundary information during data retrieval. Poor metadata design can lead to the loss of boundary information, forcing random queries to load entire compressed blocks, even if decompression itself is not the bottleneck.

2.4.3 ***Multimodal support.*** Current methods for storing multimodal datasets typically save data URLs as strings while storing raw data in separate systems, as exemplified by LAION, JourneyDB [57], and HuggingFace [15]. In Section 2.2, we analyze the shortcomings of hybrid storage systems and advocate for a single columnar storage solution. However, directly storing multimodal data in columnar formats leads to the *row group explosion* problem, where atomic multimodal data (e.g., videos, images) cannot be column-split or selectively accessed. Fine-grained atomic access necessitates small row groups, resulting in excessive metadata overhead.

2.4.4 ***Data-related motivations.*** Based on the above analysis, regarding data storage efficiency, directly applying compression algorithms in columnar storage may introduce extra access constraints beyond the row group issue, further degrading performance in inference scenarios. Thus, metadata design should support random accessible compression methods and store explicit boundary information to enable advanced compression. For multimodal data storage, this concern aligns with the fine-grained access performance issues analyzed in Section 2.3, further emphasizing the core role of row groups in current designs and the need to decouple data management from access in columnar storage. To address this, we introduce a vertical partitioning method that treats each modality type as an independent column. Each column can adopt its own management strategy and data access granularity, free from unified row group constraints. This ensures that even with reduced granularity, the row group explosion problem is avoided. More details are provided in Section 3.2.

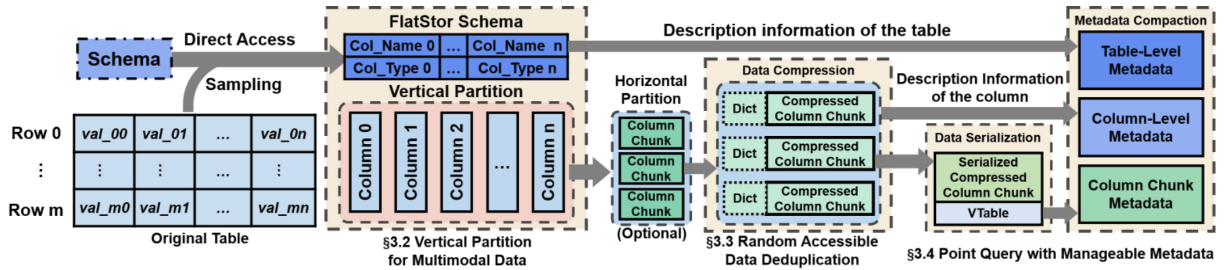


Figure 6: Workflow Generation in FlatStor

3 FLATSTOR

In this section, we first provide a brief overview of the generation process of FlatStor, a FlatBuffers-based columnar Storage format with embedded indexing for efficient multimodal data access. This is followed by a detailed explanation of each step in the workflow, covering all stages from Section 3.2 to Section 3.4.

3.1 Overview

FlatStor introduces a novel columnar layout that enables efficient point queries and multimodal data storage, while preserving the benefits of traditional columnar formats and minimizing overhead. This is achieved by addressing three key challenges: 1) supporting multimodal data, 2) reducing storage overhead, and 3) enabling efficient point access without significantly increasing metadata overhead, thus avoiding the problem illustrated in Figure 3. To this end, FlatStor adopts three core designs: Prioritized Vertical Partitioning, Random Accessible Deduplication, and Serialization with Metadata Compaction, as shown in Figure 6. A brief overview is provided below:

(1) *Vertical Partition for Multimodal Data:* To support multimodal storage, FlatStor adopts column-based vertical partitioning over row-based horizontal partitioning. This enables independent management of each column by data type, free from unified table constraints, and allows better optimization for heterogeneous data.
(2) *Random Accessible Data Deduplication:* To reduce storage overhead, FlatStor uses FSST [35] as its core compression method for efficient and fast data encoding. FSST compresses data more effectively than LZ4 [68, 69] and supports random access, ensuring that compression does not hinder fine-grained access granularity or performance.
(3) *Point Query with Manageable Metadata:* To enable efficient point access with controlled metadata overhead, FlatStor serializes column chunks. During serialization, data offsets are recorded via an embedded index to facilitate point access. Metadata is organized in three tiers (table, column, and column-chunk levels) and compacted by scope, reducing unnecessary overhead through reuse.

By integrating these methods, FlatStor achieves configurable access granularity, manageable metadata overhead, and support for multimodal data storage. Figure 6 illustrates the workflow, and the subsequent sections provide detailed designs for each step.

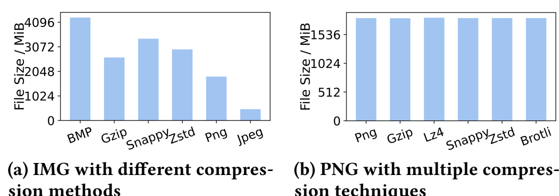
3.2 Vertical Partition for Multimodal Data

In current columnar storage, tables are typically divided horizontally into smaller subtables called *row groups*, which are further split by column to form *column chunks*. This partitioning reduces data management costs and improves concurrency and reliability. However, it forces the entire table to share the same horizontal partitioning, requiring column chunks to align with row groups. Since each column stores different data types, configurations should vary. Uniform horizontal grouping makes it difficult to ensure stable performance across columns or meet specific needs of diverse data types. For instance, differences in data scale and encoding between *string* and *image* columns prevent optimizing I/O performance for both, favoring only one grouping size. Furthermore, partitioning is typically done once in both directions, so the number of rows in a chunk is set by horizontal partitioning. Row groups often contain multiple rows for management efficiency, limiting fine-grained access to column chunks and creating a tradeoff between data access performance and metadata overhead, as shown in Figure 3.

To address limitations of globally consistent row groups in the current design, FlatStor redefines table partitioning by reversing the order of horizontal and vertical partitioning. This adjustment introduces several key benefits: ***Simplified Data Hierarchy***: Prioritizing vertical partitioning conceptualizes the table as a collection of column units, aligning naturally with columnar storage principles. *Customized Horizontal Partitioning*: Each column independently determines its horizontal partitioning size based on its data characteristics, such as modality. Combined with modality-specific configurations (e.g., encoding, indexing), this ensures optimal performance per column. *Granular Metadata Design*: Managing data at both table and column levels enables more detailed metadata, supporting efficient and flexible data management. Further details on metadata design are discussed in Section 3.4.

Although columnar organization is preferred by default in FlatStor for easier storage, input tables from other systems are often row-organized, requiring conversion. Depending on table type, we apply different strategies to process *records* and *fields* from the original table to generate the FlatStor schema:

- *Table with schema.* This type of table is already properly formatted, allowing direct extraction of descriptive information from the original schema. Typical examples include DataFrames or relational database tables. We can directly retrieve column names, data types, and other field details and convert them into column descriptions to generate a new schema in FlatStor.



(a) IMG with different compression methods

(b) PNG with multiple compression techniques

Figure 7: Comparing Parquet's image storage efficiency across compression algorithms

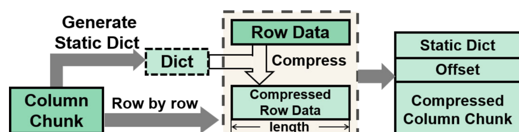


Figure 8: The compression process of FSST

- *Table without schema.* Another scenario involves receiving tables without column descriptions—for example, fragments from CSV files or table-structured data parsed from network byte streams. In such cases, we cannot directly obtain column information for a FlatStor schema. Instead, we assign placeholder names like "Column N" by record order. To infer data types, we perform random sampling on each column, initially treating all columns as *string*, then adjust types based on sampled data features—for instance, assigning *double* if decimal points appear. This enables construction of a basic FlatStor schema.

Once the schema for FlatStor is obtained using the above methods, we create vectors for each column to temporarily store its data. By iterating through the table row by row, we parse each record and append values to the corresponding vectors. This yields the schema and associated column vectors in FlatStor, completing vertical partitioning. Each column is now stored independently, allowing tailored management and configuration. Although multimodal data is stored as byte streams, original encoding formats include *magic numbers* as identifiers, distinguishing *string* from modal data and resolving modality at the storage level. The upper layer can define access interfaces for different modalities, but such designs lie beyond the scope of this paper.

3.3 Random Accessible Data Deduplication

After resolving multimodal data storage challenges, the next issue is storage overhead. As noted in Section 2.4.1, most datasets include no more than three modalities—typically text, images, and audio [41]. To address storage overhead for modality-specific complex data, we use image data as an example and apply the method in Section 3.2, storing it as *string* data in Parquet files for analysis. Figure 7 shows our results. For raw image data (BMP in Figure 7a), mainstream compression methods in columnar storage fall far short of image-specific techniques. Some methods (e.g., Lz4, Brotli) even fail on images, showing that modality-specific data requires specialized encoding or deduplication to reduce storage overhead. Moreover, Figure 7b shows that after converting images to PNG,

Table 3: Parquet file sizes for the same dataset under varying row group configurations

Row group Size	1	10	100	1,000	10,000
File Size	5.4 GiB	941.3 MiB	404.5 MiB	346.7 MiB	333.0 MiB

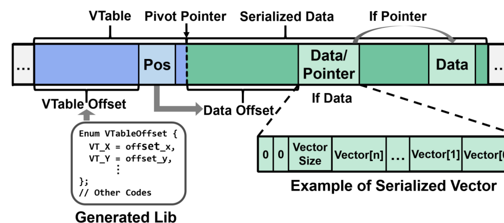


Figure 9: Serialized data layout with FlatBuffers-enabled random access

no compression method achieves further reduction, indicating that once modal data undergoes format-specific deduplication, general-purpose compression offers no additional gains.

For simple modality data like *string*, FlatStor primarily uses FSST (Fast Static Symbol Table) for compression. FSST is a lightweight, efficient algorithm for *string* data, employing a *static symbol table* to encode frequent substrings, achieving high compression ratios with fast compression and decompression. Unlike traditional dictionary methods, FSST allows access to individual items within a compressed block without traversing the entire block, making it ideal for low-latency scenarios. Figure 8 shows the FSST compression process. During compression, FSST records each item's length, converted to a global offset within the data chunk. Because the dictionary is static, decompression uses these offsets to locate and decompress the needed item directly, enabling random access.

It is important to emphasize that the data deduplication method in FlatStor is applied independently to each modality-specific column, which are managed separately. This independent deduplication does not cause loss of correlation between modalities, because modalities remain aligned at the record level through unified schema definitions and row index consistency. The schema explicitly records correspondences between modalities, similar to how Parquet handles the LAION dataset. The deduplication method in FlatStor combines the two approaches mentioned above. As described in Section 3.2, after vertical partitioning, independently managed columns and their corresponding vectors are obtained. *Magic numbers* determine whether data in each vector is modality-specific complex data and apply the appropriate deduplication method. For modality-specific complex data, deduplication treats each data item independently within the same column, without comparing different items. After deduplication, the number of items remains unchanged, and each item remains independently accessible. For simple modality data like *string*, FSST ensures random access after compression, relying on offsets, which may fail with poor layout, and row group limitations can hinder access. Therefore, the data format is adjusted to accommodate offsets and avoid row group restrictions, as discussed in Section 3.4. For other simple types, such as *int* and *double*, compression methods supporting random access are similar to FSST, so specific methods are not detailed here.

3.4 Point Query with Manageable Metadata

The final major challenge for FlatStor is supporting random point access in inference scenarios while minimizing metadata overhead. Table 3 shows Parquet file sizes for the same data sample under different row group sizes. As noted in Section 2.3, the row group is the core design element in most current columnar storage systems [1, 4, 6, 7, 11, 39], with the column chunk inside a row group as the smallest unit for management and access. This design causes metadata overhead to rise sharply as access granularity decreases—smaller chunks mean more chunks to manage for the same data volume. The core issue stems from tight coupling between data management and access mechanisms in existing systems. Therefore, addressing this requires decoupling data management from data access in columnar storage.

Inspired by KV storage and FSST, FlatStor uses a serialization-based embedded indexing method to decouple data management from access granularity. Traditional KV systems support basic set and get operations, limiting direct access to data inside values. By serializing values in a specific format, internal data becomes accessible, treating a column chunk as a value and each row as internal data within it. FSST's offset recording similarly enables direct access via offsets. FlatStor embeds offset indices for each row within a column chunk using serialization, enabling random access while retaining block-based management. The VTable records offsets of each item within the serialized data, enabling random access. Since offsets are stored in a pre-generated static lib file, parsing avoids reading irrelevant VTable data, reducing overhead.

After decoupling data management from access in FlatStor and stabilizing management overhead regardless of access granularity, we further reduce metadata overhead through consolidation. Besides the growing number of column chunks, uncontrolled metadata records cause significant redundancy. For example, data columns often use dictionary encoding to improve storage, but metadata suffers from two redundant designs: 1) at the row group level, where each row group maintains multiple dictionaries; and 2) at the data page level, where each page uses an independent dictionary. These ignore data similarity within columns, causing duplicate dictionaries and extra I/O overhead, especially at the page level. Thus, we propose that metadata components should be set at appropriate scales to improve efficiency. We classify metadata types as follows:

- **Schema**: A schema describes the overall structure of a two-dimensional dataset, such as a table. It defines a sequence of fields with names, data types, and optional metadata. For multimodal data, the schema preserves alignment semantics by assigning each modality to a dedicated field and enforcing consistent row-wise ordering. Cross-modal relationships—such as temporal or semantic alignment—can be encoded through schema metadata to support coordinated interpretation and access.
- **Dictionary**: The dictionary is used for dictionary encoding, a technique that represents data values by referencing unique entries in a dictionary. This method is particularly effective when data contains many duplicate values. During encoding, values are represented by an array of non-negative integers that index the corresponding entries in the dictionary.

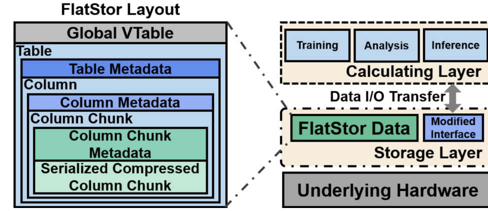


Figure 10: Integrating FlatStor with storage system

- **Statistics**: Statistical information includes data such as maximum and minimum values, as well as the number of rows in a column chunk. During a range query, a Bloom filter can efficiently sift through column chunks that meet the criteria based on the statistical information, thereby enhancing data retrieval efficiency.
- **Others**: Other information, such as data addresses, data versions, and other customized configurations, should also be included in the metadata. This type of information pertains to user-defined configurations.

FlatStor organizes metadata into three tiers: *Table*, *Column*, and *Column Chunk*. The **Schema** belongs in table-tier shared metadata. **Dictionaries** may reside in table-tier metadata if column data types are similar; otherwise, they are in column or column chunk metadata. **Statistics** belong to column or column chunk metadata. **Others** are placed according to their association with the data. As noted in Section 2.3, most columnar storage systems require loading and rewriting the entire dataset for updates, leading to poor write performance. Consequently, updates and deletions (which may trigger metadata changes) are infrequent, and although FlatStor's metadata compaction increases modification cost, the impact remains acceptable.

After completing data deduplication in Section 3.3, we observe that for a column chunk vector, regardless of the deduplication method, we obtain a deduplicated vector matching the original length. For example, during FSST compression, its output can be adjusted to form a vector of compressed row data based on recorded offsets. Deduplicated vectors from different columns are integrated using the serialization method described here, ensuring each column has the same VTable-Serialized Data structure. In serialization, original FSST offsets can be replaced by a VTable with the same function, enabling synergy with FSST format design. Deduplicated vectors from modality-specific complex data require no special handling. We then consolidate metadata across columns; for instance, encoding dictionaries from various chunks are traversed to extract common parts into column-level metadata, leaving only unique parts in chunk metadata, which are reconstructed during decoding.

4 PUT IT ALL TOGETHER

After processing a table through the generation workflow described in Section 3, we have obtained metadata structured into three tiers, along with several serialized and compressed column chunks. Since serialization is a widely adopted technique for both metadata management and data format definition [9, 14], we consistently apply unified serialization to efficiently handle and store both the three-tier metadata and the corresponding data chunks across the system.

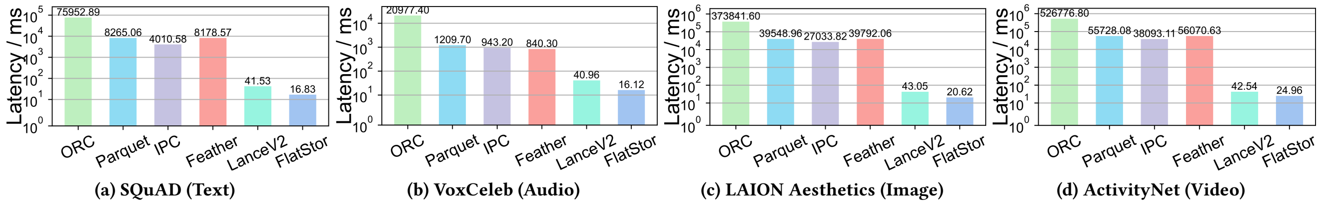


Figure 11: Query latency of different formats under inference workload scenario

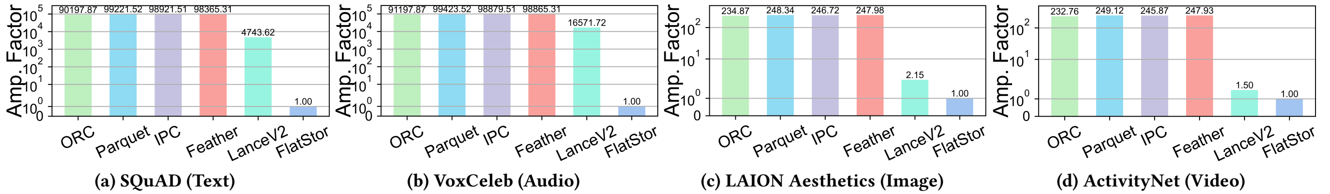


Figure 12: Query amplification of different formats under inference workload scenario

Figure 10 (left half) shows the final physical layout of FlatStor after serializing three-tier metadata alongside data, following a bottom-up hierarchy. At the lowest level, vectors correspond to *Column Chunks*, with metadata stored alongside (*Column Chunk tier*). Multiple column chunks form column data, with metadata completing the *Column tier*. Columns combine into table data, stored adjacent to table metadata (*Table tier*). The *Global VTable* records offsets of each table part, enabling efficient random access. The right half illustrates FlatStor integration with storage systems (e.g., data lakes), replacing existing data formats and extending access interfaces for point access, supporting inference while remaining compatible with traditional workflows.

5 EVALUATION

In this section, we evaluate our design using various metrics, including data access performance across different workload scenarios, ablation studies, and overhead analysis.

Methodology. Our baselines include Parquet [7] and ORC [6], two widely used columnar formats, along with IPC and Feather [4]. IPC is the columnar format in Arrow [3], while Feather is its serialization format using FlatBuffers. We also compare with LanceV2, the format used by LanceDB [17, 18], a container-based database optimized for multimodal data. As LanceV2 is still under development, we implement its core container design based on available documentation for testing. To ensure consistency across formats, Arrow is used as the default data access tool during evaluation.

Metrics. We compare performance across multiple metrics: I/O performance in inference, analytics, and model training scenarios; data skipping access efficiency; prioritized vertical partitioning impact; deduplication effectiveness; serialization impact; Arrow integration adaptability; sensitivity analysis; and overhead analysis of I/O requests, metadata storage and access latency.

Platform. We deploy a MinIO [19] storage cluster as remote storage and use a compute node for access experiments. The cluster has four machines, each with dual Intel® Xeon® E5-2620 CPUs, 64 GiB memory, and 23.8 TB HDD, running Ubuntu 20.04.6 LTS with GNU/Linux 5.15.0-105-generic x86_64, connected via 100 Mbps LAN. The compute node has dual HiSilicon KunPeng 920 CPUs, 128 GiB memory, 4 TiB HDD, and 4 TiB NVMe SSD, running Ubuntu 20.04.6 LTS with GNU/Linux 5.4.0-193-generic aarch64. Network bandwidth between the compute node and cluster is 1 Gbps. For local experiments, tests run directly on the compute node with local storage. System buffers and network caches are reset before each round to eliminate caching effects.

Dataset. We evaluate using the datasets in Figure 4, preprocessed via byte-stream conversion from Section 3.2 to meet experimental requirements. SQuAD [59] is a large-scale reading comprehension dataset with crowdworker-generated questions and text-extracted answers from Wikipedia. VoxCeleb [56] contains celebrity speech recordings from YouTube. LAION Aesthetics [60], a subset of LAION, includes images rated for aesthetic quality. ActivityNet [36] is a large-scale video understanding dataset with annotated videos of diverse human activities.

Configuration. The default configurations for different file formats, based on the datasets above, are as follows: Parquet, IPC, and Feather (via Arrow) use a row group size of 100,000 for SQuAD and VoxCeleb, or 250 for LAION Aesthetics and ActivityNet to control file size, while ORC's stripe size matches Parquet's row group count. Other settings, such as encoding and pre-cache strategies, follow Arrow's defaults. FlatStor uses the same configurations as Parquet. LanceV2's container size is set to 4 MiB. File sizes for all formats are kept below 1 GiB, and compression algorithms are avoided unless explicitly stated to prevent access performance degradation.

5.1 Inference Workload Testing

Many inference pipelines, especially retrieval-augmented ones (e.g., RAG [50]), use top-k vector search to find relevant items from a preprocessed corpus, with retrieved IDs resolved into full data entries, causing random point access. We use FAISS for vector retrieval, converting top-k results into data accesses via a reverse

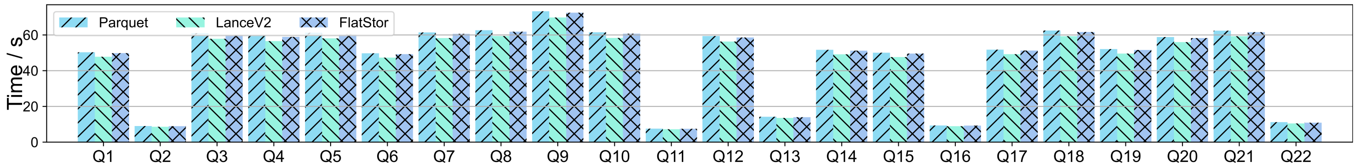


Figure 13: TPC-H benchmark results of FlatStor, Parquet and LanceV2

Table 4: MLPerf Storage benchmark results of FlatStor, Parquet and LanceV2

Workload	Dataset Size / GiB	Samples #. / s		
		Parquet	LanceV2	FlatStor
Unet3D	270	16.728	17.102	16.875
CosmoFlow	211	798.598	809.136	803.613

index to raw data in remote storage. Vector retrieval is common across methods and excluded from our results. Figures 11 and 12 show average access latency and read amplification for various datasets and formats. Parquet, ORC, IPC, and Feather exhibit poor access due to high read amplification and excessive I/O. LanceV2 accesses fixed-size containers, yielding stable latency but worse amplification for small objects (SQuAD, VoxCeleb). FlatStor supports random access via VTable, maintaining low read amplification and reducing data transfers, lowering latency by approximately 99.6% versus traditional methods and 41.3–61.6% versus LanceV2, demonstrating superior efficiency.

5.2 TPC-H Benchmark

In this subsection, we use DuckDB to run the TPC-H benchmark. Table data with scale factor sf=100 is generated and converted into various file formats. For each query, Arrow reads files directly or converts data into Arrow Tables. Relevant tables are loaded into a new DuckDB instance to avoid interference from existing tables, and SQL is executed. Results cover the entire process, from reading data in different layouts to completing TPC-H tasks. Figure 13 shows results for Parquet, LanceV2, and FlatStor. FlatStor performs similarly to Parquet. While LanceV2's container size is optimized for storage I/O, frequent offset retrieval reduces access performance. In contrast, FlatStor keeps detailed index info for column chunks and columns in the VTable, enabling efficient reading and consistently strong analytical performance.

5.3 MLPerf Benchmark

This subsection uses MLPerf Storage [34] to benchmark training workload performance. Due to limited format support in MLPerf, we enhanced compatibility of its underlying *dlio_benchmark* [10] for testing. Experiments run on a single node with four A100 accelerators. Table 4 shows the size of pre-generated training sample files, ensuring datasets prevent DRAM caching effects. MLPerf Storage measures performance in samples per second (samples/s). Results in Table 4 show FlatStor maintains a consistent sample loading rate compared to Parquet and LanceV2, preserving columnar storage benefits for training workloads.

Table 5: Access performance of different storage methods combined with serialization.

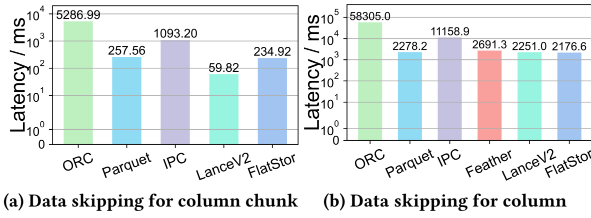
Storage Methods	Point Access / ms		Batch Access / ms	
	Caption	Image	Caption (1,000 Rows)	Image (100 Rows)
Row-based	5.13	36.47	95.34	3731.82
Horizontal (row group 100)	5.45	36.98	83.23	**3429.81**
Horizontal (row group 1,000)	5.22	37.92	**69.47**	3594.76
FlatStor	5.34	37.34	**72.76**	**3448.43**

5.4 Impact of Prioritized Vertical Partitioning

This subsection evaluates the improved flexibility in data management and access granularity enabled by prioritized vertical partitioning. Deduplication and serialization methods from Section 3 are also applied to row-based and traditional prioritized horizontal partitioning systems, for which demo versions are created. Using 10,000 caption-image pairs from LAION-Aesthetics, we assess point and batch access for specific columns. In prioritized horizontal partitioning, row groups are uniformly set to 1,000 or 100, while FlatStor sets caption column chunks to 1,000 and image chunks to 100. Table 5 shows performance results. Although serialization enables point access in all approaches, row-based storage performs worst for batch access due to row-by-row parsing. Traditional prioritized horizontal partitioning, with a single row group configuration, only supports efficient batch access for one column at a time. In contrast, FlatStor's prioritized vertical partitioning allows independent configuration of batch access granularity per column, enabling efficient batch access across multiple columns simultaneously.

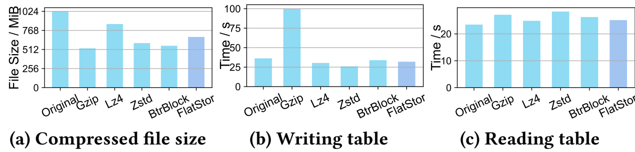
5.5 Data Skipping Access

This subsection evaluates data skipping performance across file formats. Data skipping, a key advantage of columnar storage, enables quick column retrieval via column names or Bloom filters on column chunks. The caption column from LAION Aesthetics is shuffled into a 10-column, 1,000,000-row table with unique column names, pre-converted into multiple uncompressed formats and stored in a remote cluster. Row group size is 100,000, with LanceV2 container size 4 MiB. Figures 14a and 14b show total latency for accessing column chunks and columns. Feather does not support column chunk access. LanceV2 returns data at container granularity during chunk skipping, with increased filter overhead. FlatStor achieves performance comparable to Parquet and other formats, preserving the benefits of columnar storage.



(a) Data skipping for column chunk **(b) Data skipping for column**

Figure 14: Data skipping access performance of different file formats



(a) Compressed file size **(b) Writing table** **(c) Reading table**

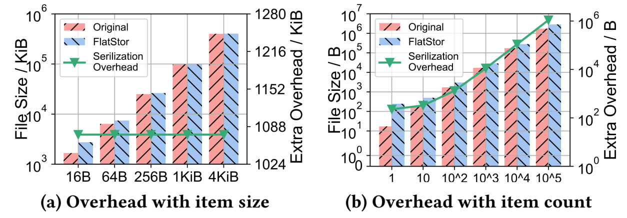
Figure 15: Compression efficiency compared to Parquet with different compression algorithms.

5.6 Data Deduplication Efficiency

This subsection evaluates the effectiveness of the data deduplication and compression scheme in FlatStor. The encoding and compression methods for modality-specific complex data follow standard practices and apply to formats like converting PNG to JPEG or using H.265 for videos. Thus, the compression process in FlatStor is consistent with conventional approaches. The focus is on how the FSST algorithm reduces storage overhead. Figure 15 compares file size and read/write performance using FSST in FlatStor with Parquet using no compression, gzip, lz4, zstd, and BtrBlock, based on the test table from Section 5.5. FlatStor reduces file size by 33.5%, improves compression by about 17%, and maintains access speed close to LZ4. BtrBlock achieves higher compression but with longer decoding time, making FlatStor more suitable for balanced workloads.

5.7 Impact of Serialization

In this subsection, we select the raw string data from the *caption* column of LAION Aesthetics. By randomly sampling sentences and truncating them, we generate data items of varying sizes (16B, 64B, 256B, 1KiB, 4KiB) and create five single-column tables of length 100,000 based on these items to evaluate the impact of item size on serialization.Additionally, we generate single-column tables of lengths 1, 10, 100, 1,000, 10,000, and 100,000 based on 4KiB-sized items to assess the impact of item count on serialization. Figure 16 presents our test results. The overhead caused by the FlatBuffers serialization in FlatStor depends solely on the number of items. Using 32-bit storage, maximum additional overhead per table is approximately $row_{num} \times column_{num} \times 4$ bytes. Since columns in FlatStor are managed independently, it is possible to configure whether each column requires embedded indexing to support point access. In scenarios where indexing is not needed (e.g., treating column chunks or column as a single data block), the additional overhead will be lower than the estimated value.



(a) Overhead with item size **(b) Overhead with item count**

Figure 16: Analysis of factors affecting serialization overhead

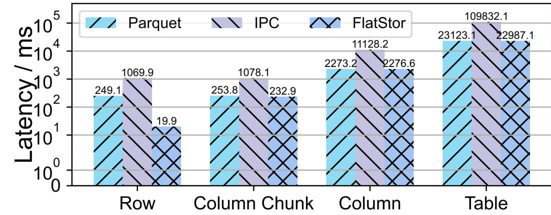


Figure 17: Integration capability testing of FlatStor and Arrow.

5.8 Adaptability to Arrow Integration

This subsection evaluates FlatStor integrated with Arrow using the testing table from Section 5.5. *Apache Arrow* provides a standardized columnar memory layout, enabling high-speed data transfer and processing across computing frameworks without memory copying, enhancing analysis efficiency and interoperability. Figure 17 demonstrates the performance of FlatStor with Arrow integration across access granularities. It achieves an 8.3% improvement in column chunk access compared to Parquet, and maintains similar column-level performance. It also excels in point access (row access), outperforming Parquet by about 9.65x. These findings align with prior experiments, confirming that integrating FlatStor into existing database or analytical systems boosts overall capabilities. This integration allows enterprises to support fine-grained access in new scenarios while preserving their current toolchains.

5.9 Sensitivity Analysis

FlatStor stores modality-specific complex data as byte streams (strings), making access to different data types equivalent to accessing string data (search is not discussed here). This subsection analyzes sensitivity by focusing on the impact of column count, rows per column, and size distribution on data I/O. Three sets of local test cases use the method from Section 5.7: 1) tables with 1,000 rows and columns numbering 10, 20, 50, 100, 200, 500, and 1,000, with fixed item size 1 KiB; 2) tables with 10 columns and rows numbering 10, 100, 1,000, 10,000, and 100,000, fixed item size 1 KiB; and 3) three single-column tables, each with 10,000 rows, data sizes from 16 B to 4 KiB, following uniform, Zipfian, and Gaussian distributions. Native Parquet lacks single item access, so to enable it, row group size is set to 1, greatly increasing metadata overhead—used as control. Except for the data distribution test (where Parquet row group size is 10,000), reads are at the column level and writes at the table level without compression. Results average multiple rounds.

Table 6: I/O performance under varying conditions (FlatStor demonstrates stable performance)

Column Num	Writing Table / ms			Reading Column/ ms		
	Parquet	LanceV2	FlatStor	Parquet	LanceV2	FlatStor
10	591.62	587.34	639.31	560.35	527.85	544.94
20	1743.89	1711.75	1799.34	567.03	551.15	553.66
50	5735.29	5713.34	5857.65	552.23	545.44	549.36
100	9628.83	9632.56	9925.71	579.76	558.22	564.88
200	22799.64	22725.45	24214.45	556.45	548.23	547.88
500	62641.98	62623.45	66116.87	598.75	583.11	585.04
1000	112822.51	112691.12	125382.45	557.34	549.15	553.86

Row Num	Writing Table / ms			Reading Column / ms		
	Parquet	LanceV2	FlatStor	Parquet	LanceV2	FlatStor
10	5.34	4.31	5.24	5.46	5.38	5.51
100	47.51	43.34	46.31	36.47	32.83	34.12
1000	591.34	587.45	609.31	529.85	512.08	514.39
10000	6948.31	6934.34	7025.34	5483.73	5290.13	5351.35
100000	79834.97	78681.45	80251.98	52082.09	51342.45	51752.93

Data Distribution	Writing Table / ms			Reading Item / ms		
	Parquet	LanceV2	FlatStor	Parquet	LanceV2	FlatStor
Uniform	192.12	189.34	194.45	25357.892	46.653	19.652
Zipfan	112.45	101.89	108.54	23262.012	42.893	17.532
Gaussian	234.12	213.76	223.13	24693.456	44.423	18.982

Table 6 presents I/O performance results for Parquet, LanceV2, and FlatStor under various column settings, row counts, and size distributions. As the number of columns increases, FlatStor experiences growing overhead from merging metadata across columns, leading to greater write latency than the other two methods. This higher write latency limits the applicability of FlatStor in scenarios with large numbers of columns. Under the same column configuration, all three methods show similar write latency, as shown in the middle and bottom tables. For read latency, LanceV2 benefits from its configuration of data access units based on containers, achieving better I/O performance than Parquet and FlatStor across different table shapes, while FlatStor consistently performs better than Parquet. When handling varying data distributions, Parquet must use a row group size of 1 to enable point access, which results in high metadata read costs and poor read performance. LanceV2, with containers holding multiple items as its minimum access unit, performs worse in data access compared to FlatStor, which supports finer-grained access to individual items.

5.10 Scalability Analysis

This subsection evaluates the scalability of FlatStor through two parts. First, we analyze how dataset size impacts the data point access performance of FlatStor, using subsets of the LAION-5B dataset at different scales. Then, we also evaluate point access performance under three network environments—1 Gbps, 10 Gbps, and high-performance IPoIB—while keeping the rest of the configurations same as Section 5.1. The results are shown in Figure 18. Figure 18a shows that access performance remains stable as dataset size increases. This is due to a simple preprocessing strategy, where large-scale data is pre-split into files of around 1–2 GiB before being stored in FlatStor, helping to maintain consistent access performance. Figure 18b presents the performance under the three network conditions. As network speed increases, data processing latency becomes a more visible bottleneck, suggesting further optimization opportunities in metadata handling and format parsing.

Table 7: Number of I/O operations of different formats for data at different levels

I/O Operations Number	Access Metadata			Access Data
	Table	Column	Column Chunk	
Parquet	1	1	/	>1
LanceV2	1	2	/	>1
FlatStor	1	2	3	>1

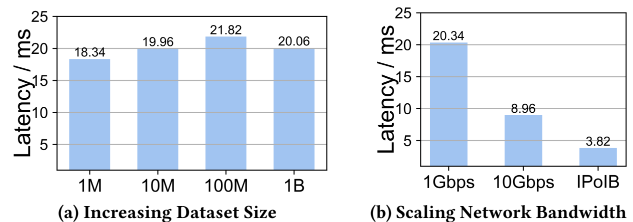


Figure 18: Scalability under Increasing Dataset Size and Network Bandwidth

5.11 Overhead Analysis

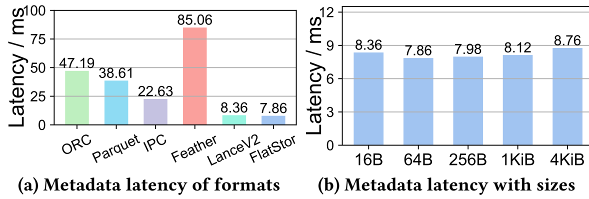
In this subsection, we evaluate the overhead of FlatStor from the perspectives of the number of I/O operations, metadata storage, and metadata access.

5.11.1 ***Number of I/O operations.*** Table 7 presents the number of I/O operations needed by Parquet, LanceV2, and FlatStor when accessing various file parts. The first two formats lack dedicated column chunk metadata. In Parquet, metadata is stored centrally, so only one access retrieves table or column metadata. LanceV2 manages data via containers; accessing table metadata requires one container, while column metadata involves first obtaining and parsing the table metadata container. In FlatStor, accessing metadata at different levels requires sequential retrieval and parsing of the Global Table, Table Metadata, and Column Metadata, involving 1, 2, and 3 I/O operations for table, column, and column chunk, respectively. Accessing actual data requires multiple I/Os proportional to data scale. This reflects how metadata design impacts overall I/O efficiency and access latency.

5.11.2 ***Total storage overhead.*** Based on previous analysis, storage efficiency of columnar formats varies across configurations for the same dataset. Several configurations are tested using the table from Section 5.9. A row group size of 100,000 is chosen for balanced data access, while a size of 1 assesses point query capabilities in formats like Parquet. Table 8 shows file sizes under these row group sizes. Results show a clear trend: as access granularity decreases, file sizes increase. FlatStor minimizes storage overhead for point queries by encoding row offsets in metadata. Metadata overhead, shown in Figure 16, grows linearly with entries and becomes negligible for larger datasets. Under point access in inference scenarios, FlatStor reduces total storage overhead by 91.6% versus Parquet with row group size 1, demonstrating higher efficiency. Comparing FlatStor, IPC, and Feather (an IPC-based format serialized by FlatBuffers) shows that even with identical serialization techniques, different workflows lead to significantly different storage outcomes.

Table 8: File sizes of the same *Table* stored in different formats different group size

Row Group Size	Parquet	IPC	Feather	FlatStor
1	5562.0MiB	1860.3MiB	2195.3MiB	538.2MiB
10	941.3MiB	529.8MiB	481.2MiB	487.7MiB
100	404.5MiB	404.1MiB	275.1MiB	482.7MiB
1000	346.7MiB	392.4MiB	239.7MiB	482.6MiB
10000	341.2MiB	391.2MiB	237.2MiB	482.3MiB
100000	339.1MiB	391.1MiB	236.6MiB	482.1MiB
Minimum Access Granularity	**Column Chunk**	**Column Chunk**	**Column**	**Row**



(a) Metadata latency of formats **(b) Metadata latency with sizes**

Figure 19: Metadata access latency of varying formats

5.11.3 *Metadata access latency.*

In Table 8, when the row group size is set to 100,000, the maximum storage overhead of FlatStor increases by approximately 42.2% compared to mainstream solutions. Since data compression is not used during writing, this overhead mainly comes from metadata storage and management. Metadata access overhead is evaluated by measuring the time to create a data handler object, representing complete metadata access latency. Figure 19a shows FlatStor reduces metadata access latency by 90.8% and 66.1% compared to Feather and IPC, respectively. FlatStor first initiates a small I/O request to obtain the footer offset (VTable), then accesses the needed metadata level. The access time in the figure includes the total time for these two I/O operations, representing a typical metadata access process. Thus, although FlatStor involves extra I/O operations, overall metadata access performance is not significantly affected, as shown in Figure 19b.

6 RELATED WORK

Columnar Layout. Currently, research on columnar storage focusing on AI scenarios is limited. Over the past decade, columnar formats such as the Capacitor [16] format developed by Google have been proposed and used in systems such as BigQuery [13] and Napa [29]. This format, based on technologies from Dremel [54] and Abadi [25], optimizes storage layout according to workload behavior. In 2019, YouTube developed the Artus format [37] for the Procella database management system, supporting adaptive encoding without block compression and providing O(1) addressing time for nested schemas. The DWRF format [12], a variant of ORC developed by Meta, offers improved support for reading and encrypting nested data. Recent studies have highlighted new data characteristics and I/O patterns in AI scenarios [52, 64]. However, current storage format designs primarily target analytical workloads and do not effectively address inference requirements. Hybrid systems like H2O [31] have explored the combination of multiple storage layouts to adapt to varied workloads, yet such approaches mainly target structured data and do not directly address challenges of heterogeneous multimodal data.

Data Caching. Caching mitigates read amplification by replacing amplified portions with data likely to be accessed soon, thereby improving overall read efficiency. However, during model training, data is loaded randomly to reduce model bias, which conflicts with caching's reliance on data locality. Similarly, in inference, the highly discrete and random nature of requests renders caching largely ineffective. Although Shade [48] is specifically designed as a cache for training workloads, its importance sampling method is impractical [46, 47], since determining data importance requires prior training, creating a paradox. NoPFS [40] optimizes training by sharing the random shuffle seed with cache nodes, generating the same sequence and enabling pre-caching to improve I/O performance. However, it still does not fully address read amplification during retrieval, as fetching blocks continues to load unnecessary data, limiting overall efficiency gains.

Data Compression. Another approach to improving I/O performance is data compression, which reduces read amplification by lowering I/O volume and minimizing transfer overhead. FSST [35] enables random access to compressed data using a static dictionary-based table, eliminating full block decompression but struggling with efficient random access in distributed storage due to lost boundary information and inconsistent block alignment. Zhang et al. [65] record index information within compressed data to enable GPU random access and processing of compressed data, but this approach is highly specific and incompatible with columnar storage systems. Zeng et al. [64] argue that query bottlenecks have shifted from storage to computation, suggesting future formats should avoid block compression or heavyweight encodings unless justified, to optimize both storage efficiency and computational performance.

7 CONCLUSION

This paper proposes a columnar format supporting multimodal data and random point access, while maintaining efficient analytical and training workloads. To our knowledge, no existing solution addresses such diverse demands within a unified columnar framework. We analyze traditional columnar storage limitations and propose key designs: vertical partitioning, random-accessible deduplication, and metadata compression. These improve access efficiency and reduce storage overhead. Compared to mainstream formats, our solution cuts latency by 99.6% and total storage overhead by 91.3% for random point access, while maintaining stable performance on traditional workloads. Notably, FlatStor's batch access performance is comparable to existing formats, reflecting a deliberate trade-off favoring low-latency point access over batch optimization. This enables FlatStor to serve large-scale enterprises with multimodal data and diverse usage, though its advantages may be limited in workloads dominated by frequent writes or full scans.

ACKNOWLEDGMENTS

This work was supported by the National Key R&D Program of China (Grant No. 2023YFB4502900) and by Huawei Cloud. The authors would like to express their sincere gratitude to Dr. Jiong Lou from Shanghai Jiao Tong University for insightful discussions and valuable feedback.

REFERENCES

[1] 2025. Amazon Redshift. https://aws.amazon.com/redshift/ Accessed: Feb 9, 2025.
[2] 2025. Amazon S3. https://aws.amazon.com/s3/ Accessed: Feb 9, 2025.
[3] 2025. Apache Arrow. https://arrow.apache.org/ Accessed: Feb 9, 2025.
[4] 2025. Apache Arrow Columar. https://arrow.apache.org/docs/format/Columnar.html Accessed: Feb 9, 2025.
[5] 2025. Apache Flink. https://flink.apache.org/ Accessed: Feb 9, 2025.
[6] 2025. Apache ORC. https://orc.apache.org/ Accessed: Feb 9, 2025.
[7] 2025. Apache Parquet. https://parquet.apache.org/ Accessed: Feb 9, 2025.
[8] 2025. Apache Spark. https://spark.apache.org/ Accessed: Feb 9, 2025.
[9] 2025. Apache Thrift. https://thrift.apache.org/ Accessed: Feb 9, 2025.
[10] 2025. DLIO Benchmark. https://github.com/argonne-lcf/dlio_benchmark Accessed: Feb 9, 2025.
[11] 2025. DuckDB. https://duckdb.org/ Accessed: Feb 9, 2025.
[12] 2025. The DWRF Format. https://github.com/facebookarchive/hive-dwrf Accessed: Feb 9, 2025.
[13] 2025. Google BigQuery. https://cloud.google.com/bigquery Accessed: Feb 9, 2025.
[14] 2025. Google Protobuf. https://github.com/protocolbuffers/protobuf Accessed: Feb 9, 2025.
[15] 2025. Hugging Face. https://huggingface.co/ Accessed: Feb 9, 2025.
[16] 2025. Inside Capacitor, BigQuery's next-generation columnar storage format. https://cloud.google.com/blog/products/bigquery/inside-capacitorbigquerys-next-generation-columnar-storage-format Accessed: Feb 9, 2025.
[17] 2025. LanceDB. https://lancedb.com/ Accessed: Feb 9, 2025.
[18] 2025. LanceV2 Format. https://blog.lancedb.com/lance-v2/ Accessed: Feb 9, 2025.
[19] 2025. MinIO. https://min.io/ Accessed: Feb 9, 2025.
[20] 2025. PostgreSQL. https://www.postgresql.org/ Accessed: Feb 9, 2025.
[21] 2025. Redis. https://redis.io/ Accessed: Feb 9, 2025.
[22] 2025. RocksDB. https://rocksdb.org/ Accessed: Feb 9, 2025.
[23] 2025. Snowflake. https://www.snowflake.com/en/ Accessed: Feb 9, 2025.
[24] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Found. Trends Databases* 5, 3 (2013), 197–280. https://doi.org/10.1561/1900000024
[25] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 671–682.
[26] Daniel J Abadi et al. 2007. Column Stores for Wide and Sparse Data.. In *CIDR*, Vol. 2007. 292–297.
[27] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. 2009. Column-oriented database systems. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1664–1665.
[28] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: how different are they really?. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 967–980.
[29] Ankur Agiwal, Kevin Lai, Gokul Nath Babu Manoharan, Indrajit Roy, Jagan Sankaranarayanan, Hao Zhang, Tao Zou, Min Chen, Zongchang Chen, Ming Dai, et al. 2021. Napa: Powering scalable data warehousing with robust query performance at Google. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2986–2997.
[30] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*. Morgan Kaufmann, 169–180. http://www.vldb.org/conf/2001/P169.pdf
[31] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. 2014. H2O: a hands-free adaptive store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 1103–1114. https://doi.org/10.1145/2588555.2610502
[32] Michael R Anderson and Michael Cafarella. 2016. Input selection for fast feature engineering. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 577–588.
[33] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. 2022. Amazon Redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*. 2205–2217.
[34] Oana Balmau. 2022. Characterizing I/O in machine learning with mlperf storage. *ACM SIGMOD Record* 51, 3 (2022), 47–48.
[35] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2649–2661.
[36] Fabian Caba Heilbron, Victor Escorcia, Bernard Ghanem, and Juan Carlos Niebles. 2015. Activitynet: A large-scale video benchmark for human activity understanding. In *Proceedings of the ieee conference on computer vision and pattern recognition*. 961–970.
[37] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew Mccormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, et al. 2019. Procella: Unifying serving and analytical data at YouTube. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2022–2034.
[38] George P. Copeland and Setrag Khoshafian. 1985. A Decomposition Storage Model. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, USA, May 28-31, 1985*, Shamkant B. Navathe (Ed.). ACM Press, 268–279. https://doi.org/10.1145/318898.318923
[39] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*. 215–226.
[40] Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, and Torsten Hoefler. 2021. Clairvoyant prefetching for distributed machine learning I/O *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 92, 15 pages. https://doi.org/10.1145/3458817.3476181
[41] Chenzhuang Du, Jiaye Teng, Tingle Li, Yichen Liu, Tianyuan Yuan, Yue Wang, Yang Yuan, and Hang Zhao. 2023. On uni-modal feature learning in supervised multi-modal learning. In *International Conference on Machine Learning*. PMLR, 8632–8656.
[42] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1917–1923.
[43] Ahmed A. Harby and Farhana Zulkernine. 2022. From Data Warehouse to Lakehouse: A Comparative Review. In *2022 IEEE International Conference on Big Data (Big Data)*. 389–395. https://doi.org/10.1109/BigData55660.2022.10020719
[44] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. 2011. RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 1199–1208.
[45] Todor Ivanov and Matteo Pergolesi. 2020. The impact of columnar file formats on SQL-on-hadoop engine performance: A study on ORC and Parquet. *Concurrency and Computation: Practice and Experience* 32, 5 (2020), e5523.
[46] Tyler B Johnson and Carlos Guestrin. 2018. Training deep models faster with robust, approximate importance sampling. *Advances in Neural Information Processing Systems* 31 (2018).
[47] Angelos Katharopoulos and François Fleuret. 2018. Not all samples are created equal: Deep learning with importance sampling. In *International conference on machine learning*. PMLR, 2525–2534.
[48] Redwan Ibne Seraj Khan, Ahmad Hossein Yazdani, Yuqi Fu, Arnab K Paul, Bo Ji, Xun Jian, Yue Cheng, and Ali R Butt. 2023. SHADE: Enable Fundamental Cacheability for Distributed Deep Learning Training. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 135–152.
[49] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM Manag. Data* 1, 2, Article 118 (June 2023), 26 pages. https://doi.org/10.1145/3589263
[50] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2021. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. arXiv:2005.11401 [cs.CL] https://arxiv.org/abs/2005.11401
[51] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. 2023. A deep dive into common open formats for analytical dbmss. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3044–3056.
[52] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. 2023. A deep dive into common open formats for analytical dbmss. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3044–3056.
[53] Dipankar Mazumdar, Jason Hughes, and JB Onofre. 2023. The Data Lakehouse: Data Warehousing and More. *arXiv preprint arXiv:2310.08697* (2023).
[54] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: a decade of interactive SQL analysis at web scale. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3461–3472. https://doi.org/10.14778/3415478.3415568
[55] Derek G Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. 2021. tf. data: A machine learning data processing framework. *arXiv preprint arXiv:2101.12127* (2021).
[56] Arsha Nagrani, Joon Son Chung, and Andrew Zisserman. 2017. Voxceleb: a large-scale speaker identification dataset. *arXiv preprint arXiv:1706.08612* (2017).
[57] Junting Pan, Keqiang Sun, Yuying Ge, Hao Li, Haodong Duan, Xiaoshi Wu, Renrui Zhang, Aojun Zhou, Zipeng Qin, Yi Wang, Jifeng Dai, Yu Qiao, and Hongsheng Li. 2023. JourneyDB: A Benchmark for Generative Image Understanding. arXiv:2307.00716 [cs.CV]
[58] Ippokratis Pandis. 2021. The evolution of Amazon redshift. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3162–3174.

[59] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* (2016).
[60] Christoph Schuhmann, Romain Beaumont, Richard Vencu, Cade Gordon, Ross Wightman, Mehdi Cherti, Theo Coombes, Aarush Katta, Clayton Mullis, Mitchell Wortsman, et al. 2022. Laion-5b: An open large-scale dataset for training next generation image-text models. *Advances in Neural Information Processing Systems* 35 (2022), 25278–25294.
[61] Christoph Schuhmann, Richard Vencu, Romain Beaumont, Robert Kaczmarczyk, Clayton Mullis, Aarush Katta, Theo Coombes, Jenia Jitsev, and Aran Komatsuzaki. 2021. Laion-400m: Open dataset of clip-filtered 400 million image-text pairs. *arXiv preprint arXiv:2111.02114* (2021).
[62] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 1–10.
[63] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.
[64] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *Proceedings of the VLDB Endowment* 17, 2 (2023), 148–161.
[65] Feng Zhang, Yihua Hu, Haipeng Ding, Zhiming Yao, Zhewei Wei, Xiao Zhang, and Xiaoyong Du. 2022. Optimizing random access to hierarchically-compressed data on GPU. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.
[66] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, et al. 2022. Understanding data storage and ingestion for large-scale deep recommendation model training: Industrial product. In *Proceedings of the 49th annual international symposium on computer architecture*. 1042–1057.
[67] Siyun Zhao, Yuqing Yang, Zilong Wang, Zhiyuan He, Luna K. Qiu, and Lili Qiu. 2024. Retrieval Augmented Generation (RAG) and Beyond: A Comprehensive Survey on How to Make your LLMs use External Data More Wisely. arXiv:2409.14924 [cs.CL] https://arxiv.org/abs/2409.14924
[68] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory* 23, 3 (1977), 337–343. https://doi.org/10.1109/TIT.1977.1055714
[69] Jacob Ziv and Abraham Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory* 24, 5 (1978), 530–536. https://doi.org/10.1109/TIT.1978.1055934

TIMEST: Temporal Information Motif Estimator Using Sampling Trees

Yunjie Pan
University of Michigan
Ann Arbor, Michigan, USA
panyj@umich.edu

Omkar Bhalerao
University of California, Santa Cruz
Santa Cruz, California, USA
obhalera@ucsc.edu

C. Seshadhri
University of California, Santa Cruz
Santa Cruz, California, USA
sesh@ucsc.edu

Nishil Talati
University of Michigan
Ann Arbor, Michigan, USA
talatin@umich.edu

ABSTRACT

The mining of pattern subgraphs, known as motifs, is a core task in the field of graph mining. Edges in real-world networks often have timestamps, so there is a need for *temporal motif mining*. A temporal motif is a richer structure that imposes timing constraints on the edges of the motif. Temporal motifs have used to analyze social networks, financial transactions, and biological networks.

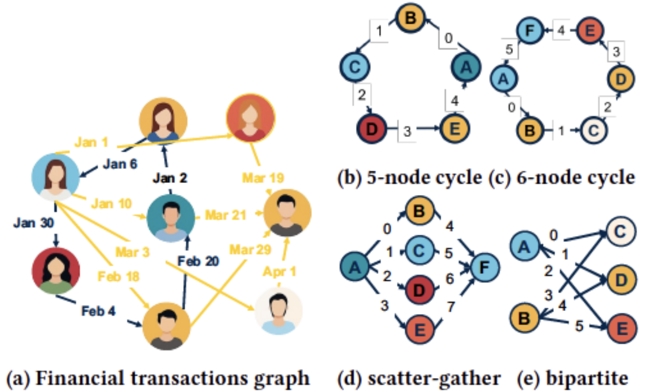
Motif counting in temporal graphs is particularly challenging. A graph with millions of edges can have trillions of temporal motifs, since the same edge can occur with multiple timestamps. There is a combinatorial explosion of possibilities, and state-of-the-art algorithms cannot manage motifs with more than four vertices.

In this work, we present TIMEST: a general, fast, and accurate estimation algorithm to count temporal motifs of arbitrary sizes in temporal networks. Our approach introduces a temporal spanning tree sampler that leverages weighted sampling to generate substructures of target temporal motifs. This method carefully takes a subset of temporal constraints of the motif that can be jointly and efficiently sampled. TIMEST uses randomized estimation techniques to obtain accurate estimates of motif counts.

We give theoretical guarantees on the running time and approximation guarantees of TIMEST. We perform an extensive experimental evaluation and show that TIMEST is both faster *and* more accurate than previous algorithms. Our CPU implementation exhibits an average speedup of 28× over state-of-the-art GPU implementation of the exact algorithm, and 6× speedup over SOTA approximate algorithms while consistently showcasing less than 5% error in most cases. For example, TIMEST can count the number of instances of a financial fraud temporal motif in four minutes with 0.6% error, while exact methods take more than *two days*.

PVLDB Reference Format:
Yunjie Pan, Omkar Bhalerao, C. Seshadhri, and Nishil Talati. TIMEST: Temporal Information Motif Estimator Using Sampling Trees. PVLDB, 19(1): 15 - 28, 2025.
doi:10.14778/3772181.3772183

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 1 ISSN 2150-8097.
doi:10.14778/3772181.3772183



(a) Financial transactions graph (b) 5-node cycle (c) 6-node cycle (d) scatter-gather (e) bipartite

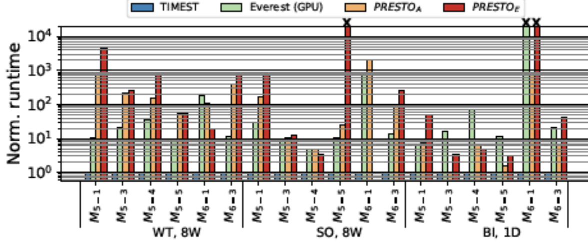
Figure 1: (a) An example of financial transactions with timestamps. Different money-laundering and gambling pattens [6, 28] including (b) 5-node simple-cycle; (c) 6-node simple-cycle; (d) scatter-gather; and (e) bipartite.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/pyjhzwh/SpanningTreeSampling/.

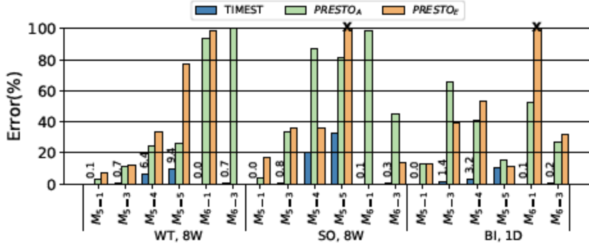
1 INTRODUCTION

A central tool in the analysis of large networks is *motif mining* [28, 33, 35, 41, 55]. A motif is a small pattern graph that indicates some special structure in a larger input graph. Motifs offer valuable insight into the graph structure, and are analogous to queries that detail a set of edge constraints. The algorithmic problem of motif mining, often called subgraph counting/finding in the algorithms literature, has a rich history in the data mining literature [5, 36, 52]. (See survey [50].)

Most real-world graphs are *temporal*, where edges come with timestamps. There is a surge of recent work on mining temporal graphs, especially searching for temporal motifs (refer to survey [14]). These works show temporal motifs can capture richer information than standard motifs [23, 38]. Temporal motifs have been used for user behavior characterization on social networks [24, 26, 41], detecting financial fraud [15, 28], characterizing function of biological networks [17, 38], and improving graph neural networks [8, 28].



(a) Runtime of prior works normalized to TIMEST. TIMEST has an average speedup of 28× compared to previous works.



