SIGMOD Officers, Committees, and Awardees

Chair
Divyakant Agrawal
Department of Computer Science
UC Santa Barbara
Santa Barbara, California
USA
+1 805 893 4385
agrawal <at> cs.ucsb.edu

Vice-Chair
Fatma Ozcan
Systems Research Group
Google
Sunnyvale, California
USA
+1 669 264 9238
Fozcan <at> google.com

Secretary/Treasurer
Rachel Pottinger
Department of Computer Science
University of British Columbia
Vancouver
Canada
+1 604 822 0436
Rap <at> cs.ubc.ca

SIGMOD Executive Committee:
Divyakant Agrawal (Chair), Fatma Ozcan (Vice-chair), Rachel Pottinger (Treasurer), Juliana Freire (Previous SIGMOD Chair), Chris Jermaine (SIGMOD Conference Coordinator and ACM TODS Editor in Chief), Rada Chirkova (SIGMOD Record Editor), Angela Bonifati (2022 SIGMOD PC co-chair), Amr El Abbadi (2022 SIGMOD PC co-chair), Floris Geerts (Chair of PODS), Sihem Amer-Yahia (SIGMOD Diversity and Inclusion Coordinator), Sourav S Bhowmick (SIGMOD Ethics)

Advisory Board:
Yannis Ioannidis (Chair), Phil Bernstein, Surajit Chaudhuri, Rakesh Agrawal, Joe Hellerstein, Mike Franklin, Laura Haas, Renee Miller, John Wilkes, Chris Olsten, AnHai Doan, Tamer Özsu, Gerhard Weikum, Stefano Ceri, Beng Chin Ooi, Timos Sellis, Sunita Sarawagi, Stratos Idreos, and Tim Kraska

SIGMOD Information Directors:
Sourav S Bhowmick, Nanyang Technological University
Byron Choi, Hong Kong Baptist University

Associate Information Directors:
Huiping Cao (SIGMOD Record), Georgia Koutrika (Blogging), Wim Martens (PODS)

SIGMOD Record Editor-in-Chief:
Rada Chirkova, NC State University

SIGMOD Record Associate Editors:
Lyublena Antova, Marcelo Arenas, Manos Athanassoulis, Renata Borovica-Gajic, Vanessa Braganhoto, Susan Davidson, Aaron J. Elmore, Wook-Shin Han, Wim Martens, Kyriakos Mouratidis, Dan Olteanu, Tamer Özsu, Kenneth Ross, Pınar Tözün, Immanuel Trummer, Yannis Velegrakis, Marianne Winslett, and Jun Yang

SIGMOD Conference Coordinator:
Chris Jermaine, Rice University

PODS Executive Committee:
Floris Geerts (chair), Pablo Barcelo, Leonid Libkin, Hung Q. Ngo, Reinhard Pichler, Dan Suciu

Sister Society Liaisons:
Raghu Ramakrishnan (SIGKDD), Yannis Ioannidis (EDBT Endowment), Christian Jensen (IEEE TKDE)

SIGMOD Awards Committee:
Yanlei Diao (Chair), Volker Markl, Renée Miller, H. V. Jagadish, Stefano Ceri, Samuel Madden, Wang-Chiew Tan
Jim Gray Doctoral Dissertation Award Committee:
Vanessa Braganholo (co-chair), Wolfgang Lehner (co-chair), Bailu Ding, Daniel Deutch, Joy Arulraj, Azza Abouzied, Gustavo Alonso, Huanchen Zhang

SIGMOD Edgar F. Codd Innovations Award
For innovative and highly significant contributions of enduring value to the development, understanding, or use of database systems and databases. Recipients of the award are the following:

Dan Suciu (2022)

SIGMOD Systems Award
For technical contributions that have had significant impact on the theory or practice of large-scale data management systems.


SIGMOD Contributions Award
For significant contributions to the field of database systems through research funding, education, and professional services. Recipients of the award are the following:

SIGMOD Jim Gray Doctoral Dissertation Award

SIGMOD has established the annual SIGMOD Jim Gray Doctoral Dissertation Award to recognize excellent research by doctoral candidates in the database field. Recipients of the award are the following:

- **2006 Winner**: Gerome Miklau. **Honorable Mentions**: Marcelo Arenas and Yanlei Diao
- **2007 Winner**: Boon Thau Loo. **Honorable Mentions**: Xifeng Yan and Martin Theobald
- **2008 Winner**: Ariel Fuxman. **Honorable Mentions**: Cong Yu and Nilesh Dalvi
- **2009 Winner**: Daniel Abadi. **Honorable Mentions**: Bee-Chung Chen and Ashwin Machanavajjhala
- **2010 Winner**: Christopher Ré. **Honorable Mentions**: Soumyadeb Mitra and Fabian Suchanek
- **2011 Winner**: Stratos Idreos. **Honorable Mentions**: Todd Green and Karl Schnaitterz
- **2012 Winner**: Ryan Johnson. **Honorable Mention**: Bogdan Alexe
- **2013 Winner**: Sudipto Das. **Honorable Mention**: Herodotos Herodotou and Wenchao Zhou
- **2014 Winners**: Aditya Parameswaran and Andy Pavlo.
- **2015 Winner**: Alexander Thomson. **Honorable Mentions**: Marina Drosou and Karthik Ramachandra
- **2016 Winner**: Paris Koutris. **Honorable Mentions**: Pinar Tozun and Alvin Cheung
- **2017 Winner**: Peter Bailis. **Honorable Mention**: Immanuel Trummer
- **2018 Winner**: Viktor Leis. **Honorable Mention**: Luis Galárraga and Yongjoo Park
- **2019 Winner**: Joy Arulraj. **Honorable Mention**: Bas Ketsman
- **2020 Winner**: Jose Faleiro. **Honorable Mention**: Silu Huang
- **2021 Winner**: Huanchen Zhang. **Honorable Mentions**: Erfan Zamanian, Maximilian Schleich, and Natacha Crooks
- **2022 Winner**: Chenggang Wu. **Honorable Mentions**: Pingcheng Ruan and Kexin Rong

A complete list of all SIGMOD Awards is available at: [https://sigmod.org/sigmod-awards/](https://sigmod.org/sigmod-awards/)

[Last updated: September 30, 2022]
Editor’s Notes

Welcome to the December 2022 issue of the ACM SIGMOD Record!

This issue starts with the Database Principles column featuring an article by Ketsman, Koch, Neven, and Vandevenoort. The article provides an introduction to concurrency control for database theorists, with a focus on the notion of robustness that allows one to achieve transaction serializability at lower concurrency-control costs. The authors introduce the theory of robustness, present transaction programs and templates, highlight recent results in the area, and outline open problems.

The Surveys column presents an article by Schlegel and Sattler. The article discusses system support for machine-learning (ML) artifact management as an essential building block to achieve comparability, reproducibility, and traceability of artifacts created and used within the ML lifecycle. Using criteria that enable systematic assessment of artifact-management systems, the authors assess more than 60 systems and platforms from academia and industry that support the management of ML lifecycle artifacts. The article presents the assessment results and findings.

The Systems and Prototypes column features an article by Benedikt, Cooper, Germano, Gyorkei, Tsamoura, Moore, and Ortiz. In the context of reasoning-based query planning, which has been studied in relational data integration, the Semantic Web, and query reformulation, the authors overview PDQ 2.0, a platform whose goal is to provide infrastructure support for reasoning-based optimization in the relational realm.

The Reminiscences on Influential Papers column features contributions by Tilmann Rabl, Avrilia Floratou, Arun Kumar, and Danica Porobic.

The Advice to Mid-Career Researchers column presents a contribution by Laks V.S. Lakshmanan entitled “Mid-Career Researcher, huh? What just Changed?”. The article shares advice to newly minted associate professors on choosing a research strategy for the new career span, on supervising students and mentoring junior colleagues, on selecting service with impact, and on ways to expose graduate students and postdocs to new and exciting opportunities.

The article featured in the Future of Data(base) Education column is contributed by Aivaloglou, Fletcher, Liut, and Miedema, organizers of the very successful first international workshop on Data Systems Education: Bridging Education Practice with Education Research (DataEd’22), which was held in conjunction with SIGMOD ’22. The article summarizes the workshop’s main results, observations, and emerging research directions.

The DBrainstorming column, whose goal is to discuss new and potentially controversial ideas that might be of interest and potentially of benefit to the research community, presents in this issue an article by Pike that considers the problem of managing data and code versions, in collaborative data science and beyond. The author articulates the goal of developing approaches that consider both code and data in version merging, and lists key challenges addressing which would move us closer to addressing this goal.

The Distinguished Profiles column features an interview with Chenggang Wu, co-founder and CTO of Aqueduct and recipient of the 2022 ACM SIGMOD Jim Gray dissertation award for his thesis entitled “The Design of Any-Scale Serverless Infrastructure with Rich Consistency Guarantees.” Chenggang’s PhD is from UC Berkeley. In the interview, Chenggang discusses his thesis and the challenges that he
had to address in the dissertation work, explains how the thesis ideas and solutions have led to the founding of the Aqueduct company, and shares advice for today’s graduate students.

The Industry Perspectives column presents an article by Tian that provides the research community with an industrial perspective on the graph-database landscape. The information provided in the article can help graph researchers better understand the industry trends and challenges faced by the industry, thus inviting solutions that would help address the problems.