(b) Relative error (%) of approximate algorithms. TIMEST is almost always the most accurate while always the fastest.

Figure 2: We show the runtime improvement and estimation errors of TIMEST versus Everest [65] (GPU) and approximate algorithms [47] ($PRESTO_A$ and $PRESTO_E$) on various datasets and motifs. Everest is tested on an NVIDIA A40 GPU, while others are evaluated on a CPU using 32 threads. "x" means run out of memory (OOM) or timeout (> 1 day).

A temporal graph can be viewed as a graph database with a rich set of attributes such as vertex/node labels and edge timestamps. Mining motifs in temporal graphs is analogous to executing queries in traditional database systems.

1.1 Problem Definition

We set some notation. The input is a multigraph $G = (V(G), E(G))$, with n vertices and m edges. Each edge in G is a tuple (u, v, t) where u and v are source and destination vertices, and t is a positive integer timestamp. Following prior work [30, 65], we assume that each tuple (u, v, t) can appear only once. However, the same u, v can have multiple edges between them with different timestamps, and there can be different tuples with the same timestamp. For a temporal edge e, let $t(e)$ to denote its timestamp. A temporal motif involves a standard graph motif with a constraint on the time window the motif occurs in, and a constraint on the order of edges. We formally define temporal motifs and temporal matches, following Paranjape *et al.* [41].

Definition 1.1. *A* temporal motif *is a triple $M = (H, \pi, \delta)$ where (i) $H = (V(H), E(H))$ is a directed pattern graph, (ii) π is an ordered set/list of the edges of H, and (iii) δ is a positive integer.*

The ordered list π specifies the time ordering of edges, and δ specifies the length of the time interval in which edges must occur.

Abusing notation, for an edge e in the list π, we use $\pi(e)$ to denote its index in the list.

Definition 1.2. *Consider an input temporal graph $G = (V(G), E(G))$ and a temporal pattern $M = (H, \pi, \delta)$. An M-*match *is a pair (ϕ_V, ϕ_E) 1-1 maps $\phi_V : V(H) \to V(G)$, $\phi_E : E(H) \to E(G)$ satisfying the following conditions.*

- *(Matching the edges) $\forall (u, v) \in E(H), \phi_E(u, v) = (\phi_V(u), \phi_V(v))$*
- *(Matching the pattern) $\forall (u, v) \in E(H), (\phi_V(u), \phi_V(v)) \in E(G)$.*
- *(Edges ordered correctly) The timestamps of the edges in the match follow the ordering π. Formally, $\forall e, e' \in E(H), \pi(e) < \pi(e')$ iff $t(\phi_E(e)) < t(\phi_E(e'))$.*
- *(Edges in time interval) All edges of the match occur within δ time range. Formally, $\forall e, e' \in E(H), |t(\phi_E(e)) - t(\phi_E(e'))| \leq \delta$.*

Figure 1a shows an example of financial transactions, represented as a graph. We also four temporal motifs, which have been explicitly mentioned as indications of money laundering [51, 55]. Note that the edges of the cycle must occur in temporal order (given the edge label), for this to represent a valid cyclic money flow. More temporal motifs are given in Figure 3. We note that a natural generalization is to impose a *partial order* of time constraints, which we discuss more in future work.

1.2 The Challenge

Temporal motif counting is particularly difficult because of *combinatorial explosion*. There can be thousands of temporal edges between the same vertices, which leads to a exponentially larger search space for motifs. For example, there are trillions of temporal 5-cliques in Bitcoin graph with 100 million edges. Exact methods based on enumeration or exploration cannot avoid this massive computation [30, 41]. Many techniques for reducing the search space using graph properties like the degeneracy and dynamic programming are tailored for simple graphs [9, 44, 49, 50]. These techniques cannot incorporate ordering constraints on edges. In general, when the motif has four vertices, no exact method is able to get results on graphs with 100M edges even in a day with commodity hardware [25, 41, 42].

There are two approaches to temporal motif counting. The usual method is to design general purpose algorithms that work for (potentially) any motif. Exact motif counting algorithms rely on explicit subgraph enumeration that suffers a massive computational explosion [30, 41, 65]. There are general purpose estimators for all kinds of temporal motifs, like IS [29], ES [59] and PRESTO [47]. However, they tend to perform poorly on larger motifs, since they rely on exact algorithm in subsampled intervals. For example, our evaluation shows that PRESTO runs for more than 5 hours estimating a 5-clique (*i.e.,* M_{5-5} in Figure 3) with a high 25% estimation error.

Other methods analyze specific motifs and designs specialized algorithms. These include 2SCENT [25] (for simple cycles), DOTT [42] (for triangles), Gao *et al.* [13] (caters to 2-, 3-node, 3-edge motifs), and TEACUPS [39] (for 4-node motifs), alongside other specialized motifs such as bi-triangle [62], bi-clique [64], and butterfly [11, 45] motifs for bipartite networks. While these often work well for the

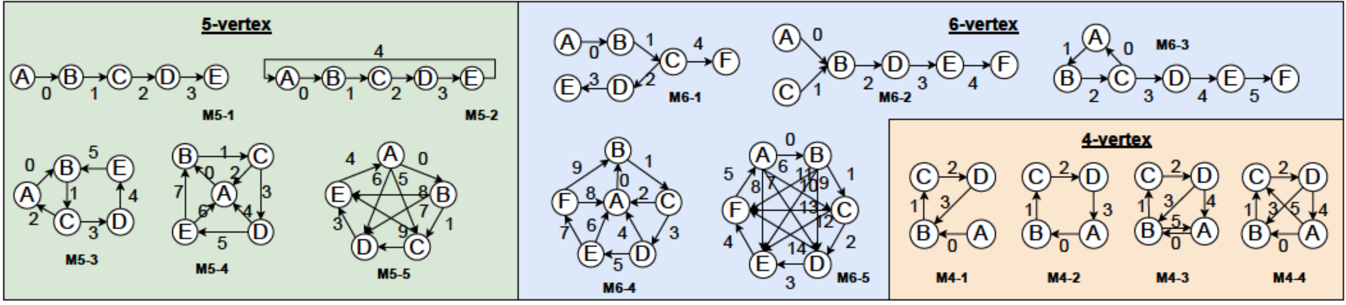


Figure 3: Connected 4-, 5- and 6-vertex temporal motifs used for evaluation.

tailored motif, they do not give general purpose algorithms. Common motifs with more than 5 nodes occurs in money laundering applications [6, 51, 55]. These motifs are shown in Figure 1, and cannot be handled efficiently by existing methods.

This leads to the main question of our paper: *do there exist general purpose temporal motif counting algorithms that can scale beyond current motif sizes?*

1.3 Our Results

Our main result is TIMEST, a general and efficient randomized (approximate) temporal motif mining algorithm. TIMEST is a general purpose method that works for any motif. As a concrete example, it is able to efficiently approximate counts for all the example motifs in Figure 3. We introduce a new technique of *temporal spanning tree sampling* that generalizes previous temporal results [40] and older non-temporal methods [21, 49, 60]. We list out some specific contributions of our work.

Temporal spanning tree sampling: Our primary contribution is a new technique of *temporal spanning tree sampling*, that generalizes a recent method of temporal path sampling [40]. To count temporal motifs of arbitrary size, we find an appropriate spanning tree of the motif. These spanning trees satisfy partial temporal constraints of this motif. We then sample many such trees, and estimate the fraction of them that "extend" to the motif.

A direct temporal spanning tree would simply take a subset of the edges *and* the corresponding temporal constraints. But it is not clear how to efficiently sample such a tree with these constraints. We show that by a careful relaxation of some of the temporal constraints, an efficient sampler can be constructed. This sampler quickly preprocesses the graph and can output any desired number of uniform random spanning trees.

Sample reduction techniques: To reduce the sample complexity, we design heuristics that choose the "best" temporal spanning tree. We can formally show that the temporal spanning tree with the fewest matches is the best choice, but finding this tree would itself require a large number of motif counts. We design heuristics to narrow down on the best spanning tree, which leads to lower error with the same sampling budget.

Practical efficiency of TIMEST: We perform a comprehensive evaluation across various datasets (Table 2) and a large collection of complex motifs (Figure 3), up to 6 vertices. We compare with a state-of-the-art exact algorithm that uses GPUs (Everest [65]) and versions of the best randomized method (Presto [47]). Results are in Figure 2a. TIMEST is *always* faster, with a typical 10x-100x speedup over both existing methods. We note that TIMEST is implemented on a CPU, and still has an average speedup of 28× over high-end NVIDIA GPU implementations. We do not give results for the motifs M_{6-4} and M_{6-5} since previous methods timeout even after running for a day. TIMEST is able to get approximate counts even for this motifs. On the specific money-laundering bipartite motif in Figure 1, TIMEST takes *4 minutes* on an example dataset, while the best exact method takes *two days*.

Low error of TIMEST: Despite the variety of complex constraints in Figure 3, in *all* cases, TIMEST has significantly lower error. We plot the errors of previous work and TIMEST in Figure 2b. TIMEST typically has an error of less than 5%, while previous methods have 20% error or more. The only hard motifs for TIMEST are M_{5-4} and M_{5-5}, which have more than 20% error. But other methods have twice as much (or much more) error.

Theoretical analysis of TIMEST: We give a comprehensive theoretical analysis of TIMEST, giving bounds on all running time steps, and an analysis of the sample complexity with respect to output error.

2 HIGH-LEVEL IDEAS IN TIMEST

The final TIMEST algorithm is fairly complex with multiple moving parts. In this section, we give an overview of various procedures in TIMEST. We will denote the input graph as G and the motif as M. A depiction of the TIMEST pipeline is given in Figure 4. There are two primary steps.

Preprocessing: This is the main novelty in TIMEST. We first choose an appropriate spanning tree of M and identify of a set of *relaxed temporal constraints.* This process itself requires computations involving the input graph G. Our aim is to select a temporal "submotif" T of M that can be quickly sampled. We set up the temporal constraints to allow for a dynamic programming bottom-up approach to sample uniform random instances of T. This approach requires computing a series of edge weights, called the *sampling weights.* Since there are multiple temporal edges between the same pair of vertices, the edge weights can be complicated to compute.

Sampling: These weights allow us to apply a common paradigm in approximate motif counting that involves sampling a substructure, and extending to the motif [9, 21, 40, 49, 60]. We show how the weights can be used to subsample subtrees of T, which can then be incrementally extended to a uniform random sample of T. A quick validation verifies that the sample satisfies the temporal constraints

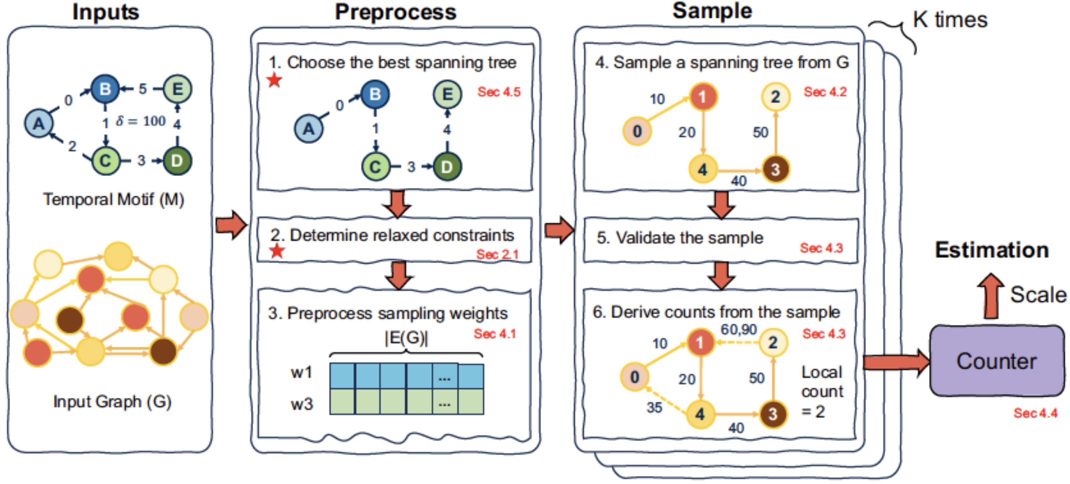
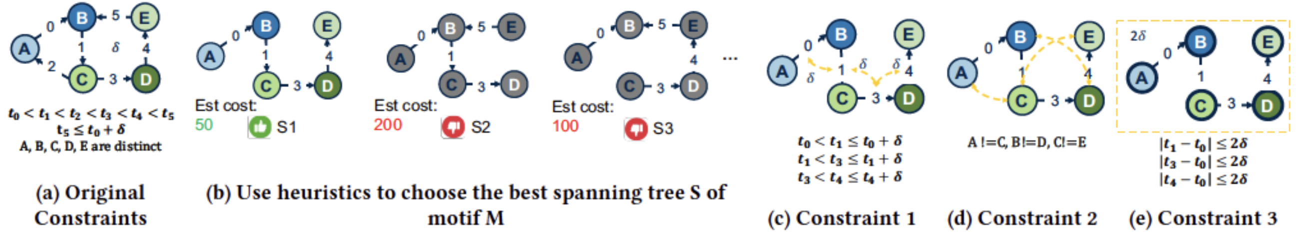


Figure 4: Overall process of estimating temporal motif M counts in the input graph G.



(a) Original Constraints (b) Use heuristics to choose the best spanning tree S of motif M (c) Constraint 1 (d) Constraint 2 (e) Constraint 3

Figure 5: An example of mining motif $M_{5\text{-}3}$. (a) The standard approach strictly enforces all constraints for a temporal motif match. (b) Multiple spanning trees of the motif exist, and we use a heuristic to select the optimal one that has the lowest estimated sampling cost (Est cost). We then relax specific constraints and apply three partial constraints to the spanning tree: (c) enforcing edge order and δ-window constraints for adjacent edges; (d) ensuring distinct end vertices for adjacent edges; and (e) constraining all edges to fall within a 2δ time window.

of M. The next step is to count the number of matches to M that involve the sampled match to T. Again, temporal graphs bring in challenges. In the typical setting, a single T can only extend to a small number of matches to M. Here, the sampled T could have many temporal edges going between the vertices, and hence an unbounded number of matches to M come from a *single* sampled T. We devise an algorithm using a careful binary searching over various edge lists to compute this number of matches. The sampling step is repeated for a collection of samples. Standard randomized algorithms techniques tell us how to rescale the total number of matches found, to get an estimate for the total count of M.

We go deeper into the preprocessing step, which requires many new ideas.

2.1 A Closer Look at Preprocessing

Consider mining the temporal motif $M_{5\text{-}3}$ as shown in Figure 5. We fix the specific spanning tree shown in the figure. The original motif imposes an ordering constraint on the edges and a total time window constraint. It is not clear how to sample such a spanning tree, so we relax some of the constraints.

We relax the requirements for all edges to be ordered correctly and to fall within a specified time interval from Definition 1.2. These constraints are only applied to certain pairs of *adjacent edges*. This forms the base constraint (Constraint 1), as illustrated in Figure 5c. In complex spanning trees, these constraints leave out multiple ordering conditions.

We note that most efficient mechanisms to sample trees use dynamic programming, and technically sample tree homomorphisms (where non-adjacent pattern vertices can be matched to the same vertex in G). To address this, we introduce a straightforward constraint ensuring that the end vertices of adjacent edges are distinct. This approach minimizes memory overhead and significantly reduces invalid samples, making it particularly effective in skewed graphs. This is described as Constraint 2 is illustrated in Figure 5d.

Constraints 1 and 2 focus on adjacent edges, while Constraint 3 introduces a global constraint of the time interval. For example, in the chosen spanning tree of $M_{5\text{-}3}$, the sequential timing conditions of Constraint 1 implies $t_0 < t_4 \leq t_0 + 3\delta$, potentially violating the δ time interval constraint. (We use t_i to denote the timestamp the edge labeled i.) To reduce the affect of these violations, we apply a

2δ sliding window approach, partitioning the input graph into overlapping subgraphs based on timestamps. This results in intervals like $[0, 2\delta]$, $[\delta, 3\delta]$, and so on, ensuring that any temporal motif matches within a δ-time window must reside entirely within a single interval and will not span across the boundary of two intervals. This approach effectively reduces the time window length to 2δ, significantly reducing the occurrence of invalid matches due to δ-window violations. An illustration of Constraint 3 is provided in Figure 5e. Note that this approach leads to the under/over counting of some temporal motifs due to boundary conditions. Since this overcounting factor can be determined for any match, we can apply corrections to the final estimate.

All in all, these constraints create a subset of matches for the spanning tree. We prove that by a multi-pass preprocessing over the graph, we can construct an efficient sampler for these matches of the spanning tree. This process involves a dynamic program that constructs weights for various subtrees, and then computes weights for larger subtrees. The constraints discussed above allow for this dynamic programmin approach to work.

Choose a Spanning Tree: The first step of preprocessing requires selecting the best spanning tree. As shown in Figure 5b, a motif can have multiple valid spanning tree candidates. We try to determine the spanning tree that will lead to the fewest number of samples required for convergence. This is an important distinction from previous spanning tree based sampling works that fix a choice [21, 40, 49, 60]. Those methods could afford to do that because they deal with smaller and non-temporal motifs. We need to design some heuristics to select a smaller pool of spanning trees, and then exactly determine the sampling cost for these trees. By selecting the tree with the lowest sampling cost, we ensure efficient and accurate motif counting during execution. The details are elaborated in Section 4.5.

3 RELATED WORK

Static Motif Mining. There have been a lot of research in static motif mining, including exact counting and enumeration approaches [12, 19, 20, 31, 32, 53, 56, 61], and approximate algorithms [3, 10, 18, 21, 21, 29, 43, 47–49, 57–60, 62–64] Although static motif mining can serve as a first step for temporal motif matching, as illustrated by Paranjape *et al.* [41], this often results in orders of magnitude redundant work [30, 65] due to the presence of temporal constraints.
Exact Temporal Motif Mining. Temporal motif mining was first formally introduced by Paranjape *et al.* [41], who suggested an algorithm to list and count instances of temporal motifs. A newer exact counting method [30] introduced a backtracking algorithm by listing all instances on edges sorted chronologically. Everest [65] further refines this approach, employing system-level optimizations on GPUs to enhance performance substantially. There are also numerous algorithms designed for general temporal motif counting [34, 54] and others tailored to specific motifs [7, 13, 25, 42]. Despite these advancements, scalability remains a significant issue.
Approximate Temporal Motif Mining. To approximate temporal motif counts, various sampling-based methods balance efficiency and accuracy. There are several general frameworks for estimating temporal motifs. Our work differs fundamentally from those methods such as IS [29], PRESTO [47], and ES [59] in how it handles motif structure and temporal constraints. IS [29] partitions the timeline into non-overlapping intervals and performs exact motif counting on a sampled subset of intervals. Its performance is highly sensitive to interval size and may miss cross-interval patterns. PRESTO [47] improves upon IS by avoiding rigid partitioning and instead using uniform sampling across the entire graph. While more flexible, it still treats motif instances in a uniform sampling space and does not incorporate structural information. ES [59] samples individual edges uniformly and enumerates all local motif instances around them. It does not exploit the motif's topology beyond local neighborhoods, leading to inefficiencies and high variance for larger motifs. In contrast, TIMEST introduces a spanning tree–based sampling framework that leverages the motif's structural backbone and performs weighted sampling along constrained subspaces.

There are methods that rely on an exact counting for some sampled graph, such as Ahmed *et al.* [4] and OdeN [46]. These are often less efficient and have large estimator variance. There are methods tailored to the unique characteristics of specific motifs. Cai *et al.* [11] and Pu *et al.* [45] specialize in butterfly motifs in bipartite graphs, Oettershagen *et al.* [37] explore 2- and 3-node motifs and Pan *et al.* [40] focus on 4-node motifs.

Table 1: Summary of notations

Symbol	Definition
δ	maximum time window
$M = (H, \pi, \delta)$	temporal motif
G	input graph with $m = \lvert E(G)\rvert$ edges and $n = \lvert V(G)\rvert$ vertices
ϕ_V, ϕ_E	vertex map $\phi_V : V(H) \to V(G)$ and edge map $\phi_E : E(H) \to E(G)$
$t(e)$	time stamp of edge e
S	spanning tree of a temporal motif
P	sampled edges from G that is mapped to S
$w_{s,e}$	sampling weight of edge e in G that is mapped to s in S
W	total sampling weights
$C, \hat{C}$	ground truth and estimated motif count

4 THE ALGORITHMIC DETAILS OF TIMEST

In line with previous work [30, 65], we store the graph G as a sorted list of incoming and outgoing edges by timestamp for efficient binary search operations. For ease of algorithm explanation, we adjust the timestamps of the edges in G to begin at time 0. We summarize the important symbols in Table 1.

We start with some important definitions for temporal graphs.

Definition 4.1 (Temporal outlists and degrees). *For a vertex v and timestamp t and t', the* temporal outlist $\Lambda_v^+[t, t']$ *is the set of outedges (v, w, t''), where w is any vertex and $t'' \in [t, t']$. The* temporal inlist $\Lambda_v^-[t, t']$ *is defined similarly.*

With $\alpha \in \{+, -\}, \beta \in \{<, >\}$ and fixed δ, we use the notation $\Lambda_v^\alpha[t, (\beta, \delta)]$ to represent $\Lambda_v^\alpha[t - \delta, t]$ when β is $<$, and $\Lambda_v^\alpha[t, t + \delta]$ when β is $>$.

The temporal out-degree $d_v^+[t, t'] = \lvert\Lambda_v^+[t, t']\rvert$. *And the* temporal in-degree $d_v^-[t, t'] = \lvert\Lambda_v^-[t, t']\rvert$.

Definition 4.2 (Multi-edge list). *We define multi-edge list $El_{u,v}[t, t']$ as the collection of edges $e = (u, v, t'')$ with $t'' \in [t, t']$.*

To simplify the notation with $\alpha \in \{+, -\}, \beta \in \{<, >\}$ and δ, we use $El_{u,v}^\alpha[t, (\beta, \delta)]]$ to present different cases. Here, α represents the edge direction: + means $u \to v$ direction, and − means $v \to u$ direction. β specifies the time range: the timestamp must fall in $[t - \delta, t]$ (<) or fall in $[t, t + \delta]$ (>).

Definition 4.3 (Multiplicity). *Given a pair u, v of distinct vertices and timestamps $t < t'$, the* multiplicity $\sigma_{u,v}[t, t']$ *is the number of edges (u, v, t'') which satisfy $t'' \in [t, t']$. $\sigma_{u,v}[t, t'] = |El_{u,v}[t, t']|$. We denote the maximum δ-multiplicity as σ_δ, defined as $\max_{u,v,t} \sigma_{u,v}[t, t + \delta]$.*

Let S be a spanning tree of the motif we wish to count. We first define the notion of a leaf edge. An edge $s \in S$ is called a leaf edge if any one of its endpoints has 0 indegree and outdegree.

We introduce the process of assigning weights to edges S. It will be convenient to root the tree at an edge (rather than a vertex), and we define parent/children using this rooting. Note that this rooting is independent of the actual directions of the edges. The height of a leaf edge is set to zero. The height of an edge is the length of the longest path down the tree to a leaf. (Equivalently, the height of an edge is one plus the maximum height of its children.)

Definition 4.4. *The dependency list of an edge $s \in E(S)$ consists of triples $\langle s', \alpha, \beta \rangle$, where $s' \neq s$ is a child of s, the parameter α (+ or -) determines the direction (incoming or outgoing) of s' with respect to the vertex at which they meet and β (<, >) gives the relative time-order between s' and s. We denote this list by $D(s)$.*

Consider the tree S in Figure 4, where the edges s_i is the edge labeled i. Then, $D(s_1) = \{\langle s_0, +, < \rangle\}$.

Definition 4.5. *A spanning tree of a temporal motif M is represented as a tuple $S = (L, O, D)$, where (i) L is the list of edges in $E(H)$ that form the directed spanning tree of M; (ii) O is an ordering of the edges in L; (iii) D is the dependency of the edges in L.*

The order O can be any topological sorting based on the dependency D. Now we introduce the notion of a *partial match* to a tree in G, which will be central throughout the paper.

Definition 4.6. *A subgraph H of G is a δ-partial match to tree S if there exists a pair $\phi = (\phi_V, \phi_E)$ of mappings such that $\phi_V : V(S) \to V(G)$ and $\phi_E : E(S) \to E(G)$ with the following properties:*

1. *$\phi_E(u_i, u_j) = (\phi_V(u_i), \phi_V(u_j))$*
2. *$(u_i, u_j) \in E(S)$ if and only if $(\phi_V(u_i), \phi_V(u_j)) \in E(G)$, with $\phi_V(u) \neq \phi_V(v)$*
3. *Suppose $e_1 = (u, v)$ and $e_2 = (x, y)$ are dependent edges in S. Assume that the height of $e_1 >$ height of e_2. Then, if $t(u, v) < t(x, y)$, then $t(\phi_E(x, y)) \in [t(\phi_E(u, v)), t(\phi_E(u, v)) + \delta]$. Otherwise, $t(\phi_E(x, y)) \in [t(\phi_E(u, v)) - \delta, t(\phi_E(u, v))]$.*
4. *Further $|\phi_E(u, v) \cap \phi_E(x, y)| = 1$*

In particular, a partial match ϕ is a homomorphism (property (2)) of S into G, which respects the relative time order between pairs of adjacent edges along every path from the root of S to any of its leaves (property (3), corresponds to Constraint 1). It also ensures that matches to adjacent edges intersect only in one vertex (property (4), corresponds to Constraint 2). Given a partial match ϕ, an edge $e = (u, v, t) \in E(G)$, and an edge $s = (x, y) \in E(S)$, we say that ϕ matches s to e if $\phi_E(x, y) = (u, v)$. We will often refer to e as the *match* of s in G under ϕ.

Overview: Figure 4 illustrates the overall process of estimating temporal motif counts for a given motif M in an input temporal graph G. The algorithm begins by selecting the best spanning tree of motif M to optimize sampling efficiency (Section 4.5), then determines relaxed temporal constraints (Section 2.1) and precomputes edge sampling weights (Section 4.1). In the sampling phase, it repeatedly samples spanning trees from G (Section 4.2), validates them against relaxed constraints, and derives local counts using dynamic programming (Section 4.3). The final estimate is obtained by scaling the aggregated local counts from K samples (Section 4.4). We explain the spanning tree selection at the end, since it requires many concepts from the other components.

4.1 Preprocess Sampling Weights

During the preprocessing phase, for each edge of the input graph, we compute a set of $|E(S)|$ non-negative weights, one for every one of the $|E(S)|$ edges in the spanning tree S. We also refer to the root edge of S as the center edge.

Furthermore, for an edge $s \in S$, v_s denotes the lower-level end-point of the edge s and T_s to denote the subtree rooted at v_s.

Definition 4.7 (s-weight of edge e $w_{s,e}$). *Given an edge $e = (u, v, t) \in E(G)$ and an edge $s \in E(S)$, the s-weight of e is defined as the number of partial matches (in G) ϕ to the subtree T_s which are rooted at $\phi_V(v_s)$, where v_s the end-point of s with lower height. We will denote the s-weight of an edge e by $w_{s,e}$.*

Consider S in Figure 4. Suppose $e = (v_2, v_3, 40) \in E(G)$ is a match to $s_1 = (B, C)$. Then $\phi_v(B) = v_2, \phi_V(C) = v_3$. In this case, w_{e,s_1} is the number of partial matches in G to the subtree T_B that are rooted at v_2. Since the subtree T_B consists of a single edge (A, B) that happens before (B, C), w_{e,s_1} assume a value equal to the number the inedges of v_2 with timestamps in the interval $[40 - \delta, 40]$.

More generally, suppose $e = (u, v, t) \in E(G)$ is a match to the edge $s = (x, y)$ under ϕ_E. Consider an arbitrary triple $\langle s_1, \alpha_1, \beta_1 \rangle$ in the list $D(s)$. Assume w.l.o.g. $\phi_V(v_s) = u$. We claim that any match to s_1 in G under ϕ_E will either be an in-edge or out-edge of u (depending on α_1), that does not intersect v, and whose timestamp is either in $[t - \delta, t]$ or $[t, t + \delta]$, depending on β_1. We denote this list by L_{e,s,s_1}, which is formally defined below.

Claim 4.8. *The partial match ϕ_E will map s_1 to an edge in $\Lambda_u^{\alpha_1}[t, (\beta_1, \delta)] \setminus El_{uv}[t, (\beta_1, \delta)]$ if $\phi_V(v_s) = u$.*
The partial match ϕ_E will map s_1 to an edge in $\Lambda_v^{\alpha_1}[t, (\beta_1, \delta)] \setminus El_{vu}[t, (\beta_1, \delta)]$ if $\phi_V(v_s) = v$.

Thus, when looking for the potential matches to s_1, it suffices to restrict to the edge list given in Claim 4.8. Let us denote this list $L_{e,s,s_1} = \Lambda_u^{\alpha_1}[t, (\beta_1, \delta)] \setminus El_{uv}[t, (\beta_1, \delta)]$ (when $\phi_V(v_s) = u$) or $\Lambda_v^{\alpha_1}[t, (\beta_1, \delta)] \setminus El_{vu}[t, (\beta_1, \delta)]$ (when $\phi_V(v_s) = v$) for simplicity.

Suppose the list $D(s)$ contains k triples $\langle s_1, \alpha_1, \beta_1 \rangle, \cdots, \langle s_k, \alpha_k, \beta_k \rangle$. Then any partial match ϕ to the subtree T_{v_s} can be obtained by first picking a match to each of the k edges $s_1, s_2, \ldots, s_k$, and then extend them recursively into partial matches to the subtrees $T_{s_1}, \cdots T_{s_k}$.

Claim 4.9. *Suppose the list $D(s)$ contains triples $< s_i, \alpha_i, \beta_i >$ for $1 \leq i \leq k$. Then*

$$w_{s,e} = \Big(\sum_{e \in L_{e,s,s_1}} w_{s_1,e} \Big) \Big(\sum_{e \in L_{e,s,s_1}} w_{s_2,e} \Big) \cdots \Big(\sum_{e \in L_{e,s,s_k}} w_{s_k,e} \Big)$$

Proof. Let Match($e = (u, v, t), s$) denote the collection of partial matches to the subtree T_s, which are rooted at $\phi_V(v_s)$ in G. Here $\phi_v(v_s) \in \{u, v\}$. Then Match(e, s) $= \{(\phi_1, \ldots, \phi_k)\}$, where ϕ_i is a partial match to the subtree T_{s_i} in G. Every such match is rooted at a suitable end-point of some edge $e_i \in L_{e,s,s_i}$. This endpoint

Algorithm 1 : PREPROCESS(S)

Input: Spanning tree $S = (L, O, D)$
Output: Total sampling weight W; W_i for each subgraph G_i; $w_{i,s,e}$ for e in G_i that are mapped to s.

1: num_{subg} = time span of G / δ
2: **for** $i \in [0, num_{\text{subg}})$ **do** ▷ partition to subgraphs
3: build subgraph G_i whose timestamp in $[i\delta, (i+2)\delta]$
4: $W_i, w_{i,s,e}$ = PREPROCESSSUBGRAPH(S, G_i)
5: **return** $W, W_i, w_{i,s,e}$

Algorithm 2 : PREPROCESSSUBGRAPH(S, G_i)

Input: Spanning tree $S = (L, O, D)$, and current subgraph G_i
Output: W for the current subgraph G_i; $w_{s,e}$ for edge $e \in G_i$ mapped to edge $s \in L$.

1: **for** $h \in [1, H]$ **do** ▷ skip leaf edges in S
2: **for** $e = (u, v, t) \in E(G_i)$ **do**
3: **for** $s \in O[h]$ **do**
4: calculate $w_{s,e}$ according to Claim 4.9
5: **if** s is the "center" edge in S **then**
6: W += $w_{s,e}$
7: **return** $W, w_{s,e}$

is given by $\phi_{Vi}(v_{s_i})$. Note that the s weight of e is precisely the size of the set Match(e, s) i.e. $w_{s,e} = |\text{Match}(e, s)|$. Recall that the number of partial matches to the subtree T_{s_i}, which can be obtained from an edge e in the list L_{e,s,s_i}, is exactly equal to e's s_i weight. Hence the number of partial matches ϕ_i with the given property is

$$\sum_{e \in L_{e,s,s_i}} w_{s_i,e}.$$

As a result, we can conclude that the number of tuples in the set Match(e, s) is given by

$$\Big(\sum_{e \in L_{e,s,s_1}} w_{s_1,e}\Big)\Big(\sum_{e \in L_{e,s,s_1}} w_{s_2,e}\Big) \cdots \Big(\sum_{e \in L_{e,s,s_k}} w_{s_k,e}\Big).$$ □

Claim 4.9 allows us to use dynamic programming to efficiently compute the $w_{s,e}$ values, where the choices for s vary in the increasing order of their height in S.

Finally, let W denote the total number of partial-matches to S in G and c be the center edge of S. Then,

CLAIM 4.10. *$W = \sum_{e \in E(G)} w_{c,e}$, where c is the center edge in S*

PROOF. Since any edge in G could potentially be the center edge in a match to S, it follows that the total number of matches to S in G is $\sum_{e \in E(G)} w_{c,e}$. This completes the proofs. □

We detail the PREPROCESS procedure in Algorithm 1. First, we segment the entire timespan of the input graph G into overlapping intervals of length 2δ. Let $I = \{[0, 2\delta], [\delta, 3\delta], \ldots, [(q-2)\delta, q\delta]\}$ denote the resulting collection of intervals. Here $q = T/\delta$ where T is the time span of G. Each interval I_i corresponds to a subgraph G_i, consisting of the edges in G which fall within the interval I_i. Next, we invoke the subroutine PREPROCESSSUBGRAPH to get the overall sampling weight W_i for each subgraph G_i, as well as the s-weights for the edges in each subgraph. Here s varies across the edges of S.

Algorithm 3 : SAMPLESUBG$(S, G_i, W, w_{s,e})$

Input: Spanning tree $S = (L, O, D)$, sampling weights W for current subgraph G_i, and $w_{s,e}$ for edges in $E(G_i)$
Output: Sampled edges P where $P[i]$ is the edge in $E(G_i)$ that is mapped to s_i in S, partial match vertex map ϕ_V and edge map ϕ_E.

1: c is the center edge in S
2: Sample center edge e with sampling probability $p_e = w_{c,e}/W_i$
3: $P[c] = e$, Update ϕ_V and ϕ_E
4: **for** $h \in [H, 0]$ **do**
5: **for** $s \in O[H]$ **do**
6: **for** $\langle s', \alpha', \beta' \rangle \in D(s)$ **do**
7: Use binary search to find the $L_{e,s,s'}$
8: $W_x = \sum_{e \in L_{e,s,s'}} w_{s',e}$
9: Compute $p_{s',e} = w_{s',e}/W_x$ for $e \in L_{e,s,s'}$
10: Sample $e_{s'}$ using sampling weight $p_{s',e}$
11: $P[s'] = e_{s'}$, update ϕ_V and ϕ_E
12: **return** P, ϕ_V and ϕ_E

In the PREPROCESSSUBGRAPH procedure (in Algorithm 2), we compute the s-weight for every edge $e \in E(G)$ and every $s \in E(S)$. These weights are stored in a 2D array and are computed in the order given by O. More concretely, for every $e \in E(G)$, first we compute its s-weight for every $s \in E(S)$ which is at *height 1* in S. (Leaf edges with 0 height are skipped because the weights are constant 1s.) Next, for each edge, we compute its associated s-weight for every $s \in E(S)$ which is at *height 2*. We use the relation given by Claim 4.9 to do so. We repeat these steps until we reach the center edge in S. The total sampling weight for the current subgraph is then calculated by Claim 4.10.

4.2 Sample

The goal of the SAMPLE procedure is to sample a partial match to spanning tree S from graph G. First, we sample an interval from I with probability proportional to its overall weight W_i. Next, we construct the graph G_i by including those edges of G whose timestamp belongs to the sampled interval I_i.

Then we invoke the SAMPLESUBG function on G_i (Algorithm 3). In this subroutine, first we sample a match $e = (u, v, t) \in E(G_i)$ to the center edge c of S. This is done by sampling an edge $e \in E(G)$ with probability proportional to $w_{c,e}$ (line 2). Next, for every $s \in D(c)$, we use binary search to get the list of potential matches to s in G. For each $s \in D(c)$, we sample its match in G by picking an edge from the list of candidate matches to s with probability proportional their associated s-weight. Having done this for every $s \in D(c)$, we recursively build a match to S, now by sampling matches to the edges in $D(s)$ for every $s \in D(c)$. Note that the order in which we sample a match to the spanning tree is exactly the opposite of how we compute the edge weights.

Let $\mathcal{G} := \{G_1, G_2, \ldots, G_{|I|}\}$ be the family of subgraphs of G obtained from the intervals in I. For a fixed $G_k \in \mathcal{G}$ containing ϕ, our algorithm outputs a uniform random partial match to S.

LEMMA 4.11. *Fix a partial match ϕ to S in G. If this partial match is present in N_ϕ-many subgraphs of $\mathcal{G}$, then the subroutine SAMPLESUBG will output ϕ with probability N_ϕ/W.*

Algorithm 4 : ValidateAndDeriveCnt(M, P, ϕ_E, ϕ_V)

Input: Temporal motif M, sampled spanning tree P, partial vertex map ϕ_V and edge map ϕ_E
Output: Number of motif instances *cnt* that extend from $\phi_E(S)$

1: Verify if ϕ_V is a 1-1 map. If not, **return** 0
2: Verify if edges in $\phi_E(S)$ are in δ time range. If not, **return** 0
3: Verify if edges in $\phi_E(S)$ have the correct edge order. If not, **return** 0
4: **return** DeriveCnt(M, P, ϕ_E, ϕ_V)$/N_\phi$

Algorithm 5 : DeriveCnt(M, P, ϕ_E, ϕ_V))

Input: Temporal motif M, sampled spanning tree P, partial vertex map ϕ_V and edge map ϕ_E
Output: Number of motif instances *cnt* that extend from $\phi_E(S)$

1: For every edge e of M that's not in the sampled spanning tree P, use binary search to find the lists (in G) of potential matches
2: Number these lists as $L_1, L_2, \cdots L_l$ in the time ordering.
3: Use the ListCount algorithms (Alg. 4 of [39]) to count all possible combinations without enumeration.
4: **return** cnts

Note that $N_\phi \leq 2$ for every partial match ϕ to S. For a fixed $G_k \in \mathcal{G}$ containing ϕ, our algorithm outputs a uniform random partial match to S.

4.3 Validate and Derive the Motif Counts

After sampling a partial match ϕ to S, we verify if it respects all ordering and temporal constraints. If the sampled match fails to meet any required conditions, we set its count to 0. Otherwise, we obtain the number of instances of the target motif induced by it. This can be done by a combination of the merge-technique used in the merge-sort algorithm and dynamic programming DeriveCnt (Algorithm 5), which uses the ListCount algorithm (Alg. 4 from Pan *et al.* [39]). Given a collection of time ordered edge lists $L_1, L_2, \cdots L_l$, it counts the number of combinations $(e_1, e_2, \ldots, e_l)$ where for all i, $e_i \in L_i$, and $t(e_i) < t(e_{i+1})$. This is done with pointer traversals and dynamic programming. Note that by modifying the DeriveCnt function to list the sampled motif instance to reconstruct individual motif instances, our algorithm can be adapted to support random motif listings. Finally, we rescale this count to ensure that our estimate is unbiased at the end. The full procedure is detailed in Algorithm 4.

4.4 Overall Estimate Procedure

Integrating the procedures described previously, we outline the Estimate process in Algorithm 6. First, analyze the motif and use heuristics to choose a spanning tree S of the motif. Given S, for every $s \in E(S)$ and $e \in E(G)$, we compute the weight $w_{s,e}$, as given by the Preprocess subroutine. Next, we invoke the SampleSubg procedure to sample a partial match to S and feed it as input to the ValidateAndDeriveCnt subroutine, which in turn returns the number of matches to the target motif induced by the partial match to S. We run the procedure multiple times, average the individual counts, rescale the average and return the final result.

Algorithm 6 Estimate(M, G, k)

Input: Temporal motif M, input graph G, and sample number k
Output: Motif count estimate $\widehat{C}$

1: Use heuristic to choose a spanning tree S
2: $W, W_i, w_{i,s,e}$ = Preprocess(S)
3: num_{subg} = time span of G / δ
4: init sample counts $samp_i = 0$ for every subgraph G_i, $cnt = 0$
5: **for** $j \in [1, k]$ **do** ▷ sample subgraphs
6:     Sample G_i with probability $p_{i,\delta} = W_i/W$
7:     $samp_i$ += 1
8: **for** $i \in [1, num_{\text{subg}}]$ **do**
9:     **for** $j \in [1, samp_i]$ **do** ▷ sample edge in subgraphs
10:         P, ϕ_E, ϕ_V = SampleSubg ($S, G_i, W_i, w_{i,s,e}$)
11:         cnt += ValidateAndDeriveCnt(M, P, ϕ_E, ϕ_V)
12: **return** $\widehat{C} = (cnt/k) \cdot W$

Next, we show that the estimate produced by our algorithm is unbiased and bounded.

Lemma 4.12. *Let $\widehat{C}$ denote the estimate produced by TIMEST and C be the true motif count. Then* $\mathrm{E}[\widehat{C}] = C$.

Proof. Let $\mathcal{S}$ denote the set of partial matches to the spanning tree S in G. For every $1 \leq i \leq k$, let ϕ_i denote the partial match to S output by the subroutine SampleSubg when invoked for the i-th time. For every $\phi \in \mathcal{S}$, let M_ϕ be the number of instances of the target motif H that extend from ϕ, N_ϕ be the number of subgraphs in $\mathcal{G}$ which contain ϕ and C be the total number of occurrences of H in G. Further, let the random variable Y_i denote the outcome of DerivecNT when invoked on ϕ_i and let $Y = \sum_{i \leq k} Y_i$.
$\mathrm{E}[Y] = \mathrm{E}[\sum_{i \leq k} Y_i] = \sum_{i \leq k} \mathrm{E}[Y_i]$.
Note that $\mathrm{E}[Y_i] = \sum_{\phi \in \mathcal{S}} \mathrm{E}[Y_i | \phi_i = \phi] \Pr(\phi_i = \phi)$.
From Lemma 4.11, $\Pr(\phi_i = \phi) = N_\phi/W$. Further, $\mathrm{E}[Y_i | \phi_i = \phi] = M_\phi/N_\phi$. This is because once we obtain a partial match ϕ, we determine the number of subgraphs in $\mathcal{G}$ that contain ϕ i.e. N_ϕ. Subsequently, we divide the value returned by the DerivecNT subroutine when invoked on ϕ with N_ϕ.
Therefore, $\mathrm{E}[Y_i] = \sum_{\phi \in \mathcal{S}} (M_\phi/N_\phi)(N_\phi/W) = \sum_{\phi \in \mathcal{S}} (M_\phi/W) = C/W$. Note that $\sum_{\phi \in \mathcal{S}} M_\phi = C$ because every instance H_G of H in G has a unique spanning tree T_{H_G} such that the subroutine DeriveCnt will account for H_G only when it is invoked with T_{H_G} as its input.
Therefore, $\mathrm{E}[Y] = kM/W$. So, $\widehat{C} = WY/k$, is an unbiased estimate of C. □

We now prove the following result. Let B denote the maximum number of matches to the target motif M resulting from matches to S in G.

Theorem 4.13. *Let $Y_1, Y_2, \ldots, Y_k$ be k random variables, where Y_i is the outcome of the subroutine* ValidateAndDeriveCnt *on the i-th sampled partial match to S. Suppose $k = (3B/\varepsilon^2)(W/C)\ln(2/\gamma)$ for any $\varepsilon, \gamma \in (0,1)$. Then with probability at least $1-\gamma$, $\widehat{C} \in [(1-\varepsilon)C, (1+\varepsilon)C]$.*

Proof. Recall that the random variables Y_i denote the outcome of the DerivecNT when invoked on partial match ϕ_i and $Y =$

$\sum_{i \le k} Y_i$. First, note that the k random variables $Y_1, \ldots, Y_k$ are independent of each other. Also, observe that the random variables $Y_1, \ldots, Y_k$ may not necessarily assume values in the interval $[0, 1]$. Hence we cannot directly use the Chernoff Bound. However, dividing them by B ensures that $0 \le Y_i/B \le 1$ for all $1 \le i \le k$.

Before applying the Chernoff Bound, observe that the event $|Y/B - kM/BW| \ge (\epsilon kM/WB)$ is identical to $|WY/k - C| \ge \epsilon C$. Similarly, since $\widehat{C} = WY/K$, the events $|Y/B - kM/WB| \ge (\epsilon kM/WB)$ and $|\widehat{C} - C| \ge \epsilon C$ are also identical.

Note that $E[Y/B] = \sum_{i \le k} E[Y_i]/B = Mk/WB$. Using Chernoff Bound, for any $\epsilon \in (0, 1)$, $\Pr(|Y/B - (kM/WB)| \ge (\epsilon kM/WB)) \le 2\exp(-(\epsilon^2/3)(kM/WB))$. Since $k = (3B/\epsilon^2)(W/C)\ln(2/\gamma)$, it follows that $\Pr(|Y/B - (kM/BW)| \ge (\epsilon kM/BW)) \le$
$2\exp(-(\epsilon^2/3)(3B/\epsilon^2)(W/C)(\ln(2/\gamma))(C/WB)) = \gamma$. □

We present the time and space complexity as follows.

Claim 4.14. *The* Preprocess *procedure takes $O(|E(S)|^2 m d_{\max})$ time. The* SampleSubg *procedure takes $O(|E(S)|^2 d_{\max})$ time per sample. The storage complexity is $O(m|V(S)|)$. Here, m is the number of edges in G, where $d_{\max}$ is the maximum number of simple edges incident on any vertex in G.*

Proof. PreprocessSubgraph computes the s-weights ($w_{s,e}$) for every edge $e \in E(G)$ and every $s \in E(S)$. To obtain $w_{s,e}$, we perform binary searches on the edges incident to one endpoint of e (say u) to locate candidate matches for edges in S, forming the set $D(s)$, in $O(\log m)$ time. For every $s' \in D(s)$, we retrieve the precomputed s'-weights of the matches incident on u (in $O(1)$ time each), and multiply them to get the s-weight of e. Thus, computing a single $w_{s,e}$ takes $O(|E(s)|d_{\max} + \log m)$ time. Repeating this for all $e \in E(G)$ and $s \in E(S)$ yields a total complexity of $O(|E(S)|^2 m d_{\max})$.

During the sampling phase, once we have sampled an edge e that matches s, we use binary search to determine potential matches to every $s' \in D(s)$, fetch their corresponding s'-weights and subsequently sample matches to every s' from their corresponding list of potential matches using distributions defined using s'-weight values. So the first step takes $O(|E(S)|\log m)$ time, the second step takes $O(|E(S)|d_{\max})$ and final step takes $O(|E(S)|\log m)$ time. We do these steps for every s in $E(S)$, so the overall time to sample an entire partial match is $O(|E(S)|^2 d_{\max})$ per sample.

Regarding the space complexity, we store the sampling weight $w_{s,e}$ in a vector for each edge in the graph G, whose size equal to the total edge count (m) multiplying the number of edges in the spanning tree of the motif ($|V(S)|$). □

4.5 Choosing the Spanning Tree

Figure 5b shows that a motif can have many spanning trees. Our goal is to select the spanning tree that minimizes runtime while having the best estimation quality. The estimation error when choosing 54 different spanning trees of the M_{6-4} motif is shown in Figure 6. The results reveal large variation in accuracy: the worst-case estimation error exceeds 350% while the best error is less than 1%.

In Theorem 4.13, given the motif M and input graph G, the number of samples k to take to reach convergence is proportional to the total sampling weight W. This weight W is exactly the number of instances of chosen spanning tree S in the graph. A smaller W indicates a rarer spanning tree, reducing the search space. For example, consider the candidate spanning trees S_1 and S_2 of motif M_{5-3} in Figure 5b, or those of motif M_{6-4} in Figure 6. S_1 follows a path-like structure and S_2 has a more tree-like structure. The experiments in Table 7 show that the sampling weight W for S_2 is around 5-10

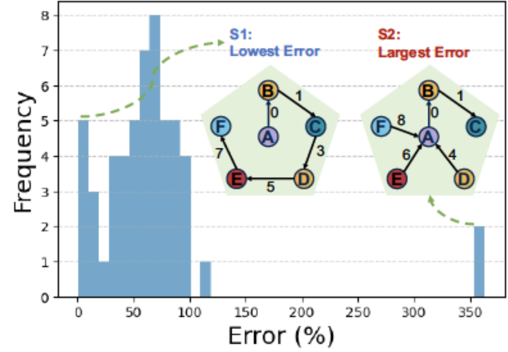


Figure 6: Histogram of estimation errors (%) across different spanning tree choices for motif M_{6-4} and with 1e8 samples. Two highlighted examples, S1 and S2, correspond to the lowest and highest observed errors, emphasizing that the choice of spanning tree significantly impacts estimation accuracy.

Algorithm 7 : getBestSPTree(M, G, n_c)

Input: Temporal motif M, input graph G, number of candidate spanning trees n_c
Output: Selected spanning tree S_{best}

1: DFS to enumerate all spanning trees of M in an array $S_{all}[]$
2: $C_losseness$ = getConstraintLooseness(S_{all}, M)
3: S_{topk} = selectTopK($C_losseness, S_{all}, n_c$)
4: Init an array $est_runtime[n_c]$
5: **for** $i \in [0, n_c)$ **do**
6: $\quad curW$ = PreprocessSubgraph($S_{topk}[i], G$)
7: $\quad est_runtime[i] = curW$
8: S_{best} = tree with min $est_runtime$
9: **return** S_{best}

Algorithm 8 : getConstraintLooseness(S_{all}, M)

Input: all spanning trees $S_{all}[]$
Output: Constraint looseness $C_lossness[]$

1: Initialize $C_lossness[]$ with 0
2: **for** $i \in [0, len(S_{all}))$ **do**
3: `/*For every node, compute the absdiff of the edges incident to the node*/`
4: $\quad$ **for** $u \in nodes(M)$ **do**
5: $\quad\quad E(u) \leftarrow$ edges if M that are incident on u
6: $\quad\quad$ **if** $|E(u)| < 2$ **then**
7: $\quad\quad\quad$ continue
8: $\quad\quad$ `/*Consider all pairs of incident edges*/`
9: $\quad\quad$ **for** each unordered pair (e_1, e_2) in $E(u)$ **do**
10: $\quad\quad\quad C_lossness[i] += |t(e_1) - t(e_2) - 1|$
11: **return** $C_lossness$

Table 2: Temporal graph datasets used in the evaluation.

Dataset	$\|V\|$	$\|E_{temporal}\|$	Time span (year)
wiki-talk (WT)	1.1M	7.8M	6.4
stackoverflow (SO)	2.6M	63.5M	7.6
bitcoin (BI)	48.1M	113.1M	7.1
reddit-reply (RE)	8.4M	636.3M	10.1

times larger than that for S_1. This indicates that S_1 appears less frequently in the input graph and introduces stricter constraints compared to S_2, so we prefer S_1 to S_2.

However, calculating W for *all* spanning trees is time-consuming. Each calculation takes approximately 10% of the overall runtime. We propose a heuristic that approximates how tightly a spanning tree enforces temporal constraints, which serves as a rough proxy for W without accessing the graph. This proxy, denoted as *C_looseness*, is computed in Algorithm 8. It examines all edge pairs incident to each node and computes how tightly the spanning tree orders them. A lower value indicates stricter constraints and is thus preferred.

Overall, our heuristics have the following steps.

- Finding Candidates: We use DFS to find all spanning trees and evaluate them based on motif-only information. Then we select the top n_c candidates with the smallest constraint losseness (GETCONSTRAINTLOOSENESS).
- Evaluating Candidates: For the selected candidate spanning trees, we calculate the actual sampling weight using the input graph (PREPROCESSSUBGRAPH) . A smaller W means fewer samples are required for convergence (indicated by Theorem 4.13).
- Choosing the Best Tree: We use the sampling weight W (which affects the number of samples) to estimate the total runtime. We select the spanning tree with the smallest estimated runtime, making it applicable across different graphs and motifs.

5 EVALUATION

In this section, we comprehensively evaluate the performance of TIMEST across different graphs, motifs, and δ values. We assess both the accuracy and the runtime of TIMEST and compare them with existing methods, highlighting its accuracy and efficiency.

5.1 Experiment Setup

Benchmarks. We evaluate the temporal motif counts across a spectrum of datasets, encompassing medium to large-scale graphs such as wiki-talk (WT), stackoverflow (SO) [27], bitcoin (BI) [22], and reddit-reply (RE) [16], detailed in Table 2. To showcase the versatility of TIMEST, we mine temporal motifs encompassing 5-vertex and 6-vertex connected graphs in Figure 3. We set δ as $8W$, $16W$ for WT and SO, and as $1D$ for BI and RE, where H represents hour, D represents day, and W represents week.

Baselines. For exact algorithms, we use the state-of-the-art Everest [65], implemented in [2], which builds on the BT algorithm [30] with GPU-specific optimizations. We exclude the original BT CPU version due to its significantly lower performance (over 20× slower than Everest).

For approximate algorithms on 4-vertex motifs, we compare against PRESTO [47], ES [59] and IS [29]. PRESTO is a sampling framework that uniformly samples intervals of length $c\delta$ and performs exact temporal motif counting on these sampled intervals to derive estimated counts. We explore two versions of PRESTO, referred to as *PRESTO-A* and *PRESTO-E*. For 5- for 6-vertex motifs, we omit the comparison with IS [29] as PRESTO was already shown to outperform it in their study. And we do not provide a quantitative comparison with ES [59] because the public code [1] is limited to motifs with 4 or fewer edges.

TIMEST The number of samples k affects the accuracy and runtime of TIMEST which varies from 10 million to 1 trillion. For the detailed settings per case, please refer to the `reproduce.py` in https://github.com/pyjhzwh/SpanningTreeSampling/. For a fair comparison, the runtime of TIMEST include all steps in Figure 4, which consists of preprocess and sample procedure.

Accuracy Metrics The accuracy of approximate algorithms is defined as $\frac{|C-\hat{C}|}{C}$, where C and $\hat{C}$ are the exact count and the estimated count, respectively. And error = 1- accuracy. We run it 5 times to get the average(avg) and standard deviation(std) of error.

Discussion Of The Samples TIMEST uses more samples than prior work, but the runtime per sample is extremely small (i.e., 0.1 microsecond per sample). PRESTO samples time windows and then runs an expensive algorithm on this sample. Each sample in TIMEST is a spanning tree that processes few edges. So, even including preprocessing time, TIMEST is faster.

Hardware Platforms. Both TIMEST and PRESTO run on a CPU platform with 32 threads. Specifically, the CPU platform utilized is an AMD EPYC 7742 64-core CPU with 64MB L2 cache, 256MB L3 cache, and 1.5TB DRAM memory. Everest [2] runs on a single NVIDIA A40 GPU with 10k CUDA cores and 48GB memory.

5.2 Results

TIMEST is fast. In Table 3, we list the runtime of Everest (GPU) and TIMEST (CPU) on counting the 5-vertex and 6-vertex temporal motifs in Figure 3. We focus on the cases where Everest runs more than 60 seconds. For RE dataset with δ=1 day, Everest is faster than TIMEST. With large number of nodes and edges in input graph and motifs, Everest takes a large amount of runtime, and even timeout (>1 day of execution time) in some cases. On the other hand, TIMEST completes execution in less than 30 minutes in all cases and exhibits a geomean speedup of 28× over Everest.

Further, we also compare TIMEST and Everest (GPU) on the bipartite money laundering pattern of Figure 1 [6]. We use the WT graph with δ set to 4W. Everest takes two days to complete, while TIMEST runs in *four minutes.* The error observed is less than 0.6%.

TIMEST is accurate. Our comparison between TIMEST and PRESTO [47] focuses on runtime performance and relative error, with findings summarized in Table 4. Despite PRESTO's runtime being 6× slower compared to TIMEST, our results show that TIMEST consistently outperforms PRESTO in accuracy for all evaluated temporal motifs involving 5 and 6 vertices.

PRESTO employs an approximation strategy that involves uniform sampling across graph partitions within a time window of size $c\delta$ ($c > 1$ is a parameter), and applying an exact algorithm on these partitions. PRESTO's inefficiency is because of the reliance on uniform sampling and the use of the slow backtracking (BT) algorithm [30] as the subroutine on the sampled time window.

Table 3: Runtime (in second) of Everest (GPU) and TIMEST (CPU), the speedup of TIMEST over Everest, and the estimation error (%) of TIMEST. We set the timeout limit as 1 day. We mark the speedup and error as *No Exact* for the case that Everest timeout. The error(%) is represented as avg ± std.

Graph	δ	Motif	Everest (GPU)	TIMEST (CPU)	Speedup (×)	Error(%)
WT	8W	$M_{5\text{-}1}$	9.9E1	9.3E0	10.6	0.1±0.0
		$M_{5\text{-}2}$	1.7E3	8.8E0	193.3	1.3±0.7
		$M_{5\text{-}3}$	1.5E3	7.1E1	21.1	0.7±0.5
		$M_{5\text{-}4}$	2.5E3	7.1E1	34.9	6.4±4.9
		$M_{5\text{-}5}$	3.4E3	4.3E2	8.0	9.4±5.3
		$M_{6\text{-}1}$	1.8E4	1.0E2	176.1	0.0±0.0
		$M_{6\text{-}2}$	1.2E4	1.4E1	852.9	0.1±0.1
		$M_{6\text{-}3}$	4.1E3	3.3E2	11.8	0.7±0.6
		$M_{6\text{-}4}$	5.5E4	1.3E3	42.5	7.4±3.8
		$M_{6\text{-}5}$	6.8E4	1.3E3	51.8	72.3±16.7
SO	8W	$M_{5\text{-}1}$	9.0E2	3.0E1	30.5	0.0±0.0
		$M_{5\text{-}2}$	7.7E3	2.9E1	265.0	0.9±0.3
		$M_{5\text{-}3}$	6.6E2	8.0E1	8.2	0.8±0.4
		$M_{5\text{-}4}$	4.0E2	8.1E1	5.0	20.4±12.6
		$M_{5\text{-}5}$	6.7E3	6.4E2	10.5	32.7±16.7
		$M_{6\text{-}1}$	3.1E4	4.2E1	734.8	0.1±0.1
		$M_{6\text{-}2}$	9.4E4	3.2E1	2980.1	0.2±0.0
		$M_{6\text{-}3}$	2.1E3	1.5E2	13.6	0.3±0.2
		$M_{6\text{-}4}$	2.9E3	1.0E3	2.8	33.1±18.5
		$M_{6\text{-}5}$	1.5E5	1.0E3	140.1	100.0±0
BI	1D	$M_{5\text{-}1}$	3.7E2	5.4E1	6.8	0.0±0.0
		$M_{5\text{-}2}$	2.1E3	4.9E1	43.1	2.2±0.7
		$M_{5\text{-}3}$	1.9E3	1.1E2	16.7	1.4±0.6
		$M_{5\text{-}4}$	7.7E3	1.1E2	68.3	3.2±3.1
		$M_{5\text{-}5}$	1.0E4	8.9E2	11.3	10.0±5.4
		$M_{6\text{-}1}$	5.1E4	6.2E1	812.7	0.1±0.0
		$M_{6\text{-}2}$	timeout	4.7E1	No Exact	No Exact
		$M_{6\text{-}3}$	3.0E3	1.4E2	20.9	0.2±0.2
		$M_{6\text{-}4}$	timeout	8.3E2	No Exact	No Exact
		$M_{6\text{-}5}$	timeout	8.2E2	No Exact	No Exact
RE	1D	$M_{5\text{-}1}$	6.3E1	3.9E2	0.2	0±0
		$M_{5\text{-}2}$	3.4E2	3.9E2	0.9	0.6±0.4
		$M_{5\text{-}3}$	5.5E2	4.1E2	1.4	0.4±0.5
		$M_{5\text{-}4}$	9.6E3	4.2E2	23.1	2.0±1.3
		$M_{5\text{-}5}$	9.9E3	1.3E3	7.8	19.9±10.6
		$M_{6\text{-}1}$	5.1E2	5.1E2	1.0	0.1±0.0
		$M_{6\text{-}2}$	3.1E3	3.9E2	7.9	0.0±0.0
		$M_{6\text{-}3}$	1.0E3	8.5E2	2.2	0.1±0.0
		$M_{6\text{-}4}$	timeout	8.5E2	No Exact	No Exact
		$M_{6\text{-}5}$	timeout	8.7E2	No Exact	No Exact

Both PRESTO and TIMEST struggle to accurately estimate counts for the most complex 6-clique motif ($M_{6\text{-}5}$), showing low convergence and accuracy. Despite generating approximately 1 billion samples, with around 100 million being valid (i.e., not violating the constraints defined in Definition 1.2), only 1 - 100 of these valid samples contribute to the final count. The 6-clique encompasses more constraints (such as edge mapping and ordering) than any chosen spanning tree can cover. Developing accurate algorithms for such complex motifs would require new ideas.

For smaller motifs with 4-vertex in Figure 3, we compare TIMEST with PRESTO, ES and IS baselines regarding runtime and relative error. The result is shown in Table 5. TIMEST is always faster by an order of magnitude with comparable or lower error.

Improved performance with additional constraints. Introducing specific constraints on sampled edges helps reduce the chances of them violating the requirements of Definition 1.2. In Table 6, we show how the three constraints in Section 2 reduce invalid samples and improve performance. We categorize three distinct invalid sample types because of violating: a) 1-1 vertex map ϕ_V b) δ time interval c) edge orders.

Our experimental approach employs Constraint 1 as the foundational condition, with Constraints 2 and 3 subsequently integrated

Table 4: Runtime (in second) and relative error (%) of approximate algorithms on various datasets and temporal motifs. The lowest error is highlighted in green block. All run in 32-thread. We set the timeout limit as (1 day). If the program runs out of memory (OOM), it will be killed. TIMEST is the fastest and the most accurate in most cases.

Dataset	Motif	PRESTO-A		PRESTO-E		TIMEST	
		Time (s)	Error	Time (s)	Error	Time (s)	Error
WT $\delta = 8W$	$M_{5\text{-}1}$	8.0E3	3.0%	4.0E4	7.1%	1.0E1	**0.1%**
	$M_{5\text{-}2}$	1.5E4	28.1%	1.6E4	2.2%	8.8E0	**1.3%**
	$M_{5\text{-}3}$	1.5E4	11.5%	1.9E4	12.0%	6.7E1	**0.7%**
	$M_{5\text{-}4}$	1.1E4	24.1%	5.1E4	33.6%	6.4E1	**6.4%**
	$M_{5\text{-}5}$	2.4E4	25.8%	2.4E4	77.1%	4.8E2	**9.4%**
	$M_{6\text{-}1}$	1.1E4	93.8%	2.0E3	98.8%	6.4E1	**0.0%**
	$M_{6\text{-}2}$	2.5E3	96.1%	timeout	N/A	1.1E1	**0.1%**
	$M_{6\text{-}3}$	1.4E5	107.5%	2.6E5	10.4%	2.0E2	**0.7%**
	$M_{6\text{-}4}$	1.3E4	89.5%	timeout	N/A	6.9E2	**7.4%**
	$M_{6\text{-}5}$	timeout	N/A	timeout	N/A	6.9E2	72.3%
SO $\delta = 8W$	$M_{5\text{-}1}$	4.9E3	3.8%	2.3E4	17.2%	2.6E1	**0.0%**
	$M_{5\text{-}2}$	OOM	N/A	OOM	N/A	2.5E1	**0.9%**
	$M_{5\text{-}3}$	8.0E2	33.4%	1.0E3	17.2%	6.6E1	**0.8%**
	$M_{5\text{-}4}$	4.0E2	87.3%	2.8E2	35.7%	6.9E1	**20.4%**
	$M_{5\text{-}5}$	1.5E4	81.5%	OOM	N/A	4.8E2	**32.7%**
	$M_{6\text{-}1}$	8.6E4	98.4%	timeout	N/A	3.2E1	**0.1%**
	$M_{6\text{-}2}$	OOM	N/A	OOM	N/A	2.9E1	**0.2%**
	$M_{6\text{-}3}$	1.4E4	45.0%	4.0E4	13.88%	1.2E2	**0.3%**
	$M_{6\text{-}4}$	1.3E3	81.4%	3.9E3	270.8%	8.7E2	**33.1%**
	$M_{6\text{-}5}$	OOM	N/A	OOM	N/A	8.8E2	100%
BI $\delta = 1D$	$M_{5\text{-}1}$	4.1E2	12.9%	2.7E3	13.2%	3.9E1	**0.0%**
	$M_{5\text{-}2}$	8.2E2	47.3%	1.3E3	41.7%	3.6E1	**2.2%**
	$M_{5\text{-}3}$	1.2E2	65.9%	3.8E2	39.4%	4.1E1	**1.4%**
	$M_{5\text{-}4}$	7.7E2	41.1%	5.4E2	53.3%	9.6E1	**3.2%**
	$M_{5\text{-}5}$	1.4E3	15.3%	2.8E3	11.5%	7.8E2	**10.0%**
	$M_{6\text{-}1}$	5.5E4	52.4%	timeout	N/A	5.2E1	**0.1%**
	$M_{6\text{-}2}$	timeout	N/A	timeout	N/A	5.2E1	NE
	$M_{6\text{-}3}$	8.5E2	26.5%	5.9E3	32.2%	1.2E2	**0.2%**
	$M_{6\text{-}4}$	NE	NE	NE	NE	7.9E2	NE
	$M_{6\text{-}5}$	NE	NE	NE	NE	8.1E2	NE
RE $\delta = 1D$	$M_{5\text{-}1}$	3.2E2	102.5%	1.6E3	14.4%	3.2E2	**0.0%**
	$M_{5\text{-}2}$	3.7E2	92.8%	2.4E3	45.8%	2.9E2	**0.6%**
	$M_{5\text{-}3}$	1.8E4	925.0%	1.9E3	95.5%	4.2E2	**0.4%**
	$M_{5\text{-}4}$	6.8E4	90.6%	5.6E4	97.6%	3.3E2	**2.0%**
	$M_{5\text{-}5}$	1.1E4	98.6%	2.5E4	25.8%	1.2E3	**19.9%**
	$M_{6\text{-}1}$	1.1E4	98.6%	4.5E4	99.9%	1.2E3	**0.1%**
	$M_{6\text{-}2}$	3.6E4	40.4%	1.3E5	6.0%	3.5E2	**0.0%**
	$M_{6\text{-}3}$	3.4E3	76.5%	1.1E3	timeout	4.5E2	**0.1%**
	$M_{6\text{-}4}$	NE	NE	NE	NE	7.9E2	NE
	$M_{6\text{-}5}$	NE	NE	NE	NE	8.1E2	NE

in a phased manner. These experiments were conducted on the $M_{5\text{-}5}$ motif (5-cliques), utilizing the WT graph with $\delta = 8W$, and the RE graph with $\delta = 1D$. The objective is to achieve high accuracy with high efficiency, striving for an increase in the rate of valid samples while minimizing runtime for an equivalent number of samples.

From the table, we can see that adding Constraint 2 (distinct end vertices on dependent edges) reduces the number of invalid samples due to violating 1-1 vertex map. For RE graph, it significantly reduces the percentage of invalid samples due to violating the vertex map from 75% to 2%. Adding Constraint 3 (2δ window) reduces the invalid samples due to violating the δ constraints, increasing the valid sample rate by around 5%. Integrating all three constraints, we observe that the runtime remains largely unchanged, suggesting that the additional computational overhead introduced by these new constraints is minimal. More crucially, this integration boosts the valid sample rate from 1% to 33% for the RE graph, concurrently decreasing the error rate dramatically from 95% to 4%. *This demonstrates not only the feasibility of applying these constraints without significant efficiency losses but also their effectiveness in enhancing the precision of the sampling process.*

Table 5: Runtime (s) and relative error (%) of all algorithms on each dataset. The lowest error is highlighted in green block. Because not all sampling algorithms support multi-threading, for a fair comparison, we report the runtime for single-threaded implementations. Because TIMEST output count for $M_{4\text{-}1}$-$M_{4\text{-}4}$ at once, we consider the runtime of TIMEST as the total runtime for $M_{4\text{-}1}$-$M_{4\text{-}4}$. TIMEST is always the fastest.

Dataset	Motif	PRESTO-A		PRESTO-E		IS		ES		TIMEST		
		Time (s)	Error	Time (s)	Error	Time (s)	Error	Time (s)	Error	Time (s)	Speedup over PRESTO-A (×)	Error
WT, δ=8W	M4-1	3.65E+03	31.08%	3.62E+03	8.65%	1.32E+04	1.62%	2.22E+02	3.64%	5.16E+01	292	**0.26%**
	M4-2	3.76E+03	4.69%	4.04E+03	7.92%	1.44E+04	0.80%	2.71E+02	2.42%			**0.63%**
	M4-3	3.65E+03	75.88%	3.96E+03	34.45%	1.32E+04	**0.57%**	2.35E+02	8.34%			2.35%
	M4-4	3.99E+03	77.56%	3.74E+03	27.54%	1.45E+04	12.63%	5.01E+02	**2.27%**			4.30%
ST, δ=8W	M4-1	3.69E+03	13.16%	4.10E+03	19.17%	7.99E+04	0.68%	1.87E+03	0.75%	3.70E+02	41	**0.24%**
	M4-2	3.88E+03	86.36%	3.91E+03	12.84%	> 1D	-	1.50E+03	0.50%			**0.37%**
	M4-3	3.62E+03	38.68%	4.01E+03	20.29%	8.59E+04	1.45%	2.24E+02	**0.95%**			1.30%
	M4-4	3.85E+03	**0.52%**	4.07E+03	10.05%	8.12E+04	0.56%	3.12E+02	1.76%			4.74%
BI, δ=1D	M4-1	3.62E+03	5.33%	3.60E+03	19.10%	1.80E+03	1.81%	8.66E+03	2.12%	4.60E+02	32	**0.07%**
	M4-2	3.65E+03	18.50%	3.61E+03	8.94%	2.01E+03	**0.23%**	1.22E+03	1.48%			0.53%
	M4-3	3.64E+03	25.39%	3.67E+03	19.33%	1.79E+03	20.56%	5.24E+02	6.23%			**1.84%**
	M4-4	3.67E+03	41.71%	3.63E+03	31.99%	1.69E+03	16.61%	6.17E+02	**6.32%**			7.60%
RE, δ=1D	M4-2	3.60E+03	26.83%	3.61E+03	23.17%	3.67E+03	38.34%	1.85E+03	8.90%	2.42E+03	6	**0.22%**
	M4-3	3.60E+03	160.88%	3.60E+03	53.66%	5.91E+03	21.37%	1.08E+03	1.45%			**0.44%**
	M4-5	3.61E+03	89.03%	3.61E+03	46.35%	4.22E+03	33.11%	6.04E+02	2.96%			**1.92%**
	M4-7	3.60E+03	104.42%	4.54E+03	344.62%	7.29E+03	151.53%	1.01E+03	27.94%			**4.57%**

Table 6: The percentage of invalid and valid sample rate, relative error, and runtime when enforcing different constraints (C) for estimating temporal motif count of $M_{5\text{-}5}$.

Graph, δ		WI, δ=8W			RE, δ=1D		
Constraints		C1	C1+2	C1+2+3	C1	C1+2	C1+2+3
Invalid sample rate (%)	vertex map	42.9%	36.3%	36.6%	75.5%	2.0%	1.9%
	δ interval	14.6%	17.1%	10.3%	0.8%	23.1%	13.6%
	edge order	34.4%	35.8%	39.8%	23.0%	45.4%	51.1%
Valid samples rate (%)		8.1%	10.8%	13.3%	0.8%	29.5%	33.4%
Error (%)		5.1%	0.9%	2.9%	95.92%	12.5%	4.0%
Runtime (s)		320.1	332.7	361.0	1259.2	1221.7	1148.1

Table 7: Sampling weight W, estimation error (%) and runtime (s) for choosing different spanning tree (S_1 to S_3) when estimating motif count for $M_{5\text{-}3}$.

G	δ	W			Error			Runtime (s)		
		S_1	S_2	S_3	S_1	S_2	S_3	S_1	S_2	S_3
WT	4W	1.1E12	8.9E12	1.4E12	**4.9%**	9.5%	14.5%	**9.9**	12.4	11.7
	8W	6.5E12	5.2E13	8.0E12	**0.8%**	13.4%	3.4%	**14.2**	16.4	17.7
SO	4W	3.1E12	2.6E13	4.0E12	**3.1%**	33.1%	5.8%	25.4	**19.8**	27.0
	8W	1.6E13	1.5E14	2.0E13	**4.4%**	21.8%	11.4%	30.0	**20.7**	31.1
BI	1D	2.7E13	1.3E15	1.2E14	**1.2%**	110.6%	3.9%	42.9	**33.6**	48.1
RE	1D	7.3E11	4.9E12	5.3E12	**0.3%**	3.5%	6.8%	327.3	**251.5**	336.0

Effect of different spanning trees on performance. We examined using three distinct spanning trees for motif $M_{5\text{-}3}$, using 20 million samples each for our analysis, as seen in Table 7. The trees S_1 and S_2 are displayed in Figure 5b. The total sampling weight W is similar for S_1 and S_3 in certain graphs, but S_2 has a W value about 10 times higher. Theorem 4.13 states that for a given graph and motif, the number of samples k needed to reach convergence is proportional to W. Smaller W implies the same error with fewer samples, which is corroborated by Table 7.

Runtime comparison, given equal sample numbers, reveals minor differences between the trees, with S_3 having slightly longer runtimes than S_1 and S_2. Because the ValidateAndDeriveCnt() function has time complexity linear to the overall number of potential candidates (see Section 4.3). Here the candidates refer to the non-spanning-tree edges of the motif.

Table 8: Peak memory usage (GB) for PRESTO and TIMEST.

Dataset	Motif	PRESTO-A	PRESTO-E	TIMEST
WT $\delta = 8W$	$M_{5\text{-}1}$	1.5	3.7	2.9
	$M_{5\text{-}3}$	15.9	37.2	2.8
	$M_{6\text{-}3}$	2.7	6.9	3.0
SO $\delta = 8W$	$M_{5\text{-}1}$	12.4	13.1	20.1
	$M_{5\text{-}3}$	98.1	111.6	20.1
	$M_{6\text{-}3}$	26.8	30.8	21.0
BI $\delta = 1D$	$M_{5\text{-}1}$	4.2	5.1	40.3
	$M_{5\text{-}3}$	8.3	19.2	40.3
	$M_{6\text{-}3}$	5.7	11.9	42.1

Peak memory usage In Table 8, we show the maximum RAM memory used for various algorithms in a single run. We can see that TIMEST and PRESTO have comparable peak memory usage. Notice that the peak memory usage of TIMEST does not grow with the motif's complexity or the number of samples. Because most of the memory is used to store the preprocessing weights.

6 CONCLUSION AND FUTURE WORK

This paper presents TIMEST—a general algorithm for counting motifs in temporal graphs that is both fast and accurate. TIMEST works for any motif with arbitrary size and shape. The key idea behind TIMEST is to estimate the motif counts based on an underlying spanning tree structure. To improve estimation accuracy and achieve high performance, we show how to intelligently design a constrained algorithm and present the heuristics for choosing spanning trees. TIMEST is designed under the assumption of total temporal orderings following temporal motif mining problems in prior works [30]. A natural next step is to try to count motifs with partial order time constraints. In some special cases, the spanning tree of TIMEST already suffices, but fully supporting arbitrary partial orderings remains an interesting direction for future work.

7 ACKNOWLEDGEMENTS

Both OB and CS are supported by NSF grants CCF-1740850, CCF-2402572, and DMS-2023495.

REFERENCES

[1] 2021. ES Code Repository. https://github.com/jingjing-cs/Temporal-Motif-Counting.

[2] 2023. Everest Code Repository. https://github.com/yichao-yuan-99/Everest.

[3] Nesreen K Ahmed, Nick Duffield, Jennifer Neville, and Ramana Kompella. 2014. Graph sample and hold: A framework for big-graph analytics. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1446–1455.

[4] Nesreen K Ahmed, Nick Duffield, and Ryan A Rossi. 2021. Online sampling of temporal networks. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 15, 4 (2021), 1–27.

[5] Uri Alon. 2007. Network motifs: theory and experimental approaches. *Nature Reviews Genetics* 8, 6 (2007), 450–461.

[6] Erik Altman, Jovan Blanuša, Luc Von Niederhäusern, Béni Egressy, Andreea Anghel, and Kubilay Atasu. 2024. Realistic synthetic financial transactions for anti-money laundering models. *Advances in Neural Information Processing Systems* 36 (2024).

[7] Hanjo D Boekhout, Walter A Kosters, and Frank W Takes. 2019. Efficiently counting complex multilayer temporal motifs in large-scale networks. *Computational Social Networks* 6, 1 (2019), 8.

[8] Giorgos Bouritsas, Fabrizio Frasca, Stefanos P Zafeiriou, and Michael Bronstein. 2022. Improving graph neural network expressivity via subgraph isomorphism counting. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2022).

[9] Marco Bressan, Flavio Chierichetti, Ravi Kumar, Stefano Leucci, and Alessandro Panconesi. 2018. Motif counting beyond five nodes. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 12, 4 (2018), 1–25.

[10] Marco Bressan, Stefano Leucci, and Alessandro Panconesi. 2019. Motivo: fast motif counting via succinct color coding and adaptive sampling. 12, 11 (2019).

[11] Xinwei Cai, Xiangyu Ke, Kai Wang, Lu Chen, Tianming Zhang, Qing Liu, and Yunjun Gao. 2024. Efficient Temporal Butterfly Counting and Enumeration on Temporal Bipartite Graphs. *Proc. VLDB Endow.* 17, 4 (mar 2024), 657–670. https://doi.org/10.14778/3636218.3636223

[12] Xuhao Chen et al. 2022. Efficient and Scalable Graph Pattern Mining on {GPUs}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 857–877.

[13] Zhongqiang Gao, Chuanqi Cheng, Yanwei Yu, Lei Cao, Chao Huang, and Junyu Dong. 2022. Scalable motif counting for large-scale temporal graphs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2656–2668.

[14] Aris Gionis, Lutz Oettershagen, and Ilie Sarpe. 2024. Mining Temporal Networks. https://miningtemporalnetworks.github.io/.

[15] László Hajdu and Miklós Krész. 2020. Temporal network analytics for fraud detection in the banking sector. In *International Conference on Theory and Practice of Digital Libraries*. Springer, 145–157.

[16] Jack Hessel, Chenhao Tan, and Lillian Lee. 2016. Science, askscience, and badscience: On the coexistence of highly related communities. In *Proceedings of the international AAAI conference on web and social media*, Vol. 10. 171–180.

[17] Yuriy Hulovatyy, Huili Chen, and Tijana Milenković. 2015. Exploring the structure and function of temporal networks with dynamic graphlets. *Bioinformatics* 31, 12 (2015), i171–i180.

[18] Shweta Jain and C Seshadhri. 2017. A fast and provable method for estimating clique counts using turán's theorem. In *Proceedings of the 26th international conference on world wide web*. 441–449.

[19] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: a pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[20] Kasra Jamshidi and Keval Vora. 2021. A deeper dive into pattern-aware subgraph exploration with peregrine. *ACM SIGOPS Operating Systems Review* 55, 1 (2021), 1–10.

[21] Madhav Jha, C Seshadhri, and Ali Pinar. 2015. Path sampling: A fast and provable method for estimating 4-vertex subgraph counts. In *Proceedings of the 24th international conference on world wide web*. 495–505.

[22] Dániel Kondor, István Csabai, János Szüle, Márton Pósfai, and Gábor Vattay. 2014. Inferring the interplay between network structure and market effects in Bitcoin. *New Journal of Physics* 16, 12 (2014), 125003.

[23] Lauri Kovanen, Márton Karsai, Kimmo Kaski, János Kertész, and Jari Saramäki. 2011. Temporal motifs in time-dependent networks. *Journal of Statistical Mechanics: Theory and Experiment* 2011, 11 (2011), P11005.

[24] Lauri Kovanen, Kimmo Kaski, János Kertész, and Jari Saramäki. 2013. Temporal motifs reveal homophily, gender-specific patterns, and group talk in call sequences. *Proceedings of the National Academy of Sciences* 110, 45 (2013), 18070–18075.

[25] Rohit Kumar and Toon Calders. 2018. 2scent: An efficient algorithm to enumerate all simple temporal cycles. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1441–1453.

[26] Mayank Lahiri and Tanya Y Berger-Wolf. 2007. Structure prediction in temporal networks using frequent subgraphs. In *2007 IEEE Symposium on computational intelligence and data mining*. IEEE, 35–42.

[27] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[28] Penghang Liu, Rupam Acharyya, Robert E Tillman, Shunya Kimura, Naoki Masuda, and Ahmet Erdem Sarıyüce. 2023. Temporal Motifs for Financial Networks: A Study on Mercari, JPMC, and Venmo Platforms. *arXiv preprint arXiv:2301.07791* (2023).

[29] Paul Liu, Austin R Benson, and Moses Charikar. 2019. Sampling methods for counting temporal motifs. In *Proceedings of the twelfth ACM international conference on web search and data mining*. 294–302.

[30] Patrick Mackey, Katherine Porterfield, Erin Fitzhenry, Sutanay Choudhury, and George Chin. 2018. A chronological edge-driven approach to temporal subgraph isomorphism. In *2018 IEEE international conference on big data (big data)*. IEEE, 3972–3979.

[31] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. 2019. Graphzero: Breaking symmetry for efficient graph mining. *arXiv preprint arXiv:1911.12877* (2019).

[32] Daniel Mawhirter and Bo Wu. 2019. Automine: harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 509–523.

[33] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. 2002. Network motifs: simple building blocks of complex networks. *Science* 298, 5594 (2002), 824–827.

[34] Seunghwan Min, Jihoon Jang, Kunsoo Park, Dora Giammarresi, Giuseppe F Italiano, and Wook-Shin Han. 2023. Time-Constrained Continuous Subgraph Matching Using Temporal Information for Filtering and Backtracking. *arXiv preprint arXiv:2312.10486* (2023).

[35] What Is Data Mining. 2006. Data mining: Concepts and techniques. *Morgan Kaufinann* 10, 559-569 (2006), 4.

[36] Mark EJ Newman. 2003. The structure and function of complex networks. *SIAM review* 45, 2 (2003), 167–256.

[37] Lutz Oettershagen, Nils M. Kriege, Claude Jordan, and Petra Mutze. 2023. *A Temporal Graphlet Kernel For Classifying Dissemination in Evolving Networks*. 19–27. https://doi.org/10.1137/1.9781611977653.ch3

[38] Raj Kumar Pan and Jari Saramäki. 2011. Path lengths, correlations, and centrality in temporal networks. *Physical Review E* 84, 1 (2011), 016105.

[39] Yunjie Pan, Omkar Bhalerao, C Seshadhri, and Nishil Talati. 2024. Accurate and Fast Estimation of Temporal Motifs using Path Sampling. *arXiv preprint arXiv:2409.08975* (2024).

[40] Yunjie Pan, Omkar Bhalerao, C. Seshadhri, and Nishil Talati. 2024. Accurate and Fast Estimation of Temporal Motifs Using Path Sampling. In *International Conference on Data Mining (ICDM)*. 809–814.

[41] Ashwin Paranjape, Austin R Benson, and Jure Leskovec. 2017. Motifs in temporal networks. In *Proceedings of the tenth ACM international conference on web search and data mining*. 601–610.

[42] Noujan Pashanasangi and C Seshadhri. 2021. Faster and generalized temporal triangle counting, via degeneracy ordering. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 1319–1328.

[43] Aduri Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. 2013. Counting and sampling triangles from a graph stream. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1870–1881.

[44] Ali Pinar, Comandur Seshadhri, and Vaidyanathan Vishal. 2017. Escape: Efficiently counting all 5-vertex subgraphs. In *Proceedings of the 26th international conference on world wide web*. 1431–1440.

[45] Jiaxi Pu, Yanhao Wang, Yuchen Li, and Xuan Zhou. 2023. Sampling Algorithms for Butterfly Counting on Temporal Bipartite Graphs. *arXiv preprint arXiv:2310.11886* (2023).

[46] Ilie Sarpe and Fabio Vandin. 2021. OdeN: simultaneous approximation of multiple motif counts in large temporal networks. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 1568–1577.

[47] Ilie Sarpe and Fabio Vandin. 2021. Presto: Simple and scalable sampling techniques for the rigorous approximation of temporal motif counts. In *Proceedings of the 2021 SIAM International Conference on Data Mining (SDM)*. SIAM, 145–153.

[48] T. Schank and D. Wagner. 2005. Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study. In *Experimental and Efficient Algorithms*. Springer, 606–609.

[49] C Seshadhri, Ali Pinar, and Tamara G Kolda. 2014. Wedge sampling for computing clustering coefficients and triangle counts on large graphs. *Statistical Analysis and Data Mining: The ASA Data Science Journal* 7, 4 (2014), 294–307.

[50] C Seshadhri and Srikanta Tirthapura. 2019. Scalable subgraph counting: The methods behind the madness: WWW 2019 tutorial. In *Proceedings of the Web Conference (WWW)*, Vol. 2. 75.

[51] Shiva Shadrooh and Kjetil Nørvåg. 2024. SMoTeF: Smurf money laundering detection using temporal order and flow analysis. *Applied Intelligence* (2024), 1–18.

[52] Shai S Shen-Orr, Ron Milo, Shmoolik Mangan, and Uri Alon. 2002. Network motifs in the transcriptional regulation network of Escherichia coli. *Nature genetics* 31, 1 (2002), 64–68.

[53] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. 2020. Graphpi: High performance graph pattern matching through effective redundancy elimination. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[54] Xiaoli Sun, Yusong Tan, Qingbo Wu, Baozi Chen, and Changxiang Shen. 2019. Tm-miner: Tfs-based algorithm for mining temporal motifs in large temporal network. *IEEE Access* 7 (2019), 49778–49789.

[55] Toyotaro Suzumura and Hiroki Kanezashi. 2021. Anti-Money Laundering Datasets: InPlusLab Anti-Money Laundering DataDatasets.

[56] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. 2015. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 425–440.

[57] Charalampos E Tsourakakis, U Kang, Gary L Miller, and Christos Faloutsos. 2009. Doulion: counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 837–846.

[58] Ata Turk and Duru Turkoglu. 2019. Revisiting wedge sampling for triangle counting. In *The World Wide Web Conference*. 1875–1885.

[59] Jingjing Wang, Yanhao Wang, Wenjun Jiang, Yuchen Li, and Kian-Lee Tan. 2020. Efficient sampling algorithms for approximate temporal motif counting. In *Proceedings of the 29th ACM international conference on information & knowledge management*. 1505–1514.

[60] Pinghui Wang, Junzhou Zhao, Xiangliang Zhang, Zhenguo Li, Jiefeng Cheng, John CS Lui, Don Towsley, Jing Tao, and Xiaohong Guan. 2017. MOSS-5: A fast method of approximating counts of 5-node graphlets in large graphs. *IEEE Transactions on Knowledge and Data Engineering* 30, 1 (2017), 73–86.

[61] Yihua Wei and Peng Jiang. 2022. STMatch: accelerating graph pattern matching on GPU with stack-based loop optimizations. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.

[62] Yixing Yang, Yixiang Fang, Maria E Orlowska, Wenjie Zhang, and Xuemin Lin. 2021. Efficient bi-triangle counting for large bipartite networks. *Proceedings of the VLDB Endowment* 14, 6 (2021), 984–996.

[63] Xiaowei Ye, Rong-Hua Li, Qiangqiang Dai, Hongzhi Chen, and Guoren Wang. 2022. Lightning Fast and Space Efficient k-clique Counting. In *Proceedings of the ACM Web Conference 2022*. 1191–1202.

[64] Xiaowei Ye, Rong-Hua Li, Qiangqiang Dai, Hongchao Qin, and Guoren Wang. 2023. Efficient Biclique Counting in Large Bipartite Graphs. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.

[65] Yichao Yuan, Haojie Ye, Sanketh Vedula, Wynn Kaza, and Nishil Talati. 2024. Everest: GPU-Accelerated System For Mining Temporal Motifs. In *50th International Conference on Very Large Databases (VLDB 2024)*. ACM.

ConANN: Conformal Approximate Nearest Neighbor Search

Sonia Horchidan
KTH Royal Institute of Technology
Stockholm, Sweden
sfhor@kth.se

Fabian Zeiher
KTH Royal Institute of Technology
Stockholm, Sweden
zeiher@kth.se

Henrik Boström
KTH Royal Institute of Technology
Stockholm, Sweden
bostromh@kth.se

Paris Carbone
KTH Royal Institute of Technology
Stockholm, Sweden
parisc@kth.se

ABSTRACT

Approximate Nearest Neighbor (ANN) search is widely used in applications such as recommendation systems, search engines, and natural language processing. Indexing techniques like the Inverted File (IVF) offer efficiency at the cost of accuracy, yet lack formal mechanisms to quantify or control approximation error. Existing approaches that attempt to provide such guarantees typically rely on restrictive assumptions about underlying data distributions, which limits their generalizability. We introduce ConANN, the first framework to provide formal, distribution-free error guarantees for IVF-based ANN search by leveraging recent advances in Conformal Risk Control. Empirical evaluation across five standard benchmarks demonstrates that ConANN: (1) tightly controls approximation error, achieving a worst-case False Negative Rate deviation within 0.03 percentage points of the target; (2) provides formal guarantees without requiring expansion of the search space, and in some cases even reduces the number of probed clusters; (3) dynamically adapts the cluster probes required per query; and (4) incurs negligible overheads when compared to existing state-of-the-art baselines. ConANN is integrated into the FAISS vector search library, facilitating adoption in real-world ANN systems.

PVLDB Reference Format:
Sonia Horchidan, Fabian Zeiher, Henrik Boström, and Paris Carbone. ConANN: Conformal Approximate Nearest Neighbor Search. PVLDB, 19(1): 29 - 42, 2025.
doi:10.14778/3772181.3772184

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/soniahorchidan/conann.

1 INTRODUCTION

The rise of unstructured data, such as text, images, and audio, has transformed how we represent and process information. Today, much of this data is encoded into dense vector embeddings, enabling powerful semantic similarity tasks across domains such as search engines, recommendation systems, and natural language processing [12, 27]. The advent of large pre-trained large language models has further amplified the need to scale vector search within massively large, high-dimensional embedding spaces and meet the demands of modern applications [12, 41]. To address this need, *Approximate Nearest Neighbor* (ANN) methods have become a key component of data-intensive applications, striking a critical balance between computational efficiency and retrieval accuracy, and enabling scalable processing of billion-scale datasets [25, 26].

Modern ANN Methods, most notably, the *Inverted File* (IVF) *index* [6, 24, 44], are widely adopted in production vector-search systems. IVF partitions the space into clusters, allowing the system to narrow the search to a few centroid-neighboring partitions. This design reduces search latency by avoiding computation across the entire dataset. However, this efficiency comes at the cost of approximation error, as true nearest neighbors may lie outside the probed clusters. Despite its practical impact, this trade-off remains fundamentally heuristic: the number of clusters probed is tuned

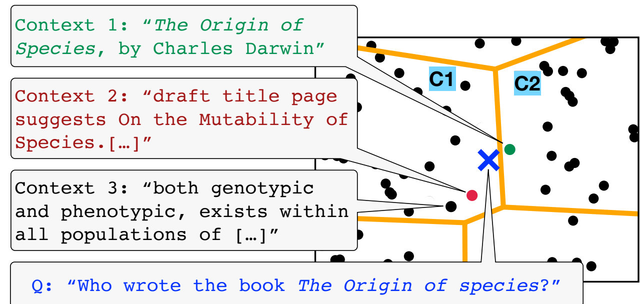


Figure 1: Example of approximate IVF search in a RAG-based LLMs using vector search for encoded passages. While an exact kNN search ($k = 1$) successfully retrieves Context 1 as relevant, the IVF-based approximate index mistakenly retrieves Context 2 if configured to probe only cluster ($C1$).

Table 1: Accuracy of RAG-based natural query answering using kNN and AkNN ($k = 1$) to retrieve the most relevant context based on its embedding proximity to the query.

Setup	Precision	Recall	F1	Search (ms)
LLM	0.051	0.209	0.082	0
kNN RAG	**0.118**	**0.558**	**0.195**	5.58
AkNN RAG	0.062	0.279	0.102	**0.146**

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 1 ISSN 2150-8097.
doi:10.14778/3772181.3772184

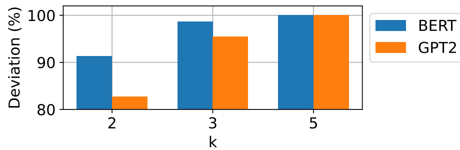


Figure 2: Local uniformity empirical experiment for the BERT and GPT-2 pre-trained embeddings.

manually, and there are no formal guarantees that the returned neighbors meet any accuracy thresholds.

RAG and the Case for Reliable ANN. The consequences of this trade-off are particularly apparent in Retrieval-Augmented Generation (RAG) pipelines, where approximate retrieval is used to condition large language models (LLMs) on relevant external knowledge [30]. In RAG systems, the semantic quality of the retrieved context directly influences the model's output. Accuracy losses in the retrieval accuracy can cascade into factually incorrect or irrelevant query answers. Figure 1 illustrates a scenario in which an IVF index, configured to search a single cluster, retrieves Context 2 as the most relevant context to a query Q encoded in Cluster C1, which the model then uses to answer the question. However, Context 1 is the more accurate response.

To assess the real-world impact, we implemented a simple RAG-based question answering system using the pre-trained LLaMA model[1]. As our query workload, we selected a subset of around 12k questions from the Natural Questions dataset [28], which consists of real user-issued search queries and associated short answers. We encoded both questions and candidate contexts using a dense embedding model and compared the accuracy of approximate IVF-based search against exact kNN retrieval. We configured the IVF index with 100 total clusters. Table 1 shows that, while AkNN achieved a 38× speedup, it exhibited a considerable drop in accuracy, measured here in Precision, Recall, and F1 score, showing that efficiency gains come at the cost of unpredictable accuracy loss. In high-stakes domains like healthcare or finance, even small drops in retrieval reliability can undermine user trust, legal compliance, or safety. This motivates a stronger requirement: *ANN systems must offer statistical accuracy guarantees that are enforceable at runtime.*

Limitations in Prior Work. While the interest in reliable ANN search is growing, existing methods fall into two categories: *best-effort* and *bounded-error* approaches. Best-effort methods, including those that can be facilitated by widely-used vector search libraries such as AnalyticDB-V [48] or Faiss [13, 25], as well as recent techniques that terminate early to minimize query latency [21, 31], rely heavily on sampling heuristics and empirically tuned parameters. These approaches optimize for performance but lack formal guarantees over standard metrics like recall or false negative rate, making their accuracy unpredictable. Further, the lack of guarantees can cause overly pessimistic results. In contrast, bounded-error methods aim to provide formal accuracy guarantees but do so under restrictive and often unrealistic assumptions. For instance, Auncel [50] assumes local uniformity in the distribution of data points to derive theoretical error bounds. These assumptions are rarely satisfied, for example, in real-world embeddings generated by deep language models like BERT or GPT, which are optimized for semantic similarity tasks and often exhibit varying density and non-linear structures across different regions [37]. To empirically assess the validity of the local uniformity assumption, we conducted a study on embeddings from BERT and GPT-2, analyzing local uniformity by comparing min-max normalized k nearest-neighbor distances to uniform distributions using a Kolmogorov-Smirnov (KS) test. Using a significance level of 0.05, we report the percentage of vectors for which the null hypothesis of uniformity was rejected. Figure 2 shows that for the majority of vectors, the local uniformity assumption is rejected, even for small values of k, and the rejection rate increases with larger neighborhoods. These findings question the applicability of methods that depend on such assumptions.

Contributions. We address this gap by introducing ConANN, a novel framework that leverages recent advances in uncertainty quantification for Machine Learning models [47] and Conformal Risk Control (CRC) [4] to provide formal, distribution-free bounded-error guarantees for IVF-based ANN search. ConANN treats ANN search as a black-box predictive task and dynamically calibrates the number of clusters probed per query to satisfy a user-specified recall tolerance. Unlike prior methods, ConANN makes no assumptions about the underlying vector data distribution and adapts automatically to varying query hardness and dataset structure. Furthermore, it requires no adjustments to the core IVF search algorithm. Our main contributions are the following:

- We adapt Conformal Risk Control to state-of-the-art Approximate Nearest Neighbor search methods, formulating the search problem as a black-box machine learning task.
- We introduce ConANN, the first framework offering formal recall guarantees for IVF-based ANN search, without requiring manual tuning or distribution assumptions.
- We evaluate ConANN on five standard ANN benchmarks, demonstrating tight recall control with a maximum deviation of 0.03 FNR from target error rates.
- We show that ConANN achieves these guarantees without increasing the search space relative to baselines, reducing cluster probes by up to 62.8%, with runtime overhead below 1.97 ms and calibration time under 4 minutes.
- We integrate ConANN into Faiss, showcasing its generality and ease of adoption in modern vector search systems.

2 PRELIMINARIES

This section reviews Approximate Nearest Neighbor (ANN) search, then introduces Conformal Prediction (CP) and Conformal Risk Control (CRC) as tools for principled error quantification. Table 2 summarizes the main notations used.

2.1 Approximate Nearest Neighbor Search

Exhaustive nearest-neighbor search is prohibitive on today's billion-vector datasets, so production systems accept approximate answers to cut latency. A standard choice for approximate search is the Inverted File Index (IVF) [6, 25]. During index construction, a vector quantizer (e.g., K-means) partitions the vector space into P cells, with the resulting centroids forming a coarse representation of the space. The search procedure in IVF balances precision and speed using a user-defined parameter, p, where $p \leq P$, which determines the number of clusters to probe. For a given query vector $\mathbf{x}$, the algorithm begins by examining the cluster with the closest centroid

[1]https://huggingface.co/ahxt/LiteLlama-460M-1T

Table 2: Summary of Notations

Notation	Description
k	Number of closest neighbors to be retrieved
$\mathbf{x}$	Query vector
P	Total number of clusters
p	Number of clusters searched
C_p	The p-th closest centroid to $\mathbf{x}$ (Voronoi cell)
S_p	ANN Search result after probing $C_1, ..., C_p$
S_{GT}	Exact kNN result set
$\mathcal{D}$	Dataset of vectors
N	$\|\mathcal{D}\|$
d	Vector dimensionality
M	Calibration dataset size
α	Error bound
α_p	Error after probing $C_1, ..., C_p$
$\pi(\mathbf{x}, y)$	The non-conformity score of input $\mathbf{x}$ with respect to label y
$\mathbf{q}^{(i)}$	The i-th element of the vector $\mathbf{q}$

and retrieves an intermediate list of k nearest neighbors. It then continues probing additional clusters, one at a time, until p clusters are searched or the desired accuracy is achieved.

We denote the AkNN search as S_p, aiming to approximate the ground truth S_{GT}. The quality of approximation is evaluated using recall, which quantifies the fraction of retrieved results present in S_{GT}, or equivalently, the False Negative Rate (FNR). As p approaches P, the search space is expanded and the FNR approaches 0.

$$FNR(S_p, S_{GT}) = 1 - \frac{|S_p \cap S_{GT}|}{k} \quad (1)$$

2.2 Conformal Prediction Methods

We build upon the Conformal Risk Control (CRC) framework [4], an extension of CP that supports arbitrary loss functions while preserving formal coverage guarantees. CP provides a model-agnostic approach for constructing prediction sets under a user-specified error rate. This section provides a brief introduction of both methods.

2.2.1 Conformal Prediction (CP). CP is a statistical framework that provides distribution-free, model-agnostic uncertainty quantification for predictive models [3]. The framework is applicable to both classification and regression problems, and allows for controlling the error rate, by turning point predictions into set predictions, i.e., sets of class labels for classification problems or prediction intervals for regression problems. The framework guarantees that an output set prediction will contain the true target (class label or regression value) with a user-defined probability (confidence level) $(1 - \alpha)$, where α, also known as *significance level*, represents the error bound. Originally, the framework was developed for an online, *transductive* setting, requiring retraining and calibration for each new observation. A recent computationally cheaper framework variant [47] known as *inductive* or *split* conformal prediction divides batches of the training set into proper training and calibration sets. The former is used to fit a model (using any algorithm), and the latter is used to compute so-called *non-conformity scores*, which encapsulate how unlikely a label y is for input x; when forming a prediction set $C(x_{\mathrm{M+1}})$ for a test object, labels with a higher non-conformity score than the $(1 - \alpha)$ percentile (denoted $\hat{s}$) are rejected. The prediction sets are, therefore, defined as follows:

$$C(x_{\mathrm{M+1}}) = \{y : \pi(x_{\mathrm{M+1}}, y) \leq \hat{s}\}\,. \quad (2)$$

For a given calibration set $\{(x_i, y_i), \ldots, (x_M, y_M)\}$, and a test example (x_{M+1}, y_{M+1}), the framework guarantees that:

$$\mathbb{P}(y_{\mathrm{M+1}} \in C(x_{\mathrm{M+1}})) \geq 1 - \alpha, \quad (3)$$

This guarantee holds under the assumption of *exchangeability*, i.e., any permutation of $\{(x_i, y_i), \ldots, (x_{M+1}, y_{M+1})\}$ is equally probable. It should be noted that the framework makes no assumptions about the underlying algorithm and how the non-conformity scores are computed. In other words, the coverage rate in Eq. (3) is guaranteed no matter how we choose to implement the framework. However, for the informativeness (defined as how narrow the prediction sets are) of the prediction sets, the choice of learning algorithm and definition of non-conformity may have a large impact. These lenient requirements make CP broadly applicable across various Machine Learning models, with use-cases including classification, regression, anomaly detection, and active learning [43].

The key to CP's statistical guarantees lies in adjusting the prediction set size to fit the desired confidence level α. Higher accuracy requirements (lower α) result in larger prediction sets. The set size is tailored to the specific test instance, based on its non-conformity. Queries resembling the calibration data typically yield smaller, more informative sets, while outliers or less familiar inputs produce larger sets, reflecting greater model uncertainty. As the error probability is mathematically guaranteed, the efficiency (informativeness) of the predictions is critical for CP; a set containing all possible labels ensures full coverage but lacks practical utility.

2.2.2 Conformal Risk Control. Traditional CP methods offer coverage guarantees, as defined in Eq. (3). Conformal Risk Control [4] extends CP by generalizing the loss function to a broader class of non-increasing loss metrics (i.e., monotonic functions where the loss decreases as the prediction set size increases). This flexibility enables CRC to address application-specific performance criteria.

The goal of CRC is to construct prediction sets $C_\lambda(X_i)$ with user-defined guarantees on a specified loss function f. Given a calibration dataset $\{(x_i, y_i), \ldots, (x_M, y_M)\}$, a test example (x_{M+1}, y_{M+1}), and a prediction loss function $f(x, y)$, CRC ensures the following expectation bound:

$$\mathbb{E}\left[f(C_{\hat{\lambda}}(x_{M+1}), y_{M+1})\right] \leq \alpha, \quad (4)$$

where $\alpha \in (-\infty, B]$ is a user-specified risk upper bound. The new parameter $\hat{\lambda}$ is optimized during calibration and controls the conservativeness of the prediction set, balancing between tighter predictions and stricter adherence to the loss constraint. To obtain $\hat{\lambda}$, CRC evaluates the empirical risk over the calibration set for each candidate λ and identifies the smallest λ such that the expected calibration loss remains below α. Formally, this involves solving:

$$\hat{\lambda} = \inf \left\{ \lambda \in [0, 1] : \frac{M}{M+1} \sum_{i=1}^{M} f(C_\lambda(x_i), y_i) + \frac{B}{M+1} \leq \alpha \right\} \quad (5)$$

The correction term $\frac{B}{M+1}$ accounts for sampling variability in the empirical risk estimate and ensures that the guarantee in Eq. (4) holds under finite-sample conditions. This adjustment prioritizes the risk constraint, potentially at the cost of larger prediction sets. Note that the correction term depends solely on the calibration set size M and does not scale with the total dataset size N. In practice, $\hat{\lambda}$ can be efficiently computed using binary search.

3 PROBLEM STATEMENT

We formulate the problem of providing statistical error guarantees for the IVF search. We define the optimization task and examine applying conformal methods to IVF, highlighting their benefits, prerequisites, and alignment with vector search requirements.

3.1 Objective

We reinterpret IVF search as a black-box approximation to the exact kNN retrieval, akin to how machine learning models approximate complex target functions. This analogy motivates the use of CP to quantify the approximation error.

We formalize the problem of providing statistical error guarantees for the AkNN error in IVF. Let $\mathbf{x}$ be the query vector, and S_p the AkNN result for x with parameter k, after probing clusters $C_1, C_2, ..., C_p$. Our goal is to find S_p corresponding to $\mathbf{x}$ such that the expected FNR, as defined in Eq. (1), is bounded by a user-defined threshold α. At the same time, we aim to minimize p, the number of clusters probed, to reduce computation time.

Importantly, achieving an FNR much lower than α is suboptimal in this setting. Since FNR can always be reduced by increasing p, overshooting the target leads to unnecessary computation without improving statistical guarantees. Thus, in addition to satisfying the constraint $\mathbb{E}[\text{FNR}(S_p, S_{GT})] \leq \alpha$, we aim to match α as closely as possible from below. In summary, we formulate this as the following constrained optimization problem:

$$\begin{aligned} \text{minimize} \quad & p \\ \text{subject to} \quad & \mathbb{E}\left[\text{FNR}(S_p, S_{GT})\right] \leq \alpha \\ & \left|S_p\right| = k; \quad p \leq P \end{aligned} \tag{6}$$

This formulation naturally gives rise to three key desiderata. First, the solution must guarantee *validity*: the expected False Negative Rate (FNR) across queries must remain below the user-specified threshold α, ensuring statistical reliability. Second, the solution should promote *efficiency* by minimizing p, the number of clusters probed during search. Third, the solution should exhibit *adaptivity*, adjusting p dynamically based on the difficulty of each query rather than relying on a fixed probing depth. We stress one final consideration: since the computational cost in IVF is primarily determined by cluster probes, reducing p leads directly to faster query processing. Therefore, the optimal solution must achieve tight calibration, minimizing unnecessary computation by keeping the actual FNR close to α without overshooting.

3.2 Applicability of Conformal Methods to IVF

First, we discuss the applicability of conformal methods to IVF. The design of IVF presents a unique opportunity for the application of conformal prediction techniques. While common statistical or learning-based methods for uncertainty quantification (UQ), such as ensembles [29] offer no formal guarantees or require complex assumptions about the underlying data distribution, conformal methods are distribution-agnostic, meaning they do not impose any assumptions about the distribution of the data. This makes them particularly well-suited to applications like IVF search, where the true distribution of the data is not known and is difficult to model due to high dimensionality and large data volumes.

The alignment of conformal methods with IVF systems goes beyond statistical benefits. Conformal methods require a calibration phase, but this step can be carried out offline, concurrently with the IVF index construction, and does not require constant updates during runtime. IVF indexes are typically built offline and retrained periodically, which aligns perfectly with the need for a fixed distribution in conformal prediction. As a result, once the index is built, the calibration step can be inherently embedded.

Regarding runtime overhead, conformal methods only require the computation of non-conformity scores, which can be directly derived from intermediary results already produced by the IVF search, followed by a simple threshold comparison to decide when to stop the probing. Conformal methods can treat the IVF search algorithms as black-box processes, making minimal modifications and introducing negligible performance penalties. In contrast, alternative UQ methods, such as Bayesian inference [17] or ensembles [29], typically require multiple forward passes or model replications, incurring substantial computational overheads that are often prohibitive in latency-sensitive ANN systems like IVF.

However, conformal methods do introduce specific requirements. Notably, conformal prediction assumes the data to be exchangeable. The calibration dataset must mirror the distribution of query vectors, which can be addressed at runtime by sampling from the query distribution.

Lastly, we highlight why CRC aligns particularly well with IVF. While standard inductive conformal prediction could, in principle, provide recall guarantees using class-conditional (Mondrian) conformal classifiers [47], doing so would require constructing a separate calibration set per class (i.e., per IVF cluster). This is infeasible in practice due to (1) scale, as IVF systems may contain thousands of clusters [25], and (2) sparsity, since many clusters might have too few samples for reliable calibration [23]. ConANN ultimately adopts Conformal Risk Control (CRC) as it is the only practical UQ framework that satisfies all three criteria essential for IVF integration: formal recall guarantees, distribution-free assumptions, and minimal runtime overhead.

4 CONANN OVERVIEW

Incorporating Conformal Prediction with CRC for AkNN Search using Inverted File Indexing (IVF) presents two principal challenges. The first challenge is to select an appropriate non-conformity score, which must encapsulate the uncertainty inherent in the model with respect to a given query $\mathbf{x}$. The second stems from the structure of IVF, which incorporates an intermediate classification step that departs from the standard conformal prediction setup. Specifically, IVF first maps the query vector $\mathbf{x}$ to a subset of centroids, which are subsequently probed to identify the k-nearest neighbors. In contrast to conventional conformal prediction methods, which focus on minimizing the size of the prediction set (i.e., the set of nearest neighbors, in our case), IVF requires minimizing the number of clusters probed while ensuring that the result set has size k and satisfies statistical error guarantees.

To address these challenges, we frame the IVF-based AkNN problem as a multi-label classification task, treating each IVF cluster (Voronoi cell) as a distinct class. This enables the application of Conformal Ranking Classification with FNR as a function of search

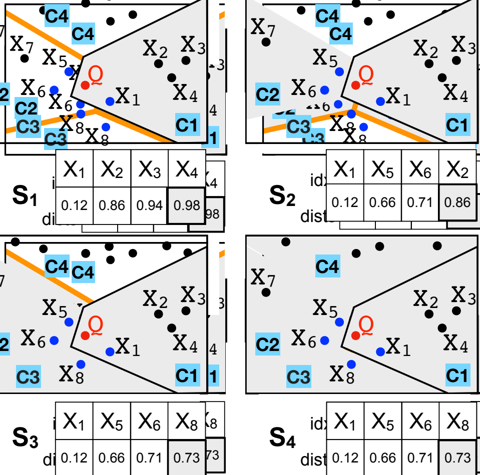


Figure 3: Example of non-conformity score computation before regularization.

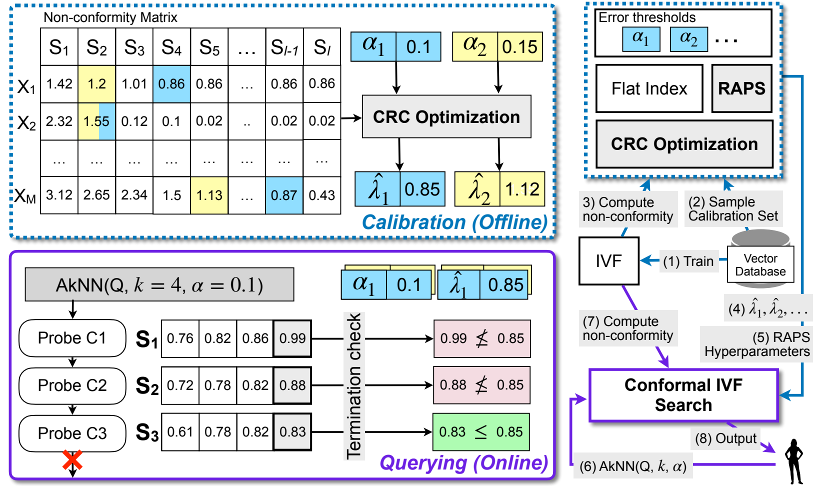


Figure 4: Overview of ConANN.

depth. In this framework, each search result S_p corresponds to a specific FNR value, which asymptotically approaches zero as the number of clusters probed p increases. We propose an effective calibration strategy and query algorithm to meet these challenges.

Figure 4 illustrates the full ConANN pipeline, containing an offline calibration stage and an online querying stage. We begin by training an IVF index on a sample dataset (Step 1). To compute the FNR required by the calibration process, we also require a flat index to obtain the ground-truth nearest neighbors for each query. We then sample the calibration query vectors (Step 2) and evaluate their approximate kNN results $S_1, \ldots, S_P$, where each S_p corresponds to probing p clusters. For each query vector, we compute the non-conformity score given the results S_p at each search depth (Step 3). CRC is applied to these scores to compute acceptance thresholds $\hat{\lambda}_j$ for different user-specified levels α_j (Step 4), as well as hyperparameters (Step 5), which will be described later in this section. At inference time (Steps 6, 7, 8), ConANN computes the non-conformity scores of the query vector , evaluates them against the optimized $\hat{\lambda}_j$, and returns the optimal set of kNN to meet the guarantee. We now detail the key components of the system.

4.1 Non-conformity Scores

We define non-conformity scores tailored to IVF-based AkNN search, where each score quantifies the quality of the result set S_p obtained after probing p clusters. Rather than computing the scores over all neighbors, we focus on a scalar summary: the distance to the k-th nearest neighbor in S_p. This definition aligns naturally with LLM retrieval scenarios. For example, as illustrated in Figure 1, the embedding corresponding to Context 1 lies closer to the query embedding, indicating higher semantic similarity and a greater likelihood of relevance. In contrast, Context 3 is embedded farther from the query, suggesting lower relevance. Further, this definition reduces computational and storage overhead, as the number of scores grows with the number of cluster probes P instead of the total database size N.

Formally, let S_p denote the set of database vectors retrieved after probing the top p clusters for a given query $\mathbf{x}$. We define the non-conformity score $\pi((\mathbf{x}, k), p)$ as the distance to the k-th nearest neighbor in S_p:

$$\pi((\mathbf{x}, k), p) = \min_{\mathbf{v} \in S_p} (d(\mathbf{x}, \mathbf{v}))^{(k)} \quad (7)$$

This score serves as a proxy for approximation quality: as p increases, S_p approaches the ground-truth set $S_{GT} = S_P$, and the k-th distance decreases accordingly. Larger scores indicate that relevant vectors remain unretrieved, implying a lower likelihood that the true k nearest neighbors are present in S_p. This behavior aligns with prior findings that query vectors located near Voronoi cell boundaries often require probing more clusters to retrieve their true neighbors [50]. To ensure robustness across vector spaces of varying scales and dimensions, we also apply min-max normalization to $\pi(q, k)$ across all calibration queries. These scores are then passed to the calibration procedure, but the same methodology is also followed at query time.

Figure 3 illustrates the computation of non-conformity scores in a 2-dimensional example using an IVF index with $P = 4$ clusters and a query vector $\mathbf{Q}$ seeking $k = 4$ neighbors. As clusters $C_1, C_2, \ldots, C_P$ are probed incrementally, we compute intermediate result sets S_p and record the distance to the k-th nearest neighbor in each set. In our example, after probing C_1 and C_2, the closest four neighbors found are X_1, X_5, X_6, and X_2, with corresponding distances of 0.12, 0.66, 0.71, and 0.86 to the query vector $\mathbf{Q}$. Our algorithm, therefore, records the distance to X_2 to build the nonconformity score of $\mathbf{Q}$ with respect to S_2. The search process continues until all P clusters are probed or an early stopping criterion is met. Once S_p contains all ground truth neighbors (in this example, S_3), the score stabilizes and remains unchanged, even if additional clusters are probed.