The article by Perscheid, Plattner, Ritter, Schlosser, and Teusner featured in the Research Centers column presents the Enterprise Platform and Integration Concepts (EPIC) research center at the Hasso Plattner Institute (HPI) at the University of Potsdam. Research at EPIC focuses on next-generation database management and enterprise software, with the goals of maximizing performance and of improving cost efficiency and business value. The article motivates the research topics pursued at the EPIC center, highlights the key contributions, and outlines collaboration activities across the projects.

The issue closes with an article by Bhowmick in the Reports column. The author presents issues related to the emergence of dense co-authorship networks formed by conference review-board members, and advocates for rethinking the traditional way review are formed, with the goal of mitigating the issues.

On behalf of the SIGMOD Record Editorial board, I hope that you enjoy reading the December 2022 issue of the SIGMOD Record!

Your submissions to the SIGMOD Record are welcome via the submission site:
https://mc.manuscriptcentral.com/sigmodrecord

Prior to submission, please read the Editorial Policy on the SIGMOD Record’s website:
https://sigmodrecord.org/sigmod-record-editorial-policy/

Rada Chirkova
December 2022

Past SIGMOD Record Editors:

Concurrency control for database theorists

Bas Ketsman
Vrije Universiteit Brussel,
Belgium

Christoph Koch
École Polytechnique Fédérale
de Lausanne, Switzerland

Frank Neven
UHasselt, Data Science
Institute, ACSL, Diepenbeek,
Belgium

Brecht Vandevoort
UHasselt, Data Science
Institute, ACSL, Diepenbeek,
Belgium

ABSTRACT

The aim of this paper is to serve as a lightweight introduction to concurrency control for database theorists through a uniform presentation of the work on robustness against Multiversion Read Committed and Snapshot Isolation.

1 Introduction

In this paper, we take a simplistic approach and view a transaction as a sequence of reads and writes to database objects. For instance, $T_1 = R_1[t] W_1[v] C_1$ is a transaction that first reads an object $t$, writes to an object $v$ and then commits. Transactions are considered to be atomic: they are executed completely or not at all, and once committed they can not be rolled back. A transaction workload then consists of a set of transactions. At its core, database concurrency control is a balancing act between two conflicting desires: the wish to increase transaction throughput via concurrent access, that is, interleaving of the execution of transactions, and the desire for data consistency for which concurrent access sometimes needs to be restricted.

The holy standard in concurrency control for guaranteeing data consistency is serializability. A concurrent execution of a transaction workload is serializable when it is equivalent to a serial, that is, non-interleaved, execution of the transactions. Serializability guarantees that no data anomalies can occur. There are several concurrency control protocols that guarantee serializability: for instance, Strict Two-Phase Locking (S2PL) and Serializable Snapshot Isolation (SSI). As these protocols restrict concurrent access and typically have a negative effect on transaction throughput, databases offer a way to trade in data consistency for an increased level of concurrency through the mechanism of isolation levels that are less strict than serializability. Examples of such weaker isolation levels are, for instance, Multiversion Read Committed (RC) and Snapshot Isolation (SI). These isolation levels are less restrictive but can induce data anomalies and therefore, in general, do not guarantee serializability.

However, there are situations when a group of transactions can be executed at an isolation level lower than serializability without causing any errors. In this way, we get the higher isolation guarantees of serializability for free in exchange for a lower isolation level, which is typically implementable with a less expensive concurrency control mechanism. This formal property is called robustness [12, 10]: a set of transactions $T$ is called robust against a given isolation level if every possible interleaving of the transactions in $T$ that is allowed under the specified isolation level is serializable. There is a famous example that is part of database folklore: the TPC-C benchmark [16] is robust against Snapshot Isolation (SI), so there is no need to run a stronger, and more expensive, concurrency control algorithm than SI if the workload is just TPC-C. This has played a role in the incorrect choice of SI as the general concurrency control algorithm for isolation level Serializable in Oracle and PostgreSQL (before version 9.1, cf. [13]).

In this paper, we present a gentle introduction to the theory of robustness. In particular, we consider the isolation levels that are offered by systems like Postgres and Oracle: RC, SI and SSI. A main technical tool in the study of robustness is that of a split schedule. It is the canonical form of a counterexample schedule witnessing non-robustness and lies at the basis of polynomial time algorithms for the robustness problem.

A more complete survey and more high level account on robustness can be found in [19]. A much more detailed exposition can be found in Vandevoort’s Phd Thesis [17]. For deeper exploration of the theoretical aspects of concurrency control, we refer to the excellent but rather outdated book by Pa...
General references for concurrency control are, e.g., [22, 5, 6], or any recent textbook on database systems.

This paper is further organized as follows. We introduce the necessary terminology on transactions and schedules in Section 2 and discuss serializability in Section 3.2. We define the isolation levels RC, SI, and SSI in Section 4. In Section 5, we consider the robustness problem. We discuss transaction programs and templates in Section 6, and conclude in Section 7.

2 Definitions

2.1 Transactions

We fix an infinite set of objects Obj. For an object \( t \in \text{Obj} \), we denote by \( R[t] \) a read operation on \( t \) and by \( W[t] \) a write operation on \( t \). We also assume a special commit operation denoted by \( C \). A transaction \( T \) over \( \text{Obj} \) is a sequence of read and write operations on objects in \( \text{Obj} \) followed by a commit. In the sequel, we leave the set of objects \( \text{Obj} \) implicit when it is clear from the context and just say transaction rather than transaction over \( \text{Obj} \).

Formally, we model a transaction as a linear order \( (T, \leq_T) \), where \( T \) is the set of (read, write and commit) operations occurring in the transaction and \( \leq_T \) encodes the ordering of the operations. As usual, we use \( <_T \) to denote the strict ordering. For a transaction \( T \), we use \( \text{first}(T) \) to refer to the first operation in \( T \).

When considering a set \( \mathcal{T} \) of transactions, we assume that every transaction in the set has a unique id \( i \) and write \( T_i \) to make this id explicit. Similarly, to distinguish the operations of different transactions, we add this id as a subscript to the operation. That is, we write \( W_i[t] \) and \( R_i[t] \) to denote a \( W[t] \) and \( R[t] \) occurring in transaction \( T_i \); similarly \( C_i \) denotes the commit operation in transaction \( T_i \). This convention is consistent with the literature (see, e.g. [3, 12]). To avoid ambiguity of notation, we assume that a transaction performs at most one write and one read operation per object. The latter is a common assumption (see, e.g. [12]). All our results carry over to the more general setting in which multiple writes and reads per object are allowed.

2.2 Schedules

A (multiversion) schedule \( s \) over a set \( \mathcal{T} \) of transactions is a tuple \( (O_s, \leq_s, \triangleleft_s, v_s) \) where

- \( O_s \) is the set containing all operations of transactions in \( \mathcal{T} \) as well as a special operation \( op_0 \), conceptually writing the initial versions of all existing objects,
- \( \leq_s \) encodes a linear ordering of \( O_s \) (with \( a \leq_s b \) and \( b \leq_s a \) meaning \( a = b \)),
- \( \triangleleft_s \) is a version order providing for each object \( t \) a total order over all write operations on \( t \) occurring in \( s \), and,
- \( v_s \) is a version function mapping each read operation \( a \) in \( s \) to either \( op_0 \) or to a write operation in \( s \).

We require that \( op_0 \leq_s a \) for every operation \( a \in O_s \), \( op_0 \triangleleft_s a \) for every write operation \( a \in O_s \), and that \( a \triangleleft_T b \) implies \( a \triangleleft_s b \) for every \( T \in \mathcal{T} \) and every \( a, b \in T \). We furthermore require that for every read operation \( a \), \( v_s(a) \triangleleft_s a \) and, if \( v_s(a) \neq op_0 \), then the operation \( v_s(a) \) is on the same object as \( a \). Intuitively, \( op_0 \) indicates the start of the schedule, the order of operations in \( s \) is consistent with the order of operations in every transaction \( T \in \mathcal{T} \), and the version function maps each read operation \( a \) to the operation that wrote the version observed by \( a \). If \( v_s(a) = op_0 \), then \( a \) observes the initial version of this object. The version order \( \triangleleft_s \) represents the order in which different versions of an object are installed in the database. For a pair of write operations on the same object, this version order does not necessarily coincide with \( \leq_s \). For example, under RC and SI the version order is based on the commit order instead.

We say that a schedule \( s \) is a single version schedule if \( \triangleleft_s \) agrees with \( \leq_s \) and every read operation always reads the last written version of the object. Formally, for each pair of write operations \( a \) and \( b \) on the same object, \( a \triangleleft_s b \) iff \( a \leq_s b \), and for every read operation \( a \) there is no write operation \( c \) on the same object as \( a \) with \( v_s(a) \triangleleft_s c \). A single version schedule over a set of transactions \( T \) is single version serial if its transactions are not interleaved with operations from other transactions. That is, for every \( a, b, c \in O_s \) with \( a \triangleleft_s b \triangleleft_s c \) and \( a, c \in T \) implies \( b \in T \) for every \( T \in \mathcal{T} \).

The absence of aborts in our definition of schedule is consistent with the common assumption [12, 4] that an underlying recovery mechanism will rollback aborted transactions. We only consider isolation levels that only read committed versions. Therefore there will never be cascading aborts.