4.2 Conformal IVF Search

The goal of offline calibration is to determine the CRC acceptance threshold $\hat{\lambda}$ corresponding to a user-specified error tolerance α. The procedure begins by sampling a calibration set of size M and

computing non-conformity scores as described in Section 4.1. These scores are subsequently passed to the CRC optimization routine outlined in Figure 4. Prior to detailing the calibration algorithm, we highlight a key empirical observation: direct application of CRC often results in overly conservative prediction sets, a trend we revisit in Section 6. This behavior is consistent with prior work [5], which observes that conformal methods tend to be conservative when the number of classes is large (i.e., > 1000). In our setting, IVF clusters are treated as class labels; thus, the effective label space scales with dataset size. For example, with 1M vectors, P should be set to a value in the order of thousands[2]. Such settings pose challenges for classical conformal prediction techniques. To mitigate this, we adopt the Regularized Adaptive Prediction Sets (RAPS) framework [5], originally developed for multi-class classification, and adapt it to the ANN setting as detailed in Section 4.2.1. RAPS penalizes the inclusion of low-ranked predicted classes, promoting tighter, more focused prediction sets.

4.2.1 Regularization. We now describe the regularization step used to improve ConANN's efficiency. Our approach builds on the RAPS algorithm [5], originally developed for conformal classification, which adds lightweight regularization to conformity scores to produce more efficient (smaller) prediction sets without sacrificing coverage guarantees. Given a test example $(\mathbf{x}, y)$, the RAPS scores are computed by summing up the predicted probability of a class y (e.g., the softmax score returned by the model during inference) with the following regularization term:

$$\beta(\mathbf{x}, y) = \gamma \cdot (o_{\mathbf{x}}(y) - c_{reg})^{+} \quad (8)$$

The regularized term includes: (1) $o_{\mathbf{x}}(y)$, defined as the ranking of y among all the labels, given the scores π, and (2) two hyperparameters γ and c_{reg}, which are designed to promote small prediction set sizes. $(z)^{+}$ denotes the positive part of z. The hyperparameters are tuned during calibration on a small data sample using a grid search, where the conformal calibration procedure is run with different values to find the combination that minimizes prediction set size. As shown in [5], RAPS is generally robust to the choice of hyperparameters: the theoretical validity is maintained regardless of γ and c_{reg}, but good choices can substantially improve efficiency.

In our setting, we adapt RAPS regularization to the IVF-based AkNN search. The non-conformity score $\pi((\mathbf{x}, k), p)$ for a query vector $\mathbf{x}$ and candidate set S_p is defined as the (normalized) distance to the k-th nearest neighbor. To align with RAPS, which assumes that higher scores are better, we take the complement $1-\pi((\mathbf{x}, y), p)$. Our regularized non-conformity score $\hat{\pi}((\mathbf{x}, y), S_p)$ then becomes:

$$\hat{\pi}((\mathbf{x}, k), S_p) = (1 - \pi((\mathbf{x}, k), S_p)) + \gamma \cdot (p - c_{\text{reg}})^{+} \quad (9)$$

Here, p is the position (rank) of S_p among all candidate clusters or vectors. Crucially, in IVF search, the candidates are processed in order of increasing distance (i.e., better matches first), so the ranking $o_x(y)$ is known in advance and simplifies to p directly.

This regularization promotes early termination by slightly penalizing candidates that appear later in the ranking, thereby reducing computational cost without violating the formal risk guarantees of CRC. As we will show in the following sections, this leads to faster termination in the IVF search.

[2] According to the Faiss guidelines, P should be set as $O(\sqrt{N})$.

Algorithm 1 Calibration Phase (Offline)

1: **Input:** Error bound α, total number of clusters P, calibration vectors $\mathcal{X} = \{x_i\}_{i=1}^{M}$, $\{k_i\}_{i=1}^{M}$, ground truths $\mathcal{S} = \{S_{GT_i}\}_{i=1}^{M}$.
2: **Output:** Optimal acceptance threshold $\hat{\lambda}$.
3: **for** $i \leftarrow 1$ **to** M **and for** $p \leftarrow 1$ **to** P **do**
4: $\quad S_{i,p} \leftarrow$ `IVF_scan`(x_i, p)
5: $\quad$ Compute nonconformity $\pi_{i,p} \leftarrow$ `score`$(k_i, S_{i,p})$
6: Fit hyperparameters γ, c_{reg} using $\pi, \mathcal{S}$
7: **for** $i \leftarrow 1$ **to** M **and for** $p \leftarrow 1$ **to** P **do**
8: $\quad \hat{\pi}_{i,p} \leftarrow (1 - \pi_{i,p}) + \gamma \cdot \max(0, p - c_{\text{reg}})$
9: **for** each λ in a discretized grid over $[0, 1]$ **do**
10: $\quad$ **for** $i \leftarrow 1$ **to** M **do**
11: $\quad\quad$ Find $p' \leftarrow \min\{p : \hat{\pi}_{i,p} \le \lambda\}$
12: $\quad\quad$ Select $\hat{S}_{i,\lambda} \leftarrow S_{i,p'}$
13: $\quad\quad R_{\lambda,i} \leftarrow$ `FNR`$(\hat{S}_{i,\lambda}, S_{GT_i}) - \alpha \frac{M+1}{M} - \frac{1}{M}$
14: $\hat{\lambda} \leftarrow$ `infimum`$(R_{\lambda}, \lambda \in [0, 1])$
15: **return** $\hat{\lambda}$

4.2.2 Calibration. The offline calibration phase computes an acceptance threshold $\hat{\lambda}$ to provide distribution-free guarantees on the FNR during the AkNN search. Following the CRC framework, we frame the IVF search process as a structured prediction task and calibrate $\hat{\lambda}$ to satisfy a user-specified risk level α. Algorithm 1 summarizes the procedure.

Given a calibration set of vectors $\mathcal{X}$, corresponding k values, and their corresponding ground-truth nearest neighbors $\mathcal{S}$, we first simulate the IVF search behavior (Line 4), which scans only the p-th closest cluster to a given query vector. For each query $x_i \in \mathcal{X}$ and each possible number of probed clusters $p \in \{1, 2, ..., P\}$, we record the intermediate search result $S_{i,p}$ and compute a non-conformity score $\pi_{i,p}$ (Line 5), as described in Section 4.1, measuring the search quality relative to the ground truth. We employ a simplified version of the RAPS regularization to strengthen the relationship between the number of clusters probed and the search quality, as described by Eq. (9). γ and c_{reg} are hyperparameters fitted to a sample of 1000 vectors of additional unseen calibration data (Line 6). The regularization results in a matrix of scores, named non-conformity matrix in Figure 4, which is then fed to the CRC Optimization step, aiming to identify the optimal $\hat{\lambda}$ acceptance thresholds.

Next, we evaluate candidate thresholds sampled from a discretized grid over $[0, 1]$, with arbitrary precision. For each query and each candidate λ, we select the smallest number of clusters p' such that the regularized non-conformity score $\hat{\pi}_{i,p} \le \lambda$. We then compute the corresponding empirical FNR across the calibration set. To account for the finite size of the calibration data, we adjust the empirical risk using the CRC correction, with $B = 1$ to denote the upper bound of the FNR. Finally, we select the smallest λ that satisfies the risk constraint, as defined in Eq. (5). Figure 4 shows the selection of λ values for two error bounds $\alpha_1 = 0.1$ and $\alpha_2 = 0.15$.

This calibration step is performed once, at the IVF index build time. It produces the acceptance threshold $\hat{\lambda}$ that governs the per-query behavior during online search, and the RAPS hyperparameters. For instance, the example provided in Figure 4 identifies $\hat{\lambda}_1 = 0.85$ for a requested FNR of 10%, and $\hat{\lambda}_2 = 1.12$ for 15%. We

Algorithm 2 IVF Search (Online)

1: **Input:** Query vector $\mathbf{x}$, k, optimal threshold $\hat{\lambda}$, calibrated hyperparameters γ, c_{reg}.
2: **Output:** Predicted output $\hat{S_p}$.
3: $p \leftarrow 1$
4: **for** $p \leftarrow 1$ **to** P **do**
5: $\quad \hat{S}_p \leftarrow \texttt{IVF_scan}(\mathbf{x}, p)$
6: $\quad$ Compute nonconformity score $\pi_p \leftarrow \texttt{score}(k, \hat{S}_p)$
7: $\quad \hat{\pi}_p \leftarrow (1 - \pi_p) + \gamma \cdot \max(0, p - c_{\text{reg}})$
8: $\quad$ **if** $\hat{\pi}_p < \hat{\lambda}$ **then**
9: $\quad\quad$ **Break**
10: **return** $\hat{S}_p$

stress the following two aspects: (1) computing non-conformity scores does not add extra cost because they are based on intermediate results already produced by the standard IVF search, and (2) although different risk levels α require different thresholds $\hat{\lambda}$, all optimizations (Lines 11–18) can reuse the same set of precomputed non-conformity scores. As we will see later, the most time-consuming part of calibration is computing these scores.

4.2.3 Online IVF Search. The online search phase iteratively probes the IVF index to find the k-nearest neighbors of a query $\mathbf{x}$ while ensuring that the error guarantee, controlled by the calibrated threshold $\hat{\lambda}$, is satisfied. The procedure is summarized in Algorithm 2.

The IVF search is configured to scan all P clusters. As each cluster is probed, we compute the non-conformity score for the current set of nearest neighbors using Eq. (7), and apply the regularization described in Eq. (9). At every iteration, the regularized non-conformity score of the k-th farthest vector in the current result set is compared to $\hat{\lambda}$ (Line 8). According to the incremental nature of IVF search, these scores are computed without extra overhead, enabling early stopping: if the score drops below $\hat{\lambda}$, it indicates that the error guarantee has been met, and the search terminates early, returning the current set of k nearest neighbors S_p. This modification allows the CRC framework to minimize the number of clusters probed, rather than adjusting the prediction set size, aligning with the optimization objective described in Eq. (6). As clusters are probed sequentially, the search can terminate as soon as the guarantee is met, minimizing p. For example, in Figure 4, the search stops after probing the third cluster, where the non-conformity score (0.71) falls below $\hat{\lambda}_1 = 0.85$. If the stopping criterion is not met, the search continues probing one more cluster at a time until either the threshold is satisfied or all P clusters are exhausted.

This procedure ensures that the search probes only as many clusters as necessary to achieve the desired accuracy, reducing computation time and adapting to the difficulty of each query. Moreover, it naturally integrates with the standard IVF search process, since intermediate nearest neighbor results are already available as clusters are incrementally scanned.

5 EXPERIMENTS

In the empirical evaluation, we aim to answer the following research questions:

Table 3: Datasets summary.

Dataset	d	N	Query
BERT [12]	768	30522	10k
FASTTEXT [8]	300	1M	10k
GIST [24]	960	1M	1k
SIFT1M [33]	128	1M	10k
DEEP10M [42]	96	10M	10k

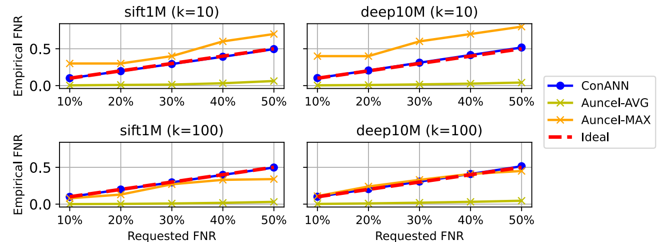


Figure 5: Validity results for Auncel.

(RQ1) Validity: Does ConANN accurately control the FNR across varying configurations? How does it compare against state-of-the-art Bounded-Error ANN methods and Best-Effort ANN methods?
(RQ2) Efficiency: How does ConANN compare to the baselines in terms of the average number of clusters searched? Does it expand the search space to provide formal error guarantees?
(RQ3) Adaptivity: Is ConANN capable of adapting the search space to the particularities of each query?
(RQ4) Overhead: What is the performance overhead of integrating conformal methods into the IVF search?

5.1 Setup

The following section outlines the experimental setup, including hardware and software configurations.

5.1.1 Hardware. The experiments were conducted on Google Cloud Platform (GCP) using a virtual machine with an `Intel(R) Xeon(R) CPU @ 2.80GHz`, 64 vCPUs, 256 GB RAM, and Ubuntu 20.04.6 LTS. The Intel-MKL SIMD instruction set was used throughout all the experiments to speed up the computation.

5.1.2 Datasets. We use high-dimensional datasets, ranging from 96 to 960 dimensions and 30k to 10M vectors, widely used in ANN research [19, 24, 40]. A summary is provided in Table 3. BERT [12] and FASTTEXT [8] are pre-trained word embeddings. The SIFT1M [33] and GIST [24] datasets are based on image data using their respective SIFT and GIST descriptors. The DEEP10M [42] dataset contains image embeddings generated by a convolutional neural network. Each dataset includes a uniform sample of 10k query vectors, with the exception of GIST, where we sample only 1k query vectors due to the prohibitively large dimensionality.

5.1.3 Baselines. Auncel [50], a Bounded-Error-type method, provides a different type of error guarantee than our method. Specifically, it targets a bound on the *maximum* false negative rate (FNR), whereas we provide guarantees on the *average* FNR. Despite this difference, we include Auncel in our evaluation because it is the only existing method that attempts to control the approximation

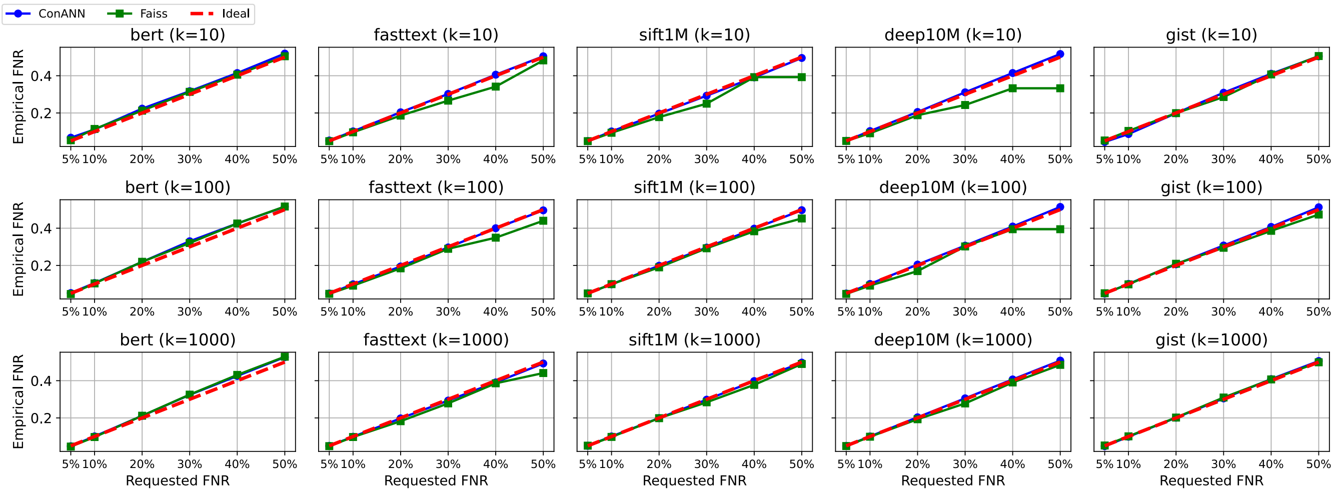


Figure 6: Validity results. ConANN offers effective FNR control over the tested datasets.

error. Our evaluation uses the hyperparameters reported in the original paper and tests the same datasets, as the tuning procedure is not disclosed. Since no other existing system offers formal false negative rate (FNR) guarantees under arbitrary data distributions, we compare ConANN against Faiss [25]. While Faiss does not natively support FNR-constrained configurations, it can be adapted into a best-effort baseline by tuning its probing strategy [50]. Specifically, we perform an offline calibration on a configuration query sample: starting from a small number of clusters, we incrementally increase the probe count until the empirical average FNR falls below the target threshold α. The smallest such probe count is then fixed and applied uniformly to all test-time queries.

5.1.4 Configuration. ConANN is integrated into Faiss 1.9.0 and uses Faiss' core features like distance computation and memory management. Parallel computing based on OpenMP is enabled with 60 threads for both ConANN and Faiss to ensure fairness. We used the L2-squared Euclidean distance as the distance metric in all the experiments. For the validity, efficiency, and overhead experiments the IVFFlat index was used, storing the vectors in their original state inside the index. Similar to related studies, the number of clusters P was fixed to 1024, except for the BERT dataset, where P was set to 128 clusters [50]. We trained the index on half of the dataset in the interest of time. For all datasets, we choose 50% of the query dataset for calibration and use the remaining for RAPS hyperparameter tuning (1000 query vectors) and testing. The min-max normalization of the non-conformity scores uses 0 as the minimum value. It approximates the maximum distance geometrically via the diagonal of the hyperrectangle containing all vectors, considering the dataset range and dimensionality. Exact distance computation is also feasible but costly, requiring quadratic time relative to dataset size. We use Brent's numerical optimization method [9] to find the optimal $\hat{\lambda}$. Without loss of generality, we calibrate and evaluate ConANN on three fixed values of k. However, the method's guarantees hold beyond this setting. Additional experiments calibrating over a range of k values, demonstrating runtime robustness in case of dynamic k changes from one query to the next, are available in our public GitHub repository.

5.1.5 Metrics. We measure empirical FNR using Eq. (1) to validate the adherence to the theoretical bound. For each query, we log the number of clusters searched to assess ConANN's adaptivity and compare its efficiency against Faiss. Two metrics are used: (1) the per-query cluster ratio, computed as ConANN's reported p divided by Faiss' static p, and (2) the average cluster ratio, defined as Faiss' static p over ConANN's average p. For example, a ratio of 1.2 implies Faiss searches 20% more clusters. We also measure ConANN's latency overhead by comparing end-to-end query times where p matches. Finally, we report ConANN's calibration time.

5.2 Experimental Results

We evaluate ConANN on five standard ANN datasets under varying k and FNR settings, and observe that it:

(1) Consistently meets user-defined FNR bounds across diverse datasets and calibration sizes (Figure 6), whereas Auncel and Faiss frequently deviate from the target FNR.
(2) Provides formal error guarantees proven empirically, without increasing the search demand and even decreasing the number of clusters needed to be probed up to 62.8% when compared to Faiss (Figure 7),
(3) Dynamically adapts the search space per query (Figure 8), regardless of dataset characteristics, and
(4) Introduces minimal latency (less than 1.97 ms in our experiments, Figure 9), with calibration completing in under 4 minutes regardless of dataset size or complexity.

5.2.1 Validity. We begin by evaluating Auncel, which aims to guarantee a bound on the *maximum* false negative rate (FNR). Using the hyperparameters reported in the original paper, we assess Auncel on the SIFT1M and DEEP10M datasets, as no tuning method is provided to test other datasets. As shown in Figure 5, Auncel frequently violates its maximum FNR targets, particularly for $k = 10$, and offers no control over the average FNR. In fact, the average FNR often approaches that of exhaustive search, indicating highly conservative behavior. Due to these shortcomings, we omit Auncel from subsequent comparisons.

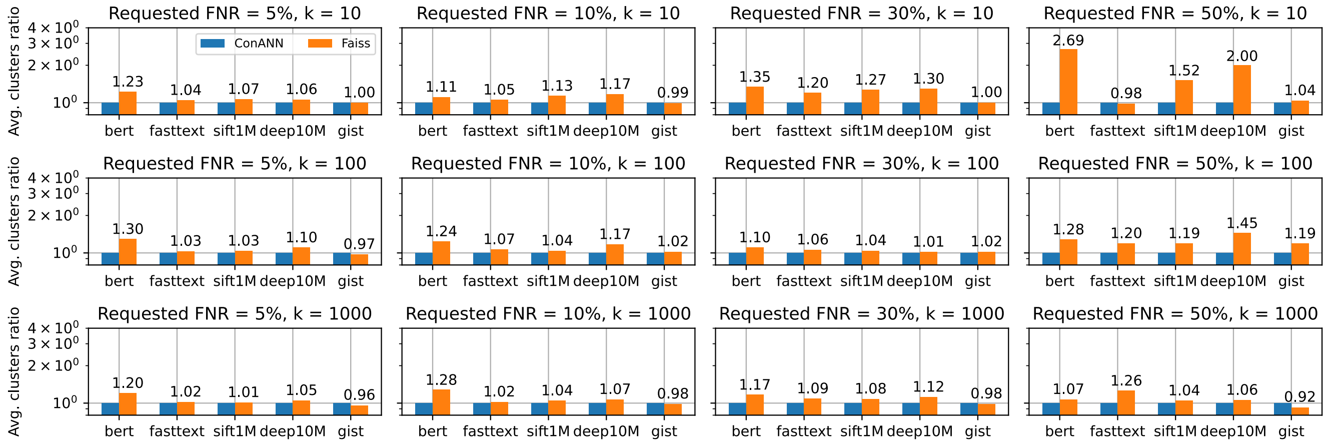


Figure 7: Efficiency results. ConANN matches or even reduces the probing depth when compared to standard methods.

We assess the validity of ConANN by measuring how well the empirical FNR tracks the requested FNR. Figure 6 presents results for $k \in \{10, 100, 1000\}$ and requested FNRs ranging from 5% to 50%. The red dashed line indicates the ideal behavior, where empirical FNR exactly matches the user-specified threshold. ConANN closely follows this ideal FNR across all conditions, indicating that it reliably satisfies the used-defined constraint. Notably, ConANN maintains this alignment even under challenging settings, and deviates at most by only 0.03 percentage points. For instance, on the GIST dataset with $k = 10$, where calibration data is limited to 500 vectors, ConANN still tracks the target line well.

In comparison, Faiss tends to undershoot the target FNR across several settings, which is particularly noticeable on FASTTEXT, SIFT1M, and DEEP10M for $k = 10$ and $k = 100$. This conservative tendency leads to lower empirical FNRs than requested, indicating unnecessarily large search spaces. While this preserves a pessimistically low FNR, it undermines efficiency, as we will note in the next section. The trend attenuates for larger values of k, where both methods converge more closely to the ideal.

On the BERT dataset, ConANN exhibits marginal deviations of the requested FNR across targets. This behavior likely arises from the nature of the non-conformity scores. Although the calibration sample is large and uniformly drawn, the scores on BERT are less discriminative, potentially due to dense, semantically clustered embeddings and small distances that cause precision losses during the min-max normalization. As a result, the correspondence between quantile thresholds and true query difficulty can be slightly weakened, leading to negligible perturbations from the target FNR, bounded in expectation. This behavior aligns with the theoretical expectations of CRC [4]. However, further investigations could help refine the quality of the non-conformity scores.

Overall, these findings affirm ConANN's capacity to maintain reliable FNR control, with only slight deviations that remain within the expected probabilistic bounds of the CRC framework.

5.2.2 Efficiency. We measure the efficiency of the methods using the ratio of average clusters searched by each method to achieve the desired FNR. First, we vary the requested FNR and the k value. The results are showcased in Figure 7.

Table 4: Average clusters ratio for ConANN against Faiss for variable total number of clusters P ($k = 100$, requested FNR $= 10\%$). A ratio of 1.1 indicates that Faiss searched 10% more clusters on average than ConANN. Our results across datasets show that ConANN is not sensitive to varying P.

P	fasttext	sift1M	deep10M	gist
512	1.09	1.06	1.22	1.01
768	1.06	1.11	1.15	1.00
1024	1.07	1.04	1.17	1.02
1536	1.05	1.05	1.15	1.00
2048	1.01	1.02	1.13	1.00

Across all experiments, ConANN either matches or reduces the search space compared to Faiss, achieving the same empirical FNR. This demonstrates ConANN's ability to adapt the search space size based on the characteristics of each query vector. In contrast, Faiss often searches more clusters than necessary. For instance, a Faiss-to-ConANN ratio of 1.35 means Faiss searches 35% more clusters, incurring unnecessary computational overhead without any gain in accuracy. This inefficiency stems from Faiss's conservative strategy for approximating the FNR. As illustrated in Figure 6 for the SIFT1M dataset at $k = 10$, Faiss' deviation from the target FNR leads directly to excess cluster searches. Specifically, while Faiss initially searches only 7% more clusters than ConANN at a 5% requested FNR, this gap widens substantially, to 52%, as the FNR target decreases. A similar trend is observed for the DEEP10M dataset at $k = 10$, where Faiss searches twice as many clusters as ConANN for 50% requested FNR because of the empirical FNR deviation.

ConANN's efficiency advantage is even more pronounced on BERT, where the larger calibration data sample (about one-third of the entire dataset) allows ConANN to better capture the data distribution. For smaller calibration data sizes, the results remain valid but are more conservative. For example, on GIST, where the calibration sample consists of 500 vectors only, the CRC framework is more conservative, leading to less efficient search behavior. While the validity property is still met, the efficiency gains observed for other datasets are not present on GIST. This is expected and consistent with the design of the CRC framework, which prioritizes valid coverage over aggressive optimization. However, the number of clusters searched by ConANN does not deviate significantly from Faiss in our experiments, showing the efficiency of RAPS.

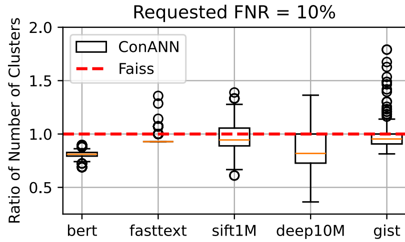


Figure 8: Adaptivity results (k = 100, requested FNR = 10%). ConANN adjusts the search effort by query difficulty.

We also assess the robustness of our method across varying P values, as shown in Table 4. We observe minimal changes in the average clusters searched ratio compared to Faiss, demonstrating that ConANN is not sensitive to P. On the DEEP10M dataset, the improvement becomes more pronounced as P decreases, peaking at a 22% ratio for $P = 512$, suggesting that an optimal P choice can yield further efficiency gains. However, since the data distribution is unknown, selecting the ideal P can be challenging. Nonetheless, our results show that ConANN remains both efficient and valid, regardless of P selection. Across diverse datasets and varying total number of clusters, ConANN's empirical FNR deviates by no more than 0.002 percentage points from the target, demonstrating robustness across all conditions.

Our results demonstrate ConANN's ability to guarantee the user-defined error threshold without searching unnecessary clusters, and, in many cases, even reducing the search space. This holds consistently across different values of k, requested FNRs, and datasets.

5.2.3 Adaptivity. Figure 8 highlights ConANN's ability to adapt the size of the search space to meet a fixed FNR target across diverse datasets. Specifically, we evaluate the ratio of the number of clusters searched by ConANN to that of Faiss, under a requested FNR of 10% with $k = 100$. This ratio, centered around the red dashed line (baseline Faiss performance), quantifies ConANN's adaptivity: values below 1 indicate more efficient search due to adaptive behavior, while values above 1 imply overcompensation. We observe that ConANN effectively tailors the number of clusters searched across datasets. For example, on the BERT dataset, the method consistently selects fewer clusters than Faiss, leveraging the availability of high-quality calibration data, as discussed above. In contrast, datasets like DEEP10M and SIFT1M exhibit a wider spread in the number of clusters searched, reflecting the intrinsic variability in query difficulty and data distribution. Despite this, ConANN maintains an overall balanced ratio below 1, indicating robust adaptivity without excessive conservativeness. Interestingly, while FASTTEXT and GIST show some outlier queries requiring substantially more clusters, the median remains close to or below 1.0, reinforcing ConANN's ability to adapt to both easy and hard queries. The wide spread on GIST is likely caused by the small calibration sample used, as discussed above; the queries corresponding to the outliers are likely not well represented in the calibration distribution, leading the method to behave conservatively in order to maintain validity. These results confirm that ConANN dynamically adjusts the retrieval process in response to query-level and dataset-level characteristics, rather than relying on static thresholds.

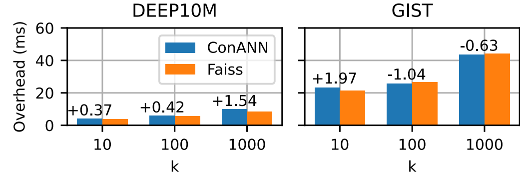


Figure 9: Latency overhead (requested FNR = 10%). ConANN introduces minimal latency overheads.

Table 5: Calibration breakdown per dataset, measured in seconds (k = 100, requested FNR = 10%). ConANN completes the calibration in under 4 minutes for all the tested datasets.

Step	bert	fasttext	sift1m	deep10m	gist
Index Building	0.727	25.610	10.043	20.229	52.013
Compute Scores	2.107	68.268	33.874	212.666	20.807
Pick Hyperparameters	2.131	5.186	4.617	5.570	0.556
Regularize Scores	0.011	0.155	0.154	0.150	0.018
Optimize (sec)	1.015	5.021	3.981	6.270	0.821
Total	**5.264**	**78.630**	**42.626**	**224.656**	**22.202**

5.2.4 Runtime Overhead. As described in Algorithm 2, ConANN extends the IVF search by adding a lightweight termination check at runtime with a single if statement. We stress that the non-conformity scores computation is already performed by Faiss, which, by design, probes the clusters iteratively, starting from 1, and computes intermediary steps. Therefore, the integration with ConANN is straightforward. We first measure the runtime cost of ConANN by measuring the per-query latency for a requested FNR of 10% across different values of k and averaging the results. We chose two datasets for this experiment, DEEP10M and GIST, to perform the measurements on a very high-dimensional dataset (GIST) and a large dataset with low dimensionality (DEEP10M). We compare only the queries where both methods searched the same number of clusters to quantify the overhead added by ConANN on top of the index search. For the DEEP10M dataset, 727, 585, and 629 queries were used for k = 10, 100, and 1000, respectively. For GIST, 308, 38, and 55 queries were found for k = 10, 100, and 1000, respectively. Figure 9 shows the total latency incurred by ConANN and Faiss, with the annotated values indicating the additional overhead (in milliseconds) introduced by ConANN relative to Faiss. On both datasets, ConANN introduces minimal overheads, of a maximum of 1.97ms on GIST for k = 10. The measured overhead is always negligible and can also be attributed to system-level factors such as CPU scheduling, which is likely the case for the GIST dataset at k = 100 and k = 1000, where we measure a negative overhead. Overall, ConANN imposes negligible runtime latency overhead across datasets and k values, while providing formal error guarantees. This reinforces its practicality as a plug-in extension for conventional ANN methods in latency-sensitive applications.

5.2.5 Calibration Overhead. We evaluate the one-time calibration cost of ConANN by profiling the end-to-end time across its four main stages: non-conformity score computation, RAPS hyperparameter selection, score regularization, and $\hat{\lambda}$ optimization. All measurements are performed for a fixed configuration with a requested FNR of 10% and k = 100. Table 5 shows a detailed breakdown for five datasets. The total calibration time ranges from 5.26s (BERT) to 224.66s (DEEP10M), with the compute scores step being the

Table 6: **RAPS improvement in terms of average number of clusters searched. ConANN achieves notable reductions in average number of clusters searched when employing RAPS.**

Method	bert	gist1m	deep10m
w/o RAPS	123.56	1008.03	991.5
ConANN	33.79	66.33	9.37

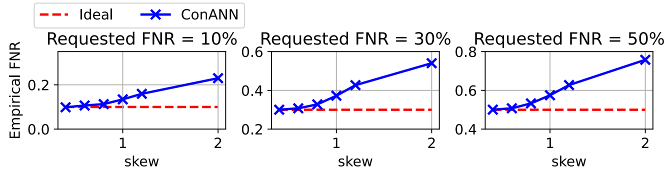


Figure 10: Validity under power-law skew on SIFT1M (k = 100). Skew controls the bias towards popular queries.

dominant contributor. Importantly, the score computation is independent of downstream optimization and can be reused across multiple calibration runs, amortizing its cost. While calibration for large datasets like DEEP10M incurs higher cost, it remains a one-time offline process. For smaller or more structured datasets such as GIST and BERT, the overhead is minimal.

6 DISCUSSION

The effect of regularization. The regularization step (RAPS) refines the non-conformity scores, enhancing their discriminative power. As IVF search increases the number of cluster probes, relying solely on the k-th nearest neighbor distance often results in many equal non-conformity scores. RAPS mitigates this by distinguishing between these probes. In our experiments, the regularization step significantly improves search efficiency. Without RAPS, ConANN is more conservative in the number of clusters searched, leading to valid but less efficient retrieval. As shown in Table 6, RAPS results in notable reductions: from 123.56 to 33.79 on BERT (-72.7%), from 1008.03 to 66.33 on GIST1M (-93.4%), and from 991.5 to 9.37 on DEEP10M (-99.1%). These findings align with previous work [5], which reported up to 10x reductions in prediction set size, and confirm that RAPS enhances efficiency across various datasets while maintaining ConANN's adherence to FNR constraints.

Query Distribution Shift. As noted in Section 3.2, ConANN assumes the calibration and test queries come from the same distribution. To test robustness, we simulate a power-law skew in the test queries (0.0 for uniform, up to a factor of 2) while keeping the calibration queries uniformly sampled. Figure 10 shows that ConANN can tolerate small drifts, but its empirical FNR increases with skew, deviating from the target. At 10% target FNR, the observed FNR rises to 22% under strong skew, and the gap grows at higher targets. These results highlight the need for drift adaptation techniques. A naive solution is to recalibrate periodically, ensuring the calibration sample matches the current query distribution. More principled alternatives could target drift detection, where a violation of the exchangeability assumption is detected using conformal test martingales [47]. Others could leverage recent methods like Weighted Conformal Prediction [7, 46] and Adaptive Conformal Inference [20], which adjust nonconformity scores to account for distribution shifts, for instance by penalizing the impact of old queries. Integrating such techniques for ConANN is an important direction for future work.

Personalized Querying. State-of-the-art methods for error bounding in ANN often rely on sampling techniques that require the sample distribution to match the data distribution. In contrast, ConANN operates by ensuring that the query distribution aligns with the calibration data, opening the door for personalized predictions based on specific workloads. This distinction could enable ConANN to adapt more effectively to varying query characteristics. Further experiments are needed to explore this potential fully, as our current setup follows the ANN literature to sample queries uniformly at random. That said, this insight suggests opportunities for optimization in real-world applications. For example, if an application frequently queries vectors close to cluster centroids, ConANN could leverage this characteristic to reduce the number of clusters searched, ultimately decreasing query latency and improving efficiency.

Marginal Coverage. ConANN guarantees marginal coverage, similar to other conformal methods: the false negative rate (FNR) constraint holds on average over the query distribution, not for individual queries. It is well-established that achieving conditional coverage (i.e., for each query) without assumptions on the data distribution, or with finite calibration data, is provably impossible for conformal methods [16]. While this limits fine-grained control, it is a necessary trade-off for distribution-free guarantees. Crucially, ConANN still provides practical, per-query adaptivity while ensuring global reliability, addressing a gap in the literature where state-of-the-art methods either provide formal guarantees or adapt to data distributions, but not both. Therefore, ConANN represents a step forward in error bounding for ANN systems.

7 RELATED WORK

Error Guarantees for ANN. Auncel [50] is the only known method offering bounded-error guarantees for IVF. It assumes a uniform distribution of result vectors within a hypersphere around the query, computing error by comparing the hypersphere's volume to that of searched clusters. As shown in Section 1, this assumption breaks down on complex, non-uniform datasets (e.g., word embeddings), limiting practical applicability. To date, no other existing method offers explicit recall guarantees for ANN search [14]. Prior work instead focuses on bounding distance errors, targeting primarily latency improvements. Locality-Sensitive Hashing methods [11, 22, 35] follow this trend but suffer from high memory overhead and poor recall in practice [14, 45]. Conceptually related to LSH, ADSampling [18] leverages distance-based random projections and adaptive dimension sampling via hypothesis testing to refine distance estimates, but lacks auto-tuning, hindering real-world usability. RaBitQ [19] proposes a drop-in replacement for PQ [24] with tighter distance bounds, but still does not address recall guarantees directly, as even minor distance approximation errors can lead to poor recall [14]. LEAT [31] predicts p per query to optimize efficiency, introducing iterative probing later used in Auncel and ConANN, though without error bounds. A recent development, the Subspace Collision Framework [49], proposes an ANN search method using random subspace sampling. It scores vectors by their frequency as close neighbors across sampled subspaces, selecting top candidates for re-ranking via exact distances.

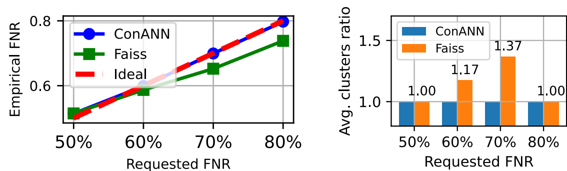


Figure 11: Validity (left) and efficiency (right) results for IVF-PQ on SIFT1M ($k = 100$). ConANN effectively controls the recall of a quantized IVF index.

While offering theoretical guarantees without query sampling, it assumes uniformly distributed distances across dimensions and is incompatible with existing index structures.

Applications of Conformal Methods. Conformal prediction methods have been widely used in applications such as medical diagnosis [34], image classification [5], and natural language processing [15], where reliable uncertainty quantification is critical. Next, a related line of work applies conformal prediction with kNN for regression tasks, as explored in prior research [38]. kNN-based regression compares data points to their neighbors to quantify their dissimilarity. In contrast, our work directly quantifies the approximation error introduced by the approximate kNN search. Lastly, applications of conformal prediction to data-management problems are still rare. dbET introduces a strategy to extend existing DBMS query plan selection methods with conformal prediction [32]. Their work produces execution time interval estimations guaranteed to cover the true execution time with probability $(1 - \alpha)$, which can be later fed into query optimization techniques.

Approximate Query Processing. Approximate query processing (APQ) with error bounds has been an active research area in relational DBMS systems [1, 10, 39]. BlinkDB [2] allows users to trade accuracy for speed on a per-query basis. It uses a sampling strategy that ensures sufficient representation of underrepresented data. The sample size is dynamically adjusted based on user time constraints or error bounds. This concept was extracted into a middleware approach for VerdictDB [39], which uses query rewriting to apply APQ on existing database systems such as SparkSQL. However, these methods are not tailored for vector data.

8 FUTURE WORK

Improved Retrieval with Confidence. A promising direction for future work is the incorporation of confidence-aware retrieval into ConANN. Rather than returning nearest neighbors based solely on vector similarity, ConANN could be extended to output calibrated confidence scores that quantify the likelihood of a result being relevant to a given query. For instance, in IVF-based document retrieval, this approach would allow ConANN to return not only top-ranked candidates but also an associated measure of confidence grounded in prior knowledge. This departs from conventional distance-based methods by enabling the system to differentiate between semantically ambiguous and highly relevant candidates, even if they are similarly distant in the latent embedding space. Such confidence estimates could further support adaptive ranking or filtering strategies, down-weighting or excluding low-confidence results to enhance both the precision and interpretability of retrieval outcomes.

Applicability beyond IVF. ConANN's core components, including the non-conformity score and ranked cluster probing, naturally extend to IVF-based indexing variations. This enables straightforward adaptation within the broader family of IVF methods. Our preliminary experiments on the SIFT1M dataset show that ConANN can effectively control the error rate even when IVF is used in combination with a vector compression method like Product Quantization [24, 26] (IVF-PQ)[3], without incurring additional search overheads, thereby demonstrating its plug-and-play nature. The experiment, depicted in Figure 11, reveals several key observations. First, due to the coarse quantization inherent to IVF-PQ, the minimum achievable FNR is approximately 50%. Additionally, when the requested FNR exceeds 80%, only one cluster needs to be searched. At the 50% FNR level, both ConANN and Faiss search all clusters, as this is the only way to ensure the target recall. As the allowed error increases, ConANN begins to search fewer clusters, mirroring trends observed in our main experiments. At the 80% requested FNR, both ConANN and Faiss search only a single cluster per query. However, ConANN exhibits a distinct behavior in this regime: to satisfy the validity constraint, it abstains from searching on a subset of queries (roughly 800), prioritizing a reliable expected guarantee. The average cluster ratio in Figure 11 excludes these abstentions. These preliminary results suggest that ConANN can generalize to compressed and quantized search settings, which may enable scalability to real-world billion-scale datasets. Nonetheless, the CRC framework is a distribution-free wrapper for any black-box model and is, therefore, not limited to IVF. This opens the possibility of extending the approach to fundamentally different ANN indexes, such as Hierarchical Navigable Small World (HNSW) graphs [36].

Metric-agnostic Design. Moreover, ConANN is distance metric-agnostic, meaning it can be used with diverse distance measures beyond Euclidean distance, such as cosine similarity, Manhattan distance, or even domain-specific metrics. This broad applicability makes ConANN a versatile tool for improving reliability in nearest neighbor search tasks, paving the way for adaptable, efficient, and robust retrieval systems across a wide range of domains.

9 CONCLUSIONS

We presented ConANN, the first distribution-free framework that delivers formal error guarantees for IVF-based Approximate Nearest Neighbor search. Leveraging Conformal Risk Control, ConANN dynamically adapts search effort per query, ensuring rigorous error control without assumptions on data distribution. Empirical results demonstrate that ConANN reliably satisfies target error rates, reduces search costs, and incurs minimal computational overhead. Integrated into the FAISS library, ConANN not only enhances the robustness of ANN systems but also highlights the broader potential of conformal methods in data management tasks beyond traditional predictive models.

ACKNOWLEDGMENTS

This work was supported by the Wallenberg Foundation (WASP-NEST Data-Bound), the Google Cloud Research Credits program, Vinnova (2023-01406), the National Academic Infrastructure for Supercomputing in Sweden (NAISS), partially funded by the Swedish Research Council through grant agreement no. 2022-06725, and Digital Futures.

[3]The IVF-PQ index was configured with 256 clusters per 8 subvectors.

REFERENCES

[1] Sameer Agarwal, Henry Milner, Ariel Kleiner, Ameet Talwalkar, Michael I. Jordan, Samuel Madden, Barzan Mozafari, and Ion Stoica. 2014. Knowing when you're wrong: building fast and reliable approximate query processing systems. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 481–492. https://doi.org/10.1145/2588555.2593667

[2] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *EuroSys*. ACM, 29–42.

[3] Anastasios N. Angelopoulos and Stephen Bates. 2021. A Gentle Introduction to Conformal Prediction and Distribution-Free Uncertainty Quantification. *CoRR* abs/2107.07511 (2021). arXiv:2107.07511 https://arxiv.org/abs/2107.07511

[4] Anastasios Nikolas Angelopoulos, Stephen Bates, Adam Fisch, Lihua Lei, and Tal Schuster. 2024. Conformal Risk Control. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=33XGfHLtZg

[5] Anastasios Nikolas Angelopoulos, Stephen Bates, Michael I. Jordan, and Jitendra Malik. 2021. Uncertainty Sets for Image Classifiers using Conformal Prediction. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=eNdiU_DbM9

[6] Artem Babenko and Victor S. Lempitsky. 2015. The Inverted Multi-Index. *IEEE Trans. Pattern Anal. Mach. Intell.* 37, 6 (2015), 1247–1260. https://doi.org/10.1109/TPAMI.2014.2361319

[7] Rina Foygel Barber, Emmanuel J. Candès, Aaditya Ramdas, and Ryan J. Tibshirani. 2023. Conformal prediction beyond exchangeability. *The Annals of Statistics* 51, 2 (April 2023). https://doi.org/10.1214/23-AOS2276

[8] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomás Mikolov. 2017. Enriching Word Vectors with Subword Information. *Trans. Assoc. Comput. Linguistics* 5 (2017), 135–146. https://doi.org/10.1162/TACL_A_00051

[9] Richard P Brent. 2013. *Algorithms for minimization without derivatives*. Courier Corporation.

[10] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. 2017. Approximate Query Processing: No Silver Bullet. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 511–519. https://doi.org/10.1145/3035918.3056097

[11] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry (SCG '04)*. Association for Computing Machinery, New York, NY, USA, 253–262. https://doi.org/10.1145/997817.997857

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/V1/N19-1423

[13] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss library. (2024). arXiv:2401.08281 [cs.LG]

[14] Karima Echihabi, Themis Palpanas, and Kostas Zoumpatianos. 2021. New Trends in High-D Vector Similarity Search: AI-driven, Progressive, and Distributed. *Proc. VLDB Endow.* 14, 12 (2021), 3198–3201. https://doi.org/10.14778/3476311.3476407

[15] Adam Fisch, Tal Schuster, Tommi S. Jaakkola, and Regina Barzilay. 2021. Efficient Conformal Prediction via Cascaded Inference with Expanded Admission. In *International Conference on Learning Representations*. https://openreview.net/forum?id=tnSo6VRLmT

[16] Rina Foygel Barber, Emmanuel J Candes, Aaditya Ramdas, and Ryan J Tibshirani. 2021. The limits of distribution-free conditional predictive inference. *Information and Inference: A Journal of the IMA* 10, 2 (2021), 455–482.

[17] Yarin Gal and Zoubin Ghahramani. 2016. Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 (JMLR Workshop and Conference Proceedings)*, Maria-Florina Balcan and Kilian Q. Weinberger (Eds.), Vol. 48. JMLR.org, 1050–1059. http://proceedings.mlr.press/v48/gal16.html

[18] Jianyang Gao and Cheng Long. 2023. High-Dimensional Approximate Nearest Neighbor Search: with Reliable and Efficient Distance Comparison Operations. *Proceedings of the ACM on Management of Data* 1, 2 (June 2023), 1–27. https://doi.org/10.1145/3589282

[19] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 2, 3 (May 2024), 1–27. https://doi.org/10.1145/3654970

[20] Isaac Gibbs and Emmanuel Candes. 2021. Adaptive Conformal Inference Under Distribution Shift. In *Advances in Neural Information Processing Systems*, Vol. 34. Curran Associates, Inc., 1660–1672. https://proceedings.neurips.cc/paper/2021/hash/0d441de75945e5acbc865406fc9a2559-Abstract.html

[21] Anna Gogolou, Theophanis Tsandilas, Karima Echihabi, Anastasia Bezerianos, and Themis Palpanas. 2020. Data Series Progressive Similarity Search with Probabilistic Quality Guarantees. In *SIGMOD Conference*. ACM, 1857–1873.

[22] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing (STOC '98)*. Association for Computing Machinery, New York, NY, USA, 604–613. https://doi.org/10.1145/276698.276876

[23] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2010. Improving Bag-of-Features for Large Scale Image Search. *Int. J. Comput. Vis.* 87, 3 (2010), 316–336. https://doi.org/10.1007/S11263-009-0285-2

[24] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 1 (2011), 117–128. https://doi.org/10.1109/TPAMI.2010.57

[25] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[26] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. Searching in one billion vectors: Re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 861–864. https://doi.org/10.1109/ICASSP.2011.5946540 ISSN: 2379-190X.

[27] Yehuda Koren, Robert M. Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *Computer* 42, 8 (2009), 30–37. https://doi.org/10.1109/MC.2009.263

[28] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur P. Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. 2019. Natural Questions: a Benchmark for Question Answering Research. *Trans. Assoc. Comput. Linguistics* 7 (2019), 452–466.

[29] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. 2017. Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 6402–6413. https://proceedings.neurips.cc/paper/2017/hash/9ef2ed4b7fd2c810847ffa5fa85bce38-Abstract.html

[30] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *NeurIPS*.

[31] Conglong Li, Minjia Zhang, David G. Andersen, and Yuxiong He. 2020. Improving Approximate Nearest Neighbor Search through Learned Adaptive Early Termination. In *SIGMOD Conference*. ACM, 2539–2554.

[32] Yifan Li, Xiaohui Yu, Nick Koudas, Shu Lin, Calvin Sun, and Chong Chen. 2023. dbET: Execution Time Distribution-based Plan Selection. *Proc. ACM Manag. Data* 1, 1 (2023), 31:1–31:26. https://doi.org/10.1145/3588711

[33] David G. Lowe. 2004. Distinctive Image Features from Scale-Invariant Keypoints. *Int. J. Comput. Vis.* 60, 2 (2004), 91–110. https://doi.org/10.1023/B:VISI.0000029664.99615.94

[34] Charles Lu, Andréanne Lemay, Ken Chang, Katharina Höbel, and Jayashree Kalpathy-Cramer. 2022. Fair Conformal Predictors for Applications in Medical Imaging. In *AAAI*. AAAI Press, 12008–12016.

[35] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases (VLDB '07)*. VLDB Endowment, Vienna, Austria, 950–961.

[36] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems* 45 (Sept. 2014), 61–68. https://doi.org/10.1016/j.is.2013.10.006

[37] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *ICLR (Workshop Poster)*.

[38] H. Papadopoulos, V. Vovk, and A. Gammerman. 2011. Regression Conformal Prediction with Nearest Neighbours. *Journal of Artificial Intelligence Research* 40 (April 2011), 815–840. https://doi.org/10.1613/jair.3198

[39] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. 2018. VerdictDB: Universalizing Approximate Query Processing. In *SIGMOD Conference*. ACM, 1461–1476.

[40] Yun Peng, Byron Choi, Tsz Nam Chan, Jianye Yang, and Jianliang Xu. 2023. Efficient Approximate Nearest Neighbor Search in Multi-dimensional Databases. *Proc. ACM Manag. Data* 1, 1 (2023), 54:1–54:27. https://doi.org/10.1145/3588908

[41] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[42] Yandex Research. 2020. Benchmarks for Billion-Scale Similarity Search. https://research.yandex.com/blog/benchmarks-for-billion-scale-similarity-search. Accessed: 2024-12-19.

[43] Glenn Shafer and Vladimir Vovk. 2008. A Tutorial on Conformal Prediction. *J. Mach. Learn. Res.* 9 (2008), 371–421. https://doi.org/10.5555/1390681.1390693

[44] Sivic and Zisserman. 2003. Video Google: a text retrieval approach to object matching in videos. In *Proceedings Ninth IEEE International Conference on Computer Vision*. 1470–1477 vol.2. https://doi.org/10.1109/ICCV.2003.1238663

[45] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. 2009. Quality and efficiency in high dimensional nearest neighbor search. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data (SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 563–576. https://doi.org/10.1145/1559845.1559905

[46] Ryan J Tibshirani, Rina Foygel Barber, Emmanuel Candes, and Aaditya Ramdas. 2019. Conformal Prediction Under Covariate Shift. In *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/hash/8fb21ee7a2207526da55a679f0332de2-Abstract.html

[47] Vladimir Vovk, Alexander Gammerman, and Glenn Shafer. 2022. *Algorithmic Learning in a Random World*. Springer Nature.

[48] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. AnalyticDB-V: A Hybrid Analytical Engine Towards Query Fusion for Structured and Unstructured Data. *Proc. VLDB Endow.* 13, 12 (2020), 3152–3165. https://doi.org/10.14778/3415478.3415541

[49] Jiuqi Wei, Xiaodong Lee, Zhenyu Liao, Themis Palpanas, and Botao Peng. 2025. Subspace Collision: An Efficient and Accurate Framework for High-dimensional Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–29.

[50] Zili Zhang, Chao Jin, Linpeng Tang, Xuanzhe Liu, and Xin Jin. 2023. Fast, Approximate Vector Queries on Very Large Unstructured Datasets. In *NSDI*. USENIX Association, 995–1011.

RayDB: Building Databases with Ray Tracing Cores

Xuri Shi
Fudan University
xrshi23@m.fudan.edu.cn

Kai Zhang*
Fudan University
zhangk@fudan.edu.cn

X. Sean Wang
Fudan University
xywangCS@fudan.edu.cn

Xiaodong Zhang
The Ohio State University
zhang@cse.ohio-state.edu

Rubao Lee
Freelance Researcher
lee.rubao@ieee.org

ABSTRACT

Ray tracing (RT) cores are a new type of hardware that have been actively integrated into modern GPUs. Recent studies have demonstrated that RT cores can deliver much higher performance than GPU CUDA cores and CPUs for general-purpose data processing tasks like table scan. In this paper, we propose RayDB, the first query engine that leverages RT cores to accelerate database query processing. RayDB transforms the core part of the query execution that involves multiple operators into one single ray tracing job. With a set of proposed encoding and ray launching techniques, RayDB effectively exploits RT cores to accelerate diverse workloads. Experimental results show that RayDB outperforms the state-of-the-art GPU-based query engine by up to 18.3×.

PVLDB Reference Format:
Xuri Shi, Kai Zhang, X. Sean Wang, Xiaodong Zhang, and Rubao Lee. RayDB: Building Databases with Ray Tracing Cores. PVLDB, 19(1): 43 - 55, 2025.
doi:10.14778/3772181.3772185

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/LonelySlim/myOptixDB/tree/fin.

1 INTRODUCTION

With the ever-increasing data volume from applications, modern processors have been intensively studied to enhance the performance of database engines. Representative processors for which database systems have been designed include GPU CUDA cores [4, 13, 32], Tensor cores [12, 15], and FPGAs [26, 33]. Specifically, GPUs were originally designed to accelerate computer graphics. After they were found useful in general-purpose computing for the massive number of cores, the programming model evolved from OpenGL to CUDA/OpenCL, and GPUs are used to accelerate a broad class of data processing tasks. Crystal [32] is the state-of-the-art implementation of GPU databases, which has made a notable advancement in utilizing CUDA cores. Experimental results show that Crystal is 16× faster than the GPU-based HeavyDB and 61× faster than the CPU-based MonetDB. However, Crystal saturates more than 97% of the GPU memory bandwidth for queries in the Star Schema Benchmark (SSB). Since the approach has already tried to minimize the amount of data accessed, it has become difficult to further improve the query performance on CUDA cores.

Recently, commodity GPUs have incorporated ray tracing (RT) cores to boost the real-time rendering of 3D scenes. As an important type of computing resource, RT cores have been adopted in mobile, desktop, and workstation processors, which are under fast development. RT cores efficiently trace rays through a 3D space to identify intersected objects. With user-defined functions (a.k.a., shaders), RT cores can perform customizable operations upon ray-object intersections, providing versatility for various tasks. RT cores have been used to accelerate data processing tasks such as K-nearest neighbor search [25, 38], scan operator [14, 21], and range minimum queries [22]. Specifically, existing work like RTScan [21] has shown that RT cores can bring up to 4.6× higher performance than CUDA cores and CPU for database operators. Therefore, we believe that RT cores have the potential to become another critical computing resource for general-purpose data processing tasks.

Unlike database implementations on CUDA cores, accelerating a data processing program with RT cores requires the program to be transformed into an efficient RT job. In an RT job, data records are transformed into primitives, such as triangles or spheres, positioned in a three-dimensional space with a bounding volume hierarchy (BVH), while a query is converted into rays in a specified region. If the task does not fit such a job transformation, or the mapping is inefficient (e.g., lack of parallelism with a limited number of rays), it may result in even lower performance than CPUs and CUDA cores [14, 21]. Due to the above reason, exploring RT cores to accelerate database queries is quite challenging because an operator like `Join` or `GroupBy` is hard to transform into an independent RT job. Moreover, since the execution of an operator depends on the output of its previous operator in the query plan, the BVHs for the following operators have to be built during query execution, which is very time-consuming. Therefore, simply implementing independent RT-based operators like CUDA-based databases cannot exploit the performance advantage of RT cores.

In this paper, we propose RayDB, a query engine that utilizes ray tracing cores to achieve unprecedented performance. Instead of implementing an RT job for each operator, the main idea of RayDB is to map the core query execution containing multiple operators into a single RT job. To be specific, RayDB maps and encodes the data attributes involved in the core operators, i.e., `Aggregation`, `GroupBy`, and `Scan`, to the coordinates x, y, z in the 3D space, respectively. RayDB pre-builds a set of BVHs for query execution to select from. When building a BVH, the attributes

*Dr. Kai Zhang is the corresponding author.
This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 1 ISSN 2150-8097.
doi:10.14778/3772181.3772185

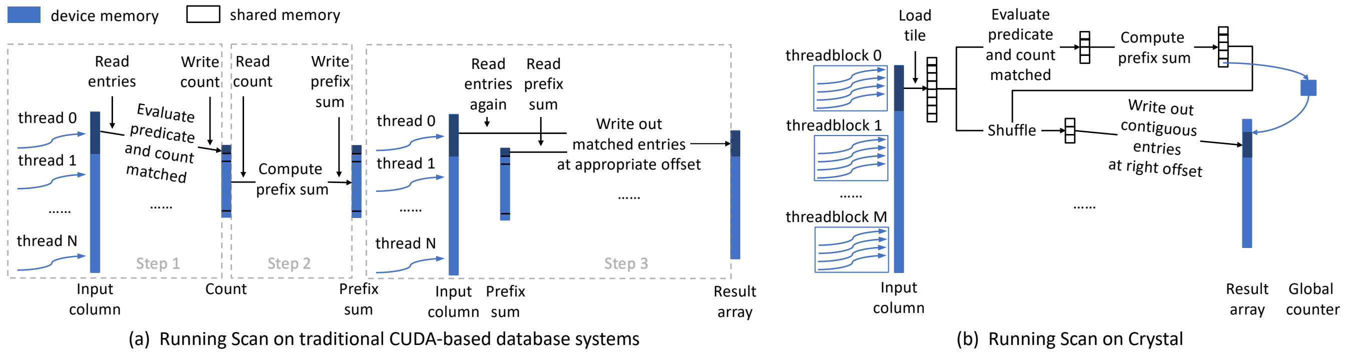


Figure 1: Running Scan on CUDA-based database systems

involved are extracted and encoded as the primitive coordinates of each data record. With a selected BVH for a query, rays in RayDB are launched in the region specified by `Scan` so that they intersect only with primitives that satisfy the predicates, significantly reducing the amount of data accessed. For each primitive, the data attributes involved in `Aggregation` and `GroupBy` are encoded in their 3D coordinates, which can be accessed directly in query processing. Therefore, another main advantage of RayDB is that it can retrieve all the attributes of a data record needed for the three operators with only one memory access, i.e., access to the coordinates of a primitive, dramatically reducing the number of random memory accesses. Moreover, the entire process is accelerated by ray tracing cores, which are designed to perform tasks like this efficiently. After RT processing, RayDB uses CUDA cores to execute other operators like `Having` and `OrderBy` if any. In addition, RayDB natively supports parallel execution on multiple GPUs by partitioning the 3D space into independent regions. It enables flexible scaling out to support workloads of any size.