Example 2.1. As a running example, consider
the set of transactions \( T = \{ T_1, T_2, T_3 \} \) with
\[
T_1 = R_1[t] W_1[v] C_1; \quad T_2 = R_2[v] W_2[q] C_2; \quad \text{and}, \quad T_3 = R_3[q] W_3[t] W_3[q] C_3.
\]

Let \( s_1 \) be the schedule over \( T \) where the ordering \( \leq s_1 \) of operations is
\[
\{ R_3[q] \rightarrow op_0, R_3[t] \rightarrow R_1[t], R_2[v] \rightarrow W_3[v] \}.
\]
The version order \( \ll s_1 \) equals
\[\bullet op_0 \ll_s W_3[t] \text{ for object } t, \]
\[\bullet op_0 \ll_s W_1[v] \text{ for object } v, \text{ and,} \]
\[\bullet op_0 \ll_s W_2[q] \ll_s W_3[q] \text{ for object } q. \]
Furthermore, the version function \( v_{s_1} \) is
\[\{ R_3[q] \rightarrow op_0, R_1[t] \rightarrow C_1, R_2[v] \rightarrow W_3[v] \}. \]

Figure 1: Schedules \( s_1 \) and \( s_2 \) from Example 2.1 with solid (resp., dashed) arrows representing their version function (resp., version order).

### 3 Serializability

As explained in the introduction, a schedule is serializable when it is equivalent to a serial schedule. We therefore need to address precisely what equivalence in this context means. Furthermore, the equivalent serial schedule must additionally be single version, as multiversion serial schedules can still exhibit concurrency anomalies.

**Example 3.1.** Towards a multiversion serial schedule exhibiting a concurrency issue, consider the set of transactions \( T = \{ T_a, T_b \} \) with
\[
T_a = W_a[t] W_a[v] C_a; \quad \text{and}, \quad T_b = R_0[t] W_0[v] C_0.
\]

Let \( s \) be the schedule over \( T \) where the ordering \( \leq s \) of operations is
\[\{ R_0[t] \rightarrow op_0, R_0[v] \rightarrow W_0[v], \}. \]

Although the schedule \( s \) executes \( T_a \) before \( T_b \) in a serial fashion according to \( \leq s \), the version function \( v_s \) implies that \( T_b \) observes the original value of \( t \) and the updated value of \( v \). In other words, \( s \)
exhibits a concurrency anomaly where \( T_b \) observes only a partial update of \( T_a \).

Most of the literature considers conflict serializability even though it is not the most general notion. We define view, final-state, and conflict serializability.

3.1 View and final-state serializability

We start with view equivalence which requires that each read operation reads the result of the same write operation (as defined by the respective version function) in both schedules.

In essence this means that every operation must ‘view’ the same values in equivalent schedules. We introduce the graph \( D \), to make this explicit.\(^1\) For a schedule \( s \), \( D(s) \) has as nodes \( O_s \setminus \{ \text{op}_0 \} \) and there is an edge \( o \rightarrow_D o' \) iff

- \( o = \text{R}_i[t] < v, \text{W}_i[v] = o' \), that is, \( o' \) writes a value that can depend on an earlier read \( o \) in the same transaction; or,
- \( o' \) reads the value written by \( o \), that is, \( o = \text{W}_i[t] \) and \( o' = \text{R}_j[t] \) with \( i \neq j \), and \( o = v_i(o') \).

Intuitively, the edge \( o \rightarrow_D o' \) indicates that \( o \) must occur before \( o' \) when read dependencies need to be preserved.

The latter leads to the following notion of equivalence. Two schedules \( s \) and \( s' \) are view equivalent if they are over the same set \( \mathcal{T} \) of transactions and \( D(s) = D(s') \).

We now turn to final-state equivalence which only enforces dependencies for operations that contribute to the final value of at least one object. In other words, dependencies for a write operation to an object that is overwritten without being read, can be discarded. In this context, define \( LW(s) \subseteq O_s \) as those write operations \( \text{W}_i[t] \) that are the last in \( s \) to write \( t \). That is, \( \text{W}_i[t] \in LW(s) \) iff \( \text{W}_i[t] \in O_s \) and there is no \( \text{W}_j[t] \in O_s \) with \( \text{W}_i[t] < \text{W}_j[t] \). Now define \( D_1 \) as the graph obtained from \( D \) by removing every connected component (in \( D \)) not containing a write operation from \( LW(s) \).

Two schedules \( s \) and \( s' \) are final-state equivalent if they are over the same set \( \mathcal{T} \) of transactions and \( D_1(s) = D_1(s') \).

**Definition 3.1.** A schedule \( s \) is final-state serializable (resp., view serializable) if it is final-state equivalent (resp., view equivalent) to a single version serial schedule.

**Theorem 3.1.** [15] Deciding whether a schedule \( s \) is final-state or view serializable is \( \text{NP-complete}. \)

Notice that schedules that are view serializable are also final-state serializable, but not vice versa, as the next example shows.