The contributions of this paper are as follows.

- We propose RayDB, a query engine that leverages RT cores for acceleration by effectively transforming the core part of a query into a ray tracing job.
- We propose a set of encoding and ray-launching mechanisms to enable efficient query execution across diverse workloads.
- We implement the prototype of RayDB, evaluate its performance, and quantitatively analyze the advantages of RT cores.

Experimental results show that RayDB can significantly enhance query performance. Compared with the state-of-the-art CUDA-based method, RayDB improves the query performance by up to 18.3×. In addition to the substantial speedup, the usage of the memory bandwidth drops from 97.4% to only 36.7% on average, which proves that RayDB has broken the limitation of memory bandwidth in GPU-based query execution. To our knowledge, it is the first work that demonstrates that RT cores can be effectively used in building a database engine with unprecedented performance.

2 BACKGROUND AND MOTIVATION

2.1 An Analysis of GPU-based Databases

As a general-purpose computing device, GPU CUDA cores have been intensively studied to build high-performance query engines, such as Crystal [32], HeavyDB [13], BlazingSQL [4], etc. Among them, Crystal is the state-of-the-art. Figure 2 compares the query runtime of Crystal with HeavyDB and MonetDB [5] on SSB flat. The GPU used in the comparison is NVIDIA GeForce RTX 4090, and the CPU used is Intel Xeon Gold 5318Y. The results show that Crystal is, on average, 16× faster than HeavyDB and 61× faster than MonetDB.

The key to Crystal's high performance is that its *tile-based execution model* aims at efficiently utilizing the GPU shared memory, which has an order of magnitude higher bandwidth than the device memory (36618 GB/sec versus 1008 GB/sec in the RTX 4090). Taking the `Scan` operator as an example, the traditional CUDA-based database systems execute the operator in three steps, as shown in Figure 1a. First, multiple CUDA threads are launched to scan their assigned entries and count the matches, storing the results in the *count* array. Then, prefix sums are computed over the *count* array to produce the *prefix sum* array, which provides write offsets. Finally, the same number of threads are launched to read the allocated entries again and write matched entries to the *result* array using the offsets. There are three main performance issues with this approach, including 1) reading the input column twice from the device memory, 2) reading and writing intermediate structures like *count* array and *prefix sum* array in the device memory, 3) each thread writes to a different location in the *result* array, resulting in uncoalesced memory accesses.

Figure 1b demonstrates how Crystal works. Crystal uses a thread block as the basic execution unit, with each block processing a tile of entries. It first loads the tile from device memory into shared memory. Threads then evaluate the predicate in parallel and record match counts of each thread in the *count* array in shared memory. A prefix sum is computed over the *count* array, and a global counter is atomically updated to get the block's output offset. Matched entries are shuffled to form a contiguous array in shared memory, enabling coalesced writes to the *result* array in device memory. By leveraging shared memory, Crystal 1) avoids repeatedly reading input columns from the device memory, 2) eliminates accesses to intermediate structures in device memory, and 3) enables coalesced memory access when writing the *result* array back to the device memory, leading to significant performance improvements.

It is worth noting that existing CUDA-based database systems generally do not use traditional indexes. This design choice is driven

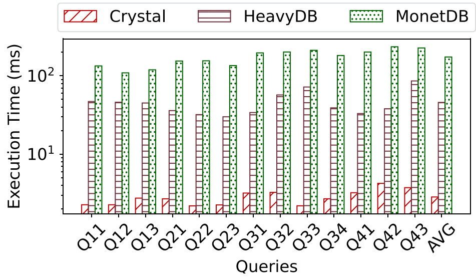


Figure 2: Comparison of query runtime for Crystal, MonetDB, and HeavyDB on SSB flat

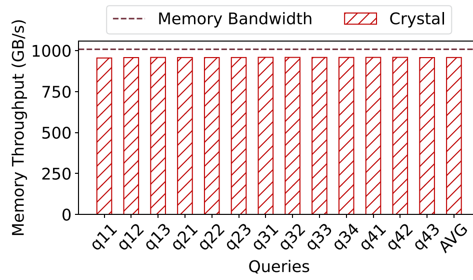


Figure 3: Query memory throughput of Crystal on SSB flat

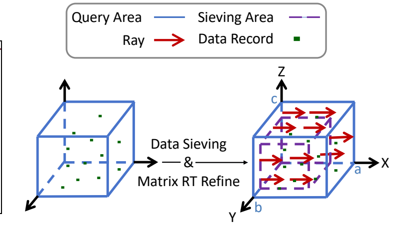


Figure 4: The execution of RTScan

by the architectural and performance characteristics of GPUs: 1) Avoid thread divergence. On GPUs, the SIMT (Single Instruction, Multiple Threads) architecture requires all threads in a warp (typically 32) to execute the same instruction each cycle. When threads diverge due to conditional branches (e.g., if), execution becomes serialized, and inactive threads idle, reducing efficiency [3, 6, 31]. Traditional index structures like B+ trees cause severe divergence because each thread follows a different search path based on its key. As a result, threads within a warp execute different code paths, making indexes poorly suited for efficient execution on GPUs. 2) Avoid uncoalesced memory accesses. To amortize memory access latency and improve bandwidth utilization, GPU hardware employs memory access coalescing, which combines memory accesses from multiple threads within a warp into one or a few large memory transactions when the accessed addresses are contiguous. However, in traditional index structures (such as B+ trees and hash indexes), data and index nodes are typically scattered across memory. Each thread accesses different paths and nodes depending on its query key, resulting in highly irregular and non-contiguous memory accesses across threads. In such scenarios, memory access coalescing fails, leading to a large number of small and fragmented memory transactions. This causes severe memory bandwidth underutilization and a significant drop in performance [6, 31]. As a result, full table scans are generally preferred over indexes in CUDA-based database systems.

Figure 3 shows the memory throughput of Crystal on the 13 queries of SSB flat. It can be observed that its memory throughput is very close (an average of 97.4%) to the GPU device memory bandwidth. As Crystal already makes highly efficient use of memory bandwidth, the potential for further optimization is minimal.

2.2 Background of Ray Tracing

Ray tracing is a rendering technique used in computer graphics to simulate the way rays interact with objects in a scene. It works by tracing the paths of rays as they travel through a 3D space. Objects in the three-dimensional space are represented as primitives, whose types include triangles, spheres, and even custom primitives. All primitives in the space are wrapped by bounding volumes, which are usually Axis-Aligned Bounding Boxes (AABBs). AABBs are then organized hierarchically as a tree known as the Bounding Volume Hierarchy (BVH). A ray tracing job utilizes a BVH to traverse the space and find intersected primitives with the rays.

Although BVH avoids a large number of potential ray-primitive intersection tests, the BVH traversal and intersection tests are still time-consuming. Since the Turing architecture [27], NVIDIA GPUs are equipped with dedicated hardware, i.e., RT cores, to speed up BVH traversal and intersection tests. Taking the classic Turing architecture as an example, each Streaming Multiprocessor (SM) integrates one RT core, which operates alongside CUDA cores and Tensor cores [27]. The RT core essentially adds a dedicated pipeline (ASIC) to the SM. It can access the BVH and configure some L0 buffers to reduce the delay of BVH and triangle data access. The request is made by SM. The instruction is issued, and the result is returned to the SM's local register. The interleaved instruction and other arithmetic or memory I/O instructions can be concurrent. Because it is an ASIC-specific circuit logic, $performance/mm^2$ can be increased by an order of magnitude compared to the use of shader code[1]. Besides the mainstream desktop and workstation GPUs like NVIDIAs RTX 40 series and AMDs RX 7000, data center GPUs like NVIDIA A40 and T4 also support ray tracing. Specifically, the NVIDIA RTX 4090 GPU integrates 128 RT cores. OptiX programming model [29] is an application framework for building ray tracing jobs. In OptiX, each ray is mapped to a CUDA thread. CUDA threads generate rays with the specified ray origins and directions. Then, the control is transferred to RT cores, which accelerate BVH traversal and ray-triangle intersection tests. For NVIDIA GPUs, triangles are the built-in primitives, which allows RT cores to accelerate intersection tests when the primitives are triangles. When other types of primitives are used, control is transferred to CUDA threads to perform intersection tests defined in the Intersection Shader. Overall, OptiX leverages both CUDA threads and RT cores to achieve efficient ray tracing jobs. RT cores specialize in accelerating BVH traversal/ray-triangle intersections, while CUDA threads provide flexibility for custom intersection logic through the Intersection Shader. This enables extensibility for complex ray tracing scenarios.

2.3 Expedite Data Processing with RT Cores

RT cores have been utilized to accelerate various data processing tasks like K-nearest neighbor search [25, 38] and range minimum

[1] https://www.reddit.com/r/nvidia/comments/97qogl/former_nvidia_gpu_architect_what_is_rt_core_in (last accessed 2025/10/6)

queries [22]. RTScan [21] and RTIndex [14] are pioneering implementations that leverage RT cores to accelerate the Scan operator. Specifically, RTScan [21] achieves significant performance improvement by mapping the evaluation of entire conjunctive predicates into a ray tracing process. Experiments show that RTScan achieves up to 4.6× higher performance than BinDex [20], which is the state-of-the-art scan approach on CPUs. Figure 4 demonstrates the approach of RTScan when evaluating a query with three predicates. For each data record, the three data attributes involved in the predicates are used as the coordinates of the corresponding primitive. Assuming that the conjunctive predicates are ($0 \leq x \leq a \wedge 0 \leq y \leq b \wedge 0 \leq z \leq c$), then the query area is a cuboid with the origin as a vertex and three edges of length a, b, and c as shown in the figure. The data records satisfying the conjunctive predicates are all in the query area. To reduce the computation overheads on intersection tests, RTScan adopts Data Sieving, which uses pre-stored results to filter most data records (the dashed area). Then, RTScan launches rays in the remaining region to intersect primitives with enhanced parallelism. These techniques add up to form the performance improvement over CUDA cores and CPUs. In addition, RTScan adopts Uniform Encoding to evenly distribute data records in space, enabling it to achieve good performance even under skewed data distributions.

After analyzing RTScan and other RT-based implementations, we summarize three key aspects of efficiently mapping a data processing task to a high-performance RT job. 1) *Evaluating multiple operators in one job:* RTScan utilizes the 3D feature to evaluate three predicates simultaneously within one RT job, whose execution time is even lower than that of evaluating one predicate. 2) *Reducing the amount of data accesses:* The evaluation of conjunctive predicates and the Data Sieving technique dramatically shrink the querying region, leading to a significantly smaller number of primitives for intersections. 3) *Enhancing the parallelism:* RTScan segments a long ray into several small rays with spacing, which aims at maximizing the utilization of RT cores while balancing their load.

2.4 Challenges of Accelerating Query Processing with RT Cores

For GPU acceleration with CUDA cores, operators are generally implemented as separate CUDA kernels and executed sequentially. RTScan demonstrates a standalone implementation of accelerating Scan with RT cores. However, accelerating database queries with each operator implemented as a separate RT job faces several critical issues that are hard to address.

Difficulties in the three-dimensional mapping of operators: In order to improve performance, a database operator needs to be effectively converted into a ray tracing job, or the performance advantage of RT hardware cannot be effectively exploited. For instance, a naive scan implementation on RT cores can be 2.3×10^4 times slower than the state-of-the-art CPU-based implementation [21]. However, the natures of some database operators make it hard to map the data in the 3D space and hard to convert operations into ray intersections. Therefore, except Scan, other operators like Join and GroupBy have not demonstrated superior performance on RT cores so far.

Inefficiency of the execution of multiple operators: A ray tracing job uses a BVH as its index, but building a BVH is a time-consuming process that takes orders of magnitude longer time than the RT job itself. For a dataset with 120 million tuples, the average time to build a BVH is 227.84 ms, while the average time to launch rays is only 0.75 ms. When the operators of a query are executed sequentially on RT cores, the BVHs for the second to the last operator can only be built online, resulting in an ultra-high query processing latency.

```
A: SELECT AVG(Math)
   FROM Score
   WHERE English >= 60
   GROUP BY Hometown;
```

We take the query A as an example, which executes the operators in the following order: Scan → GroupBy → Aggregation. For Scan, the BVH it traverses can be pre-built from the English column directly. However, for GroupBy, the BVH it traversed is built from the Hometown column filtered by the execution results of Scan. Therefore, it can only start building the BVH after the execution of Scan is complete and the results are obtained. The above process is shown in Figure 5a. As a result, the BVH building process for GroupBy has to be taken as part of the query execution, which results in ultra-low performance. An alternative scheme is shown in Figure 5b, where the BVHs for all operators have been pre-built from their corresponding attributes. In this case, GroupBy cannot get the filtered results from Scan and has to group all data in the column. Moreover, the results of GroupBy have to be further filtered according to the results of Scan. Consequently, this scheme completely negates the performance benefit of RT cores and is severely inefficient.

To conclude, due to the aforementioned issues, accelerating database queries with RT cores to outperform CUDA cores and CPUs is particularly challenging.

3 THE DESIGN OF RAYDB

3.1 Overview of RayDB

We propose RayDB, a high-performance query engine accelerated by RT cores. Specifically, RayDB leverages RT cores to accelerate three core operatorsAggregation, GroupBy, and Scanwhile delegating the remaining parts of the query to CUDA cores. Instead of implementing each operator as a separate RT job, RayDB maps the execution of these three operators to one ray tracing job. The basic idea is to use the attributes involved in these three operators as the coordinates for a data record, i.e., the attributes in Aggregation, GroupBy, and Scan are used as the X-coordinate, Y-coordinate, and Z-coordinate, respectively.

The workflow of RayDB is illustrated in Figure 7. As denormalization is widely adopted in data warehouses and favored by many technology companies [1, 2, 11], RayDB is designed to accelerate data warehouse queries on denormalized tables. During the offline phase, it performs denormalization in initialization by joining all relevant tables into a single flat table, thereby eliminating the need for Join during query execution. Next, a set of BVHs is built offline based on different combinations of attributes in the wide table. For a given query, RayDB parses the attribute composition of its three core operators and selects a pre-built BVH from the BVH

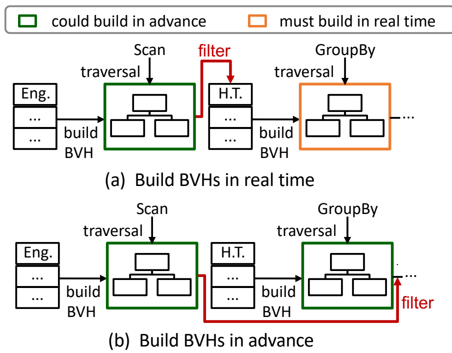


Figure 5: Two strategies of query execution with RT cores

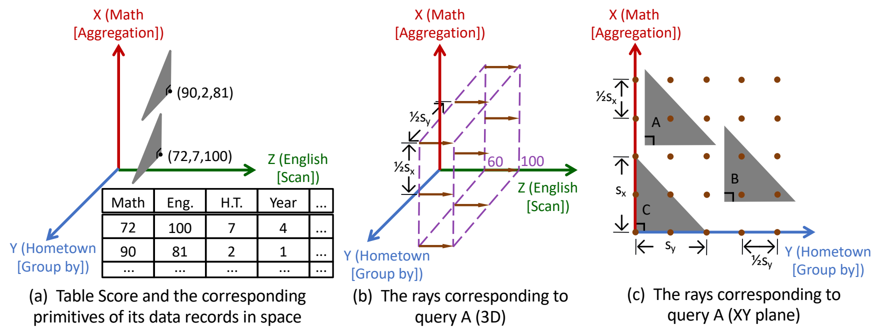


Figure 6: The design of RayDB

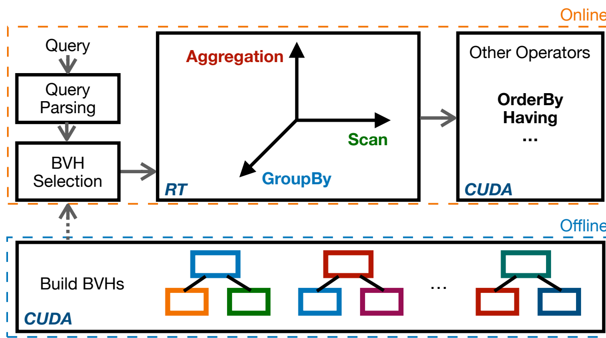


Figure 7: The workflow of RayDB

set that covers all the required attributes. It is worth noting that when a query involves excessive number of attributes such that no BVH can fully cover all of them, RayDB can utilize a BVH that includes only part of the required attributes to enable the query to still benefit from the acceleration provided by RT cores (introduced in Section 4.5). With the selected BVH, the RT cores are responsible for accelerating the execution of `Aggregation`, `GroupBy`, and `Scan`, which are fused into a single RT job. The design is based on the observation that these three operators appear consecutively in the query plan after denormalization. It is worth noting that this approach offers sufficient flexibility to remain effective even when certain core operators are absent from the query. For example, in queries without `GroupBy`, all records can be treated as belonging to the same group. RayDB determines the corresponding query area in the space based on `Scan` and launches a set of rays to intersect all primitives in the area. By only accessing data records in the query area defined by `Scan`, there is a significant reduction in the amount of data accessed for a query. Based on the coordinates of an intersected primitive, RayDB can obtain its data attributes for `GroupBy` and `Aggregation` and then perform the corresponding operations. For each data record, the data attributes involved in the three operators are stored together as coordinates, which can be retrieved by a single memory access. It dramatically reduces the number of random accesses to the device memory. Once the RT cores complete execution, the remaining operatorssuch as `OrderBy` and `Having`are handled by CUDA cores in a manner similar to existing CUDA-based database systems.

Instead of building a BVH for every query, RayDB maintains a few BVHs to support a wide range of queries. The encoding scheme allows multiple attributes to be compressed into a single coordinate (introduced in Section 4), where a query may use one or an arbitrary combination of the encoded attributes, enabling each BVH to support more queries. Furthermore, based on our observation of real-world queries, we find that certain columns tend to be used with specific types of operators. For example, in TPC-H, `l_shipdate` is predominantly used in `Scan`, while `l_extendedprice` is typically used in `Aggregation`. By pre-constructing BVHs for semantically meaningful combinations of attributes, RayDB is able to cover common OLAP queries using a few BVHs. For queries with subqueries, RayDB can accelerate them if the optimizer rewrites them into equivalent queries without subqueries through techniques like unnesting or decorrelation. If such rewriting isnt possible, RayDB currently does not support their execution.

3.2 The Mechanism of RayDB: An Example

In this subsection, we demonstrate how a BVH is constructed and how an RT job is executed. Suppose there is a denormalized table named *Score*, as shown in Figure 6a, that stores information about students' scores. Each row of the table corresponds to a student, and the table has many attributes, among which the *Math*, *English*, *Hometown*, and *Year* are used to store students' math scores, English scores, hometowns, and academic years, respectively. The course scores are in the range $[0, 100]$ and are integers. For each column, there must exist a data range that can be known in advance (e.g., maintained as metadata). We use the execution of query A (Section 2.4) as an example. The query obtains the average math score of students in each hometown who have passed English.

Building the BVH: First, it should be clarified that the BVH constructed in this example is the most suitable one from the pre-built BVH set for executing Query A. The key step in building a BVH lies in mapping data records to primitives. Based on the idea of representing data attributes involved in `Aggregation`, `GroupBy`, and `Scan` by the coordinates in three-dimensional space, RayDB makes the X-axis, Y-axis, and Z-axis represent the data attributes involved in `Aggregation`, `GroupBy`, and `Scan`, respectively. In this way, each

data record in the table corresponds to a primitive in space. For example, row 0 of the table corresponds to $(72, 7, 100)$ in space. Then, using the point as its vertex, RayDB creates a right triangle as the primitive. The reason why we choose triangles as primitives is that only the ray-triangle intersection test is hardware-supported by RT cores, while the intersection tests for other types of primitives are software-based and offloaded to CUDA cores. Therefore, the use of triangles allows RayDB to enhance performance by exploiting hardware acceleration from RT cores. Specifically, if the coordinate of a data record is (a, b, c), then the three vertex coordinates of the right triangle we create are (a, b, c), $(a + S_x, b, c)$, and $(a, b + S_y, c)$, where S_x and S_y are the two leg lengths of the right triangle. Therefore, the projections of a primitive on the XZ-plane and YZ-plane are a line segment S_x and a line segment S_y, respectively. In this case, each data record in the table is mapped to a triangle in three-dimensional space, as shown in Figure 6a. Once all primitives in the space are determined, the BVH can be built.

Executing the RT job: With the selected BVH, RayDB initiates the execution of the RT job by launching a set of rays. For query A, RayDB launches a set of parallel rays starting from the $Z = 60$ plane to the $Z = 100$ plane, along the positive direction of the Z-axis, as shown in Figure 6b. The rays launched should be dense enough to intersect all triangles in the region. Rays are launched as a two-dimensional array from the view of the XY-plane ($Z = 60$ plane), which have an interval of $\frac{1}{2}S_x$ along the X-axis and an interval of $\frac{1}{2}S_y$ along the Y-axis. Recall that when mapping data to primitives, the two legs of right triangles have lengths S_x and S_y, respectively. The design guarantees that a triangle can be intersected by at least one ray. As shown in Figure 6c, primitives may intersect one ray (triangle A) or three rays (triangle B), and in the limiting case primitives intersect at most three rays (triangle C)[2]. If the interval grows, there may be triangles that fail to intersect any ray. In turn, if the interval gets smaller, it increases the probability that a triangle is intersected by more than one ray, which degrades the performance. Rays entirely cover the query area $60 \leq Z \leq 100$, and triangles in the query area are bound to intersect rays, while triangles not in the query area are bound not to intersect any ray.

For students who pass the English examination, their triangles are in the query area $60 \leq Z \leq 100$. Thus, the set of triangles intersecting a ray is the set of students that satisfy the predicate of `Scan`. For each triangle that is intersected by a ray, the Y-coordinate of its right-angle vertex is used to find the group to which it belongs, while the X-coordinate is read to compute the aggregate function, respectively. The aggregate function is *AVG* in Query A, so we maintain two arrays in the Any Hit Shader function, whose pseudo-code is shown in Algorithm 1. The *sum* array is to store the sum of *Math* for all students in each group, and the *count* array is to store the number of students in each group. Indexing by the Y-axis coordinate, RayDB appends the X-coordinate to the corresponding element of the *sum* array and increments the corresponding element of the *count* array by 1. The Any Hit Shader, which is executed on the SM as part of the shader pipeline, will be called each time a ray finds an intersection with a triangle. Therefore, the flag bit array V_{flag} (line 3) is used to ensure that triangles are not double-counted, and the atomic operation *atomic_add* (lines 5 and 6) is used to avoid synchronization issues when tracing multiple rays in parallel. After the BVH traversal is complete, the sum of the *Math* (V_{sum}) is divided by the number of students (V_{count}) to obtain the average score of *Math* in each group. Since aggregate functions share the common characteristic of operating on a group of values to return a single scalar result, their implementations are generally similar. As a result, RayDB supports all commonly used aggregate functions, including `AVG`, `SUM`, `COUNT`, `MAX`, and `MIN`.

[2]https://forums.developer.nvidia.com/t/what-is-the-limiting-case-of-ray-triangle-intersection/309730/2 (last accessed 2025/10/6)

Algorithm 1 Pseudo-code of Any Hit Shader

Input: flag bit array V_{flag}, result arrays V_{sum}, V_{count}
Output: result arrays V_{sum}, V_{count}

1: primIdx ← get_prim_index()
2: $[a, b, c]$ ← get_prim_right_vertex_coord(primIdx)
3: flag ← atomic_bit_exch($V_{flag}[primIdx]$, 1)
4: **if** flag = 0 **then**
5: atomic_add($V_{sum}[b]$, a)
6: atomic_add($V_{count}[b]$, 1)
7: **end if**

4 ENCODING OF MULTIPLE DATA ATTRIBUTES

4.1 Challenges from Processing Multiple Attributes

Query A is a simple query where each operator involves only one data attribute. However, in real-world queries, it is common for an operator to involve multiple data attributes. When mapping Query A in the space, a primitive directly uses the value in the corresponding attribute as the coordinate on an axis. For instance, *English* is used as the coordinate on the Z-axis. However, when multiple data attributes are evaluated by one operator, the coordinate on one dimension needs to represent all data attributes involved. For instance, with the Where clause `WHERE English ≥ 60 and Math ≥ 60`, both English score and Math score should be represented by the Z-axis. To address this issue, in RayDB, we propose to encode multiple data attributes as the coordinate on each axis. In addition, the encoding scheme brings an extra benefit: it compresses multiple attributes into a single coordinate, allowing a query to utilize one or an arbitrary combination of the encoded attributes. This enables a single BVH to support a wider range of queries, thereby reducing the number of pre-built BVHs required. However, since different operators have different functionalities, an appropriate encoding scheme needs to be chosen for each one. In this section, we study how to encode attributes for `Scan`, `GroupBy`, and `Aggregation`.

4.2 Scan with Conjunctive Predicates

The encoding scheme for `Scan` needs to maintain the relative order of the encoded data and specify the ray launching area to ensure correct execution. A query generally contains multiple conjunctive predicates, like $p_1 \wedge p_2 \wedge \cdots \wedge p_n$, and the attributes involved in the predicates can be encoded in the same attribute.

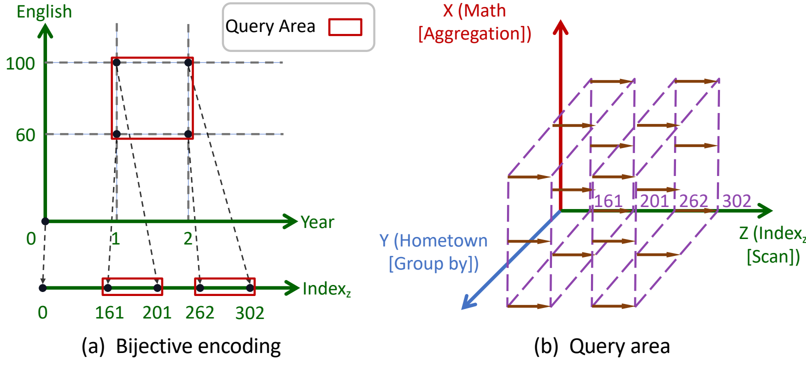


Figure 8: Processing for conjunctive predicates

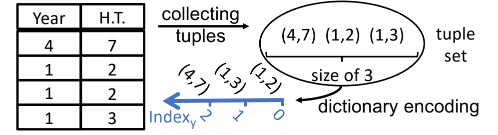


Figure 9: Encoding for GroupBy

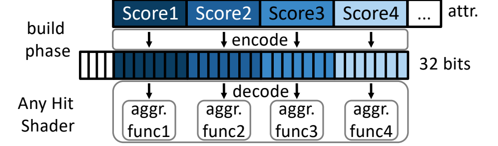


Figure 10: Encoding for Aggregation

Data Encoding: RayDB adopts Bijection Encoding to encode multiple attributes in Scan as a coordinate. It treats all data attributes involved in conjunctive predicates as n-tuples, where each n-tuple is uniquely mapped to a natural number, and each natural number uniquely corresponds to an n-tuple. Assume that conjunctive predicates involves n attributes $A_1, A_2, \cdots, A_n$, where $A_i \in [0, k_i)$ and $A_i \in \mathbb{Z}$ $(i \in [0, n])$, then the encoding rule is as follows:

$$Index_z = \sum_{i=1}^{n} \left[\left(\prod_{j=i+1}^{n} k_j \right) A_i \right]$$

If the data type of an attribute is not INTEGER, for fixed-point data types like DECIMAL(p, s) & NUMERIC(p, s), where p is the total number of digits and s is the number of fractional digits, RayDB uses Fixed-Point Encoding, which multiplies the decimal value by a fixed scaling factortypically $10^s$to convert it into an integer; for data types such as DATE, TIME, CHAR, VARCHAR, and ENUM, RayDB adopts dictionary encoding, where each distinct value is placed into a sorted dictionary and assigned an integer identifier representing its position in the sorted set of values. Therefore, it supports a wide range of data types. If the data range of an attribute is not of the form $[0, k)$, a dictionary encoding is used to convert it to the above form. Taking the query B as an example:

```
B: SELECT AVG(Math)
   FROM Score
   WHERE Year between 1 and 2 AND English >= 60
   GROUP BY Hometown;
```

For simplicity, it is assumed that the attributes involved in Scan are already encoded. Query B contains conjunctive predicates as its filtering conditions, which not only require students to have a passing score in English but also require students to be from either the first or second grade. RayDB assigns each $(Year, English)$ tuple with a natural number by the following rule:

$$Index_z = 101 \cdot Year + English$$

Here, 101 represents the range of *English*. Figure 8a depicts the above process.

Ray Launching: After Bijective Encoding, the Z-axis represents $Index_z$ generated by Bijective Encoding as an alternative. As a result, the query area changes as well. In the example, the query area is split into two parts, $161 \leq Index_z \leq 201$ and $262 \leq Index_z \leq 302$, as shown in Figure 8a. Correspondingly, as shown in Figure 8b, RayDB launches parallel rays from the $Z = 161$ plane to the $Z = 201$ plane and from the $Z = 262$ plane to the $Z = 302$ plane along the positive direction of the Z-axis to cover the entire query area.

4.3 GroupBy with Multiple Attributes

For data attributes involved in GroupBy, RayDB treats the n data attributes as n-tuples, where each distinct n-tuple represents a different group. Take query C as an example, which groups students according to two data attributes *Year* and *Hometown*.

```
C: SELECT AVG(Math)
   FROM Score
   WHERE English >= 60
   GROUP BY Year, Hometown;
```

For all distinct n-tuples appearing in the table, RayDB maps them to natural numbers $Index_y$ via dictionary encoding. The correspondence between $Index_y$ and n-tuples is maintained by a mapping table that allows efficient lookup. After dictionary encoding, each natural number in $Index_y$ represents a group, which serves as the Y-coordinate. Taking query C as an example, table *Score* is searched to find all distinct $(Year, Hometown)$ tuples. Figure 9 illustrates the above process. For simplicity of presentation, there are only 4 data records in the table. In the table, there are three different $(Year, Hometown)$ tuples $(4, 7)$, $(1, 2)$, and $(1, 3)$, making up a group set of size $K = 3$. Then, RayDB performs dictionary encoding to the group set, producing a coordinate $Index_y$ ranging from 0 to $K - 1$. As shown in the figure, $(1, 2)$ is assigned to 0, $(1, 3)$ is assigned to 1, and $(4, 7)$ is assigned to 2. This encoding scheme facilitates the implementation of aggregation because the Y-coordinate can be directly used as the index to the array that stores aggregate results for each group.

4.4 Aggregation with Multiple Attributes

For Aggregation, we choose to pack the bits of the attribute values in a 32-bit float coordinate so that they can be directly used for the aggregate function. For example, assume that there is a query containing two aggregate functions AVG(Math) and SUM(English). Considering that the data range of course scores is $[0, 100]$, we store *Math* and *English* with 7 bits each in the X-coordinate. In

Any Hit Shader, we obtain *Math* and *English* by decoding the X-coordinate and computing the two aggregate functions separately. Given that the coordinates have 32 bits and 7 bits are sufficient to store a single score, the X-coordinate can hold up to 4 scores. Therefore, the encoding can support queries with up to 4 aggregate functions, as shown in Figure 10.

For cases where an aggregate function contains multiple data attributes, e.g., `SUM(Math + English)`, the calculated result *Math + English* is directly encoded in the coordinate if the result does not exceed the expression range of a float. Multiple queries in SSB have such forms of aggregations. This scheme may help make further compressions. In this example, the range of *Math + English* is $[0, 200]$, which only needs 8 bits instead of 14 bits when being separately stored. RayDB adopts this optimization when possible to store more attributes in a coordinate.

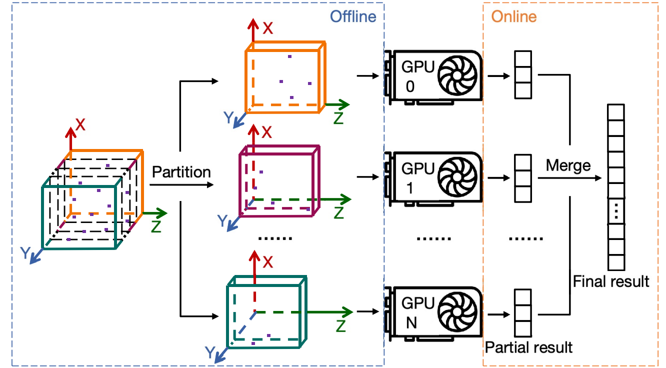


Figure 11: The workflow of parallel execution on multiple GPUs

4.5 Breaking through the Encoding Limitation

RayDB converts all data types to `INTEGER` for encoding, while BVH coordinates can only be represented using `Float`, which can accurately represent integers only within the range of $[-2^{24}, 2^{24}]$. Under this limitation, the maximum number of attributes that can be encoded into a single coordinate depends on the value range of each attribute involved in the encoding. Taking attributes from the SSB as an example, RayDB can encode up to four `LO_QUANTITY` (with a value range of 150) or six `LO_DISCOUNT` (with a value range of 010) into a single coordinate.

Although a coordinate can only encode a limited number of attributes, RayDB can support queries with more attributes. From a high-level perspective, when a BVH that covers all attributes in the query is unavailable in the pre-constructed BVH set, RayDB can utilize a BVH that encodes only part of the attributes to enable the query to still benefit from the acceleration provided by the RT cores. RayDB performs the RT job on the BVH, and upon detecting a ray-primitive intersection, the Any-Hit Shader that is called uses the primitive ID (i.e., row ID) to access the remaining uncoded attributes from the denormalized table in the GPU memory and perform the corresponding operations. Since reading attributes from the GPU memory incurs extra memory accesses, RayDB selects the optimal BVH by maximizing the attribute coveragei.e., choosing the BVH that encodes the largest number of attributes involved in the query. In particular, attributes in `Scan` are given higher weight, as the presence of `Scan` can significantly reduce memory access overhead. This selection process is formalized as $\max\{\, x + y + \alpha \cdot z \,\}$, where x, y, and z represent the number of the matched attributes in `Aggregation`, `GroupBy`, and `Scan`, respectively, and α is the weight assigned to `Scan`.

5 SCALING OUT WITH MULTIPLE GPUS

In RayDB, we store multiple BVHs in the device memory to support various queries. When BVHs exceed the device memory size of a single GPU, RayDB can only discard some BVHs. This results in more queries failing to find a BVH that encodes all of their attributes and instead choosing a BVH that covers only a part of the attributes, as described in Section 4.5. This incurs additional memory accesses, leading to performance degradation.

To cope with this dilemma, we design a parallel execution scheme for RayDB on multiple GPUs, which can not only utilize the device memory of multiple GPUs to accommodate larger data sizes but also utilize the RT cores of multiple GPUs to improve query performance. The key idea is that the 3D space in RayDB can be naturally partitioned into any number of independent regions for parallel execution. First, all primitives can be divided into several sets according to the X-axis (`Aggregation`), Y-axis (`GroupBy`), or Z-axis (`Scan`) coordinates. After that, each set can be assigned to a GPU and used to build a specific BVH, which can be viewed as a sub-BVH. In this way, each GPU is supposed to intersect primitives to get partial results for the query. After all GPUs have completed processing, the final query result can be obtained by merging the results from all GPUs.

Figure 11 shows the parallel execution workflow when partitioning the data space along the Y-axis (`GroupBy`). We evenly divide the range of Y-axis coordinates into N segments, and the primitives located in the same segment are divided into the same group and assigned to the same GPU to build the BVH. Spatially, the entire data space is partitioned into a number of equal-sized subspaces along the Y-axis, and each subspace is assigned to a GPU, and each GPU launches rays in this subspace according to the same principles as a single GPU. Since we are partitioning the space along the Y-axis, the result obtained by each GPU is the query result for the groups involved within the corresponding segment. The final merge process is as simple as concatenating the results of each GPU in order to obtain the final query result. Since each primitive belongs to only one subspace, it is neither repeatedly accessed by multiple GPUs nor omitted, which ensures the correctness of the parallel execution result. In this way, all GPUs can be executed in parallel without any dependency. This approach can not only solve the problem of insufficient device memory but also accelerate the execution of queries.

The obvious benefit of partitioning the data space along the Y-axis is the simplification of merging. If we partition the space along the X-axis (`Aggregation`) according to a similar principle and the aggregate function is `SUM` as an example, the result obtained by each GPU is not the sum of some groups but the partial sum of all groups in the subspace. During the merge process, we need to sum

the result arrays of all GPUs together to get the query result, which is more time-consuming. In addition, if the data space is partitioned along the Z-axis (`Scan`), the query region may only be in a small number of subspaces, which will cause a serious load imbalance. In summary, we believe that partitioning along the Y-axis is a good choice for most cases.

The denormalized table can also be partitionedsplit by attributes and distributed across multiple GPUs. Unlike BVHs, each GPU may need to access attributes stored on other GPUs during query execution. With technologies like NVLink and RDMA enabling direct memory access between GPUs, RayDB can still support queries involving an arbitrary number of attributes in a multi-GPU parallel execution environment.

6 EXPERIMENTAL ANALYSIS

6.1 Experiment Setup

Hardware and Software We run most experiments on a machine equipped with two Intel(R) Xeon(R) Silver 4316 CPU @ 2.30GHz, 128GB DDR4 DRAM, and an NVIDIA GeForce RTX 4090 with 128 RT cores, 16384 CUDA cores, and 24GB VRAM. The operating system is 64-bit Ubuntu Server 20.04 with Linux Kernel 5.4.0-42-generic. The GPU programming interface uses CUDA 10.1 and OptiX 7.1.

In Section 6.3, we also compare the performance between RayDB and Crystal on NVIDIA TITAN X (PASCAL), where OptiX 5.1 is used for programming. The GPU was launched in 2015 and does not have RT cores. Therefore, ray tracing jobs are only executed on CUDA cores. The experiment aims to analyze the performance benefits from RT core acceleration. In Section 6.8, to evaluate the scalability of RayDB on multiple GPUs, the experiment was run on a machine with four NVIDIA GeForce RTX 4080 SUPER.

Workloads Throughout the experiments, we adopt the Star Schema Benchmark (SSB) [28]. There are a total of 13 queries in the benchmark, divided into 4 query flights. The industry usually flattens SSB into a wide table model (SSB flat) to test the performance of query engines. In our experiments, we run the SSB flat with a scale factor of 1, 10, and 20 to evaluate the performance with different data set sizes. When the scale factor equals 20, it will generate a flat table with 120 million tuples.

Baseline We compare RayDB with Crystal [32]. Crystal is a recently proposed state-of-the-art GPU database system that delivers superior query execution performance compared to other systems. It currently supports only queries from the Star Schema Benchmark (SSB). Specifically, Crystal implements a custom operator-invocation program for each query in the SSB. For a fair comparison, we modify these programs by removing all `Join`. Besides, since Cyrstal does not implement the `OrderBy`, we also remove it from all queries.

Encoding In SSB, we adopt different encoding schemes for the `Aggregation` operator. Three queries in Flight Q4 have the aggregate function: `sum(lo_revenue - lo_supplycost)`, and we adopt the encoding optimization in Section 4.4 to make further compression. The aggregate function of queries in Flight Q1 is `sum(lo_extendedprice * lo_discount)`. However, the range of *lo_extendedprice* ∗ *lo_discount* is too large to be precisely represented as an integer by a 32-bit float. Therefore, in experiments, we use the approach in Section 4.5 to handle this situation. In evaluating Flight Q1, the X-coordinate only represents *lo_extendedprice* while *lo_discount* is stored in the GPU device memory.

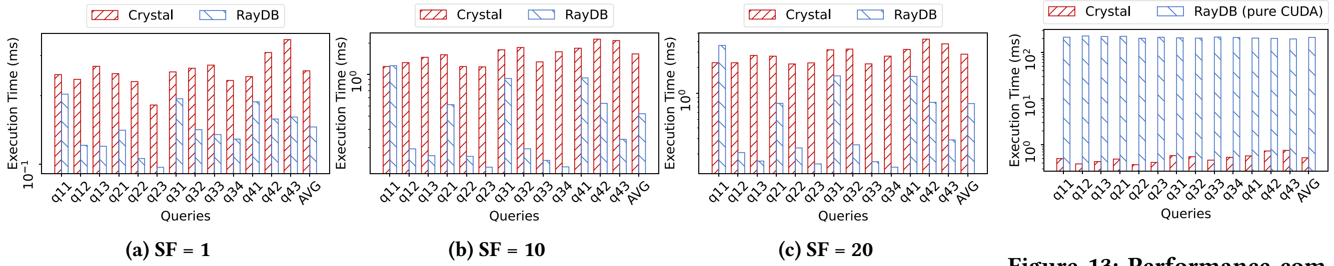
Table 1: Selectivity of queries in SSB

Query	q11	q12	q13		q21	q22	q23
Sel.(%)	1.99	0.07	0.02		0.80	0.16	0.02
Query	q31	q32	q33	q34	q41	q42	q43
Sel.(%)	3.67	0.14	5.76E-3	7.33E-5	1.59	0.38	0.04

6.2 Evaluation of Query Performance

Figure 12 illustrates the performance comparison between RayDB and Crystal. In order to ensure the fairness of the comparison between systems, the query execution time in the experiment is unified as starting after loading all input data into GPU memory and ending after the results are calculated. RayDB shows excellent performance on the SSB flat. At SF=1, RayDB is faster than Crystal on all queries, on average, by 82.08%. At SF=10, RayDB is faster than Crystal on 12 out of 13 queries and 5.4× faster on average. At SF=20, the situation is similar to that at SF=10. RayDB is faster than Crystal on 12 out of 13 queries, at least 1.0× faster and at most 18.3× faster. Over the entire SSB flat, RayDB is on average 8.5× faster than Crystal. It can be seen that RayDB maintains its performance advantage over Crystal in all SF cases. Considering that Crystal is by far the state-of-the-art GPU database system delivering superior query execution performance compared to other systems, the performance improvement is reasonably satisfactory.

Table 1 records the selectivity of each query in the SSB flat. Referring to Figure 12, it is observed that the query execution time of RayDB has a certain positive correlation with the selectivity of the query. The selectivity of q11, q31, and q41 is significantly higher than the other queries, and their execution time is also longer. The fundamental reason for the positive correlation between query execution time and query selectivity under certain conditions is that a lower selectivity implies a smaller number of data records in the query area, that is, a smaller amount of data to be accessed. Among the above queries, although q11 does not have the highest selectivity (1.99%, the maximum selectivity is 3.67% of q31), its execution time is particularly long, and it is even the only query for which RayDB has a longer execution time than Crystal. The reason is the particularity of Flight q1. The queries in Flight q1 do not include the `GroupBy` operator. In our implementation, we treat such queries as if all data records belong to the same unique group. Recall that atomic operations are used in Any Hit Shader to avoid synchronization issues, but they also limit parallelism, which affects performance. In the presence of only one group, all atomic operations target the same scalar value, further preventing the parallelism advantage of RT cores from being fully exploited and slowing down the execution of q11. This is also confirmed by the shift of the execution time of q11 from SF=1 to SF=20: when the dataset is small (SF=1), there are also fewer atomic operations, and the restriction of parallelism is not obvious. Therefore, RayDB is faster than Crystal. When the dataset becomes larger (SF=20), the



(a) SF = 1

(b) SF = 10

(c) SF = 20

Figure 12: Query execution time of RayDB and Crystal

Figure 13: Performance comparison of RayDB (pure CUDA) and Crystal

atomic operations increase accordingly, and the disadvantage of limited parallelism is further highlighted. At this time, RayDB is inferior to Crystal.

6.3 The Performance Gains from RT Cores and BVHs in RayDB

In order to disentangle the gains obtained from RT cores and BVHs, we implement a pure CUDA version of RayDB where the entire ray tracing process is computed by CUDA cores on the GPU. By comparing it with Crystal, we can evaluate the performance impact of the BVH index in the absence of RT cores, and then verify the contribution of RT cores. To achieve this, we switch to OptiX 5.1, which uses CUDA cores to compute BVH traversal and ray-triangle intersection tests. Since OptiX 5.1 is an old version, it does not support NVIDIA RTX 4090. Therefore, the experiments in this subsection are performed on an NVIDIA TITAN X (Pascal). Limited by the device memory size of TITAN X, we conduct experiments with a scale factor of 1. The performance results are shown in Figure 13.

On the 13 queries of SSB, RayDB (pure CUDA) is 258×-588× slower than Crystal, and 423× slower on average. Recall that RayDB is faster than Crystal on all queries with the same scale factor as in Section 6.2. This demonstrates that the BVH index alone does not work well due to the features of CUDA, and performance gains can only be achieved by cooperating with RT cores. With the application of hardware-accelerated ray tracing technology, the BVH traversal and ray-triangle intersection tests that originally needed to be computed by CUDA cores during the ray tracing process are offloaded to the RT core, which is specialized hardware designed for this purpose, freeing CUDA cores from thousands of instructions per ray, which could be an enormous amount of instructions for an entire ray tracing process. The presence of RT cores considerably accelerates the ray tracing process and makes the ray-tracing-based database possible.

6.4 GPU Memory Bandwidth Occupancy

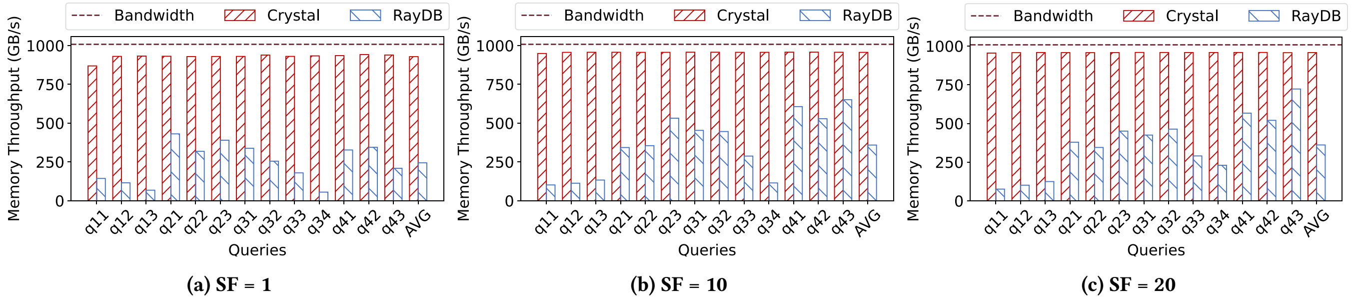
Figure 14 presents a comparison of the memory throughput of Crystal and RayDB on 13 queries of SSB flat. At SF=20, Crystal achieves a memory throughput of 97.11% to 97.51% of the memory bandwidth, with an average of 97.41%. The situation is rather similar in other SF cases. It can be argued that Crystal saturates the memory bandwidth. In contrast, at SF=20, RayDB's memory throughput is only 7.72% to 73.53% of the memory bandwidth, with an average of 36.74%. This ratio is even smaller in other SF cases. It can be seen that the memory bandwidth occupied by RayDB is considerably smaller than that of Crystal in all SF cases. This is due to the fact that RayDB drastically reduces the amount of data that needs to be accessed and the number of random memory accesses, freeing its performance from the memory bandwidth constraints. Based on this, we identify that the limiting factor for the performance of RayDB is the number of RT cores. The latest RTX 4090 is equipped with 128 RT cores versus 16384 CUDA cores. More RT cores mean that we can launch more rays to expedite intersection tests, and the performance is expected to be further enhanced. With the fast development of the ray tracing technique, the number of RT cores has also been increasing. The RTX 2080 Ti has only 68 RT cores, while this number grows to 84 on the RTX 3090 Ti and 128 on the RTX 4090. Therefore, since the memory bandwidth is no longer a limitation, RayDB is expected to benefit from the evolving architecture and achieve higher performance on future GPUs.

It is worth noting that there is no clear positive correlation between query memory throughput and query selectivity, which seems to contradict the conclusions we obtained. While RayDB reduces the amount of data that needs to be accessed with the help of ray tracing, it also increases the overhead of BVH traversal and ray-triangle intersection tests. Since the BVH is stored in the device memory, BVH traversal and ray-triangle intersection tests also require access to the memory, and this partial memory access is closely related to the BVH structure. The BVH structures corresponding to different queries show great differences, and their effect on the memory throughput shows strong stochasticity, ultimately shaping the results shown in Figure 14. Nevertheless, overall, the query memory throughput is still significantly degraded.

6.5 BVH Construction Overhead

Figure 15 shows the time of building the BVH in RayDB on each query of SSB flat for scale factors 1, 10, and 20. Experimental results show that the BVH building time scales linearly with the dataset size. From SF=1 to SF=10, the dataset size increases by about 9×, and the average building time increases by 9.95×. The dataset size roughly doubles from SF=10 to SF=20, while the average building time also grows by 1.01×. At a certain dataset size, the BVH building time is relatively stable, and the difference in building time between different queries does not exceed 11%.

In the three cases of SF=1, SF=10, and SF=20, the average BVH building time is 10.30 ms, 112.85 ms, and 227.84 ms, respectively,



(a) SF = 1 **(b) SF = 10** **(c) SF = 20**

Figure 14: Query memory throughput of RayDB and Crystal

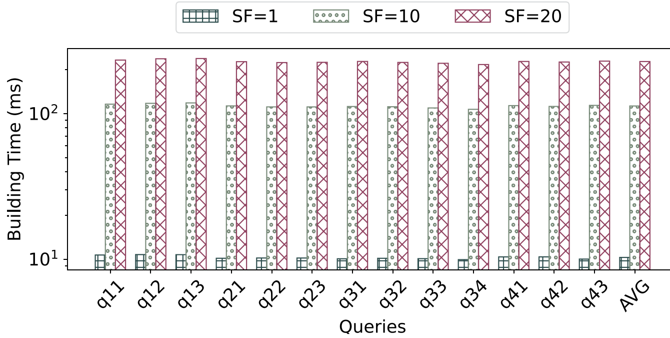


Figure 15: BVH building time of RayDB

while the average query execution time is 0.26 ms, 1.58 ms, and 2.85 ms, respectively. Therefore, the average execution time is 2.50%, 1.40%, and 1.25% of the average BVH building time, respectively. Due to the huge gap between the building time and the execution time, it is imperative to construct the BVH in advance. In RayDB, we build BVHs offline, which can be viewed as indexes or materialized views. For a given query, RayDB chooses a BVH that covers the attributes in the three operators.

6.6 Comparison between Query-Level and Operator-Level Ray Tracing

In Section 2.4, we explain why we choose to use ray tracing to accelerate the entire query rather than each operator. In the following, we conduct experiments to verify our conclusions. In view of the fact that there have been few studies on accelerating a single operator using ray tracing, which mainly focuses on the Scan operator (e.g., RTScan), we implement the Scan operator of the query by RTScan and the other operators by RayDB. RTScan transmits its execution result to RayDB in the form of a bit vector, where each bit records whether the corresponding data record satisfies predicates. RayDB enforces the Z-coordinate to be 0 for all primitives. Accordingly, rays are launched at the $Z = 0$ plane as the query area. In this way, all primitives will be intersected by rays, and RayDB needs to use the bit vector to determine whether a data record satisfies the predicates and should continue to participate in the following computation. Since RTScan suffers from out-of-memory issues when SF=20, we down-regulate SF to 16 in this experiment. The performance comparison with the case where RayDB implements the entire query is shown in Figure 16. The performance of ray tracing for individual operators exhibits a noticeable decrease when compared to ray tracing for the entire query. On the 13 queries of SSB flat, ray tracing for the entire query is 13.3× - 555.1× faster than ray tracing for individual operators and 194.9× faster on average. The reason for this staggering performance gap is that ray tracing is used to accelerate Scan and other operators separately, which means that RayDB (disable Scan) requires access to the entire dataset, taking away the benefit of ray tracing. Overall, this result is sufficient to demonstrate the inefficiency of ray tracing for individual operators and the necessity of choosing ray tracing for the entire query.

6.7 Comparison between Encoding and Splitting

In Section 4, faced with the case where a single Scan and GroupBy operator involves multiple data attributes, we give a solution that is encoding. However, there is a more intuitive way, which is to split the data attributes involved in the operator into two parts. The first part contains only one data attribute, which is still used for RayDB. The data attributes in the other part are stored in device memory in the form of arrays that can be accessed according to primitive indices. We refer to this approach as splitting.

Figure 17 is a plot of the performance comparison between splitting and encoding on the Scan operator when SF=20. In the experiments, splitting on Scan uniformly selects the first data attribute involved in the Scan operator for RayDB and stores the remaining data attributes in the device memory. When RT cores detect a primitive that intersects a ray, the corresponding Any Hit Shader reads from the device memory the remaining data attributes of the corresponding primitive and determines whether they satisfy the conjunctive predicates contained in Scan. Experimental results show that encoding on Scan performs significantly better than splitting on Scan. on the 13 queries of SSB flat, encoding on Scan is 3.3× to 176.9× faster than splitting on Scan, with an average of 31.1×. The reason for such a large performance difference is the difference in the amount of data that needs to be accessed. For example, if the conjunctive predicates are $p_1 \wedge p_2$, p_1 has a selectivity of 50% and p_2 has a selectivity of 20%, then encoding on Scan will access 50%×20%=10% of the data records, and splitting on Scan will access 50% of the data records. In this case, the amount of data they need to access differs by a factor of 5. Due to the different selectivity of each predicate in the conjunctive predicates, the performance difference between encoding on Scan and splitting on Scan on each

Figure 16: Performance comparison between RayDB and RayDB+RTScan

Figure 17: Performance improvement with encoding on Scan

Figure 18: Performance improvement with encoding on GroupBy

Figure 19: Scalability on multiple GPUs

query of SSB flat also shows a large difference. Overall, however, encoding on `Scan` outperforms splitting on `Scan` by a wide margin.

Figure 18 illustrates the performance comparison of splitting and encoding on the `GroupBy` operator when SF=20. Splitting on `GroupBy` is analogous to splitting on `Scan`. Since the queries in Flight q1 do not involve the `GroupBy` operator, the experiment is performed only on the remaining 10 queries of SSB flat. It can be observed that the difference in performance between splitting on `GroupBy` and encoding on `GroupBy` is smaller, 74.61% on average, compared to the difference between splitting on `Scan` and encoding on `Scan`, which is 31.1× on average. This is because splitting on `GroupBy` does not change the amount of data records accessed compared to encoding on `GroupBy` but only increases the overhead of accessing the remaining data attributes stored in the device memory. However, only the remaining data attributes of primitives that intersect rays will be accessed, so this overhead is not apparent in most SSB queries with generally low selectivity. Regardless, given the slight performance benefit, encoding on `GroupBy` remains the better option.

6.8 Scalability on Multiple GPUs

We evaluate the performance of RayDB on a single RTX 4080 SUPER and four RTX 4080 SUPER with a scale factor (SF) of 20, and the experimental result is shown in Figure 19. In this section, we only adopt the parallel execution strategy where the data space is equally partitioned along the Y-axis as an example of parallel execution. For the same reasons as in Section 6.7, we exclude Flight q1 from our experiments. The experimental result indicates that the performance of four GPUs working in parallel is 2.1× than that of a single GPU on average and can reach 3.3× at most.

While parallel execution on multiple GPUs significantly enhances performance, further improvements are constrained by the following two main factors: Firstly, the preparatory work before launching rays (e.g., determining the query region) needs to be performed on each GPU and cannot be parallelized. The time spent in this part has small differences between queries and is relatively fixed, approximately 0.07 ms in this experiment. As a result, for queries with long execution times, the performance improvement is large because the time taken for preparatory work is a small proportion of the execution time and vice versa. Secondly, the simple strategy of partitioning the space equally leads to a certain load imbalance between each GPU, which ultimately affects the performance. It is our future study to explore a more reasonable way of partitioning to achieve load balancing among GPUs.

7 RELATED WORK

Applications of RT cores Since RT cores are originally designed to render physically correct reflections, refractions, shadows, and indirect lighting in real time [27], there are numerous studies that leverage it to accelerate rendering workloads, including graphic rendering [8, 16], ambient occlusion [7], and simulation in physics [10, 37]. In addition, there are also many studies that creatively use RT cores to accelerate non-rendering workloads such as data processing, including K-nearest neighbor search [25, 38], database scan [14, 21], range minimum queries [22], point location search [24, 34], and rendering of unstructured meshes [23]. In summary, RT cores demonstrate significant potential for accelerating both rendering and non-rendering tasks.

Accelerating queries with GPUs GPUs offer strong parallelism and high-bandwidth memory, making them an attractive candidate for accelerating database queries. Therefore, GPU-accelerated database systems have been widely and deeply studied for more than 10 years [9, 17–19, 30, 35, 36]. There are three types of computational cores on the GPU: CUDA cores, Tensor cores, and RT cores. CUDA cores are responsible for integer and floating-point operations, so most GPU database systems are based on CUDA cores. Among them, Crystal [32] is the state-of-the-art CUDA-based GPU database that provides superior query execution performance compared to other systems. Tensor cores provide significant speedups to matrix operations. To leverage the computational power of Tensor cores, TCUDB [15] maps query operators to efficient matrix operators and implements a Tensor-based GPU database. RT cores accelerate Bounding Volume Hierarchy (BVH) traversal and ray-triangle intersection tests in ray tracing, and RayDB is the first study to make use of them to accelerate database queries, filling the gap in related directions.