**Example 3.2.** We consider the set of transactions \( \mathcal{T} = \{ T_4, T_5, T_6 \} \) with

\[
T_4 = \text{R}_4[t] \text{W}_4[t] C_4; \quad T_5 = \text{R}_5[t] \text{W}_5[t] C_5; \quad \text{and,} \quad T_6 = \text{W}_6[t] C_6,
\]

and the single-version schedule \( s_3 \) over \( \mathcal{T} \).

\[
\text{op}_0 \quad \text{R}_4[t] \quad \text{W}_4[t] \quad C_4 \quad \text{R}_5[t] \quad \text{W}_5[t] \quad C_5 \quad \text{W}_6[t] \quad \text{C}_6
\]

Schedule \( s_3 \) is final-state serializable because graph \( D_1(s_3) = (O_{s_3}, \emptyset) = D_1(s_4) \) with \( s_4 \) being the following single-version serial schedule over \( \mathcal{T} \).

\[
\text{op}_0 \quad \text{R}_4[t] \quad \text{W}_4[t] \quad C_4 \quad \text{R}_5[t] \quad \text{W}_5[t] \quad C_5 \quad \text{W}_6[t] \quad \text{C}_6
\]

We note that \( s_3 \) is not view-serializable because in every single-version serial schedule \( s \) over \( D(s) \) must have at least one of the following edges: \( \text{W}_5[t] \rightarrow \text{R}_4[t], \text{W}_4[t] \rightarrow \text{R}_5[t], \text{W}_6[q] \rightarrow \text{R}_4[q] \) or \( \text{W}_6[t] \rightarrow \text{R}_5[t] \). \( D(s_3) \) has no such edge.

3.2 Conflict Serializability

Let \( a_j \) and \( b_j \) be two operations on the same object \( t \) from different transactions \( T_j \) and \( T_i \) in a set of transactions \( \mathcal{T} \). We then say that \( b_i \) is conflicting with \( a_j \) if:

- (ww-conflict) \( b_i = \text{W}_i[t] \) and \( a_j = \text{W}_j[t] \); or,
- (wr-conflict) \( b_i = \text{W}_i[t] \) and \( a_j = \text{R}_j[t] \); or,
- (rw-conflict) \( b_i = \text{R}_i[t] \) and \( a_j = \text{W}_j[t] \).

In this case, we also say that \( b_i \) and \( a_j \) are conflicting operations. Furthermore, commit operations and the special operation \( \text{op}_0 \) never conflict with any other operation. When \( b_i \) and \( a_j \) are conflicting operations in \( \mathcal{T} \), we say that \( a_j \) depends on \( b_i \) in a schedule \( s \) over \( \mathcal{T} \), denoted \( b_i \rightarrow_s a_j \) if: \(^2\)

\(^1\)The notation \( D \) and \( D_1 \) comes from [15].

\(^2\)Throughout the paper, we adopt the following convention: a \( b \) operation can be understood as a ‘before’ while an \( a \) can be interpreted as an ‘after’.
• (ww-dependency) $b_i$ is ww-conflicting with $a_j$ and $b_i \ll_s a_j$; or,
• (wr-dependency) $b_i$ is wr-conflicting with $a_j$ and $b_i = v_s(a_j)$ or $b_i \ll_s v_s(a_j)$; or,
• (rw-antidependency) $b_i$ is rw-conflicting with $a_j$ and $v_s(b_i) \ll_s a_j$.

Intuitively, a ww-dependency from $b_i$ to $a_j$ implies that $a_j$ writes a version of an object that is installed after the version written by $b_i$. A wr-dependency from $b_i$ to $a_j$ implies that $b_i$ either writes the version observed by $a_j$, or it writes a version that is installed before the version observed by $a_j$. A rw-antidependency from $b_i$ to $a_j$ implies that $b_i$ observes a version installed before the version written by $a_j$.

Example 3.3. Consider schedule $s_2$ as defined in Example 2.1. In this schedule, the dependency $W^1_3[v] \rightarrow_s W^2_3[v]$ is a ww-dependency since $W^1_3[v] \ll_w W^2_3[v]$. Schedule $s_2$ furthermore has a wr-dependency from $W^1_3[v]$ to $R^2_3[v]$, as $v_s(R^2_3[v]) = W^1_3[v]$. The dependency $R^1_3[t] \rightarrow_s W^3_3[t]$ is a rw-antidependency, witnessed by $v_s(R^1_3[t]) = o_{p_0} \ll_s W^3_3[t]$.

Two schedules $s$ and $s'$ are conflict equivalent if they are over the same set of transactions and for every pair of conflicting operations $a_j$ and $b_i$, $b_i \rightarrow_s a_j$ iff $b_i \rightarrow_{s'} a_j$.

Definition 3.2. A schedule $s$ is conflict serializable if it is conflict equivalent to a single version serial schedule.

A serialization graph $SeG(s)$ for schedule $s$ over a set of transactions $T$ is the graph whose nodes are the transactions in $T$ and where there is an edge from $T_i$ to $T_j$ if $T_j$ has an operation $a_j$ that depends on an operation $b_i$ in $T_i$, thus with $b_i \rightarrow a_j$. Since we are usually not only interested in the existence of dependencies between operations, but also in the operations themselves, we assume the existence of a labeling function $\lambda$ mapping each edge to a set of pairs of operations. Formally, $(b_i, a_j) \in \lambda(T_i, T_j)$ iff there is an operation $a_j \in T_j$ that depends on an operation $b_i$ in $T_i$. For ease of notation, we choose to represent $SeG(s)$ as a set of quadruples $(T_i, b_i, a_j, T_j)$ denoting all possible pairs of transactions $T_i$ and $T_j$ with all possible choices of operations with $b_i \rightarrow a_j$. Henceforth, we refer to these quadruples simply as edges. Notice that edges cannot contain commit operations.

A cycle $\Gamma$ in $SeG(s)$ is a non-empty sequence of edges

$$(T_1, b_1, a_2, T_2), (T_2, b_2, a_3, T_3), \ldots, (T_n, b_n, a_1, T_1)$$

in $SeG(s)$, in which every transaction is mentioned exactly twice. Note that cycles are by definition simple. Here, transaction $T_i$ starts and concludes the cycle. For a transaction $T_i$ in $\Gamma$, we denote by $\Gamma[T_i]$ the cycle obtained from $\Gamma$ by letting $T_i$ start and conclude the cycle while otherwise respecting the order of transactions in $\Gamma$. That is, $\Gamma[T_i]$ is the sequence

$$(T_i, b_i, a_{i+1}, T_{i+1}) \cdots (T_n, b_n, a_1, T_1) (T_1, b_1, a_2, T_2) \cdots (T_{n-1}, b_{n-1}, a_i, T_i).$$

Theorem 3.2. (Implied by [1]). A schedule $s$ is conflict serializable iff $SeG(s)$ is acyclic.

The previous Theorem essentially extends the well known characterization of conflict serializability for single version schedules based on acyclicity of conflict graphs (see, e.g., [15]) towards multiversion schedules. In brief, the conflict graph $CG(s)$ for a single version schedule $s$ over a set of transactions $T$ is the graph whose nodes are the transactions in $T$ and where there is an edge from $T_i$ to $T_j$ if $T_j$ has an operation $a_j$ that is conflicting with an operation $b_i$ in $T_i$ and $a_i \ll_s b_j$. Note in particular that $CG(s)$ is defined solely in terms of conflicting operations and $\ll_s$, whereas $SeG(s)$ takes into account $\ll_s$ and $v_s$ as well. It can be proven that, if $s$ is a single version schedule, $CG(s)$ and $SeG(s)$ are identical.

Corollary 3.1. Deciding whether a schedule $s$ is conflict serializable is in PTIME.

Example 3.4. The serialization graphs for schedules $s_1$ and $s_2$ in Example 2.1 are given in Figure 2. Since $SeG(s_1)$ contains cycles, we conclude that $s_1$ is not conflict serializable. The serialization graph $SeG(s_2)$ on the other hand is acyclic, thereby implying that $s_2$ is conflict serializable. Indeed, $s_2$ is conflict equivalent to the single version serial schedule $T_1 \cdot T_3 \cdot T_2$.

Notice that conflict serializability implies view serializability but not vice versa.

Example 3.5. We consider the set of transactions $T = \{T_4, T_6, T_7\}$ with

$T_4 = R_4[t] \cdot W_4[t] \cdot C_4$;
$T_6 = W_6[t] \cdot C_6$; and,
$T_7 = W_7[t] \cdot C_7$.

We consider the following single-version schedule $s_5$ over $T$. 

SIGMOD Record, December 2022 (Vol. 51, No. 4)
A transaction exhibiting a concurrent write always exhibits a concurrent write. In schedule $s_1$ (and $s_2$) the transaction $T_3$ is equivalent with the single-version serial schedule as defined in Example 2.1. Indeed, $s_1$ is conflict-serializable since $\{T_4 \rightarrow T_7, T_7 \rightarrow T_4\} \subseteq ScG(s_5)$. But $s_5$ is view-equivalent with the single-version serial schedule $s_6$.

The relationship between the three notions for serializability is graphically depicted in Figure 3.

$$op_0$$

$$R_4[t] \rightarrow W_4[t] \rightarrow C_4$$

$$\rightarrow W_7[t] \rightarrow C_7$$

$$s_5$$

$$op_0$$

$$R_4[t] \rightarrow W_4[t] \rightarrow C_4$$

$$\rightarrow W_7[t] \rightarrow C_7$$

$$s_6$$

Indeed, $D(s_5) = D(s_6) = \{R_4[t] \rightarrow_D W_4[t]\}$. 