8 CONCLUSION

In this paper, we propose RayDB, a query engine that accelerates database queries with ray tracing cores. Instead of implementing each operator independently like CUDA-based implementations, RayDB maps the core operators in a query as one RT job. The approach brings several performance advantages, including accessing data with an optimized sequential access pattern, reducing the amount of data to be accessed, and exploiting the parallelism of RT cores. RayDB breaks the memory bandwidth restriction on query performance and significantly outperforms the state-of-the-art CUDA-based GPU and CPU query engines.

REFERENCES

[1] Analytics at Meta. 2022. Using LogTime Denormalization for Data Wrangling at Meta. https://medium.com/meta-analytics/using-log-time-denormalization-for-data-wrangling-at-meta-3b6fc050268a.

[2] Burak Bacioglu and Meenakshi Jindal. 2021. Elasticsearch Indexing Strategy in Asset Management Platform (AMP). https://netflixtechblog.com/elasticsearch-indexing-strategy-in-asset-management-platform-amp-99332231e541

[3] Piotr Bialas and Adam Strzelecki. 2016. Benchmarking the cost of thread divergence in CUDA. In *Parallel Processing and Applied Mathematics: 11th International Conference, PPAM 2015, Krakow, Poland, September 6-9, 2015. Revised Selected Papers, Part I 11*. Springer, 570–579.

[4] BlazingSQL. 2021. BlazingSQL. https://github.com/BlazingDB/blazingsql.

[5] Peter A Boncz et al. 2002. *Monet: A next-generation DBMS kernel for query-intensive applications.* Ph.D. Dissertation. Ph. d. thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands.

[6] Niladrish Chatterjee, Mike O'Connor, Gabriel H Loh, Nuwan Jayasena, and Rajeev Balasubramonia. 2014. Managing DRAM latency divergence in irregular GPGPU applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 128–139.

[7] Kevin Egan, Frédo Durand, and Ravi Ramamoorthi. 2011. Practical filtering for efficient ray-traced directional occlusion. In *Proceedings of the 2011 SIGGRAPH Asia Conference*. 1–10.

[8] Heiko Friedrich, Johannes Günther, Andreas Dietrich, Michael Scherbaum, Hans-Peter Seidel, and Philipp Slusallek. 2006. Exploring the use of ray tracing for future games. In *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*. 41–50.

[9] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined query processing in coprocessor environments. In *Proceedings of the 2018 International Conference on Management of Data*. 1603–1618.

[10] Franco Fuschini, Hassan El-Sallabi, Vittorio Degli-Esposti, Lasse Vuokko, Doriana Guiducci, and Pertti Vainikainen. 2008. Analysis of multipath propagation in urban environment through multidimensional measurements and advanced ray tracing simulation. *IEEE Transactions on Antennas and Propagation* 56, 3 (2008), 848–857.

[11] Google Cloud. 2025. *Use nested and repeated fields.* Google LLC.

[12] Dong He, Supun Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query processing on tensor computation runtimes. *arXiv preprint arXiv:2203.01877* (2022).

[13] HeavyDB. 2022. HeavyDB. https://github.com/heavyai/heavydb.

[14] Justus Henneberg and Felix Schuhknecht. 2023. RTIndeX: Exploiting Hardware-Accelerated GPU Raytracing for Database Indexing. *arXiv preprint arXiv:2303.01139* (2023).

[15] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. 2022. Tcudb: Accelerating database with tensor processors. In *Proceedings of the 2022 International Conference on Management of Data*. 1360–1374.

[16] Henrik Wann Jensen and Per Christensen. 2007. High quality rendering using ray tracing and photon mapping. In *ACM SIGGRAPH 2007 courses*. 1–es.

[17] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L Wolf, Paolo Costa, and Peter Pietzuch. 2016. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 International Conference on Management of Data*. 555–569.

[18] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. 2021. The art of balance: a RateupDB experience of building a CPU/GPU hybrid database product. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2999–3013.

[19] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics. *Proceedings of the VLDB Endowment* 9, 14 (2016), 1647–1658.

[20] Linwei Li, Kai Zhang, Jiading Guo, Wen He, Zhenying He, Yinan Jing, Weili Han, and X Sean Wang. 2020. Bindex: A two-layered index for fast and robust scans. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 909–923.

[21] Yangming Lv, Kai Zhang, Ziming Wang, Xiaodong Zhang, Rubao Lee, Zhenying He, Yinan Jing, and X Sean Wang. 2024. RTScan: Efficient Scan with Ray Tracing Cores. *Proceedings of the VLDB Endowment* 17, 6 (2024), 1460–1472.

[22] Enzo Meneses, Cristóbal A Navarro, Héctor Ferrada, and Felipe A Quezada. 2024. Accelerating range minimum queries with ray tracing cores. *Future Generation Computer Systems* 157 (2024), 98–111.

[23] Nate Morrical, Will Usher, Ingo Wald, and Valerio Pascucci. 2019. Efficient space skipping and adaptive sampling of unstructured volumes using hardware accelerated ray tracing. In *2019 IEEE Visualization Conference (VIS)*. IEEE, 256–260.

[24] Nate Morrical, Ingo Wald, Will Usher, and Valerio Pascucci. 2020. Accelerating unstructured mesh point location with RT cores. *IEEE transactions on visualization and computer graphics* 28, 8 (2020), 2852–2866.

[25] Vani Nagarajan, Durga Mandarapu, and Milind Kulkarni. 2023. Rt-knns unbound: Using rt cores to accelerate unrestricted neighbor search. In *Proceedings of the 37th International Conference on Supercomputing*. 289–300.

[26] Xuan-Thuan Nguyen, Hong-Thu Nguyen, Trong-Thuc Hoang, Katsumi Inoue, Osamu Shimojo, Toshio Murayama, Kenji Tominaga, and Cong-Kha Pham. 2016. An efficient FPGA-based database processor for fast database analytics. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1758–1761.

[27] NVIDIA. 2018. NVIDIA TURING GPU ARCHITECTURE. https://images.nvidia.cn/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[28] Patrick ONeil, Elizabeth ONeil, Xuedong Chen, and Stephen Revilak. 2009. The star schema benchmark and augmented fact table indexing. In *Performance Evaluation and Benchmarking: First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers 1*. Springer, 237–252.

[29] Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al. 2010. Optix: a general purpose ray tracing engine. *Acm transactions on graphics (tog)* 29, 4 (2010), 1–13.

[30] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. 2022. Query processing on heterogeneous CPU/GPU systems. *ACM Computing Surveys (CSUR)* 55, 1 (2022), 1–38.

[31] Charitha Saumya, Kirshanthan Sundararajah, and Milind Kulkarni. 2021. CFM: SIMT Thread Divergence Reduction by Melding Similar Control-Flow Regions in GPGPU Programs. *CoRR* (2021).

[32] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A study of the fundamental performance characteristics of GPUs and CPUs for database analytics. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*. 1617–1632.

[33] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. 2012. Database analytics acceleration using FPGAs. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. 411–420.

[34] Ingo Wald, Will Usher, Nathan Morrical, Laura Lediaev, and Valerio Pascucci. 2019. RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location. *High Performance Graphics (Short Papers)* 7 (2019), 13.

[35] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. 2014. Concurrent analytical query processing with GPUs. *Proceedings of the VLDB Endowment* 7, 11 (2014), 1011–1022.

[36] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of processing data warehousing queries on GPU devices. *Proceedings of the VLDB Endowment* 6, 10 (2013), 817–828.

[37] Zhengqing Yun and Magdy F Iskander. 2015. Ray tracing for radio propagation modeling: Principles and applications. *IEEE access* 3 (2015), 1089–1100.

[38] Yuhao Zhu. 2022. RTNN: accelerating neighbor search using hardware ray tracing. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 76–89.

NeutronCloud: Resource-Aware Distributed GNN Training in Fluctuating Cloud Environments

Mingyi Cao*
Northeastern University
Shenyang, China
2201763@stu.neu.edu.cn

Chunyu Cao*
Northeastern University
Shenyang, China
caochunyu@stumail.neu.edu.cn

Yanfeng Zhang
Northeastern University
Shenyang, China
zhangyf@mail.neu.edu.cn

Zhenbo Fu
Northeastern University
Shenyang, China
fuzhenbo@stumail.neu.edu.cn

Xin Ai
Northeastern University
Shenyang, China
aixin0@stumail.neu.edu.cn

Qiange Wang
Northeastern University
Shenyang, China
wangqiange94@gmail.com

Yu Gu
Northeastern University
Shenyang, China
guyu@mail.neu.edu.cn

Ge Yu
Northeastern University
Shenyang, China
yuge@mail.neu.edu.cn

ABSTRACT

Graph Neural Networks (GNNs) are widely employed to learn representations from graph-structured data. To support large-scale graph training, researchers use distributed techniques, partitioning the graph across multiple computing nodes and performing parallel training by exchanging dependency vertex information via cross-node communication. However, existing GNN training systems operate on statically partitioned subgraphs, making them difficult to adapt to resource fluctuations. In practice, resource fluctuations in cloud environments often cause variability in compute and communication resources, posing challenges for aligning each worker's workload to its available resources during GNN training.

In this paper, we propose NeutronCloud, a system designed for efficient GNN training in cloud environments. First, we adopt a resource-aware workload adjustment strategy. It builds on hybrid dependency handling by obtaining dependency information through both local computation and remote communication. During training, it dynamically adjusts the ratio between locally computed and remotely fetched dependencies based on each worker's available resources, ensuring workload-resource alignment. Second, we employ a dependency-aware partial-reduce approach reusing historical vertex embeddings and skipping the stragglers during gradient aggregation to address extreme resource fluctuations that cause some workers to lag significantly behind others in the cluster. Experimental results on the resource-fluctuating environment demonstrate that NeutronCloud achieves 1.83×-4.43× speedup compared to state-of-the-art distributed GNN systems.

PVLDB Reference Format:
Mingyi Cao, Chunyu Cao, Yanfeng Zhang, Zhenbo Fu, Xin Ai, Qiange Wang, Yu Gu, Ge Yu. NeutronCloud: Resource-Aware Distributed GNN Training in Fluctuating Cloud Environments. PVLDB, 19(1): 56 - 69, 2025.
doi:10.14778/3772181.3772186

*Both authors contributed equally to this research.
This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 1 ISSN 2150-8097.
doi:10.14778/3772181.3772186

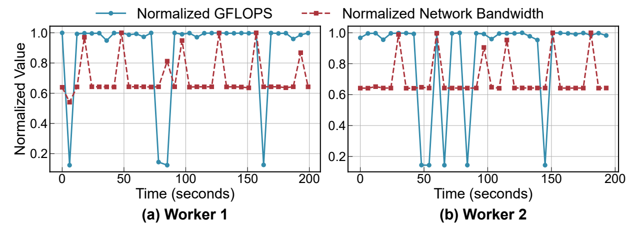


Figure 1: Normalized computational throughput (FLOPS) and network bandwidth measured over 200 seconds on two Alibaba Cloud shared GPU instances (each with a shared NVIDIA A10 GPU and up to 10 Gbps bandwidth).

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Toaoc/NeutronCloud.

1 INTRODUCTION

Graph Neural Networks (GNNs) have emerged as a fundamental approach for graph-based tasks such as social network analysis [5, 11], link prediction [22, 31, 52], and recommendation systems [49]. The GNN models iteratively perform neighbor aggregation and representation updates for each vertex to capture complex topological information. Distributed training methods are adopted to partition graph data across multiple workers to handle large-scale graphs, running GNN models in parallel. Most existing distributed GNN training systems [3, 6, 10, 16, 24, 34, 37, 38, 42, 44, 51, 54, 57] assume a stable resource environment, where computational and network resources remain consistent during training. However, in reality, resource fluctuations are ubiquitous in real-world applications.

Resource fluctuations can be categorized into computational resource fluctuations and network resource fluctuations. In cloud environments, computational resources fluctuate due to contention on shared physical hosts [35]. For example, burstable instances (e.g., AWS T-series or Alibaba Cloud "shared-core" VMs) [1, 36] may decrease CPU/GPU performance when neighboring VMs experience high demand. Likewise, network performance varies due to congestion and bandwidth contention. Multi-tenant clusters often face communication bottlenecks caused by overloaded data center switches and cross-worker network contention [28, 47, 53]. To illustrate these fluctuations, we evaluate computational power (FLOPS,

floating point operations per second) and network bandwidth over time in a two-worker cluster on Alibaba Cloud's shared GPU instances. As shown in Figure 1, FLOPS and network bandwidth drop by up to 90% and 35%, respectively.

Such fluctuations significantly impact the performance of distributed GNN training. When performing distributed GNN training in a resource-fluctuating environment, the initially assigned workload becomes mismatched with the fluctuating resources, causing computational and communication bottlenecks, and leading to the emergence of slower workers (stragglers). In addition, GNN data samples exhibit complex interdependencies, further exacerbating this issue. In distributed GNN training, graph partitioning results in local vertices needing to aggregate features from remote neighbors (called remote dependency vertices), and all worker parameters need to be synchronized at the end of each epoch. Thus, the delay caused by stragglers propagates throughout the entire cluster during training, slowing down the overall training performance.

We summarize that the key challenge in distributed GNN training under resource fluctuations is how to quickly adjust each worker's computational and communication workload to match the available resources. For Deep Neural Network (DNN) systems, there are no dependencies among input samples. They mitigate resource-load mismatches through real-time load migration to address resource fluctuations [14, 56]. However, for GNN systems, the data dependencies make workload migration difficult. Workload migration requires not only transferring vertex features but also adjusting and maintaining the dependencies between workers to ensure graph consistency. This leads to substantial additional overhead and reduces the efficiency of workload adjustment. The overhead of migration can even offset the benefits of workload adjustment.

In this paper, we propose NeutronCloud, a distributed GNN training system capable of resource-aware workload adjustment, addressing the above challenge through two critical strategies.

First, we propose a resource-aware workload adjustment strategy that adapts dependency handling based on real-time resource conditions. The fetching of remote dependencies takes up most of the time in the entire distributed GNN training process [2, 12, 32, 37, 45]. We propose a lightweight resource-aware workload adjustment strategy based on the hybrid dependency handling method. This strategy dynamically adjusts the processing of remote dependency vertices in the cluster to adjust computational and communication workloads. Specifically, we cache remote dependency vertices and their multi-hop neighbors, obtaining embeddings through local computation to address the reduction in communication resources. When computational resources are constrained, more remote dependency vertex embeddings are fetched via cross-worker communication. To enable flexible runtime adjustment, we trade additional storage for adaptability by pre-caching remote dependency vertices and their multi-hop neighbors during the preprocessing phase. This design eliminates the need for workload migration during adjustment.

Second, when some workers face extreme resource degradation (e.g., computation and communication resources decrease simultaneously), adjusting vertex dependencies processing is not enough to reduce the serious delay caused by the synchronization of the severe stragglers. To address this problem, we introduce a dependency-aware partial-reduce strategy, allowing local computation using cached historical embeddings when embeddings from severe stragglers cannot be received in time. Despite being slightly outdated, these embeddings still provide useful information for computation without stalling the process. During gradient aggregation, we synchronize gradients only from faster workers, skipping the severe stragglers. To ensure unbiased gradients and model convergence, we adopt a dependency-aware weighted gradient aggregation strategy and set a bound on severe stragglers that are skipped.

In summary, we make the following contributions.

- We propose a resource-aware workload adjustment strategy, which dynamically adjusts the number of remote dependencies for each worker by quantifying variations in computation and communication resources, ensuring a better match between resource and workload.
- We propose a dependency-aware partial-reduce approach to reduce synchronization overhead. By using history embeddings, we update only the faster worker's parameters while setting a bound to ensure convergence.
- We develop NeutronCloud, an efficient GNN training system for resource fluctuation environments. The experimental results show that NeutronCloud achieves 1.83× - 4.43× speedup compared to state-of-the-art GNN systems.

2 BACKGROUND

2.1 Resource Fluctuations

Resource fluctuations are common in real-world applications and are often caused by contention on shared physical hosts or network contention. These resource fluctuations frequently impact computing power and network bandwidth, significantly affecting the performance of distributed GNN training.

To analyze the impact of resource fluctuations on the performance of distributed GNN systems, we evaluate two GNN training systems, NeutronStar [45] and Sancus [32], on resource-fluctuating and resource-stable clusters using a two-layer GCN model. The cluster consists of four workers, each equipped with NVIDIA T4 GPUs, interconnected via a 10 Gbps network.

Motivated by existing methods for simulating resource fluctuations [25–27, 56], we emulate runtime fluctuations by injecting sleep commands into workers. Specifically, in each iteration, each worker incurs additional overhead with a 10% probability, equivalent to twice the average epoch runtime. To emulate sustained fluctuation patterns (as shown in Figure 1), each injected sleep command lasts for 5 consecutive epochs.

As shown in Figure 2, the per-epoch runtime in a resource-fluctuating environment is 3.5× slower than that in a resource-stable environment. Distributed GNN training needs to synchronize data across all workers in each epoch. Resource fluctuations on any worker can impact the efficiency of the entire cluster. This indicates that resource fluctuations can significantly degrade the efficiency of distributed GNN training.

2.2 GNN Training

A graph neural network processes a graph as input, where each vertex and edge is associated with high-dimensional features. Through

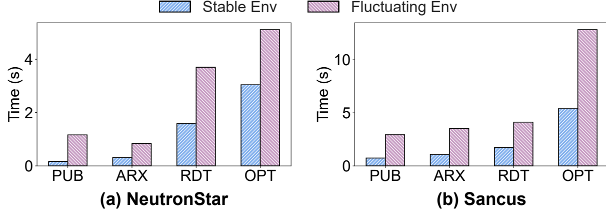


Figure 2: Per-epoch runtime comparison of NeutronStar [45] and Sancus [32] under stable environment (Stable Env) and resource-fluctuating environment (Fluctuating Env) on four datasets PUB (Pubmed), ARX (Ogbn-Arxiv), RDT (Reddit), and OPT (Ogbn-Products).

multiple layers, GNNs iteratively propagate and aggregate information from neighboring vertices, updating their representations to capture the graph structure information.

In the l-th layer, the aggregation result a_v^l of vertex v is obtained by collecting the embedding h_u^{l-1} of its neighboring vertex u in the $(l-1)$-th layer, as shown in the following formula:

$$a_v^l = Aggregate(h_u^{l-1}, u \in N(v)) \tag{1}$$

Subsequently, the update function uses the result a_v^l and h_v^{l-1}, which is the embedding of the vertex v in the $(l-1)$-th layer, to compute the representation h_v^l of the vertex v in the l-th layer, as shown in the following formula.

$$h_v^l = Update(a_v^l, h_v^{l-1}) \tag{2}$$

The final-layer vertex embeddings are fed into downstream tasks, where a task-specific loss is computed against ground truth labels. This loss triggers backward propagation to calculate gradients via automatic differentiation. These gradients drive parameter updates through gradient-based optimizers like SGD or its adaptive variants (e.g., Adam). The standard SGD update rule is:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\theta_t) \tag{3}$$

where the η denotes the learning rate controlling the step size, the $\nabla_\theta \mathcal{L}(\theta_t)$ denotes the gradient of loss function w.r.t parameters.

2.3 Distributed GNN Training Approach

Distributed GNN training partitions the input graph across multiple workers. Dependencies arise when vertices need to aggregate features from remote neighbors. The critical aspect of distributed GNN systems is efficiently handling remote dependencies.

We categorize existing dependency handling strategies into three approaches. The **Dependencies Communicated** (DepComm) approach requires each vertex to gather its neighbors' representations from remote workers via cross-worker communication [17, 38], as shown in Figure 3 (a). This method reduces storage consumption but leads to significant communication overhead. The **Dependencies Cached** (DepCache) approach caches the features of multi-hop neighbors of remote dependency vertices on the local worker in advance for multi-layer computing [57], as shown in Figure 3 (b). This eliminates inter-worker communication but results in significant redundant computation and storage overhead.

As shown in Figure 3(c), the hybrid dependency-resolution scheme assigns a subset of remote dependency vertices to DepCache (e.g., vertex 0 on worker 1) and the remainder to DepComm (e.g., vertex 1 on worker 1). The approach can balance the use of computational and communication resources. Based on this hybrid design, we

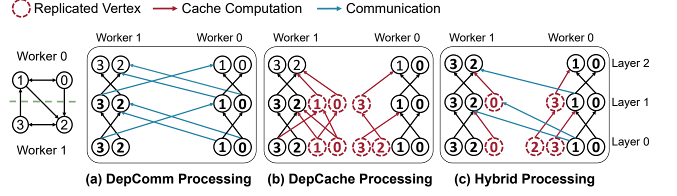


Figure 3: Hybrid dependency handling method of distributed GNN training.

Algorithm 1 DepComm-DepCache Hybrid Training for L-layer GCN

Input: $G = (V, E)$, L, $\{\mathbf{h}_v^{(0)}\}_{v \in V}$, $\{L_v\}_{v \in V_L}$; partition $\{V_1, \ldots, V_m\}$; params $\{W^{(\ell)}\}_{\ell=1}^{L}$
Output: updated $\{W^{(\ell)}\}_{\ell=1}^{L}$
1: For worker i: $E_i \leftarrow \{(u, v) \in E \mid v \in V_i\}$; $N_{\text{loc}}(v) = N(v) \cap V_i$, $N_{\text{rem}}(v) = N(v) \setminus V_i$
2: **for each worker** $i = 1, \ldots, m$ **in parallel do**
3: **for** $\ell = 1$ to L **do** ▷ Forward
4: **for** each $v \in V_i$ **do**
5: $S_{\text{loc}} \leftarrow \{\mathbf{h}_u^{(\ell-1)} : u \in N_{\text{loc}}(v)\}$
6: $F_\ell(v) \leftarrow \{u \in N_{\text{rem}}(v) : \text{DepComm}(u, v, \ell)\}$
7: $C_\ell(v) \leftarrow N_{\text{rem}}(v) \setminus F_\ell(v)$
8: $S_{\text{cache}} \leftarrow \{\text{CacheGet}(u, \ell-1) : u \in C_\ell(v)\}$
9: PullByOwner $\{\mathbf{h}_u^{(\ell-1)} : u \in F_\ell(v)\}$ from owner(u)
10: $S_{\text{comm}} \leftarrow \{\mathbf{h}_u^{(\ell-1)} : u \in F_\ell(v)\}$
11: $S \leftarrow S_{\text{loc}} \cup S_{\text{cache}} \cup S_{\text{comm}}$
12: $a_v \leftarrow \text{Aggregate}(S)$
13: $\mathbf{h}_v^{(\ell)} \leftarrow \text{Update}(a_v, \mathbf{h}_v^{(\ell-1)}; W^{(\ell)})$
14: $\hat{L}_v \leftarrow P(\mathbf{h}_v^{(L)})$ for $v \in V_{L,i}$; $loss_i \leftarrow \text{Loss}(\{\hat{L}_v\}, \{L_v\})$; $loss \leftarrow \text{AllReduceSum}(loss_i)$
15: Backprop to obtain $\{\nabla \mathbf{h}_v^{(\ell-1)}\}$ and $\{\nabla W^{(\ell)}\}$ for $v \in V_i$
16: **for** $\ell = L$ **down to** 1 **do** ▷ Backward: send grads only for DepComm neighbors
17: **for** each $v \in V_i$ **do**
18: **for** each $u \in F_\ell(v)$ **do**
19: Send $\nabla \mathbf{h}_u^{(\ell-1)}$ to owner(u)
20: **for** $\ell = 1$ to L **do** ▷ Sync & update
21: $\nabla W^{(\ell)} \leftarrow \text{AllReduceSum}(\nabla W^{(\ell)})$
22: $W^{(\ell)} \leftarrow \text{OptStep}(W^{(\ell)}, \nabla W^{(\ell)})$

propose a dynamic adjustment mechanism that adaptively adjusts the ratio between DepCache and DepComm during training. This enables each worker to adapt its computation and communication workload to fluctuating resources.

Algorithm 1 outlines the hybrid dependency handling. In the forward pass (Lines 5–13), the worker collects local neighbors, partitions remote neighbors into DepComm and DepCache, computes the DepCache branch locally, fetches DepComm embeddings via communication, and then aggregates and updates. The per-worker loss is computed and reduced across workers (Line 14). In the backward pass (Lines 15–20), local gradients are computed on each worker, and only gradients for DepComm remote neighbors are sent to their owners, while DepCache neighbors are handled locally. Finally (Lines 21–22), gradients are All-Reduced and parameters are updated.

Therefore, the exchange of vertex embeddings and gradients at each layer, combined with the synchronization of parameter updates, creates layer-wise communication barriers, introducing significant synchronization overhead to distributed GNN training.

2.4 Existing GNN Systems in Fluctuating Environments

Most existing distributed GNN systems are built on the assumption of stable resource availability. Systems such as DGL [43], AliGraph [57], MGG [46], ROC [17], and DGCL [2] rely on static, predefined task allocation and communication–computation pipelines, with optimizations for computation, memory, or communication applied at initialization. These designs assume static resource conditions and therefore cannot adapt to runtime resource fluctuations. As a result, execution plans quickly become suboptimal, and delays caused by stragglers can cascade across synchronization barriers, leading to resource underutilization.

Some recent systems attempt to mitigate the impact of stragglers caused by resource fluctuations. PipeGCN [41] adopts pipelined execution with historical embeddings to overlap computation and communication, while Sancus [32] introduces historical embeddings to reduce communication with stragglers. These approaches can partially hide the latency introduced by stragglers. However, they still rely on synchronous parameter updates (e.g., all-reduce), which are bottlenecked by the slowest worker in each iteration and limit adaptability under resource fluctuations.

While NeutronStar [45] also adopts a hybrid dependency handling strategy, it lacks runtime resource awareness and cannot adjust dependency handling based on dynamic execution conditions. Moreover, its hybrid dependency handling method requires evaluating the benefit of each dependency and performing global sorting, which incurs high overhead and makes it unsuitable for online adjustment.

3 SYSTEM OVERVIEW

We introduce NeutronCloud, a distributed GNN system capable of handling resource fluctuations through two critical strategies. First, NeutronCloud provides a resource-aware workload adjustment strategy that adjusts the computational and communication workloads of each worker to match the real-time resource conditions. Second, NeutronCloud introduces a dependency-aware partial-reduce approach to reduce synchronization overhead between fast and slow workers.

Resource-aware Workload Adjustment Strategy. NeutronCloud employs a lightweight resource-aware workload adjustment algorithm based on the hybrid dependency handling method. The strategy consists of two phases. In the preprocessing phase, remote dependency vertices and their multi-hop neighbors are cached in local CPU memory. Additionally, the costs of different handling methods for remote dependency vertices are computed, thereby determining per-vertex adjustment priorities. In the online adjustment phase, the algorithm continuously records the computation and communication time of each worker per epoch and quantifies resource fluctuations. Based on the adjustment priorities obtained in the preprocessing phase, a binary search is performed to determine the handling method for remote dependency vertices in the next epoch, thereby speeding up the adjustment process. Moreover, no workload migration is required during the adjustment process.

Dependency-aware Partial-reduce Strategy. We propose a dependency-aware partial-reduce method to address the significant

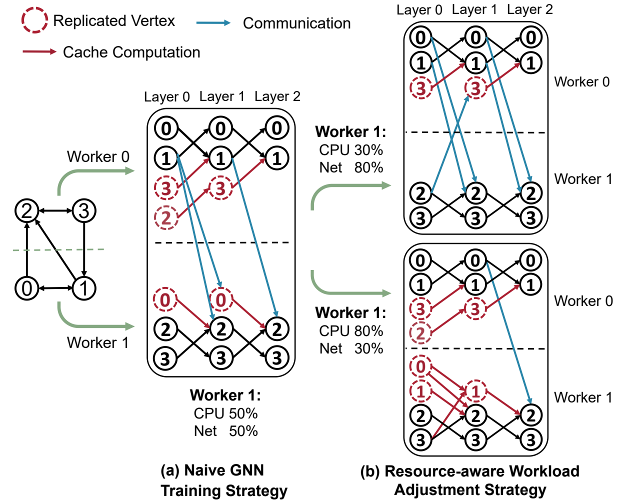


Figure 4: An illustrative example of resource-aware workload adjustment.

synchronization time caused by severe stragglers. This method consists of partial computation and partial update. Partial computation is applied to graph propagation (including both forward and backward propagation), when computing remote dependency vertices handled by the DepComm approach, if the severe stragglers fail to provide timely embedding updates, we use the locally cached historical embeddings of dependency vertices to continue forward propagation. Similarly, historical gradients are used for backward propagation. Additionally, a lightweight controller monitors embedding and gradient staleness, ensuring they are used only if their divergence from the latest embeddings is below a certain threshold.

Partial update limits gradient aggregation to faster workers and skips the stragglers. A runtime controller monitors worker progress to ensure that each update involves the majority of workers. To maintain fairness and convergence, we ensure that all workers participate in parameter updates within a bounded number of epochs, and cached embeddings are periodically refreshed. Finally, we apply dependency-aware weighted parameter synchronization to ensure unbiased aggregation.

4 RESOURCE-AWARE WORKLOAD ADJUSTMENT

In this section, we first analyze the impact of resource fluctuation on GNN training. Then, we define an optimization objective that minimizes the runtime of each worker under resource-fluctuating environments by adjusting how dependencies are handled. Finally, we propose a lightweight algorithm to approximate this objective during training.

4.1 Impact Study of Resource Fluctuations

To design our workload adjustment strategy, we analyze how resource fluctuations affect each worker's total runtime, including local subgraph computation, DepCache computation, and DepComm communication. Given the processing times of the two dependency-handling approaches, we formulate an objective to minimize the total runtime by adjusting their proportions.

Training Time under Resource Fluctuations. We decompose each worker's training time under resource fluctuations into three components: $T'_{\text{local},i}$ for local subgraph computation, $T'_{\text{cache},i}$ for DepCache computation, and $T'_{\text{comm},i}$ for DepComm communication. This decomposition is expressed as:

$$T'_i = T'_{\text{local},i} + T'_{\text{cache},i} + T'_{\text{comm},i} \tag{4}$$

Impact of Resource Fluctuations on DepCache. DepCache improves training efficiency by locally caching the features of multi-hop dependency neighbors to reduce communication. For each vertex, this results in a layer-wise recursive structure: at layer l, the embedding computation depends on its in-neighbors from layer $l-1$, and so on. As shown in Figure 4(a), computing the embedding of Vertex 3 at Layer 1 in Worker 0 requires the Layer-0 embedding of Vertex 2.

Under resource fluctuations, the cost of processing cached dependencies becomes sensitive to system performance. In particular, contention on compute and memory resources affects vertex and edge operations, which is reflected in the runtime-aware per-dimensional costs T'_v and T'_e of processing each vertex and edge. Let $V^{(l)}_{\text{cache},i}$ and $E^{(l)}_{\text{cache},i}$ denote the cached vertices and edges at layer l on worker i. Since multi-hop dependencies are cached across layers, the overall DepCache cost for worker i is computed by accumulating costs across multiple layers:

$$T'_{\text{cache},i} = \sum_{l=1}^{L-1} \left(|V^{(l)}_{\text{cache},i}| \cdot T'_v + |E^{(l)}_{\text{cache},i}| \cdot T'_e \right) \cdot d^{(l)} \tag{5}$$

where $d^{(l)}$ is the feature dimension of layer l.

Impact of Resource Fluctuations on Local Subgraph Computation. Similar to DepCache computation, the local subgraph computation time under resource fluctuations is also composed of the multi-layer vertex and edge processing costs, as shown in the following equation:

$$T'_{\text{local},i} = \sum_{l=1}^{L} \left(|V^{(l)}_{\text{local},i}| \cdot T'_v + |E^{(l)}_{\text{local},i}| \cdot T'_e \right) \cdot d^{(l)} \tag{6}$$

where $|V^{(l)}_{\text{local},i}|$ and $|E^{(l)}_{\text{local},i}|$ represent the number of vertices and edges in the local partition of worker i.

Impact of Resource Fluctuations on DepComm. The DepComm cost arises from cross-partition dependency vertices when a vertex must fetch feature vectors from remote neighbors of other workers. As shown in Figure 4(a), computing the embedding of Vertex 2 on Worker 1 of Layer 1 requires accessing the embedding of its remote neighbor Vertex 1 located on Worker 0 of Layer 0. Such communication recurs across layers, as each GNN layer aggregates information from the remote neighbors. Under bandwidth fluctuations, the per-dimension communication cost between Worker i and Worker j is denoted by $T'_{c,ij}$. The total communication cost is given by:

$$T'_{\text{comm},i} = \sum_{l=1}^{L} \left(d^{(l-1)} \cdot \sum_{\substack{j=0 \\ j\neq i}}^{m-1} T'_{c,ij} \cdot |V^{(l-1)}_{\text{comm},ij}| \right) \tag{7}$$

where $|V^{(l-1)}_{\text{comm},ij}|$ denotes the number of dependency vertices transferred from worker j to worker i at layer $l-1$, and m denotes the total number of workers.

Optimization Objective. To support decentralized optimization in resource-fluctuating environments, each worker independently minimizes its own runtime-aware training time T'_i, which reflects the current execution delay under resource fluctuations. This is achieved by dynamically adjusting the assignment of dependency vertices between DepCache and DepComm, based on locally available resource. Moreover, the additional memory consumption for DepCache must respect the worker's memory constraint C_i. The optimization problem for worker i is formulated as:

$$\min \quad T'_i = T'_{\text{comm},i} + T'_{\text{cache},i} + T'_{\text{local},i} \tag{8}$$

$$V_{\text{cache},i} \cup V_{\text{comm},i} = V_{\text{depend},i} \tag{9}$$

$$V_{\text{cache},i} \cap V_{\text{comm},i} = \emptyset \tag{10}$$

$$\sum_{v \in V_{\text{cache},i}} s(v) \leq C_i \tag{11}$$

where $V_{\text{depend},i}$ denotes the complete set of dependency vertices required by worker i, and $s(v)$ denotes the storage cost of vertex v.

The storage cost $s(v)$ of a dependency vertex v is determined by the number of neighbor vertices and edges it introduces across layers when cached locally. Formally, we define:

$$s(v) = \sum_{l=1}^{L-1} \left(|V^{(l)}_i(v)| + |E^{(l)}_i(v)| \right) \cdot d^{(l)} \tag{12}$$

where $|V^{(l)}_i(v)|$ and $|E^{(l)}_i(v)|$ denote the number of vertices and edges introduced by v at layer l.

4.2 Lightweight Resource-aware Workload Adjustment

NP-hardness. We show that the dependency adjustment problem in our formulation is NP-hard by a polynomial-time reduction from the Cardinality-Constrained Knapsack Problem (CCKP). Given any CCKP instance, we map each item to a dependency; the item size maps to the storage cost, and the item profit maps to the reduction in runtime variation. The capacity constraints are preserved. Under this mapping, selecting items in CCKP to maximize total profit is equivalent to selecting dependency handling method to minimize runtime variation subject to the memory budget. Since this reduction is in polynomial time and CCKP is NP-hard, our problem is also NP-hard.

Lightweight Adaptive Adjustment Algorithm. We propose a workload adjustment algorithm to address this NP-hard problem. The algorithm dynamically adjusts the handling of the dependencies to align the computation and communication load of each worker with its available resources. As shown in Figure 4, when a worker faces limited CPU resources, more dependencies are retrieved using the DepComm approach (upper part of Figure 4(b)). Conversely, when a worker suffers from limited communication bandwidth, the DepCache approach is preferred (lower part of Figure 4(b)). The algorithm initially sets all dependency vertices to use DepComm, and then selects the vertices with the highest benefit to convert them to DepCache. The algorithm consists of two stages: preprocessing and online adjustment. The preprocessing stage is executed before training starts to cache essential data and prioritize dependency vertices through sorting by their computation

Algorithm 2 Preprocessing Phase

Input: vertex subset V_i, edge subset E_i; set of remote dependency vertices D; number of remote workers m; T_v, T_e.

Output: $\{\mathcal{V}_{i,j}\}_{j=1}^m$: remote dependency vertices owned by worker j, sorted by $T_{\text{cache},i,j}(v)$;
$\{\mathcal{N}_{i,j}\}_{j=1}^m$: arrays aligned with $\mathcal{V}_{i,j}$. For each v, $\mathcal{N}_{i,j}(v)$.V = # multi-hop neighbor vertices, $\mathcal{N}_{i,j}(v)$.E = # multi-hop neighbor edges, $\mathcal{N}_{i,j}(v)$.S = per-vertex storage cost;
$\{\mathcal{S}_{i,j}\}_{j=1}^m$: prefix-sum arrays of storage cost over $\mathcal{V}_{i,j}$, i.e., $S_{i,j}[k] = \sum_{t=1}^{k} \mathcal{N}_{i,j}(\mathcal{V}_{i,j}[t])$.S.

1: **for** $j = 1$ to m **do**
2: $\quad \mathcal{V}_{ij} \leftarrow \{\, v \in D, v \in V_j \}$
3: $\quad$ **for** each $v \in \mathcal{V}_j$ **do**
4: $\quad\quad T_{\text{cache},ij}(v) \leftarrow \mathcal{N}_{ij}(v).V \cdot T_v \;+\; \mathcal{N}_{ij}(v).E \cdot T_e$
5: $\quad$ SORT $(\mathcal{V}_{ij})$ by $T_{\text{cache},j}(v)$ *(ascending)*
6: $\quad$ REINDEX $\mathcal{N}_{i,j}$ accordingly
7: $\quad \mathcal{S}_{ij}[1] \leftarrow \mathcal{N}_{ij}(\mathcal{V}_{ij}[1]).S$
8: $\quad$ **for** $k = 2$ to $|\mathcal{V}_{ij}|$ **do**
9: $\quad\quad \mathcal{S}_{ij}[k] \leftarrow \mathcal{S}_{ij}[k-1] + \mathcal{N}_{ij}(\mathcal{V}_{ij}[k]).S$
10: **return** $\{\mathcal{V}_{i,j}\}_{j=1}^m, \{\mathcal{N}_{i,j}\}_{j=1}^m, \{\mathcal{S}_{i,j}\}_{j=1}^m$

overhead. The online adjustment stage dynamically adjusts dependency handling approaches based on real-time resource feedback. By offloading expensive operations to preprocessing, the online adjustment remains lightweight and efficient during training.

Preprocessing Phase. Each worker i precomputes two ordered arrays, $\mathcal{N}_i$ and $\mathcal{S}_i$. Array $\mathcal{N}_i$ groups remote dependency vertices by their owner j into $\mathcal{V}_{i,j}$ and, for each $v \in \mathcal{V}_{i,j}$, stores the counts of multi-hop neighbor vertices/edges and the per-vertex storage cost. Array $\mathcal{S}_i$ contains prefix-sum arrays $S_{i,j}$ of the storage costs for each $\mathcal{V}_{i,j}$. We group by owner because vertices from the same remote worker share the same per-feature-dimension communication time $T_{\text{c},i,j}$ (Algorithm 2, Lines 1–2). Using the measured per-feature-dimension processing costs T_v and T_e, we compute $T_{\text{cache},i,j}(v)$ for $v \in \mathcal{V}_{i,j}$ (Line 4), sort $\mathcal{V}_{i,j}$ accordingly and reorder $\mathcal{N}_{i,j}$ (Lines 5–6), then build $S_{i,j}$ (Lines 7–9). These precomputations enable fast lookups during online adjustment. Although T_e and T_v may fluctuate, they typically scale proportionally; hence the relative order of $T_{\text{cache},i,j}(v)$ is empirically stable across epochs. This is because compute-bound (e.g., GEMM, related to T_v) and memory-bound (e.g., SpMM, related to T_e) operations degrade proportionally under GPU contention. We validate this proportional degradation through experiments, with detailed results available in the technical report [4] due to the page limit. Therefore, these arrays only need to be generated once before training, significantly reducing the adjustment overhead.

Online Adjustment Phase. During training, worker i adapts the handling of remote dependencies to resource fluctuations while keeping the *cached* set within budget C_i (Algorithm 3). For each owner j, since $T'_{\text{comm},i,j}$ is constant over $v \in \mathcal{V}_{i,j}$ and $\mathcal{V}_{i,j}$ is sorted by $T'_{\text{cache},i,j}(v)$, we locate the positive-benefit region $\gamma_{i,j}(v) = T'_{\text{comm},i,j} - T'_{\text{cache},i,j}(v) > 0$ by binary search (Lines 2–6). From the prefix-sum array $S_{i,j}[\cdot]$ we obtain its storage $s_{i,j} = S_{i,j}[k_j]$ (Line 7) and the total demand $S_i = \sum_{j=1}^m s_{i,j}$ (Line 9). If $S_i \le C_i$, convert all positive-benefit vertices to DepCache and leave the rest DepComm (Lines 12–13). Otherwise allocate budget proportionally,

Algorithm 3 Online Adjustment Phase

Input: $\{\mathcal{V}_{i,j}\}, \{\mathcal{N}_{i,j}\}, \{S_{i,j}\}$; updated costs $T'_v, T'_e, T'_{\text{c},i,j}$; total storage constraint C_i; number of remote workers m.

Output: $\{\mathcal{V}_{i,j}^{\text{cache}}\}_{j=1}^m, \{\mathcal{V}_{i,j}^{\text{comm}}\}_{j=1}^m$.

1: **Initialize:** $\mathcal{V}_{i,j}^{\text{cache}} \leftarrow \emptyset,\ \mathcal{V}_{i,j}^{\text{comm}} \leftarrow \emptyset$ for $j = 1, \dots, m$
2: **for** $j = 1$ to m **in parallel do**
3: $\quad$ **for** each $v \in \mathcal{V}_{i,j}$ **do**
4: $\quad\quad T'_{\text{cache},i,j}(v) \leftarrow \mathcal{N}_{i,j}(v).\text{V} \cdot T'_v \;+\; \mathcal{N}_{i,j}(v).\text{E} \cdot T'_e$
5: $\quad\quad \gamma_{i,j}(v) \leftarrow T'_{\text{c},i,j} \;-\; T'_{\text{cache},i,j}(v)$
6: $\quad$ **BinarySearch** in $\mathcal{V}_{i,j}$ for largest index k_j s.t. $\gamma_{i,j}(v) > 0$
7: $\quad s_{i,j} \leftarrow S_{i,j}[k_j]$
8: $\quad$ // k_j is the last vertex with positive benefit.
9: $S_i \leftarrow \sum_{j=1}^m s_{i,j}$
10: **if** $S_i \le C_i$ **then**
11: $\quad$ **for** $j = 1$ to m **in parallel do**
12: $\quad\quad \mathcal{V}_{i,j}^{\text{cache}} \leftarrow$ first k_j items of $\mathcal{V}_{i,j}$
13: $\quad\quad \mathcal{V}_{i,j}^{\text{comm}} \leftarrow \mathcal{V}_{i,j} \setminus \mathcal{V}_{i,j}^{\text{cache}}$
14: **else**
15: $\quad$ **for** $j = 1$ to m **do**
16: $\quad\quad C_{i,j} \leftarrow (s_{i,j}/S_i) \cdot C_i$
17: $\quad$ **for** $j = 1$ to m **in parallel do**
18: $\quad\quad k'_j \leftarrow \max\{\, k \le k_j \mid S_{i,j}[k] \le C_{i,j} \,\}$
19: $\quad\quad \mathcal{V}_{i,j}^{\text{cache}} \leftarrow$ first k'_j items of $\mathcal{V}_{i,j}$
20: $\quad\quad \mathcal{V}_{i,j}^{\text{comm}} \leftarrow \mathcal{V}_{i,j} \setminus \mathcal{V}_{i,j}^{\text{cache}}$
21: **return** $\{\mathcal{V}_{i,j}^{\text{cache}}\}_{j=1}^m,\ \{\mathcal{V}_{i,j}^{\text{comm}}\}_{j=1}^m$

$C_{i,j} = (s_{i,j}/S_i)\, C_i$ (Lines 15–16), and for each j take the largest prefix of $\mathcal{V}_{i,j}$ that fits $C_{i,j}$ using $S_{i,j}$ (Lines 18–20). This prioritizes high-benefit conversions under the memory constraint.

Complexity Analysis. We fuse benefit computation into the binary search to avoid extra passes. Each adjustment performs at most $(m-1)$ binary searches (one per remote owner $j \ne i$) over sorted lists $\mathcal{V}_{i,j}$. Let $N_{\text{dep}} = \sum_{j \ne i} |\mathcal{V}_{i,j}|$. The online time is $\sum_{j \ne i} O(\log |\mathcal{V}_{i,j}|) = O\big(m \log(N_{\text{dep}}/m)\big)$ in the balanced case, and $O(m \log N_{\text{dep}})$ in general. The preprocessing is a one-time step before training that sorts and builds prefix sums over N_{dep} items, costing $O(N_{\text{dep}} \log N_{\text{dep}})$.

Approximation guarantee. Our method achieves a $\frac{1}{2}$-approximation guarantee as it is equivalent to that of the default greedy algorithm of CCKP, which ranks items by their benefit-to-cost ratio $\gamma(v)/s(v)$ and selects the largest feasible prefix, achieving a worst-case $\frac{1}{2}$-approximation [19]. Our algorithm adopts a simplified strategy that ranks vertices by $\gamma(v)$ and selects the Top-K vertices under the memory budget. In our design, ranking by $\gamma(v)$ is effectively equivalent to ranking by $\gamma(v)/s(v)$, as vertices with higher $\gamma(v)$ typically induce lower neighborhood memory costs $s(v)$. This allows us to omit $s(v)$ from the ranking criterion.

5 DEPENDENCY-AWARE PARTIAL-REDUCE

5.1 Overall Workflow

Motivation. The previously proposed resource-aware workload adjustment strategy generally mitigates the impact of resource fluctuations. However, under extreme resource degradation (e.g., simultaneous drops in compute capacity and network bandwidth),

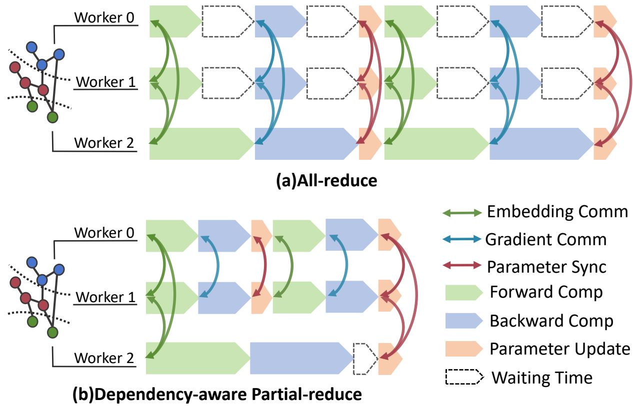


Figure 5: Partial-reduce for GNN. "Comm" indicates communication, "Comp" indicates computation, and "Sync" indicates synchronization.

some workers may lag significantly behind the rest of the cluster. When such degradation persists, workload adjustment alone may be insufficient to mitigate **severe stragglers**.

Challenges of Partial-reduce in GNNs. In distributed DNNs, partial-reduce (a variant of all-reduce) reduces synchronous overhead caused by severe stragglers by synchronizing gradients among only a subset of faster workers in each epoch. However, due to complex dependencies between training samples (vertices), partial-reduce faces the following challenges in distributed GNN training.

First, partial-reduce only works on the parameter synchronization stage. Yet workers must exchange embeddings and gradients of dependency vertices at every layer, creating implicit synchronization points. Thus, each epoch includes additional layer-wise synchronizations in addition to parameter synchronization.

Second, the number of workers involved in parameter updates may vary across iterations in partial-reduce, leading to unstable gradient scaling and slower convergence [8, 23, 33].In distributed DNN training, this is typically handled by weighted gradient averaging, with each worker's contribution proportional to its number of local training samples [59]. However, the number of training vertices for different workers is different, and the aggregation operation collects non-training vertices embeddings, which will also generate gradients for these non-training vertices during backward computation, as shown in Figure 6(a). Simply performing a weighted averaging of parameters across workers based on the number of training vertices may result in significant convergence bias.

A Two-stage Solution: Partial Computation and Partial Update. We propose a dependency-aware partial-reduce method for GNN, consisting of partial computation and partial update, as shown in Figure 5. Partial computation reuses historical vertex embeddings and gradients for forward/backward computation. Figure 5(b) illustrates that during the backward computation of the first epoch, Worker 0 and Worker 1 proceed without waiting for the gradients from the straggling Worker 2. Partial update collects the gradients from each worker based on corresponding weight values, which are computed from the number of training vertices participating in each layer. Specifically, a Breadth-First Search(BFS) traversal is initiated from the training vertices to collect the number

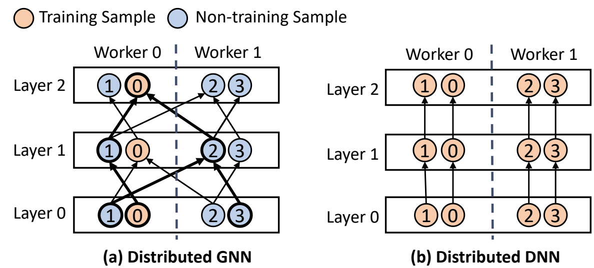


Figure 6: BFS-based partial update.

of neighbor vertices per worker at each layer. These counts are used to compute the normalization weights for gradient aggregation.

5.2 Partial Computation for Forward/Backward

We adopt a partial-computation mechanism in which workers use locally cached historical embeddings instead of waiting for remote embeddings when communication delays occur. In addition, we introduce a layer-wise timing constraint to limit the maximum runtime of each layer.

Concretely, for each GNN layer l, we estimate a baseline *runtime* $T_0^{(l)}$ from a short warm-up run of a few epochs and keep it fixed throughout training. We then set an adaptive slack $\Delta T_{\max}^{(l)} = \alpha T_0^{(l)}$, with $\alpha = 0.5$ by default, which serves as the maximum tolerable *waiting time* (elapsed runtime minus pure compute time) for remote vertex embeddings. Whenever a worker's elapsed time for layer l at an iteration exceeds $T_0^{(l)} + \Delta T_{\max}^{(l)}$, it stops waiting and instead uses *cached historical embeddings* for the unresolved vertices.

To ensure embedding version consistency, we design a dynamic staleness control mechanism to track the staleness of embeddings on each worker. The controller dynamically controls the maximum allowable delay for remote embedding synchronization. Specifically, when a worker's cached embeddings fall behind by more than K iterations, the system forces the worker to wait for the latest remote embeddings. This approach ensures that the embeddings used in the computation remain up-to-date, preventing approximation errors due to outdated embeddings.

5.3 Partial Update for Parameter Synchronization

Workflow of Partial Update. Partial update employs a dual mechanism of dynamic grouping and bounded staleness constraints, as shown in Figure 5(b). The lightweight controller monitors the per-iteration runtime (aggregated across all layers). Let T_0 denote the iteration-level baseline runtime (e.g., the sum of $T_0^{(l)}$ over layers) and let ΔT be the corresponding slack (e.g., αT_0). When the current iteration's elapsed time reaches $T_0 + \Delta T$, if the number of ready workers is $\geq M/2+1$ (M is the total number of workers), it triggers a weighted parameter synchronization within the temporary group of ready workers (e.g., workers 0–1 synchronize while bypassing worker 2). Otherwise, the controller continues waiting until the threshold is met.

To prevent parameter drift when some workers have not participated for a long time, all workers are required to join parameter synchronization every K iterations. The controller checks participation

every K iterations; if any worker is absent, the next synchronization round is performed across all workers in the cluster.

BFS-based Gradient Weighting Strategy. To address convergence degradation caused by varying worker participation across iterations and the uneven distribution of training vertices in partial update, we propose a BFS-based gradient weighting strategy. The key idea is to normalize each worker's gradient contribution according to the number of training-related vertices.

Let $n_k^{(l)}$ denote the number of vertices contributing to gradients at layer l on worker k. These vertices are determined through a backward BFS traversal starting from the training vertices. Each worker computes its local normalized gradient as:

$$g_k^{(l)} = \frac{1}{n_k^{(l)}} \sum_{v \in \mathcal{V}_k^{(l)}} \nabla \mathcal{L}_v, \tag{13}$$

where $\mathcal{V}_k^{(l)}$ is the set of training-related vertices on worker k at layer l, and $\nabla \mathcal{L}_v$ is the gradient of the loss with respect to vertex v. The parameter update rule then becomes:

$$\theta_{t+1}^{(l)} = \theta_t^{(l)} - \eta \cdot \frac{\sum_{k \in \mathcal{S}_t} n_k^{(l)} \cdot g_k^{(l)}}{\sum_{k \in \mathcal{S}_t} n_k^{(l)}}, \tag{14}$$

where η is the global learning rate and $\mathcal{S}_t$ is the set of workers active at iteration t. This dependency-aware normalization adjusts each worker's contribution by the number of vertices involved in the backward computation (training vertices and their dependencies), stabilizes gradient scaling across iterations, and improves the convergence rate under dynamic worker participation.

Bias Elimination under Partial-reduce. We demonstrate that the proposed vertex-weighted gradient aggregation yields an unbiased estimate of the global gradient. We take expectations on both sides and assume the vertex-level gradients are unbiased, i.e., $\mathbb{E}[g_k^{(l)}] = \nabla \mathcal{L}(\theta_t)$, we have:

$$\begin{aligned} \mathbb{E}\left[\frac{\sum_{k \in \mathcal{S}_t} n_k^{(l)} \cdot g_k^{(l)}}{\sum_{k \in \mathcal{S}_t} n_k^{(l)}}\right] &= \frac{\sum_{k \in \mathcal{S}_t} n_k^{(l)} \cdot \mathbb{E}[g_k^{(l)}]}{\sum_{k \in \mathcal{S}_t} n_k^{(l)}} \\ &= \frac{\sum_{k \in \mathcal{S}_t} n_k^{(l)} \cdot \nabla \mathcal{L}(\theta_t)}{\sum_{k \in \mathcal{S}_t} n_k^{(l)}} = \nabla \mathcal{L}(\theta_t). \end{aligned} \tag{15}$$

In contrast, traditional GNN training directly sums local gradients across active workers without normalization, which leads to biased gradient estimates when only a subset of training vertices is covered at each iteration. The expected aggregated gradient becomes:

$$\mathbb{E}\left[\sum_{k \in \mathcal{S}_t} g_k^{(l)}\right] = \frac{N_t^{(l)}}{N^{(l)}} \nabla \mathcal{L}(\theta_t), \tag{16}$$

where $N^{(l)} = \sum_{k=1}^{K} n_k^{(l)}$ is the total number of training vertices at layer l, and $N_t^{(l)} = \sum_{k \in \mathcal{S}_t} n_k^{(l)}$ is the number of vertices contributing to gradients at layer l among all workers involved in gradient aggregation. The factor $N_t^{(l)}/N^{(l)}$ reflects incomplete training signal coverage and causes systematic underestimation of the true gradient.

Table 1: Dataset description.

Dataset	|V|	|E|	#$\mathbb{F}$	#$\mathbb{L}$	#$\mathbb{H}$
Yelp (YP)	716,874	13,954,819	300	100	128
Reddit (RDT)	232,965	114,615,892	602	41	128
Ogbn-products (OPT)	2,449,029	61,859,140	100	47	64
Amazon (AMZ)	1,598,960	132,169,734	200	107	128

Therefore, by incorporating dependency-aware normalization, our method eliminates this bias and ensures unbiased gradient estimation under resource fluctuation and imbalanced participation. The convergence of the method is formally established, and the detailed proof is provided in the technical report [4] due to space constraints.

6 EXPERIMENTS

6.1 Experimental Setup

Environments. Our experiments are conducted on Aliyun ECS cluster with 16 GPU nodes. Each node has 16 vCPUs, 155GB DRAM, and 1 NVIDIA Tesla T4 GPU, running Ubuntu 20.04 LTS OS. The network bandwidth is 10 Gbps.

Datasets and GNN Algorithms. Table 1 lists the four graph datasets used in our evaluation: Reddit [13] is based on user interactions in a social network. Ogbn-products [15] originate from similar relationships between products in an e-commerce platform. The Yelp [50] dataset is constructed from user reviews of local businesses. The Amazon [6] dataset is derived from product co-purchasing graphs, with nodes representing products and edges capturing frequently co-reviewed or co-purchased items. The vertex feature dimensions, the number of labels of datasets, and hidden layer dimensions are listed in Table 1. We use four popular GNN models, including Graph Convolutional Network (GCN) [21], Graph Attention Network (GAT) [39], GraphSAGE [13], and TAGCN [9] to evaluate the performance, all of them are in a 2-layer structure.

The Systems for Comparisons. In our experiments, we compare NeutronCloud with two types of systems: mini-batch systems and full-graph systems. For the mini-batch system, we choose DistDGL [55] as the baseline. DistDGL reduces computational and memory overhead through sampling. In our experiment, DistDGL samples up to 10 first-hop neighbors per seed node and up to 15 second-hop neighbors per first-hop neighbor. For the full-graph system, we compare NeutronCloud with NeutronStar [45] and Sancus [32].NeutronStar adopts a hybrid dependency-handling approach designed to balance computation and communication workload, enabling high-performance GNN training. Sancus uses historical embeddings to reduce cross-worker communication. In NeutronCloud, we follow the graph partitioning strategy used in NeutronStar, adopting a chunk-based [58] approach that divides the vertex ID space into contiguous ranges. Vertex features and labels are colocated with their corresponding vertices, while edges are assigned to partitions based on their destination vertex. By default, we set the staleness bound $K = 3$ in both accuracy and runtime performance evaluations, meaning that each worker is allowed to compute with cached embeddings and delay gradient synchronization for up to 3 epochs. All experimental results are

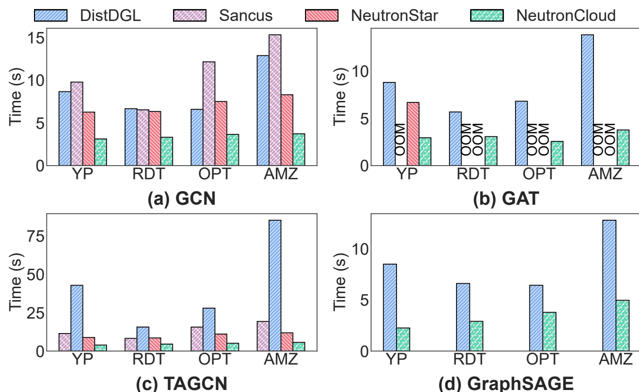


Figure 7: Per-epoch runtime of different systems on different models. "OOM" indicates the out-of-memory error.

presented in terms of the runtime per epoch, which refers to the time for forward and backward propagations, of all vertices in the graph. The results are obtained by averaging over 100 epochs. A shorter runtime per epoch indicates that the model takes less time to achieve the same accuracy.

Heterogeneity Simulation. Inspired by existing heterogeneous training methods [25–27, 56], we simulate a resource-fluctuating environment in the experiment. Specifically, for each worker, we independently add a certain probability of sleep time in each epoch to reflect resource dynamic. Specifically, each worker has a 10% probability of adding 5 seconds of sleep time within an epoch, which is partially added to both forward and backward propagations.

6.2 Overall Comparison

We compare the performance of NeutronCloud by running GCN, GAT, GraphSAGE, and TAGCN on a 16-node cluster. Recording the average runtime and the average increased training time caused by resource fluctuations (called **fluctuation-induced delay** time) per epoch. The experimental results are summarized in Figure 7 and Figure 8 respectively.

Per-epoch Time Comparison. Compared to NeutronStar, DistDGL, and Sancus, NeutronCloud demonstrates superior performance across all datasets. For the average runtime per epoch, it also achieves speedups of up to 2.10×, 2.97× and 4.15× (Figure 7).

Sancus periodically performs a full broadcast of the local embeddings of each worker to all workers, usually every few iterations, regardless of whether other partitions actually need them. This design introduces substantial redundant communication and lengthens waiting times, further degrading end-to-end performance. Thus bandwidth fluctuations exacerbate delays and harm training efficiency. NeutronStar adopts a pipeline parallelism strategy. In each worker, the workload is partitioned into multiple chunks, and chunk-level scheduling is applied to overlap the communication and compute tasks. It exhibits better adaptability and performance compared to Sancus and DistDGL in most datasets under resource-fluctuating environment.

The static workload allocation method used by DistDGL, NeutronStar and Sancus cannot match fluctuating available resources, leading to mismatches between workload and available resources. The advantages of NeutronCloud stem from two main factors: (1) the resource-aware workload adjustment strategy enables dynamic changes in workload to match the continuously changing resources;

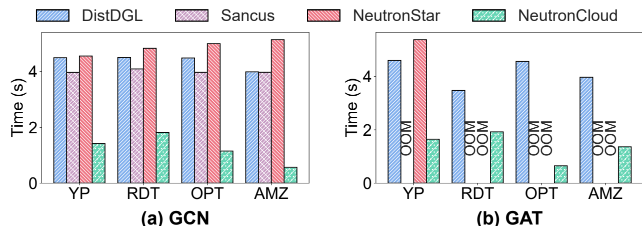


Figure 8: Fluctuation-induced delay time of different systems on GCN and GAT models. "OOM" indicates the out-of-memory error.

(2) the proposed dependency-aware partial-reduce method, which effectively alleviates the negative impact of extreme resource fluctuations on training efficiency.

Fluctuation-induced Delay Time Comparison. The fluctuation-induced delay time can directly reflect the impact of resource variability on distributed GNN training. For the average fluctuation-induced delay time per epoch, compared to DistDGL, NeutronStar, and Sancus, NeutronCloud achieves speedups of 3.89×, 4.81× and 3.87×, respectively, as shown in Figure 8. Sancus reduces communication overhead by reusing historical embeddings, making it less affected by communication resource fluctuations compared to DistDGL and NeutronStar.

6.3 Preprocessing Overhead Analysis

We compare the preprocessing time, including prioritizing dependent vertices through sorting and counting the number of training vertices, to the execution time of running GCN for 100 epochs. Table 2 shows the results. The preprocessing phase adds an average overhead of 3.91% to NeutronCloud. This overhead gradually decreases as the graph size increases (e.g., 5.66% on Yelp vs. 2.84% on Amazon). And this is typically amortized during training, while the techniques applied in this phase contribute to an average speedup of 2.08× for NeutronCloud.

Table 2: Preprocessing vs. training time breakdown (seconds and portion of total) for 100 epochs. Baseline denotes the training time before applying our approach.

Dataset	Training(Baseline)	Preprocessing	Training	Total
Yelp	675.00	18.74 / 5.66%	312.00 / 94.34%	330.74
Reddit	683.00	16.62 / 4.77%	332.00 / 95.23%	348.62
Products	750.00	8.94 / 2.39%	365.00 / 97.61%	373.94
Amazon	879.00	10.86 / 2.84%	372.00 / 97.16%	382.86

6.4 Scalability Analysis

Performance with varying cluster sizes. In this experiment, we compare NeutronCloud with other systems when training GCN on four datasets across varying cluster sizes. Figure 9 reports per-epoch runtime across cluster sizes, where NeutronCloud consistently outperforms other systems. Specifically, compared to DistDGL, NeutronStar, and Sancus, NeutronCloud achieves average speedup ratios of 2.53×-2.65× when scaling from 4 to 16 workers.

As the number of workers increases, the overall training time of distributed GNNs is expected to decrease. However, for full-graph systems such as Sancus and NeutronStar, the training time not only fails to reduce but instead increases due to resource fluctuations. In contrast, NeutronCloud experiences a slight reduction in training time because dynamic workload allocation methods

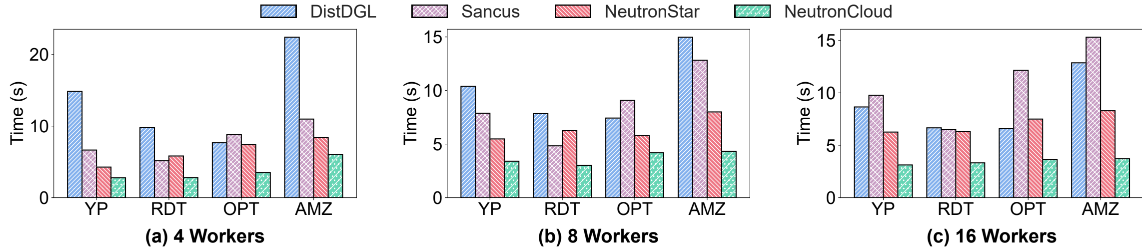


Figure 9: Per-epoch runtime of different systems with different cluster sizes on different datasets.

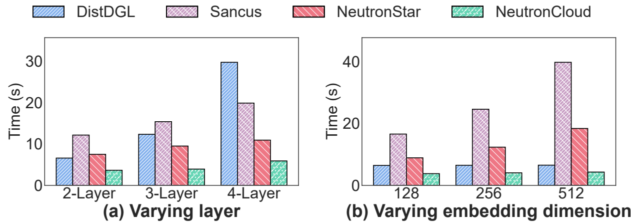


Figure 10: Per-epoch runtime of different systems on Ogbn-Products with varying model configurations.

can better match fluctuating available resources. Sancus exhibits poor scalability, possibly due to reliance on serial global broadcasting and broad, repeated embedding transfers. In large-scale clusters, growing workers amplify communication overhead and network bottlenecks, which limits Sancus's scalability. In contrast, NeutronCloud employs a dependency-aware partial-reduce method that better handles severe stragglers under resource fluctuations in large-scale clusters.

Performance with varying model layers. In this experiment, we compare NeutronCloud with baselines when training GCN with different model layers over Ogbn-products in a 16-node cluster. For the 2, 3, and 4-layer models, the DistDGL sampling strategies were set to (25,10), (25,15,10), and (25,20,15,10), respectively. The results are shown in Figure 10(a) We observe that the performance advantage of NeutronCloud over other baselines gradually increases with the model depths. For the 2-layer model, NeutronCloud achieves an average speedup of 2.40×. For the 3-layer and 4-layer models, the speedups were 3.15× and 3.42×, respectively.

Performance with varying embedding dimensions. In this experiment, we compare NeutronCloud with baselines when training GCN with different embedding dimensions on a 16-node cluster. As shown in Figure 10(b), NeutronCloud consistently outperforms baselines, and its advantage increases with the embedding size. Specifically, NeutronCloud achieves average speedups of 2.81×, 3.57×, and 5.02× over all baselines for 128-, 256-, and 512-dimensional embeddings, respectively. Larger embedding dimensions increase communication volume; NeutronCloud reduces this overhead by (i) limiting embedding exchange via partial-reduce and (ii) shifting dependency communication to local computation through resource-aware adjustment.

6.5 Performance Tolerance to Parameter Staleness (K)

We study GNN training's tolerance to parameter staleness. In our setting, historical embeddings are reused for computation, and only a subset of gradients participates in parameter synchronization. We evaluate the convergence performance of the systems on a cluster with 16 workers and show the accuracy curves for the GNN models (GCN, TAGCN) on NeutronCloud, NeutronStar, DistDGL, and Sancus under different parameters K, as shown in Figure 11. Results indicate that increasing K reduces average per-epoch time, with a modest decrease in accuracy; when the value of parameter K is 3, the model achieves a relatively ideal balance between accuracy and training overhead.

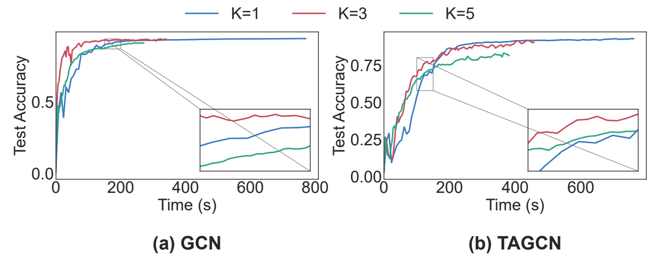


Figure 11: Accuracy for different parameters K.

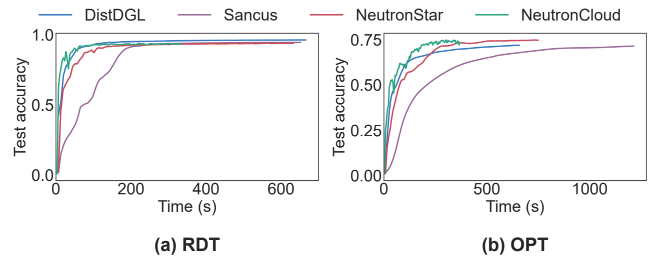


Figure 12: Time-to-accuracy.

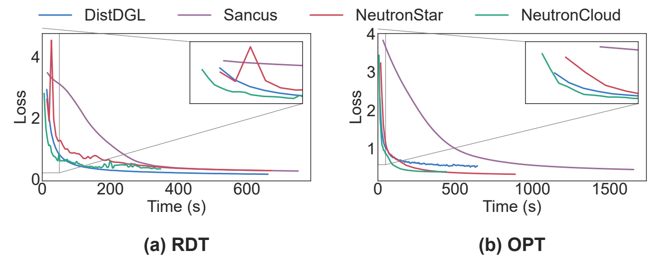


Figure 13: Time-to-loss.

6.6 Convergence Analysis

We evaluate the convergence of NeutronCloud, NeutronStar, DistDGL, and Sancus on a 16-worker cluster using a GCN model for node classification, and show the accuracy curves on two datasets in Figure 12.

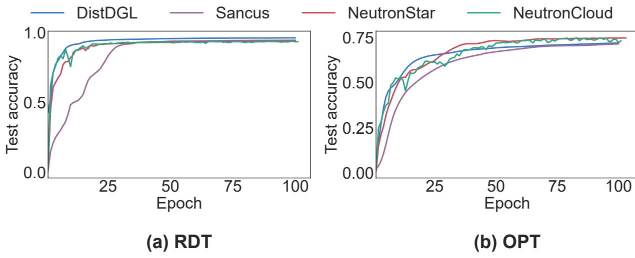


Figure 14: Epoch-to-accuracy.

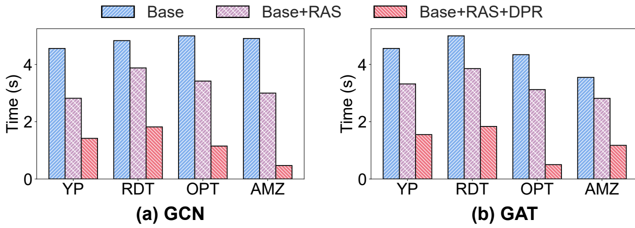


Figure 15: Performance gain analysis. "RAS" indicates the resource-aware workload adjustment strategy, "DPR" indicates the dependency-aware partial-reduce strategy.

The results show that in the early stages, NeutronCloud experiences accuracy fluctuations caused by severe stragglers. After 100 epochs, the test accuracy stabilizes, and NeutronCloud achieves the same test accuracy compared to other systems while reaching the target accuracy faster than all other systems. Similarly, Figure 13 presents the training loss over time. At convergence, NeutronCloud attains a training loss comparable to that of the baselines.

However, Sancus exhibits a long convergence time, which may be due to its cross-worker communication. It does not immediately transmit vertex embeddings after each update but instead transmits them at fixed epoch intervals. NeutronCloud employs a bounded control mechanism in partial computation, limiting the staleness of historical embeddings to at most K iterations, ensuring that deviation does not accumulate indefinitely. Additionally, periodic All-Reduce in partial update is performed, ensuring that all workers complete parameter synchronization within K epochs, preventing the model from degrading due to severe stragglers. As a result, NeutronCloud achieves higher accuracy.

Figure 14 shows the accuracy curves of the GCN model on NeutronStar, NeutronCloud, DistDGL, and Sancus across training epochs. We observe that under the same number of epochs, NeutronCloud achieves slightly lower model accuracy compared to other systems, but it eventually achieves convergence accuracy comparable to that of other systems. This is because it uses historical (stale) embeddings for forward computation and performs parameter synchronization only among a subset of workers in each iteration. However, since NeutronCloud executes each epoch faster, the system still achieves faster convergence in terms of overall training time.

6.7 Performance Gain Analysis

Training efficiency analysis. To validate the effectiveness of NeutronCloud's key designs, we conduct experiments on GNN models

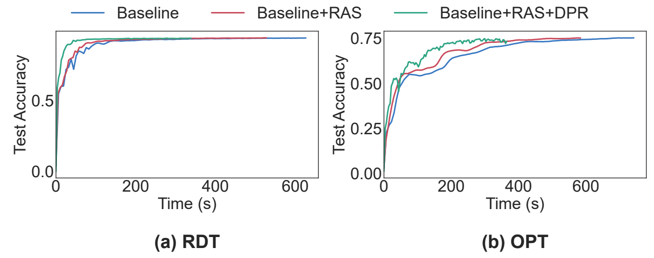


Figure 16: Test accuracy over time for the RDT and OPT datasets under three settings: Baseline, Baseline+RAS, and Baseline+RAS+DPR.

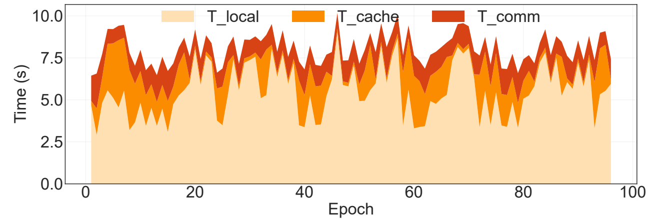


Figure 17: The distribution of overhead associated with T_local, T_cache, and T_comm during a 100-epoch training.

(GCN, GAT) across four datasets, evaluating the impact of resource-aware workload adjustment strategy (RAS) and dependency-aware partial-reduce (DPR) on system efficiency. To ensure a fair comparison, we start with a foundational framework established on the NeutronCloud codebase and gradually integrate the two optimization methods.

Figure 15 shows the average fluctuation-induced delay time per epoch. Compared to the `Baseline`, the `Baseline+RAS` achieves an average speedup of 1.40×. Figure 17 shows the per-epoch distribution of T_{local}, T_{cache}, and T_{comm} during a 100-epoch training of GCN on the Amazon dataset under `Baseline+RAS`. And the T_{local} includes the injected sleep time used to simulate resource fluctuations. By dynamically rebalancing the workload between DepCache and DepComm, NeutronCloud effectively mitigates the performance degradation introduced by these fluctuations.

Compared to the `Baseline+RAS`, the `Baseline+RAS+DPR` achieves an average speedup of 2.64×. The dependency-aware partial-reduce strategy includes two levels of mechanisms, reducing the impact of severe stragglers on overall training progress from both layer-wise synchronization and parameter synchronization dimensions under extreme conditions. It significantly reduces the prolonged synchronization overhead caused by severe stragglers.

Accuracy evaluation and analysis. To validate the effectiveness of NeutronCloud's designs, we evaluate the convergence performance of the `Baseline` and two optimization variants. Figure 16 shows the accuracy curves of the GCN model on the Reddit and Ogbn-products datasets. After 100 epochs, the test accuracy stabilizes, and both `Baseline+RAS+DPR` and `Baseline+RAS` achieve accuracy comparable to that of the `Baseline`.

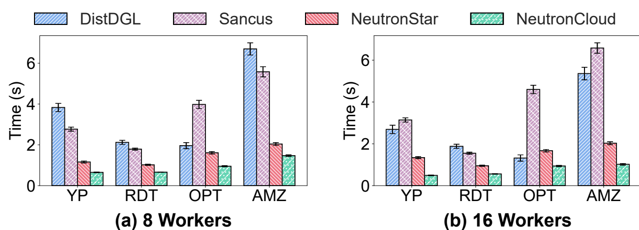


Figure 18: Per-epoch runtime comparison under real cloud heterogeneous setting.

6.8 Performance on Real Cloud GPUs

To evaluate our method in a real heterogeneous environment, we further conduct experiments on commercial cloud platforms that support GPU sharing. With the improvement of single-GPU performance, public GPU cloud providers have started offering virtual GPU containers, such as Vultr[40] and Alibaba Cloud[7], allowing multiple users to share the same physical GPU resources. By leveraging GPU virtualization technologies such as NVIDIA MIG[29] or SR-IOV[30], these platforms allocate independent computing instances to each user. These cloud providers typically allow users to request GPU resources on demand, renting computing power at a finer granularity (e.g., 1/2, 1/4, or even 1/20 of a single GPU). This approach improves the overall utilization of GPU resources while effectively reducing user costs.

Specifically, we launch 16 GPU instances on Alibaba GPU cloud provider to validate the adaptability of our method, each instance provisions a virtual GPU container with 1/3 of an NVIDIA A10 GPU and up to 20 Gbps network bandwidth. Since the computational performance of shared GPUs is affected by cluster scheduling policies, we observed significant dynamic heterogeneity among these instances. We run the GCN model on four datasets to compare the execution speed of NeutronCloud with other systems. To account for the variability of real cloud environments and ensure statistical reproducibility, we repeat each experiment multiple times across three independent periods and report the average per-epoch runtime. Figure 18 presents the experimental results with error bars that indicate the standard deviation between numerous experiments. The result demonstrates that NeutronCloud achieves stable performance across repeated trials, confirming the reproducibility of our results. Compared to DistDGL, NeutronStar, and Sancus, NeutronCloud achieves average speedup ratios of 3.90×, 1.83×, and 4.43×, respectively.

7 RELATED WORK

Existing Distributed GNN Systems. Several distributed GNN training systems have been proposed to address the scalability challenge of graph learning. Sancus [32] reduces synchronization overhead by using historical embeddings, but adopts a broadcast-based communication scheme where each worker sequentially sends all local embeddings to all others, regardless of actual demand. This incurs redundant communication and prolonged synchronization delays. NeutronStar [45] performs full-graph training with pipeline parallelism and chunk-level scheduling to accelerate training by overlapping communication and computation. However, it cannot adapt to environments with fluctuating resources due to the parameter synchronization approach of All-Reduce and static load distribution. DistDGL [55] adopts mini-batch training and reduces overhead through neighborhood sampling. However, its reliance on static METIS [18] partitioning can cause load imbalance under resource fluctuations, leading to unstable performance in resource-dynamic environments.

The Partial-Reduce Strategy for Distributed DNNs. Deploying deep learning workloads in heterogeneous environments (e.g., public clouds) exacerbates communication latency and straggler effects. Synchronous All-Reduce, originally designed for homogeneous clusters, becomes a performance bottleneck under such heterogeneity. To mitigate stragglers, recent work relaxes strict synchronization: *partial-reduce* [26] skips delayed workers during gradient aggregation; *DPAR* [20] adapts synchronization to node capacity; and *RNA* [48] allows nodes to update at their own pace. These relaxations reduce computation and communication overhead and improve throughput for distributed DNNs. While effective for distributed DNNs, these techniques cannot be directly applied to GNNs due to inter-sample dependencies between workers.

Distributed DNNs in Resource-fluctuating Environments. In multi-tenant cloud environments and shared clusters, compute and network capacity fluctuate over time, making fixed task assignment and strict synchronization ineffective. SDPipe [27] introduces a semidecentralized framework that couples heterogeneity-aware scheduling with dynamic gradient synchronization, adapting computation and communication to instantaneous worker capacity. This reduces straggler-induced stalls and synchronization latency and improves pipeline utilization for model-parallel DNNs. However, the design does not carry over to GNNs: models are small and trained in a data-parallel regime, and the dominant cost is dependency-driven communication (cross-partition neighbor-feature retrieval and consistency), not gradient synchronization. Consequently, SDPipe yields limited benefit for GNNs unless combined with mechanisms that reduce dependency communication.

8 CONCLUSION

We present NeutronCloud, a system designed for efficient GNN training in cloud environments with dynamic and fluctuating resources. Its performance and adaptability are enabled by two key components: (1) a resource-aware workload adjustment strategy that dynamically matches computational and communication workloads to real-time resource conditions, and (2) a dependency-aware partial-reduce strategy that reuses historical vertex embeddings and skips stragglers during gradient aggregation to improve training efficiency. Compared with existing distributed GNN systems such as DistDGL and Sancus, NeutronCloud achieves end-to-end training speedups ranging from 1.83× to 4.43× in the real cloud environments.

ACKNOWLEDGMENTS

This work is supported by the National Key R&D Program of China (2023YFB4503601), the National Natural Science Foundation of China (62461146205, U2241212, 62572108), the Distinguished Youth Foundation of Liaoning Province (2024021148-JH3/501), and the Liaoning Province Key R&D Program Project (2019JH2/10100027). Yanfeng Zhang is the corresponding author.

REFERENCES

[1] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. 2022. Varuna: scalable, low-cost training of massive deep learning models. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*. ACM, 472–487.

[2] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: an efficient communication library for distributed GNN training. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*. ACM, 130–144.

[3] Chunyu Cao, Xin Ai, Qiange Wang, Yanfeng Zhang, Zhenbo Fu, Hao Yuan, Mingyi Cao, Chaoyi Chen, Yingyou Wen, Yu Gu, and Ge Yu. 2025. NeutronHeter: Optimizing Distributed Graph Neural Network Training for Heterogeneous Clusters. *Proc. ACM Manag. Data* 3, 4, Article 257 (Sept. 2025), 29 pages. https://doi.org/10.1145/3749175

[4] Mingyi Cao, Chunyu Cao, Yanfeng Zhang, Zhenbo Fu, Xin Ai, Qiange Wang, Yu Gu, and Ge Yu. 2025. NeutronCloud: Resource-Aware Distributed GNN Training in Fluctuating Cloud Environments (Technical Report). https://github.com/Toaoc/NeutronCloud/blob/main/NeutronDyn_TechnicalReport.pdf. Technical Report.

[5] Sandro Cavallari, Vincent W. Zheng, Hongyun Cai, Kevin Chen-Chuan Chang, and Erik Cambria. 2017. Learning Community Embedding with Community Detection and Node Embedding on Graphs. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*. ACM, 377–386.

[6] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 257–266.

[7] Alibaba Cloud. 2025. VGPU Accelerated Instance Families - Elastic GPU Service. https://www.alibabacloud.com/help/en/elastic-gpu-service/latest/vgpu-accelerated-instance-families. Accessed: 2025-03-31.

[8] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*. 1232–1240.

[9] Jian Du, Shanghang Zhang, Guanhang Wu, José M. F. Moura, and Soummya Kar. 2017. Topology adaptive graph convolutional networks. *CoRR* abs/1710.10370 (2017).

[10] Euler 2019. Euler. https://github.com/alibaba/euler/wiki/System-Introduction.

[11] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Yihong Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph Neural Networks for Social Recommendation. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*. ACM, 417–426.

[12] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 551–568.

[13] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. 1024–1034.

[14] Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2016. Addressing the straggler problem for iterative convergent parallel ML. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC '16)*. Association for Computing Machinery, New York, NY, USA, 98–111.

[15] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

[16] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2021. Understanding and bridging the gaps in current GNN performance optimizations. In *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*. ACM, 119–132.

[17] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *Proceedings of the Third Conference on Machine Learning and Systems, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org.

[18] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20, 1 (1998), 359–392.

[19] Samir Khuller, Anna Moss, and Joseph Naor. 1999. The Budgeted Maximum Coverage Problem. *Inf. Process. Lett.* 70, 1 (1999), 39–45.

[20] HyungJun Kim, Chunggeon Song, HwaMin Lee, and Heonchang Yu. 2023. Addressing Straggler Problem Through Dynamic Partial All-Reduce for Distributed Deep Learning in Heterogeneous GPU Clusters. In *IEEE International Conference on Consumer Electronics, ICCE 2023, Las Vegas, NV, USA, January 6-8, 2023*. IEEE, 1–6.

[21] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

[22] Jérôme Kunegis and Andreas Lommatzsch. 2009. Learning spectral graph transformations for link prediction. In *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009 (ACM International Conference Proceeding Series)*, Vol. 382. ACM, 561–568.

[23] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. USENIX Association, 583–598.

[24] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. PaGraph: Scaling GNN training on large graphs via computation-aware caching. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*. ACM, 401–415.

[25] Qinyi Luo, Jiaao He, Youwei Zhuo, and Xuehai Qian. 2020. Prague: High-Performance Heterogeneity-Aware Asynchronous Decentralized Training. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*. ACM, 401–416.

[26] Xupeng Miao, Xiaonan Nie, Yingxia Shao, Zhi Yang, Jiawei Jiang, Lingxiao Ma, and Bin Cui. 2021. Heterogeneity-Aware Distributed Machine Learning Training via Partial Reduce. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 2262–2270.

[27] Xupeng Miao, Yining Shi, Zhi Yang, Bin Cui, and Zhihao Jia. 2023. SDPipe: A Semi-Decentralized Framework for Heterogeneity-aware Pipeline-parallel Training. *Proc. VLDB Endow.* 16, 9 (2023), 2354–2363.

[28] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. 2021. Synergy: Resource Sensitive DNN Scheduling in Multi-Tenant Clusters. *CoRR* abs/2110.06073 (2021).

[29] NVIDIA. 2025. Multi-Instance GPU (MIG) Technology. https://www.nvidia.com/en-sg/technologies/multi-instance-gpu/. Accessed: 2025-03-31.

[30] PCI-SIG. 2010. Single Root I/O Virtualization (SR-IOV) Specification. https://pcisig.com/specifications/iov. Accessed: 2025-03-31.

[31] Hao Peng, Jianxin Li, Yu He, Yaopeng Liu, Mengjiao Bao, Lihong Wang, Yangqiu Song, and Qiang Yang. 2018. Large-Scale Hierarchical Text Classification with Recursively Regularized Deep Graph-CNN. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*. ACM, 1063–1072.

[32] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. 2023. Sancus: Staleness-Aware Communication-Avoiding Full-Graph Decentralized Training in Large-Scale Graph Neural Networks (Extended Abstract). In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*. ijcai.org, 6480–6485.

[33] Benjamin Recht, Christopher Ré, Stephen J. Wright, and Feng Niu. 2011. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain*. 693–701.

[34] Jaeyong Song, Hongsun Jang, Hunseong Lim, Jaewon Jung, Youngsok Kim, and Jinho Lee. 2024. GraNNDis: Fast Distributed Graph Neural Network Training Framework for Multi-Server Clusters. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques, PACT 2024, Long Beach, CA, USA, October 14-16, 2024*. ACM, 91–107.

[35] Yujia Song, Ruyue Xin, Peng Chen, Rui Zhang, Juan Chen, and Zhiming Zhao. 2023. Identifying performance anomalies in fluctuating cloud environments: A robust correlative-GNN-based explainable approach. *Future Gener. Comput. Syst.* 145 (2023), 77–86.

[36] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. 2023. Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*. USENIX Association, 497–513.

[37] Alok Tripathy, Katherine A. Yelick, and Aydin Buluç. 2020. Reducing communication in graph neural network training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*. IEEE/ACM, 70.

[38] Md. Vasimuddin, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj D. Kalamkar, Nesreen K. Ahmed, and

Sasikanth Avancha. 2021. DistGNN: scalable distributed training for large-scale graph neural networks. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*. ACM, 76.

[39] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

[40] Vultr. 2025. Vultr: High Performance SSD Cloud. Retrieved from https://www.vultr.com. Accessed: 2025-03-31.

[41] Cheng Wan, Youjie Li, Cameron R. Wolfe, Anastasios Kyrillidis, Nam Sung Kim, and Yingyan Lin. 2022. PipeGCN: Efficient Full-Graph Training of Graph Convolutional Networks with Pipelined Feature Communication. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*.

[42] Xinchen Wan, Kaiqiang Xu, Xudong Liao, Yilun Jin, Kai Chen, and Xin Jin. 2023. Scalable and Efficient Full-Graph GNN Training for Large Graphs. *Proc. ACM Manag. Data* 1, 2 (2023), 143:1–143:23.

[43] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. *CoRR* abs/1909.01315 (2019).

[44] Qiange Wang, Yao Chen, Weng-Fai Wong, and Bingsheng He. 2023. HongTu: Scalable Full-Graph GNN Training on Multiple GPUs. *Proc. ACM Manag. Data* 1, 4 (2023), 246:1–246:27.

[45] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. 2022. NeutronStar: Distributed GNN Training with Hybrid Dependency Management. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 1301–1315.

[46] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Kevin J. Barker, Ang Li, and Yufei Ding. 2023. MGG: Accelerating Graph Neural Networks with Fine-Grained Intra-Kernel Communication-Computation Pipelining on Multi-GPU Platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*. USENIX Association, 779–795.

[47] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*. USENIX Association, 945–960.

[48] Donglin Yang, Wei Rang, and Dazhao Cheng. 2020. Mitigating Stragglers in the Decentralized Training on Heterogeneous Clusters. In *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*. ACM, 386–399.

[49] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*. ACM, 974–983.

[50] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K. Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

[51] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. 2020. AGL: A Scalable System for Industrial-purpose Graph Machine Learning. *Proc. VLDB Endow.* 13, 12 (2020), 3125–3137.

[52] Muhan Zhang and Yixin Chen. 2018. Link Prediction Based on Graph Neural Networks. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. 5171–5181.

[53] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. 2022. Multi-resource interleaving for deep learning training. In *SIGCOMM '22: ACM SIGCOMM 2022 Conference, Amsterdam, The Netherlands, August 22 - 26, 2022*. ACM, 428–440.

[54] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. 2022. ByteGNN: Efficient Graph Neural Network Training at Large Scale. *Proc. VLDB Endow.* 15, 6 (2022), 1228–1242.

[55] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, and George Karypis. 2022. Distributed Hybrid CPU and GPU training for Graph Neural Networks on Billion-Scale Heterogeneous Graphs. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022*. ACM, 4582–4591.

[56] Qihua Zhou, Song Guo, Haodong Lu, Li Li, Minyi Guo, Yanfei Sun, and Kun Wang. 2021. Falcon: Addressing Stragglers in Heterogeneous Parameter Server Via Multiple Parallelism. *IEEE Trans. Comput.* 70, 1 (2021), 139–155.

[57] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. *Proc. VLDB Endow.* 12, 12, 2094–2105.

[58] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 301–316.

[59] Martin Zinkevich, Markus Weimer, Alexander J. Smola, and Lihong Li. 2010. Parallelized Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems 2010. Proceedings of a meeting held 6-9 December 2010, Vancouver, British Columbia, Canada*. Curran Associates, Inc., 2595–2603.

CLaP - State Detection from Time Series

Arik Ermshaus
Humboldt-Universität zu Berlin
Berlin, Germany
ermshaua@informatik.hu-berlin.de

Patrick Schäfer
Humboldt-Universität zu Berlin
Berlin, Germany
patrick.schaefer@hu-berlin.de

Ulf Leser
Humboldt-Universität zu Berlin
Berlin, Germany
leser@informatik.hu-berlin.de

ABSTRACT

The ever-growing amount of sensor data from machines, smart devices, and the environment leads to an abundance of high-resolution, unannotated time series (TS). These recordings encode recognizable properties of latent states and transitions from physical phenomena that can be modelled as abstract processes. The unsupervised localization and identification of these states and their transitions is the task of time series state detection (TSSD). Current TSSD algorithms employ classical unsupervised learning techniques, to infer state membership directly from feature space. This limits their predictive power, compared to supervised learning methods, which can exploit additional label information. We introduce CLaP, a new, highly accurate and efficient algorithm for TSSD. It leverages the predictive power of time series classification for TSSD in an unsupervised setting by applying novel self-supervision techniques to detect whether data segments emerge from the same state. To this end, CLaP cross-validates a classifier with segment-labelled subsequences to quantify confusion between segments. It merges labels from segments with high confusion, representing the same latent state, if this leads to an increase in overall classification quality. We conducted an experimental evaluation using 405 TS from five benchmarks and found CLaP to be significantly more precise in detecting states than six state-of-the-art competitors. It achieves the best accuracy-runtime tradeoff and is scalable to large TS. We provide a Python implementation of CLaP, which can be deployed in TS analysis workflows.

PVLDB Reference Format:
Arik Ermshaus, Patrick Schäfer, and Ulf Leser. CLaP - State Detection from Time Series. PVLDB, 19(1): 70 - 83, 2025.
doi:10.14778/3772181.3772187

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ermshaua/classification-label-profile.

1 INTRODUCTION

The study and analysis of long-running biological, human-controlled, or physical processes, such as human activities or industrial manufacturing, is of great interest in their respective domains. The field of human activity recognition, for instance, aims to detect falls in the elderly [71]. To achieve this, human motions are conceptualized as an abstract process of distinct activities that transition from one to another. Data acquisition and analysis workflows are employed to detect activities from mobile sensors and report falls. Similarly, in industrial manufacturing, CNC machines are monitored by pre-installed IoT devices to detect tool wear [60]. The condition of a tool can be abstracted as either normal or faulty, and wear detection is realized by recognising the transitions in-between. From a computational perspective, such phenomena can be modelled as abstract processes with discrete states and pairwise transitions between them. The core property of these processes is that the states are distinct and can be observed and measured over time by instrumentation.

Recent advances in the digitalization of measurement devices have greatly facilitated the acquisition of biological and physical process data. Sensors, such as those in smartphones, industrial machinery, or earth observation stations, continuously record high-frequency real-valued observational data, termed *time series* (TS) [34, 43, 69]. State changes in the observed processes lead to variations in the recorded signals, which form the basis for analysis. Sensor data capture the unique characteristics of states as subsequences, distinguishable by either their shape or statistical properties. These distinct parts of TS are commonly called *segments*, and each contains the recognizable properties of the observed process state. The transitions between segments, also called *change points* (CPs), mark the time points in a recording, at which the observed process changes state. The identification of segments, CPs, and state labels forms the foundation for activity recognition [18], health assessment [38], or condition monitoring in IoT [60].

Formally, the task of recovering the sequence of discrete state labels from a TS of observations is called time series state detection (TSSD). It acts as a complex, unsupervised preprocessing step between data collection and TS knowledge discovery [65]. TSSD annotates each data point in a TS with a label that corresponds to a state in the data-generating process. Consecutive stretches of the same label mark segments, while different neighbouring labels indicate CPs. TSSD requires the analysis of observations to identify signal shifts, which are assumed to result from state changes in the observed process. This creates consecutive segments of data points that are homogeneous within themselves, yet sufficiently distinct from their neighbours. These segments are further compared to each other to assign equal labels to those sharing the same state and unequal labels to those representing different states. State labels from segments are then propagated to individual data points to form the annotation, a state sequence. Contrary to supervised problems, such as TS classification (TSC), the labels in TSSD (e.g. 0,1,2) are discovered by an algorithm, abstract, and primarily used to differentiate observations based on state affiliation. They can be mapped to semantic labels (e.g. walk, jog, run) when appropriate domain knowledge is available [2].

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 1 ISSN 2150-8097.
doi:10.14778/3772181.3772187

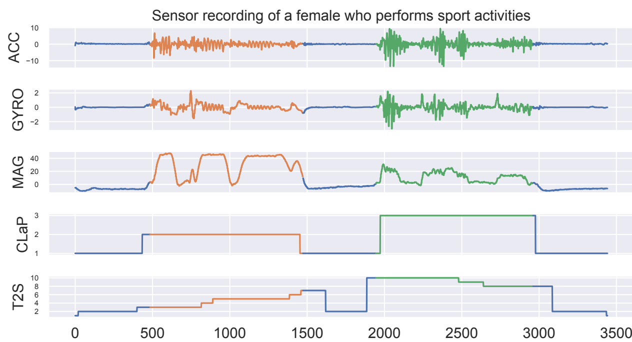


Figure 1: Top-3: human activity recording showing X-axis acceleration, gyroscope, and magnetometer measurements [18] of different motions. Activities are consecutive, differently coloured sequences. 2nd from bottom: state sequence predicted by CLaP, assigning each data point one of three possible activity labels (1 for resting, 2 for squats, and 3 for loosening legs). Bottom: state sequence from competitor method Time2State, with 10 (too many) predicted labels.

A popular approach to TSSD is a combination of TS segmentation (TSS) and clustering techniques [65]. A baseline approach could, for example, use the ClaSP (Classification Score Profile) algorithm [20] to compute a TSS and cluster the resulting segments using Time2Feat [6] to predict state labels. The central limitation of this pipeline is its reliance on distance calculations in clustering, which are known to show limited performance compared to supervised methods used in TSC [49]. Unsupervised distance-based methods determine the state membership of segments solely within the feature space, whereas supervised techniques could also utilize the label space in this computation. This issue prevails to all state-of-the-art methods [28, 33, 42, 47, 66].

We present CLaP (Classification Label Profile), a domain-agnostic TSSD algorithm that uses self-supervised change analysis to leverage the predictive power of supervised classification algorithms to the unsupervised TSSD problem. It is accurate, scalable, and capable of processing both univariate and multivariate TS. CLaP is hyper-parameter-free and learns all information from the data at hand. The routine first computes a segmentation with ClaSP, which internally uses a binary self-supervised classifier to differentiate segments based on differently shaped subsequences. CLaP then cross-validates a multi-class classifier with segment-labelled subsequences to measure the confusion between segments. This information is central to labelling segments that share the same state or not. In a subsequent merging procedure, the algorithm iteratively combines confused labels, representing the same state, using a novel merge criterion called classification gain. By design, CLaP automatically determines the number of states in a TS; an otherwise difficult to set hyper-parameter.

Unlike competing approaches, which use classical unsupervised techniques, CLaP is the first method to tackle the TSSD problem with supervised learning methods, by explicitly framing it as a self-supervised task. Segments are modelled directly in label space, which is more discriminative than feature space, used in earlier algorithms. CLaP introduces a novel merging procedure that learns states by fusing segment labels according to their mutual confusion. This process is guided by a new score that enables fair comparisons across classification problems of differing complexity.

Figure 1 shows an example of how CLaP and a state-of-the-art competitor Time2State [66] annotate a multivariate time series (MTS) with a state sequence. The recording captures smartphone sensor readings of a 30-year-old female resting (blue), performing squats (orange), resting again (blue), loosening her legs (green), and resting again (blue) [18]. CLaP accurately identifies the transitions between activities and correctly assigns 3 distinct labels to the 5 segments (segments with the same activity are visualized as equally high flat lines; ones with different activities appear as lines at varying levels in the visualization). Time2State, in comparison, correctly identifies the resting periods, but overestimates the total number of activities being 10 separated by 12 transitions.

In summary, this paper's contributions are:

(1) We present CLaP, a new domain-agnostic and hyper-parameter-free TSSD algorithm that leverages self-supervised TS classification for segmentation and labelling to predict state sequences of multivariate TS. We provide technical descriptions, a computational complexity analysis, examples, and a Python implementation.
(2) We propose two technical novelties that make TS classifiers applicable for TSSD: confused merging, a mechanism to reduce a TS state labelling to a minimal required set of labels; and classification gain, which quantifies the increase in F1 score of a classification compared to a random one.
(3) We assess the accuracy and runtime of CLaP and six state-of-the-art competitors (Time2State, HDP-HSMM, E2USD, ClaSP2Feat, TICC, AutoPlait) on 405 TS from five benchmarks. CLaP significantly outperforms all rivals in accuracy and has the best accuracy-runtime tradeoff. It achieves the highest mutual information of 60.4% with ground truth annotations, an increase of 16.1 percentage points (pp) compared to the second-best competitor Time2State.

To foster the reproducibility of our findings, we created a supporting website [11] that contains all source codes, our evaluation framework, Jupyter notebooks for exploratory analysis, raw measurements, scores, and visualizations. The remaining paper is organized as follows: Section 2 introduces the necessary definitions and background of this work. In Section 3, we present CLaP and its technical novelties in detail. Section 4 shares the results of extensive accuracy and runtime experiments of CLaP and the competitors. In Section 5, we review related works, and Section 6 concludes.

2 DEFINITIONS AND BACKGROUND

We formally define the concepts of abstract processes, time series, subsequences, state sequences, the time series state detection (TSSD) problem, and introduce ideas of TS change analysis.

Definition 1. *An* abstract process $P = (S, R)$ *consists of one or more discrete and distinct states* $s_1, ..., s_l \in S$ *that are pairwise separated by transitions* $(s_i, s_j) \in R \subseteq S \times S$*, with* $s_i \neq s_j$*.*

Following the definition of Wang et al. [65], states refer to distinct phases of real-world processes observable through sensor

measurements. We assume each state has a stationary, recognizable property that persists over time and makes it distinguishable from other states. Examples include human activities or machine conditions. This definition excludes processes with homonym states, that emit indistinguishable signals, and synonym states, which cannot be recognized by observation. It also excludes states that have drifting semantics over time.

Transitions are, by definition, changes between states and constitute the links among them in processes. We assume that a transition leads to a change in the observations of a process. Gradual changes, such as trends, can be modelled as separate states. We make no assumptions about transition causes, i.e., whether they occur randomly or systematically.

For analysis, we consider measurements emitted by one or multiple sensors observing outcomes or byproducts of a process. Human activity, for instance, can be tracked by inertial measurement units (IMUs) in smartphones [43] and industrial machinery can be monitored with IoT devices [64], which results in temporal data.

Definition 2. *A* time series *(TS) T is an ordered sequence of $n \times d \in \mathbb{N}^2$ dimensional vectors $T = (\vec{t}_1, \dots, \vec{t}_n) \in \mathbb{R}^{n \times d}$ that simultaneously measures d observable outputs of a process P.*

Each t_i has d dimensions, or channels, one for each sensor. Its values, also called data points or measurements, are equi-distant and ordered by time, e.g. 1 data vector is recorded every 10 milliseconds.

Definition 3. *Given a TS T, a* subsequence *$T_{s,e}$ of T with start offset s and end offset e is the d-dimensional slice of contiguous observations from T at position s to position e, i.e., $T_{s,e} = (\vec{t}_s, \dots, \vec{t}_e)$ with $1 \leq s \leq e \leq n$. The length of $T_{s,e}$ is $|T_{s,e}| = e - s + 1$.*

We use the terms *subsequence* and *window* interchangeably, and refer to their length as the *width*. A state from S yields its observable properties as a subsequence with shapes or statistics distinguishable from others. We call these core structures *temporal patterns*, since they provide the necessary information to recognize the same state and distinguish it from others. Temporal patterns can drift or suddenly change over time, indicating a switch from one process state to another. Note, however, that local parts of channels contribute differently to the signal, for instance in amplitude.

Definition 4. *Given a TS T of size n that captures a process $P = (S, R)$, the corresponding state sequence $Q = (s_{t_1}, \dots, s_{t_n}) \in S^n$ contains the states $s_{t_i} \in S$ that are captured at time points $t_i \in T$.*

A *change point* (CP) denotes an offset $i \in [1, \dots, n]$ for $t_i \in T$ that corresponds to a transition between states $s_{t_{i-1}}$ to s_{t_i} in Q, where $(s_{t_{i-1}}, s_{t_i}) \in R$ and $s_{t_{i-1}} \neq s_{t_i}$. For notational convenience, we consider the first and last values in T as CPs. We call the subsequence between two CPs a *segment* with variable size. A *segmentation* of T is the ordered sequence of CPs in T, i.e., $t_{i_1}, \dots, t_{i_n}$ with $1 \leq i_1 < \cdots < i_n \leq n$ at which the process P changes state.

Definition 5. *The problem of* time series state detection *(TSSD) is to recover the latent state sequence Q of a process P, only by analysing the time series T, emitted by P.*

The TSSD problem, as defined here, is unsupervised and twofold: We need to find a segmentation of T that captures the state transitions R, as well as the distinct states S from all segments, in order to predict a state sequence $\hat{Q}$. An optimal result yields a $\hat{Q}$ that is isomorphic to Q, because an algorithm does not have access to the actual state labels. To evaluate the quality of a predicted state sequence, we can measure its alignment with the ground truth annotated by domain experts — e.g., by inspecting Covering [63] or the mutual information [51] of segments.

2.1 Self-supervised Change Analysis

To label TS with their states, CLaP computes segments of T and iteratively relabels them based on mutual similarity. Contrary to classical clustering approaches, we do not compare distances, but build on self-supervised TS change analysis, first proposed by Hido et al. [29], that we shortly introduce next.

Self-supervised learning is an unsupervised learning variant, in which the data itself is used to generate supervision labels. Consider two data sets, X_A and X_B, representing subsequences from different segments. We assign label 0 to samples from X_A and 1 to ones from X_B, enabling a binary classification evaluation using cross-validation. A TS classifier, trained on the labelled subsequences, predicts labels for unlabelled instances. k-fold cross-validation evaluates the classifier by training it on $(k-1)$ parts of the data and testing it on the remaining one. This process repeats k times, covering all combinations, with the average of the k evaluation scores (e.g., F1-scores) representing the classifier's predictive power. This value measures the classifier's ability to distinguish between data sets X_A and X_B. A high score implies high dissimilarity and unique characteristics between the segments, indicating they represent different states. Conversely, a lower score indicates similarities, suggesting the segments may belong to the same state.

3 CLAP - CLASSIFICATION LABEL PROFILE

We propose the *Classification Label Profile*, short *CLaP*, a novel algorithm that formulates TSSD as a self-supervised classification problem. We first provide an overview and example of the main concept, before we explain in detail how to implement it for state detection in Subsections 3.1 to 3.3.

Definition 6. *Consider a process $P = (S, R)$, captured by a TS T of size $|T| = n$, and a window size w. A CLaP is a tuple (L, c) that annotates T with a label sequence $L \in \{1, \dots, k\}^{n-w+1}$ and an associated cross-validation score $c \in [0, 1]$.*

The label sequence L links all the $n - w + 1$ overlapping subsequences $T_{i,i+w-1}$ with one of k labels, which are used to represent the states in S. CLaP is computed from a subsequence classification problem with k classes, whose discriminatory power is summarised in the score c. We derive L from a self-supervision mechanism (see Subsection 3.1) and calculate c as the F1-score from a 5-fold cross-validation with the ROCKET classifier [12], using the subsequences from T as data and L as artificial ground truth labels. Low scores of c indicate CLaP is not able to accurately differentiate similar from dissimilar subsequences, while high scores show that it constitutes distinguishable windows.

Figure 2 shows an example of three different CLaPs. The top part illustrates a triaxial human activity recording from a 23-year-old male, switching between horizontal (blue) and downstairs (orange) walking. The true state sequence of this recording assigns, for example, 1s to the blue data points and 2s to the orange measurements.

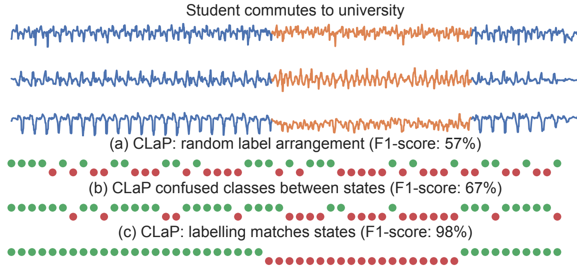


Figure 2: Top: excerpt of triaxial mobile phone acceleration recording from male student commuting to university, showing horizontal (blue) and downstairs (orange) walking motions [18]. 4th from top to bottom: three CLaPs with different labellings (green and red) and associated F1 cross-validation scores. Profiles (a to c) incrementally align more with the true state sequence, which leads to increasing F1 scores.

The bottom part displays three different CLaPs (a to c). The sequence of green and red dots shows the label configuration L, with the associated F1-score c annotated. CLaP (a) presents a random label configuration that does not match the true state sequence, resulting in poor cross-validation performance (57% F1-score). The profile in (b) somewhat aligns with Q but confuses some instances (67% F1-score). In contrast, CLaP in (c) perfectly resembles the true state sequence of the human activity recording, leading to a near-optimal performance (98% F1-score).

The true state sequence Q of a TS T constitutes a CLaP that scores the highest discriminatory power compared to all possible sequence permutations. Our implicit assumptions thereby are three-fold: (1) Process states can be conceptualized as classes of TS windows, (2) subsequences (of appropriate length) can serve as class exemplars with distinctive features, (3) and a TS classifier cross-validation score correlates with the quality of the label configuration. With these concepts, we can tackle an unsupervised problem with the predictive power of TS classification (TSC) algorithms. Much research interest continues to increase accuracy and efficacy for the task (see Middlehurst et al. [49] for a survey). However, searching a high-quality state sequence prediction $\hat{Q}$ using CLaP is non-trivial.

The naive approach of enumerating all k^{n-w+1} candidates and choosing the highest-scoring one (according to c) is computationally inefficient and yields further challenges. For instance, the number of classes k is typically not known in advance. Also, the large collection of label combinations makes the cross-validation score incomparable for different values of k. We propose efficient and accurate solutions to prune the candidates and make their scores comparable in Subsections 3.1 and 3.2, bridging the gap to enable TSC methods to be directly applicable for TSSD. Subsection 3.3 analyses the runtime and space complexity of CLaP.

3.1 Confused Merging

The TSSD problem can be divided into a segmentation followed by clustering. We use this formulation to drastically prune the amount of potential CLaPs. First, we compute a segmentation of a TS T. This enables us to label only few segments, instead of many

Algorithm 1 Classification Label Profile

```
1:  procedure CLAP(T)
2:      w ← LEARN_SUBSEQUENCE_WIDTH(T)                ▷ Run SuSS
3:      cps ← COMPUTE_SEGMENTATION(T)                 ▷ Run ClaSP
4:      X, L ← CREATE_DATA_SET(T, w, cps)
5:      y_pred ← CROSS_VAL_CLF(X, L)                  ▷ 5-fold ROCKET CV
6:      for i ∈ [1, ..., ||cps||] do
7:          merged ← false
8:          labels, conf ← CALC_CONFUSED_LABELS(L, y_pred)
9:          for (l1, l2) ∈ RANK(labels, conf) do
10:             L̂ ← REPLACE(L, l1, l2)                ▷ Merge labels
11:             ŷ_pred ← REPLACE(y_pred, l1, l2)
12:             if CGAIN(L̂, ŷ_pred) ≥ CGAIN(L, y_pred) then
13:                 L, y_pred ← L̂, ŷ_pred             ▷ Update labels
14:                 merged ← true
15:                 break
16:             end if
17:         end for
18:         if merged ≠ true then break              ▷ Early stopping
19:     end for
20:     return (L, SCORE(L, y_pred))                  ▷ Calc. F1-score
21: end procedure
```

single data points. Secondly, we propose a novel agglomerative self-supervision mechanism, that greedily clusters confused state labels with a new bespoke merge criterion. The entire process, called *Confused Merging*, combines several design choices, that we fixed using ablations (see Subsection 4.4). For each component, we considered multiple domain-agnostic methods and chose the best-performing one. CLaP is hyper-parameter-free, automatically learns the number of k classes and stops as soon as no classes must be merged. Pseudocode is provided in Algorithm 1 and the workflow is illustrated in Figure 3.

Preprocessing: Algorithm 1 receives a TS T as user input (line 1) and starts by learning its subsequence width w (line 2), needed to compute the sliding window in CLaP. Multiple techniques are available for this task and based on the idea that temporal patterns of similar size repeat throughout TS, an assumption we share in CLaP. We choose SuSS (Subsequence Summary Statistics), one of the best-performing procedures, according to [19]. For multidimensional TS, we compute one window size per dimension and then compute their average. The CLaP algorithm continues to compute a segmentation of T with the ClaSP method [20] (line 3), which can partition multivariate TS [21] from different domains [23, 70] and has an extension for very large and streaming data [22]. This preprocessing step removes the need to label individual data points in CLaP and instead only requires assigning classes to entire segments.

Self-supervised Classification: The procedure creates an initial labelled data set from T by creating a sliding window X (width of w, stride of $\frac{w}{2}$) and labelling the resulting subsequences with their corresponding segment ranks in L, i.e. label 1 for the first segment, label 2 for the second one, label 3 for the third, and so on (line 4). Subsequences that overlap neighbouring segments with at least $\frac{w}{2}$ values are disposed. The motivation behind this labelling is twofold: it encodes the CPs (from the segmentation) as a labelling

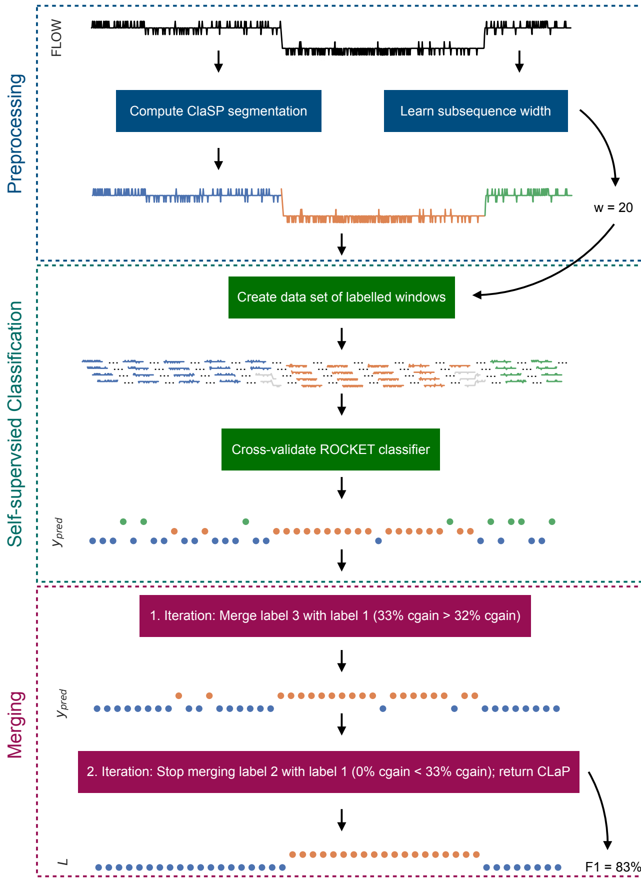


Figure 3: Workflow of CLaP for a water circuit [36]. TS captures fluid circulation flow with opened (blue and green) and closed valve (orange). Preprocessing: ClaSP segmentation divides TS into three segments, SuSS learns the subsequence width. Self-supervised Classification: labelled subsequences constitute the data set; grey ones are disposed; ROCKET classifier computes initial predicted cross-validation labels. Merging: labels are combined based on confusion as long as cgain increases. Final CLaP with artificial ground truth labels and F1-score of 83% is returned.

and first assumes all segments to capture distinct states. This enables the later merging, which fuses labels from segments sharing state. CLaP evaluates the ROCKET classifier, which is fast and accurate [4], with the labelled data set in a 5-fold cross-validation and stores the predicted labels in y_{pred} (line 5). This creates an initial measure for the quality of the labelling. Compared to the artificial ground truth L, segments representing single states are expected to mostly match. Conversely, multiple segments sharing state, are expected to confuse labels, facilitating their identification.

Merging: CLaP uses the initial labelled data set (X, L) and its cross-validated prediction labels y_{pred} to iteratively fuse segments representing the same state (lines 6–19). To achieve this, the procedure greedily merges highly confused labels, as their associated subsequences are difficult to separate. For every unique label l_1 from L, CLaP determines its most confused counterpart l_2, according to y_{pred} (line 8). This information is directly derived from a confusion matrix. The algorithm iterates the most confused label pairs with descending confusion to check if they can be merged (lines 9-17). Specifically, it creates temporary label vectors $\hat{L}$ and $\hat{y}_{pred}$, where l_2 is replaced with l_1 (lines 10–11), and checks if their classification gain (see Subsection 3.2 for details) is greater (or equal) to the one from L and y_{pred} (line 12). In this case, the method updates the current with the merged labelling (line 13) and stops merging; otherwise, it continues with less confused label pairs. After a successful merge iteration, CLaP starts another round with updated artificial ground truth and predicted labels. Once no label pair can be further merged (line 18), the procedure stops and returns the current label configuration L and its associated cross-validation score, according to y_{pred}, constituting CLaP (line 20). The central mechanism of this process is confused label pair selection. We exploit the fact that segments sharing state are confused by the classifier, because of similar temporal patterns that are differently labelled. Hence, we use it as a selection criterion, implementing it in an agglomerative fashion, to learn the amount of states and stop as soon as possible. Classification gain (see Subsection 3.2) regularises the merging to ensure that only labels from sufficiently similar segments are fused.

Workflow: Figure 3 illustrates the confused merging algorithm for water flow circulation in a testbed [36]. The preprocessing (blue frame) divides the TS into three segments (blue, orange, green), capturing two states; namely open and closed valve. It also computes the subsequence width of 20 with SuSS. The self-supervision mechanism (green frame) creates labelled subsequences using segment ranks (1, 2, 3) and uses ROCKET to compute initial predicted cross-validation labels y_{pred} that show high confusion between label 1 and 3. Merging (purple frame) finds label 3 to have the highest overall confusion (with label 1) and fuses both, increasing classification gain (from 32% to 33%). Merging even further to one class would reduce cgain to 0%, hence the procedure stops. The resulting artificial ground truth L and it's 83% F1-score constitute CLaP and correctly capture the latent states.

The procedure is hyper-parameter-free and it learns two model-parameters from the input TS: the subsequence width w and the number of classes k. By default, CLaP uses five folds for cross-validation and draws at most 1k randomly sampled, labelled subsequences from the full data set to control runtime.

3.2 Classification Gain

Confused merging relies on a merge criterion that decides if two confused classes should be combined or not. In agglomerative clustering, this is typically achieved by thresholding the distance computation. For CLaP, we could translate that into enforcing the F1-score (or e.g. cross-entropy loss) of the label configurations (Algorithm 1, line 12) to pass a minimal (maximal) value. However, this is disadvantageous for three key reasons: (a) the F1-score (also accuracy, ROC/AUC, or entropy) depends on the classifier and data; rendering threshold selection use case dependent. (b) Comparing F1-scores of classification problems with different label distributions is meaningless; more labels increase problem complexity, resulting in generally lower F1-scores. (c) The F1-score does not reflect the improvement of one labelling over another, just their total qualities.

To address these challenges, we propose a new merge criterion called *Classification Gain*, which measures the increase in F1-score over its expected random score, thereby normalizing for problem complexity. It isolates predictive power from classification difficulty, facilitating performance comparisons across different problems.

Definition 7. *Given a ground truth labelling $y_{true} \in U^n$ of size n, with unique classes $U \subset \mathbb{N}$, the random F1-score is defined as:*

$$f1_{rand}(y_{true}) := \frac{1}{\|U\|} \sum_{l \in U} \frac{2 \cdot TP(l)}{2 \cdot TP(l) + FN(l) + FP(l)} \quad (1)$$

The random (macro) F1-score provides an unbiased estimate of classification difficulty based on the label distribution y_{true}. It can be computed efficiently using probability theory in the following way: in a random classification, we expect each instance labelled $l \in U$ to be mapped to its own (or any other) class according to its likelihood of occurrence.

$$P(y_{true} = l) := \frac{\#(y_{true} = l)}{\|y_{true}\|} \quad (2)$$

$$P(y_{true} \neq l) := 1 - P(y_{true} = l) = \frac{\#(y_{true} \neq l)}{\|y_{true}\|} \quad (3)$$

These class priors enable us to define the true positives (TPs) as the expected amount of instances that are randomly assigned to their class (Equation 4), the false negatives (FNs) as the expected number of samples that are mapped to another class (Equation 5), and the false positives (FPs) as the expected amount of misclassified examples that actually belong to a given class (Equation 6).

$$TP(l) := \#(y_{true} = l) \cdot P(y_{true} = l) \quad (4)$$

$$FN(l) := \#(y_{true} = l) \cdot P(y_{true} \neq l) \quad (5)$$

$$FP(l) := \#(y_{true} \neq l) \cdot P(y_{true} = l) \quad (6)$$

Given the random macro F1-score for y_{true} as an estimate of problem difficulty, we can decouple it from the performance of a classification to make it comparable across problems.

Definition 8. *Given a ground truth labelling $y_{true} \in U^n$ and associated predictions from a classifier $y_{pred} \in U^n$ of size n, the classification gain, short* cgain, *is defined as:*

$$cgain(y_{true}, y_{pred}) := f1(y_{true}, y_{pred}) - f1_{rand}(y_{true}) \quad (7)$$

It reflects the improvement in macro F1-score of the classification over the expected random one. The central idea of *cgain* is to normalize F1-score with the associated classification problem complexity to make the measure comparable for different label distributions. By comparing *cgain* in CLaP (line 12), we enforce that the merge operation steadily improves classification quality, while not relying on thresholds. This addresses the problems (a) to (c) that suffer from greedily maximising performance.

Note that cgain is different from gini gain [8], which measures the discrepancy in purity between data sets. Classification gain measures the quality for a given classification considering its difficulty, estimated by random classification.

Example: Figure 4 (top) illustrates a satellite image TS capturing sensor observations of 3 different crops (coloured in blue, orange, green) as 9 segments. A correct segment state sequence (one label per segment) would be: 1,2,3,1,2,3,1,2,3. CLaP tries to find such sequence by merging classes from its initial sequence: 1,2,3,4,5,6,7,8,9.

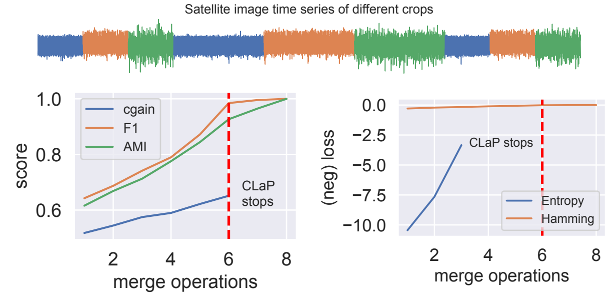


Figure 4: Top: TS capturing 9 time spans of 3 different crops (blue, orange, green) [56]. Bottom: CLaP run on TS with different merge criteria (scores and negative losses); dashed red line marks optimal amount of 6 merge operations, reducing 9 initial classes to 3. Only cgain accomplishes this; other measures misjudge label configuration quality and either merge too few (Entropy) or too many (F1, AMI, Hamming) classes.

In Figure 4 (bottom) we use different merge criteria in CLaP (line 12) for this computation and show their values for the merge operations. Generally, merging should increase the measures for 6 iterations (red dashed line) and then stop, as they begin to decrease. It protrudes that F1-score, adjusted mutual information score (AMI) and (negative) Hamming loss steadily increase, reducing the state sequence with 8 merges to just one state, namely 1. This is due to the fact that none of the measures considers problem complexity and overestimates label configuration quality with decreasing number of classes. Negative cross-entropy loss underestimates quality and merges only 3 times, as its value would decrease for a fourth iteration (line 12), resulting in the sequence: 1,2,3,1,2,4,1,5,6. Our proposed classification gain is the only criterion that merges exactly 6 times and would decrease in value thereafter; producing the ground truth result and stopping automatically.

3.3 Computational Complexity

The scalability of state detection algorithms is critical as the size of sensor recordings steadily grows. The complexity of CLaP mainly depends on the segmentation (Algorithm 1, line 3), self-supervised classification (line 5) and merging (lines 6-19).

For a TS T with d channels of size n, the initial computation of the window size w (line 2) with SuSS is in $O(d \cdot n \log w)$ [20]. Locating c CPs in T with ClaSP (line 3) is in $O(d \cdot n^2 \cdot c)$, which involves cross-validating a self-supervised k-nearest neighbour classifier multiple times. The entire preprocessing complexity is in $O(d \cdot n \log w + d \cdot n^2 \cdot c)$ which reduces to $O(d \cdot n^2 \cdot c)$ as $w \leq n$.

Creating the data set for self-supervised classification (line 4) is in $O(d \cdot n \cdot w)$, as it only requires reorganising T. Running a single validation of the ROCKET classifier is also in $O(d \cdot n \cdot w)$; it convolves the data set with a fixed set of $10k$ kernels for feature generation and uses ridge regression for classification [12]. The same complexity holds for repeating the procedure 5 times during cross-validation (line 5), which leads to a total runtime in $O(d \cdot n \cdot w)$ for the entire step. Note, that ROCKET's runtime hides five-digit constants for calculating convolutions with $10k$ kernels [12].

Table 1: Technical specifications of TS used in experiments.

Name	No. TS / Dims.	TS Length Min/Med./Max	No. States (Segs.) Min/Med./Max
TSSB [20]	75 / 1	240 / 3.5k / 20.7k	1 (1) / 3 (3) / 7 (9)
UTSA [26]	32 / 1	2k / 12k / 40k	2 (2) / 2 (2) / 3 (3)
HAS [18]	250 / 9	340 / 5k / 41.5k	1 (1) / 3 (4) / 12 (15)
SKAB [36]	34 / 8	745 / 1.1k / 1.3k	2 (2) / 2 (3) / 2 (3)
MIT-BIH [50]	14 / 1	57k / 258k / 650k	2 (3) / 2 (8) / 5 (16)
Total	405 / 9	240 / 4.8k / 650k	1 (1) / 3 (3) / 12 (16)

Table 2: List of competitors for experimental evaluation.

Competitor	State Detection Method
AutoPlait [47]	Hidden Markov Models
CLaP	Self-supervision
ClaSP2Feat	Self-supervision
E2USD [42]	Deep Learning
HDP-HSMM [33]	Hidden Markov Models
TICC [28]	Expectation Maximization
Time2State [66]	Deep Learning

The confused merging calculates a confusion matrix (line 8) for the $\lfloor \frac{2\cdot(n-w)}{w} \rfloor + 1$ labels from the cross-validation in $O(n)$; which only involves incrementing counts. The same complexity holds for replacing labels (lines 10–11) and calculating classification gain (line 12), which is mainly based on computing the confusion matrix for the random F1-score; counting labels and calculating likelihoods. This process is repeated, in the worst case, for $c+1$ confused label pairs in descending order which requires $O(n \cdot c \log c)$ time. The entire process (lines 6–19) can be repeated up to c times, resulting in a total runtime complexity of $O(n \cdot c^2 \log c)$.

Considering the sum of all steps, CLaP requires $O(d \cdot n^2 \cdot c) + O(d \cdot n \cdot w) + O(n \cdot c^2 \log c)$ time, which simplifies to $O(d \cdot n^2 \cdot c + n \cdot c^2 \log c)$ for computing n state labels with maximal $c+1$ unique classes. This runtime complexity allows for CLaP's application to medium-sized and large TS in the batch setting. For streaming purposes, however, further advancements are needed.

The space complexity mainly depends on the data set size and confusion matrix, which is in $O(d \cdot n \cdot w + c^2)$.

4 EXPERIMENTAL EVALUATION

We evaluated the accuracy, runtime, and scalability of CLaP against six competitors on five large benchmark data sets (405 TS) to assess its performance on real-world TS data. Subsection 4.1 presents the data sets, evaluation metrics, and competitors. In Subsections 4.2 and 4.3, we present accuracy and runtime results of CLaP and its competitors. Subsections 4.4 and 4.5 analyse the results of an ablation study with different design choices for CLaP and sensitivity experiments, with varying data quality. Finally, Subsection 4.6 discusses strengths and weaknesses of CLaP and Subsection 4.7 showcases a real-world use case. All experiments were conducted on an Intel Xeon 8358 with 2.60 GHz, 2 TB RAM, 128 cores, running Python 3.8. All of our experimental results, figures, raw measurements per data set, codes, Jupyter Notebooks, and TS used in the evaluation are available on our supporting website [11] for inspection and to ensure reproducibility.

4.1 Experiment Setup

Data Sets: We evaluated CLaP and its competitors on a total of 405 small to very large-sized (240 to 650k) TS from five public segmentation benchmarks (see Table 1). Ground truth CPs and states, annotated by domain experts, are available and were used for evaluation. The benchmarks can be divided into two groups: cross-domain and single-domain use cases.

UTSA [26] and TSSB [20] consist of 75 and 32 univariate TS, representing a diverse collection of cross-domain problem settings featuring biological, mechanical, and synthetic processes from sensor, device, image, spectrogram, and simulation signals. Conversely, HAS [18] contains 250 9-dimensional TS from smartphone sensors, capturing indoor and outdoor motion sequences from students. SKAB [36] consists of 34 8-dimensional TS from machine sensors observing a water circulation testbed with pumps, tanks, and valves, undergoing both normal and anomalous conditions. MIT-BIH [50] features 14 univariate high-resolution ECG recordings from patients with normal cardiac activity and prolonged types of arrhythmias.

All TS combined capture versatile processes from natural phenomena, humans, animals, and industrial machinery. The number of distinct states per process varies between 1 and 12, occurring in 1 to 16 segments. In 65% of all data sets, the number of states and segments match; in 15% a single state reoccurs, 9% show two reoccurrences, and the remaining 11% exhibit three to 14 reoccurrences. It is worth noting that the data sets vary greatly in problem complexity, which tends to increase with the number of states and transitions in the processes.

Evaluation Metrics: TSSD combines the segmentation and state identification tasks. We assessed both aspects separately with own scores, namely Covering [63] for segmentation and Adjusted Mutual Information (AMI) [51] for state identification.

Covering measures how well the predicted and annotated segments overlap in location, disregarding the specific state labels of the segments. This allows for the comparison of segmentations of different sizes (including empty segmentations). Let the interval of successive CPs $[t_{c_i}, \ldots, t_{c_{i+1}}]$ be a segment in T, and let $segs_{pred}$ and $segs_T$ be the sets of predicted and annotated segmentations, respectively. Additionally, we set $t_{c_0} = 0$ as the first and $t_{c_{n+1}} = n+1$ as the last CP to include the first and last segments. The Covering score reports the highest-scoring weighted overlap between predicted and annotated segmentations (using the Jaccard index) as a normalized measure in the interval $[0, \ldots, 1]$, with higher scores being better (Equation 8).

$$\textsc{Covering} = \frac{1}{\|T\|} \sum_{s \in segs_T} \|s\| \cdot \max_{s' \in segs_{pred}} \frac{\|s \cap s'\|}{\|s \cup s'\|} \quad (8)$$

Mutual Information (MI) reports the similarity between two different labellings of the same data. It is invariant to specific label values and allows us to quantify the agreement between predicted and annotated states, disregarding their locations. The measure is adjusted to account for chance, as the score naturally increases with a larger number of states. Let $states_{pred}$ and $states_T$ represent the

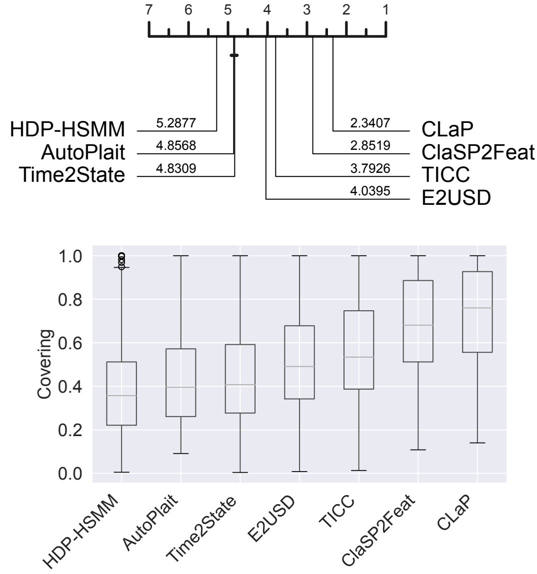


Figure 5: Covering ranks (top) and box plot (bottom) on 405 TS for CLaP (lowest rank) and six competitors.

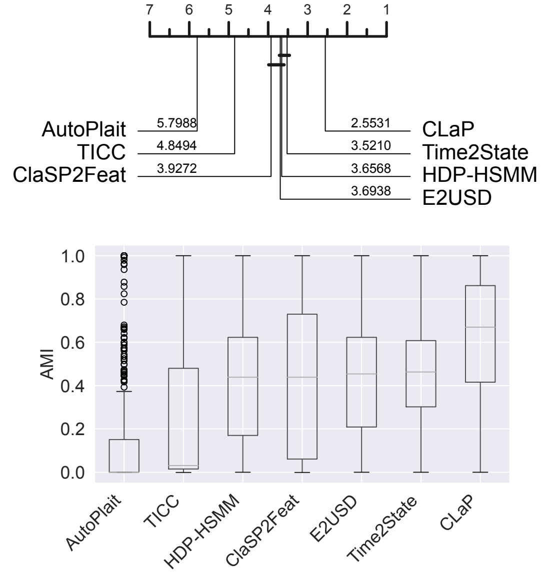


Figure 6: AMI ranks (top) and box plot (bottom) for CLaP (lowest rank) and six competitors on 405 TS.

sets of predicted and annotated states, respectively, and let H denote the entropy function. AMI measures the adjusted, weighted overlap between pairwise clusters as a normalized score within $[-1, \ldots, 1]$, with higher values indicating better agreement (Equation 10).

$$MI = \sum_{s \in states_T} \sum_{s' \in states_{pred}} \frac{\|s \cap s'\|}{\|T\|} \log\left(\frac{\|T\| \cdot \|s \cap s'\|}{\|s\| \cdot \|s'\|}\right) \quad (9)$$

$$AMI = \frac{MI - \boldsymbol{E}[MI]}{\text{AVG}(H(states_T), H(states_{pred})) - \boldsymbol{E}[MI]} \quad (10)$$

To compare different methods across multiple data sets, we aggregate scores into a single ranking. First, we compute the rank of the score for every algorithm and TS. Then, we average the ranks per method across data sets to obtain overall mean ranks. We visualize these values with Critical Difference (CD) diagrams [15] (see, e.g., Figure 5 top) which include an assessment of statistical differences between methods. The best approaches, with the lowest average ranks, are shown to the right of the diagram. Groups of algorithms that are not significantly different in rank are connected by a bar, based on a pairwise one-sided Wilcoxon signed-rank test with $\alpha = 0.05$ and Holm correction. Besides CD diagrams, we discuss summary statistics, box plots [62], and specific examples.

Competitors: We compare CLaP against six state-of-the-art state detection competitors (see Table 2). All implementations are openly available on our supporting website [11]. As a baseline, we extract segments with ClaSP [20] and learn their labels using Time2Feat [6], which extracts and selects features from TS segments before clustering them. We call this approach ClaSP2Feat. We report results for two competitors based on Hidden Markov Models (HMMs). AutoPlait [47] uses so-called multi-level chain models to learn states and applies the minimum description length (MDL) principle to automatically select the best-fitting model. HDP-HSMM [33] is a Bayesian extension of traditional HMMs with efficient state sampling. Another statistical method we compare against is TICC [28], which formulates state detection as a covariance-regularized maximum likelihood problem and computes it using a greedy heuristic. From deep learning, we report results for Time2State [66], which learns an encoder from TS windows using a latent state encoding loss and clusters the embeddings with a Gaussian mixture model (DPGMM) to predict states. E2USD [42] first creates a compact and informative TS embedding using Fast Fourier Transform (FFT) compression and decomposition techniques combined with contrastive learning, and then also uses DPGMM.

To prevent overfitting, we adopt the default hyper-parameters reported in the original publications or reference implementations: for HDP-HSMM we set $\alpha = 10k$ and $\beta = 20$; for TICC we use the annotated subsequence widths, $\lambda = 0.001$, $\beta = 2200$, a convergence threshold of 10^{-4}, and a maximum of 10 iterations; Time2State and E2USD set a sliding window of 256 and step sizes of 10 and 50, respectively. HDP-HSMM, TICC, Time2State, and E2USD output only state sequences, so we derive CPs by locating offsets at changing states. ClaSP2Feat and TICC require the number of states as a hyper-parameter. We set this value from the ground truth annotations. Note that it provides an information advantage to both methods, which is, however, necessary to run them. All other competitors infer the number of states automatically.

4.2 Comparative Analysis

We evaluated the average (Covering / AMI) rank of CLaP and the six competitors on all 405 TS to assess overall performance. Figures 5 and 6 (top) show the results. For both measures, CLaP (2.34 / 2.55) leads with a significant advantage, followed by ClaSP2Feat (2.85

Table 3: Summary performances for CLaP (highest scores) and six competitors on 405 TS.

Covering / AMI

	average (in %)	median (in %)
CLaP	**72.9** ± 21.6 / **60.4** ± 30.9	**76.0** / **67.0**
ClaSP2Feat	68.4 ± 22.9 / 44.1 ± 34.4	68.1 / 43.9
TICC	57.2 ± 24.2 / 24.4 ± 34.8	53.4 / 3.1
E2USD	50.0 ± 23.3 / 42.0 ± 27.3	49.1 / 45.4
Time2State	43.9 ± 22.6 / 44.3 ± 23.3	40.7 / 46.3
AutoPlait	45.8 ± 25.0 / 17.5 ± 32.9	39.6 / 0.0
HDP-HSMM	38.7 ± 24.1 / 41.5 ± 27.9	35.7 / 43.9

/ 3.93) and TICC (3.79 / 4.85) for Covering, and Time2State (4.83 / 3.52) as well as HDP-HSMM (5.29 / 3.66) for AMI. CLaP's lead in Covering, compared to ClaSP2Feat, confirms the substantial advantage of confused merging over clustering, as both methods utilize ClaSP for segmentation. Interestingly, the competitors' differences in average rank are not consistent across the measures. Considering the five benchmarks separately, CLaP also achieves first place for each collection, except in SKAB, where HDP-HSMM leads and CLaP scores third place because it detects wrongly located CPs that cannot be properly corrected by confused merging (see [11]). CLaP's wins in Covering are not statistically significant, except for HAS. Conversely, its wins in AMI are statistically significant, except for MIT-BIH.

Looking at individual data sets, CLaP wins or ties 204 / 193 out of 405 data sets (Covering / AMI), followed by ClaSP2Feat (146 / 94) and TICC (86 / 62). Note that the Covering counts exceed the total number of data sets because of ties (see [11]). For the 37 TS with only one state, TICC achieves the best results. However, for the 154 data sets with two states and the 214 TS with three or more states, CLaP leads by a significant margin. In a pairwise comparison of CLaP against every competitor, it outperforms each method in at least 38.3% / 56.0% of cases for Covering / AMI. In summary, the rankings show that CLaP on average surpasses state-of-the-art algorithms in terms of accuracy across the five benchmarks.

The summary statistics in Figures 5 and 6 (bottom) and Table 3 confirm the rankings. On average, CLaP scores 72.9 ± 21.6 / 60.4 ± 30.9 for Covering / AMI, representing an increase of 4.5 / 16.1 pp over the second-best competitor, ClaSP2Feat / Time2State. The differences in median are even more pronounced. Across both measures and all benchmarks, CLaP achieves the highest scores, except for SKAB, which aligns with the rankings.

4.3 Runtime and Scalability

We measured the runtime of CLaP and its six competitors on the 405 TS to determine the time needed for TSSD, the accuracy-runtime tradeoff, as well as the scalability of CLaP.

Runtime: Figure 7 (left) shows the runtime distributions of the methods on all data sets. AutoPlait (median of 0.05 seconds) leads with a substantial advantage over all other competitors due to its linear runtime complexity (with respect to TS size). Additionally, the main part of its implementation uses fast C code, while all other methods are implemented in Python. E2USD (8 seconds) and ClaSP2Feat (26 seconds) rank second and third. CLaP (38 seconds) secures fourth place. Its segmentation step takes 25 seconds, while the self-supervised classification and confused merging require 13 seconds. TICC (59 seconds) and HDP-HSMM (72 seconds) rank fifth and sixth, followed by Time2State (103 seconds) at last place. The distributions of all methods scatter widely, due to the large spans in TS lengths. CLaP is faster in median than three of its six competitors, but trades more computational costs, than e.g. AutoPlait, for higher accuracy. If runtime is a critical concern, users may choose to use the fast streaming implementation of ClaSP [22] or an even faster classifier, e.g. QUANT [14], which may, however, decrease accuracy.

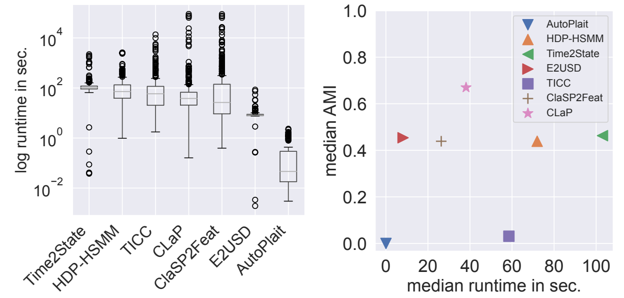


Figure 7: Runtime distribution (left) and accuracy-runtime tradeoff (right) on 405 TS for CLaP and six competitors.

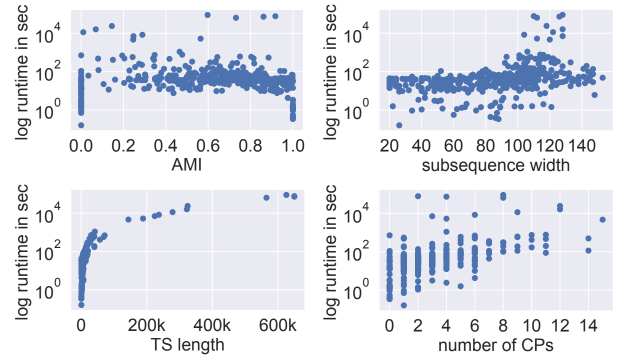


Figure 8: Scalability of CLaP considering AMI performance (top left), subsequence width (top right), TS length (bottom left), and number of CPs (bottom right).

Accuracy vs. Runtime: We investigated the tradeoff between accuracy and runtime of the approaches. Figure 7 (right) depicts the median runtimes vs. AMI scores. The optimal solution is a fast and accurate algorithm (top left), while a poor algorithm is slow and inaccurate (bottom right). CLaP (pink star), ClaSP2Feat (brown plus), and E2USD (red triangle) all exhibit traits of fast and accurate algorithms. However, CLaP stands out with a solution that is only 12 seconds slower but 21.6 pp more accurate. It offers, by far, the most desirable tradeoff between accuracy and runtime compared to the competitors on the considered TS.

Scalability: Figure 8 illustrates how CLaP's (log) runtime scales concerning four different variables, namely: AMI score, subsequence width, TS length, and number of CPs. For the first two variables, we do not observe clear relationships. As expected, CLaP becomes slower with increasing TS length. For instance, it requires less than 5 minutes to process up to 20k data points and less than 3 hours for up to 200k values. Its quadratic runtime complexity with respect to TS length can be roughly observed in the experimental runtimes. For an increasing number of CPs, CLaP also requires more runtime. However, the values are more dispersed compared to TS length. In summary, CLaP produces results for TS with tens of thousands of data points in minutes. For much larger data, runtime may become an issue, as for all other methods, except AutoPlait. In future work, we will investigate TS partitioning and hierarchical merging to further improve scalability.

4.4 Ablation Study

CLaP has four main components, namely: (a) a window size selection algorithm, (b) a segmentation procedure, (c) a classifier, and (d) a merge mechanism. We evaluated different approaches for each component, fixing the others to defaults, on 20% of randomly chosen TS (21 out of 107) from TSSB and UTSA. To prevent overfitting, we excluded the remaining 86 out of 107 data sets, as well as the 298 TS from HAS, SKAB, and MIT-BIH from the ablations. The following analysis reports the summarized results. See the box plots for visualization and more evaluations on our website [11].

(a) Window Size Selection: To determine the window size for a TS in CLaP, we tested two whole-series-based methods: the most dominant Fourier frequency (FFT) and the highest autocorrelation offset (ACF) from [19]; as well as two subsequence-based algorithms: Multi-Window-Finder (MWF) [32] and Summary Statistics Subsequence (SuSS) [20]. Our experiments show no significant differences in average rank between the methods, which aligns with the results from [19]. We choose SuSS for WSS in CLaP, as it achieves the highest average scores for Covering of 80.5% ± 17.4% and AMI of 76.1% ± 27.5%.

(b) Segmentation Procedure: We evaluated three optimization-based approaches for segmentation, namely BinSeg [61], PELT [39], and Window [61], as well as the density-based approach FLUSS [26], the density ratio estimator RuLSIF [31], and the self-supervised algorithm ClaSP [20]. For both metrics, ClaSP leads in average rank, with an insignificant advance for Covering (2.12) and a significant one for AMI (1.98). Hence, we use ClaSP in CLaP.

(c) Classifier: For self-supervised classification in CLaP, we tested six state-of-the-art classifiers according to [49], namely: the feature-based approach FreshPRINCE [48], the interval-based algorithm QUANT [14], the shapelet-based method RDST [27], the dictionary-based procedure WEASEL 2.0 [58], and two convolution-based techniques, namely ROCKET [12] and MR-Hydra [13]. Our experiments show only differences in tendency between the average ranks of the classifiers. We choose ROCKET, as it has the highest-scoring median Covering of 81.6% and AMI of 87.2%.

(d) Merge Score: To implement the merge mechanism, we evaluated four scores: our proposed cgain, F1, ROC/AUC, and AMI; we also assessed two loss functions: log and Hamming. Classification gain leads for both metrics in average rank, with an insignificant advance for Covering but a significant difference to unnormalized F1 and both losses in AMI. It achieves an increase in Covering (AMI) of 2.2 (4.1) percentage points. Therefore, we use cgain in CLaP for confused merging.

In summary, the choice of window size and classification algorithm results in only negligible differences in performance, whereas the segmentation procedure and merge score have a substantial impact. This may be partially due to the interconnections between the components; for example, imprecisely located CPs can negatively influence self-supervised classification, and improper merging can easily degrade the entire state detection process.

We investigated whether CLaP's two phases, segmentation and state identification, substantially contribute to TSSD by comparing each phase with a random baseline while keeping all other settings fixed. For segmentation, we ran CLaP with uniformly drawn CPs (without replacement), which lowered the average Covering score by 20.2 pp and AMI by 31.8 pp. For state identification, we assigned uniformly drawn labels to the segments (with replacement), causing Covering to drop by 20.5 pp and AMI by 40.9 pp. These pronounced declines show that both phases are essential to CLaP's performance; omitting either one sharply degrades accuracy.

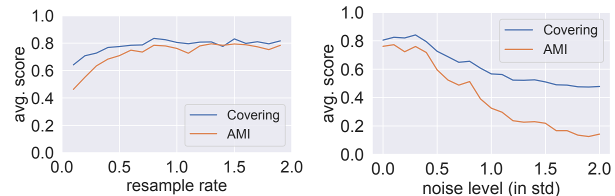


Figure 9: Sensitivity analysis of CLaP regarding sampling rate (left) and noise level (right).

4.5 Sensitivity Analysis

Real-world TS data are collected at different sampling rates and often contain noise introduced by sensor artefacts or external events. We evaluated how these factors affect CLaP's performance on the data sets from the ablation study to assess its robustness to changes in data representation.

Sampling Rate: Figure 9 (left) plots CLaP's average Covering and AMI scores when the data sets are resampled to 10% up to 190% (in 10 pp steps) of their original size. Both metrics increase steadily from 10% to roughly 50% and then plateau. This suggests that CLaP requires a minimum temporal resolution (30% to 50% for these data sets) to identify process states accurately, but maintains stable performance at higher resolutions.

Noise Level: Figure 9 (right) shows how CLaP's performance changes when white noise with increasing standard deviation is added to the standardised data. Performance remains quite stable up to a noise level of 0.5, after which it declines sharply. Thus, CLaP is resilient to moderate noise but struggles to differentiate process states under very high noise levels.

These experiments show that CLaP is robust to variations in sampling rate and to moderate noise. When data become excessively coarse or noisy, performance degrades gradually, indicating that the method's performance scales with data quality.

4.6 Discussion

CLaP's central strength is its high accuracy. The comparative analysis shows that it is significantly more accurate compared to six state-of-the-art competitors on the five benchmarks. We carefully designed this edge by splitting TSSD into a two-step process: segmentation and state identification; optimizing each part separately:

(a) Segmentation: We use ClaSP for segmentation, which is one of the most accurate TSS algorithms for univariate and multivariate [21] batch data as well as streaming TS [22]. Except for ClaSP2Feat, the other competitors either do not use a dedicated segmentation procedure (e.g. Time2State) or use one with inferior detection accuracy, such as AutoPlait.

(b) State Identification: Our proposed self-supervised classification and confused merging procedure leverages the predictive power of classification algorithms [49], which are more accurate than classical clustering strategies that do not use TS-specific feature engineering or label information, as i.e. implemented in TICC. Others rely on distance calculations [30], or use limited statistical models with assumptions (e.g. Time2State).

Both components substantially contribute to the overall performance, as the ablation study indicates for (a) and CLaP's advance over ClaSP2Feat shows for (b). Despite its strengths, CLaP has several weaknesses. First, it assumes that subsequences can fully capture the distinctive properties of process states. This assumption holds for TS with homogeneous segments that contain similar temporal patterns, but performance degrades when intra-state diversity is high, e.g. in the presence of high noise (see Figure 9, right) or concept drift. Second, CLaP's runtime is quadratic in both the TS length and the number of CPs. For very long sequences with many CPs, this leads to high latency and limits its direct use in streaming contexts. Although a streaming version of ClaSP exists [22], further work is required to make confused merging viable for real-time processing. Consequently, the current CLaP implementation is best suited to batch TS with homogeneous segments.

4.7 Satellite Image Data Use Case

We revisit the satellite image example from Subsection 3.2 in Figure 10. The TS (top) illustrates sensor data for 3 different crops (coloured in blue, orange, and green) across 9 segments. We computed the state sequences of CLaP, ClaSP2Feat, and Time2State (2nd from top to fourth). A well-fitting output annotates one distinct number to each observed state (e.g., 1 for blue, 2 for orange, and 3 for green). CLaP accurately identifies the boundaries of each segment and assigns correct state labels, resulting in the sequence: 1,2,3,1,2,3,1,2,3. ClaSP2Feat does not differentiate between the blue and orange segments and erroneously assigns different labels to the green segments. However, it correctly identifies some CPs. The result of Time2State is noisy and largely overestimates the total number of states. Nonetheless, it correctly labels many data points and identifies correct repetitions of segments. Figure 10 (bottom) shows the state sequences as diagrams, with nodes representing states and edges depicting transitions. The start and end states appear in green (blue), and transition probabilities are labelled. It protrudes that only CLaP's graph is isomorphic to the ground truth.

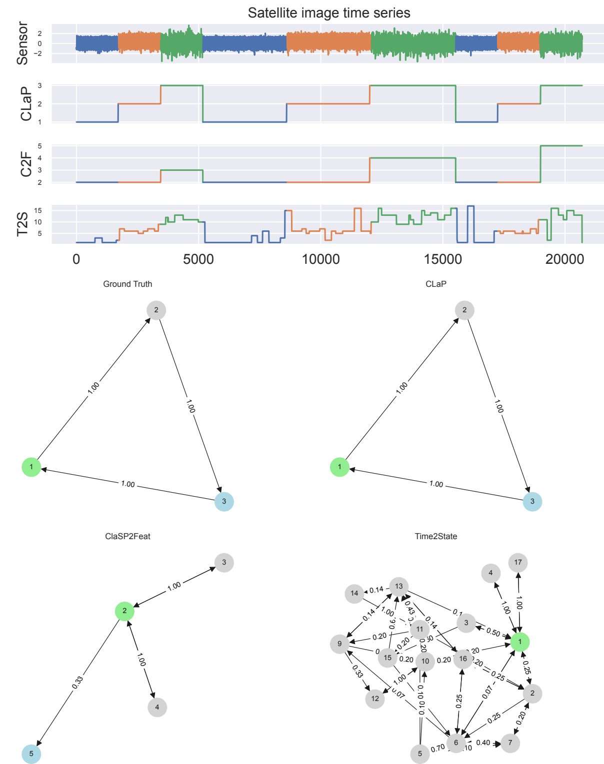


Figure 10: TS (top) shows satellite image data for 3 different crops (blue, orange, green) [56]. State sequences (2nd from top to fourth) shows results of CLaP, ClaSP2Feat, and Time2State. Bottom diagrams show graphical representations of ground truth and competitor results.

This use case demonstrates the accuracy of CLaP for long and complex time series data (20.7k data points, 9 segments, 3 states). Its results are also interpretable for human inspection.

5 RELATED WORK

The rapid growth of sensor data from IoT devices in *smart* applications, such as healthcare and factories, has driven a substantial boost in research of TS management and mining [41]. TS storage solutions include specialized databases [54], indices [17], and compression algorithms [44]. Unsupervised data analytics process the acquired data to automatically extract anomalies [7], motifs [57], or clusters [53]. This enables domain experts to make data-driven decisions, such as health professionals conducting gait analysis [73].

TSSD is a complex preprocessing step in a typical TS analysis workflow. Its main components, TS segmentation and clustering, have been researched both independently and in combination [65]. TSS can be formalized as an optimization problem, where each segment incurs a cost based on a discrepancy measure (e.g., mean-shift [52] or entropy [55]), and the number of segments [61] is penalized. Exact or approximate solvers, such as PELT [40] or Wild BinSeg [25], then compute the optimal segmentation. Adams et al. model segments as probability distributions and use a recursive

message-passing algorithm to infer the most recent CP [1]. This approach can be extended to detect short gradual changes [16], an extension of the original TSS problem [9]. A central limitation of these techniques is their dependence on domain-specific parameters, such as assumed value distributions or fitting cost functions. To overcome this, Katser et al. propose ensembling strategies, which increase robustness and accuracy [35]. Other domain-agnostic methods include FLUSS [26] and ClaSP [20], which achieve top-ranking results on recent TSS benchmarks [18, 23] and are capable of processing multivariate and streaming data. FLUSS measures the density of similar subsequences in potential segments as an arc curve, from which it extracts local minima that represent CPs [26]. ClaSP frames TSS as a collection of self-supervised subsequence classification problems and reports the segmentation with the highest cross-validation performance [20]. We use ClaSP in CLaP because it is hyper-parameter-free and makes few assumptions about segments, such as being mutually dissimilar.

After segmentation, individual segments must be assigned correct state labels, which can be achieved by clustering, for instance, by extracting equal-sized subsequences from segments, where the individual data points define the feature space. Partition-based algorithms, such as Lloyd's algorithm [45], PAM [37], or k-Shape [53], randomly partition the data and iteratively refine it until some convergence criterion is met [30]. Specifically for TS, the data is commonly assigned to clusters based on minimal distances to their centres, such as medoids. Holder et al. evaluated the accuracy of different distance measures and found MSM [59] and TWE [46] to be good choices [30]. Different classical clustering approaches include density-based methods, e.g., DBSCAN [24] and OPTICS [3], as well as hierarchical approaches, such as BIRCH [72]. These can be applied to the raw TS segments. However, Bonifati et al. report in a recent study that extracting and selecting features from TS before clustering improves performance [6]. Specifically, they compute statistics with tsfresh [10] from the data and calculate similarity measures between sequences. Features without variance are filtered, and the remaining ones are clustered. We use this approach in our baseline ClaSP2Feat, which combines an accurate segmentation and clustering procedure and achieves top-ranking results.

Besides the independent segmentation and clustering, the literature also contains integrated TSSD approaches [65]. HMMs, for instance, can model single TS with a limited number of states and probabilities for initialization, transition, and output [5]. By design, HMM states model single data points, not prolonged events. Matsubara et al. tackle this by grouping HMM states and learning transitions between groups, which reveals their CPs. AutoPlait implements this idea by iteratively refining the segmentation to optimize a cost function [47]. HDP-HSMM uses a different approach: it incorporates explicit-duration semi-Markovian properties to model flexible HMM state durations and employs a hierarchical Dirichlet process prior to learn the number of states from the TS directly [33]. Hallac et al. propose characterizing TS using a sliding window instead of modelling single data points. Each subsequence belongs to a cluster that is characterized by a correlation network. In TICC, the assignment of subsequences and the update of cluster parameters are iteratively refined using an expectation maximization algorithm [28]. The recent method Time2State also builds a sliding window but uses it to train a deep learning encoder by minimizing (maximizing) distances of intra-state (inter-state) subsequences. The resulting embedding is clustered using DPGMM to assign subsequences to state labels and to automatically learn their amount [66]. E2USD implements a similar deep-learning TSSD architecture, transforming each sliding window through compression, trend–seasonal decomposition and contrastive learning into a compact embedding that is subsequently clustered via DPGMM [42].

Our proposed CLaP also processes subsequences, but differs substantially from the existing methods. It is the first TSSD method that uses self-supervised learning [29] to exploit the predictive power of TS classification algorithms, which are more accurate than the unsupervised models used by the aforementioned methods that rely on classical techniques sensitive to hyper-parameters or TS characteristics. CLaP merges confused classes as long as their cgain increases, which clusters segments automatically in label space rather than in feature space.

Apart from state detection in single TS, current research also investigates the relationships among dimensions in MTS and across different data sets. Wang et al. examine high-dimensional TS in which a subset of channels must be automatically selected for accurate TSSD. They propose ISSD, an algorithm that identifies a subset of TS dimensions with high-quality states, quantified by channel set completeness, and solves this selection in an optimization problem [68]. StaCo is another algorithm designed to measure overall, partial, and time-lagged correlations between states in heterogeneous TS. It produces a state correlation matrix, from which state affiliations across data sets can be derived [67].

6 CONCLUSION

We proposed CLaP (Classification Label Profile), a new time series state detection algorithm that uses self-supervised classification techniques for segmentation and labelling. Our experimental evaluation shows that CLaP combines high-scoring design choices, leading to a statistically significant improvement in accuracy over six competitors on 405 univariate and multivariate TS from five benchmarks. CLaP also offers the best tradeoff between accuracy and runtime. It computes state labels for tens of thousands of data points in just a few minutes, providing a useful TS annotation that can be employed as input for advanced data analytics or as decision support for domain experts.

Limitations of CLaP include its sensitivity to the segmentation provided by ClaSP. While confused merging can correct false-positive CPs, it cannot relocate imprecise ones. It also has a quadratic runtime complexity with respect to TS size and the number of CPs. Some TS applications, such as predictive maintenance, require data analysis solutions capable of processing streaming data.

In future work, we plan to explore sampling, batching, and hierarchical techniques to improve the scalability of CLaP. We want to explore the incorporation of domain knowledge to CLaP to guide its TSSD. Experts often have specific information from visual inspection or access to exogenous data which could increase the quality of state detection, e.g. by indicating some CPs or state labels of single measurements. We further aim to investigate state detection in TS collections to identify recurring states and transitions across instances, potentially revealing even more semantics of the data-generating process.

REFERENCES

[1] Ryan P. Adams and David John Cameron MacKay. 2007. Bayesian Online Changepoint Detection. *arXiv: Machine Learning* (2007).

[2] Md Atiqur Rahman Ahad, Anindya Das Antar, and Masud Ahmed. 2021. IoT Sensor-Based Activity Recognition - Human Activity Recognition. In *Intelligent Systems Reference Library.*

[3] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. 1999. OPTICS: ordering points to identify the clustering structure. In *ACM SIGMOD Conference.*

[4] Anthony Bagnall, Michael Flynn, James Large, Jason Lines, and Matthew Middlehurst. 2020. On the usage and performance of the hierarchical vote collective of transformation-based ensembles version 1.0 (hive-cote v1. 0). In *Advanced Analytics and Learning on Temporal Data: 5th ECML PKDD Workshop, AALTD 2020, Ghent, Belgium, September 18, 2020, Revised Selected Papers 6.* Springer, 3–18.

[5] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics).* Springer.

[6] Angela Bonifati, Francesco Del Buono, Francesco Guerra, Miki Lombardi, and Donato Tiano. 2023. Interpretable Clustering of Multivariate Time Series with Time2Feat. *Proceedings of the VLDB Endowment* 16 (2023), 3994–3997.

[7] Paul Boniol, Michele Linardi, Federico Roncallo, Themis Palpanas, Mohammed Meftah, and Emmanuel Remy. 2021. Unsupervised and scalable subsequence anomaly detection in large data series. *VLDB Journal* 30 (2021), 909–931.

[8] Leo Breiman. 2004. Technical note: Some properties of splitting criteria. *Machine Learning* 24 (2004), 41–47.

[9] Louis Carpentier, Len Feremans, Wannes Meert, and Mathias Verbeke. 2024. Pattern-based Time Series Semantic Segmentation with Gradual State Transitions. In *SIAM International Conference on Data Mining.*

[10] Maximilian Christ, Nils Braun, Julius Neuffer, and A. Kempa-Liehr. 2018. Time Series FeatuRe Extraction on basis of Scalable Hypothesis tests (tsfresh - A Python package). *Neurocomputing* 307 (2018), 72–77.

[11] CLaP Code and Raw Results. 2025. https://github.com/ermshaua/classification-label-profile.

[12] Angus Dempster, Franccois Petitjean, and Geoffrey I. Webb. 2019. ROCKET: exceptionally fast and accurate time series classification using random convolutional kernels. *Data Mining and Knowledge Discovery* 34 (2019), 1454 – 1495.

[13] Angus Dempster, Daniel F. Schmidt, and Geoffrey I. Webb. 2022. Hydra: competing convolutional kernels for fast and accurate time series classification. *Data Mining and Knowledge Discovery* 37 (2022), 1779–1805.

[14] Angus Dempster, Daniel F. Schmidt, and Geoffrey I. Webb. 2024. QUANT: A Minimalist Interval Method for Time Series Classification. *Data Mining and Knowledge Discovery* (2024), 1–26.

[15] Janez Demšar. 2006. Statistical Comparisons of Classifiers over Multiple Data Sets. *Journal of Machine Learning Research* 7 (2006), 1–30.

[16] Erick Draayer, H. Cao, and Yifan Hao. 2021. Reevaluating the Change Point Detection Problem with Segment-based Bayesian Online Detection. *Proceedings of the 30th ACM International Conference on Information & Knowledge Management* (2021).

[17] Karima Echihabi, Panagiota Fatourou, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2022. Hercules Against Data Series Similarity Search. *Proceedings of the VLDB Endowment* 15 (2022), 2005–2018.

[18] Arik Ermshaus, Patrick Schäfer, Anthony Bagnall, Thomas Guyet, Georgiana Ifrim, Vincent Lemaire, Ulf Leser, Colin Leverger, and Simon Malinowski. 2023. Human Activity Segmentation Challenge @ ECML/PKDD'23. In *8th Workshop on Advanced Analytics and Learning on Temporal Data.*

[19] Arik Ermshaus, Patrick Schäfer, and Ulf Leser. 2022. Window Size Selection In Unsupervised Time Series Analytics: A Review and Benchmark. *7th Workshop on Advanced Analytics and Learning on Temporal Data* (2022).

[20] Arik Ermshaus, Patrick Schäfer, and Ulf Leser. 2023. ClaSP: parameter-free time series segmentation. *Data Mining and Knowledge Discovery* 37 (2023), 1262 – 1300.

[21] Arik Ermshaus, Patrick Schäfer, and Ulf Leser. 2024. Multivariate Human Activity Segmentation: Systematic Benchmark with ClaSP. *9th Workshop on Advanced Analytics and Learning on Temporal Data* (2024).

[22] Arik Ermshaus, Patrick Schäfer, and Ulf Leser. 2024. Raising the ClaSS of Streaming Time Series Segmentation. *Proceedings of the VLDB Endowment* 17, 8 (2024), 1953–1966.

[23] Arik Ermshaus, Sunita Singh, and Ulf Leser. 2023. Time Series Segmentation Applied to a New Data Set for Mobile Sensing of Human Activities. In *EDBT/ICDT Workshops.*

[24] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Knowledge Discovery and Data Mining.*

[25] Piotr Fryzlewicz. 2014. Wild binary segmentation for multiple change-point detection. *The Annals of Statistics* 42, 6 (2014), 2243.

[26] Shaghayegh Gharghabi, Chin-Chia Michael Yeh, Yifei Ding, Wei Ding, Paul R. Hibbing, Samuel R LaMunion, Andrew Kaplan, Scott E. Crouter, and Eamonn J. Keogh. 2018. Domain agnostic online semantic segmentation for multi-dimensional time series. *Data Mining and Knowledge Discovery* 33 (2018), 96 – 130.

[27] Antoine Guillaume, Christel Vrain, and Wael Elloumi. 2021. Random Dilated Shapelet Transform: A New Approach for Time Series Shapelets. In *International Conferences on Pattern Recognition and Artificial Intelligence.*

[28] David Hallac, Sagar Vare, Stephen P. Boyd, and Jure Leskovec. 2017. Toeplitz Inverse Covariance-Based Clustering of Multivariate Time Series Data. *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2017).

[29] Shohei Hido, Tsuyoshi Idé, Hisashi Kashima, Harunobu Kubo, and Hirofumi Matsuzawa. 2008. Unsupervised Change Analysis Using Supervised Learning. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining.*

[30] Christopher Holder, Matthew Middlehurst, and A. Bagnall. 2022. A Review and Evaluation of Elastic Distance Functions for Time Series Clustering. *Knowledge and Information Systems* 66 (2022), 765–809.

[31] Mikhail Hushchyn and Andrey Ustyuzhanin. 2021. Generalization of change-point detection in time series data based on direct density ratio estimation. *Journal of Computational Science* 53 (2021), 101385.

[32] Shima Imani, Alireza Abdoli, Ali Beyram, and Eamonn J. Keogh. 2021. Multi-Window-Finder: Domain Agnostic Window Size for Time Series Data. In *MileTS'21: 7th KDD Workshop on Mining and Learning from Time Series.*

[33] Matthew J. Johnson and Alan S. Willsky. 2010. The Hierarchical Dirichlet Process Hidden Semi-Markov Model. In *Conference on Uncertainty in Artificial Intelligence.*

[34] Ameeth Kanawaday and Aditya Sane. 2017. Machine learning for predictive maintenance of industrial machines using IoT sensor data. *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)* (2017), 87–90.

[35] Iurii D. Katser, Viacheslav Kozitsin, Victor Lobachev, and Ivan Maksimov. 2021. Unsupervised Offline Changepoint Detection Ensembles. *Applied Sciences* 11 (2021), 4280.

[36] Iurii D. Katser and Vyacheslav O. Kozitsin. 2020. Skoltech Anomaly Benchmark (SKAB).

[37] Leonard Kaufman and Peter J. Rousseeuw. 1990. *Partitioning Around Medoids (Program PAM).* John Wiley & Sons, Ltd, Chapter 2, 68–125.

[38] Bob Kemp, Aeilko H. Zwinderman, Bert Tuk, Hilbert A. C. Kamphuisen, and Josefien J. L. Oberye. 2000. Analysis of a sleep-dependent neuronal feedback loop: the slow-wave microcontinuity of the EEG. *IEEE Transactions on Biomedical Engineering* 47 (2000), 1185–1194.

[39] Rebecca Killick, Paul Fearnhead, and Idris Arthur Eckley. 2011. Optimal detection of changepoints with a linear computational cost. *J. Amer. Statist. Assoc.* 107 (2011), 1590 – 1598.

[40] Rebecca Killick, Paul Fearnhead, and Idris A. Eckley. 2012. Optimal detection of changepoints with a linear computational cost. *J. Amer. Statist. Assoc.* 107 (2012), 1590 – 1598.

[41] Rajalakshmi Krishnamurthi, Adarsh Kumar, Dhanalekshmi Gopinathan, Anand Nayyar, and Basit Qureshi. 2020. An Overview of IoT Sensor Data Processing, Fusion, and Analysis Techniques. *Sensors (Basel, Switzerland)* 20 (2020).

[42] Zhichen Lai, Huan Li, Dalin Zhang, Yan Zhao, Weizhu Qian, and Christian S. Jensen. 2024. E2Usd: Efficient-yet-effective Unsupervised State Detection for Multivariate Time Series. *Proceedings of the ACM Web Conference 2024* (2024).

[43] Oscar D. Lara and Miguel A. Labrador. 2013. A Survey on Human Activity Recognition using Wearable Sensors. *IEEE Communications Surveys & Tutorials* 15 (2013), 1192–1209.

[44] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2022. Chimp: Efficient Lossless Floating Point Compression for Time Series Databases. *Proceedings of the VLDB Endowment* 15 (2022), 3058–3070.

[45] Stuart P. Lloyd. 1982. Least squares quantization in PCM. *IEEE Transactions on Information Theory* 28 (1982), 129–136.

[46] Pierre-François Marteau. 2007. Time Warp Edit Distance with Stiffness Adjustment for Time Series Matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31 (2007), 306–318.

[47] Yasuko Matsubara, Yasushi Sakurai, and Christos Faloutsos. 2014. Autoplait: Automatic mining of co-evolving time sequences. In *Proceedings of the ACM SIGMOD international conference on Management of data.* 193–204.

[48] Matthew Middlehurst and Anthony Bagnall. 2022. The FreshPRINCE: A Simple Transformation Based Pipeline Time Series Classifier. In *International Conferences on Pattern Recognition and Artificial Intelligence.*

[49] Matthew Middlehurst, Patrick Schafer, and A. Bagnall. 2023. Bake off redux: a review and experimental evaluation of recent time series classification algorithms. *Data Mining and Knowledge Discovery* 38 (2023), 1958–2031.

[50] George B. Moody and Roger G. Mark. 2001. The impact of the MIT-BIH Arrhythmia Database. *IEEE Engineering in Medicine and Biology Magazine* 20 (2001), 45–50.

[51] Xuan Vinh Nguyen, Julien Epps, and James Bailey. 2010. Information Theoretic Measures for Clusterings Comparison: Variants, Properties, Normalization and Correction for Chance. *Journal of Machine Learning Research* 11 (2010), 2837–2854.

[52] E. S. Page. 1955. A test for a change in a parameter occurring at an unknown point. *Biometrika* 42 (1955), 523–527.

[53] John Paparrizos and Luis Gravano. 2016. k-Shape: Efficient and Accurate Clustering of Time Series. *SIGMOD Record* 45 (2016), 69–76.

[54] Tuomas Pelkonen, Scott Franklin, Paul Cavallaro, Qi Huang, Justin Meza, Justin Teller, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-Memory Time Series Database. *Proceedings of the VLDB Endowment* 8 (2015), 1816–1827.

[55] Piotr Sadri, Yongli Ren, and Flora D Salim. 2017. Information gain-based metric for recognizing transitions in human activities. *Pervasive and Mobile Computing* 38 (2017), 92–109.

[56] Patrick Schäfer, Arik Ermshaus, and Ulf Leser. 2021. ClaSP - Time Series Segmentation. *Proceedings of the 30th ACM International Conference on Information & Knowledge Management* (2021).

[57] Patrick Schäfer and Ulf Leser. 2022. Motiflets - Simple and Accurate Detection of Motifs in Time Series. *Proceedings of the VLDB Endowment* 16 (2022), 725–737.

[58] Patrick Schäfer and Ulf Leser. 2023. WEASEL 2.0: a random dilated dictionary transform for fast, accurate and memory constrained time series classification. *Machine Learning* 112 (2023), 4763–4788.

[59] Alexandra Stefan, Vassilis Athitsos, and Gautam Das. 2013. The Move-Split-Merge Metric for Time Series. *IEEE Transactions on Knowledge and Data Engineering* 25 (2013), 1425–1438.

[60] Minh-Quang Tran, Hoang-Phuong Doan, Viet Q. Vu, and Lien T. Vu. 2022. Machine Learning and IoT-based Approach for Tool Condition Monitoring: A Review and Future Prospects. *Measurement* (2022).

[61] Charles Truong, Laurent Oudre, and Nicolas Vayatis. 2020. Selective review of offline change point detection methods. *Signal Processing* 167 (2020).

[62] John Wilder Tukey. 1977. Exploratory data analysis. *Reading/Addison-Wesley* (1977).

[63] Gerrit JJ van den Burg and Christopher KI Williams. 2020. An evaluation of change point detection algorithms. *arXiv* (2020).

[64] Chen Wang, Xiangdong Huang, Jialin Qiao, Tian Jiang, Lei Rui, Jinrui Zhang, Rong Kang, Julian Feinauer, Kevin A McGrail, Peng Wang, et al. 2020. Apache IoTDB: Time-series database for internet of things. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2901–2904.

[65] Chengyu Wang, Xiong lve Li, Tongqing Zhou, and Zhiping Cai. 2024. Unsupervised Time Series Segmentation: A Survey on Recent Advances. *Computers, Materials & Continua* (2024).

[66] Chengyu Wang, Kui Wu, Tongqing Zhou, and Zhiping Cai. 2023. Time2State: An Unsupervised Framework for Inferring the Latent States in Time Series Data. *Proceedings of the ACM on Management of Data* 1 (2023), 1 – 18.

[67] Chengyu Wang, Yuan Yuan, Tongqing Zhou, and Zhiping Cai. 2024. Detecting State Correlations between Heterogeneous Time Series. *Proceedings of the 2024 2nd International Conference on Advances in Artificial Intelligence and Applications* (2024).

[68] Chengyu Wang, Tongqing Zhou, Lin Chen, Shan Zhao, and Zhiping Cai. 2025. ISSD: Indicator Selection for Time Series State Detection. *Proceedings of the ACM on Management of Data* (2025).

[69] J. H. Woollam, Jannes Munchmeyer, Frederik Tilmann, Andreas Rietbrock, Dietrich Lange, Thomas Bornstein, Tobias Diehl, Carlo Giunchi, Florian Haslinger, Dario Jozinovi'c, Alberto Michelini, Joachim Saul, and Hugo Soto. 2022. SeisBench—A Toolbox for Machine Learning in Seismology. *Seismological Research Letters* (2022).

[70] Yongchao Ye, Xinting Zhu, Xuejin X. Shen, Xiaoyu Chen, Lishuai Li, and S. Joe Qin. 2024. Diffusion-based Method for Satellite Pattern-of-Life Identification. *arXiv* (2024).

[71] Jie Yin, Qiang Yang, and Jeffrey Junfeng Pan. 2008. Sensor-Based Abnormal Human-Activity Detection. *IEEE Transactions on Knowledge and Data Engineering* 20 (2008), 1082–1090.

[72] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. 1996. BIRCH: an efficient data clustering method for very large databases. In *ACM SIGMOD Conference*.

[73] Lin Zhou, Eric Fischer, Clemens Markus Brahms, Urs Granacher, and Bert Arnrich. 2023. DUO-GAIT: A gait dataset for walking under dual-task and fatigue conditions with inertial measurement units. *Scientific Data* 10 (2023).