The relationship between the three notions for serializability is graphically depicted in Figure 3.

**4 Isolation Levels**

Most generally, an isolation level corresponds to a set of allowed schedules. In this section, we define the isolation levels RC, SI, and SSI.

Let $s$ be a schedule for a set $\mathcal{T}$ of transactions. Two transactions $T_i, T_j \in \mathcal{T}$ are said to be concurrent in $s$ when their execution overlaps. That is, if $first(T_i) <_s C_j$ and $first(T_j) <_s C_i$. We say that a write operation $W_j[t]$ in a transaction $T_j \in \mathcal{T}$ respects the commit order of $s$ if the version of $t$ written by $T_j$ is installed after all versions of $t$ installed by transactions committing before $T_j$ commits, but before all versions of $t$ installed by transactions committing after $T_j$ commits. More formally, if for every write operation $W_j[t]$ in a transaction $T_i \in \mathcal{T}$ different from $T_j$ we have $W_j[t] <_s W_i[t]$ iff $C_j <_s C_i$. For examples, consider schedules $s_1$ and $s_2$ from Example 2.1. In $s_1$ all transactions respect the commit order, while in schedule $s_2$ we have $W_3[q] <_s W_2[q]$ and $C_2 <_s C_3$.

We next define when a read operation $a \in T$ reads the last committed version relative to a specific operation. For RC this operation is $a$ itself while for SI this operation is $first(T)$. A read operation $R_j[t]$ in a transaction $T_j \in \mathcal{T}$ is read-last-committed in $s$ relative to an operation $a_j \in T_j$ (not necessarily different from $R_j[t]$) if the following holds:

- $v_s[R_j[t]] = op_0$ or $C_i <_s a_j$ with $v_s[R_j[t]] \in T_i$; and
- there is no write operation $W_k[t] \in T_k$ with $C_k <_s a_j$ and $v_s[R_j[t]] \ll_s W_k[t]$.

So, $R_j[t]$ observes the most recently installed version of $t$ (according to $\ll_s$) that is committed before $a_j$ in $s$. The latter can be observed in schedule $s_2$ (w.r.t both the read itself as well as the start of the transaction), while in schedule $s_1$ there is a read $R_1[t]$ with $v_s[R_1[t]] = W_2[t]$ and $R_1[t] <_s C_2$.

A transaction $T_j \in \mathcal{T}$ exhibits a concurrent write in $s$ if there are two write operations $b_i$ and $a_j$ in $s$ on the same object with $b_i \in T_i$, $a_j \in T_j$ and $T_i \neq T_j$ such that $b_i <_s a_j$ and $first(T_j) <_s C_i$. That is, transaction $T_j$ writes to an object that has been modified earlier by a concurrent transaction $T_i$.

A transaction $T_j \in \mathcal{T}$ exhibits a dirty write in $s$ if there are two write operations $b_i$ and $a_j$ in $s$ with $b_i \in T_i$, $a_j \in T_j$ and $T_i \neq T_j$ such that $b_i <_s a_j <_s C_i$. That is, transaction $T_j$ writes to an object that has been modified earlier by $T_i$, but $T_j$ has not yet issued a commit. Notice that by definition a transaction exhibiting a dirty write always exhibits a concurrent write. In schedule $s_1$ (and $s_2$) the transaction $T_3$...
witnesses a concurrent write since $w_2[q] \leq_{s_1} w_3[q]$ and $\text{first}(T_3) <_{s_1} C_2$. But $T_3$ does not exhibit a dirty write since $C_2 <_{s_1} w_3[q]$.

**Definition 4.1.** Let $s$ be a schedule over a set of transactions $\mathcal{T}$. A transaction $T_i \in \mathcal{T}$ is allowed under isolation level read committed (RC) in $s$ if:

- each write operation in $T_i$ respects the commit order of $s$;
- each read operation $b_i \in T_i$ is read-last-committed in $s$ relative to $b_i$; and
- $T_i$ does not exhibit a dirty write in $s$.

A transaction $T_i \in \mathcal{T}$ is allowed under snapshot isolation (SI) in $s$ if:

- each write operation in $T_i$ respects the commit order of $s$;
- each read operation in $T_i$ is read-last-committed in $s$ relative to $\text{first}(T_i)$; and
- $T_i$ does not exhibit concurrent writes in $s$.

**Definition 4.2.** We then say that the schedule $s$ is allowed under RC (respectively, SI) if every transaction is allowed under RC (respectively, SI) in $s$.

The latter definitions correspond to the ones in the literature (see, e.g., [12, 18]).

While RC and SI are defined on the granularity of a single transaction, SSI enforces a global condition on the schedule as a whole. For this, recall the concept of dangerous structures from [7]: three transactions $T_1, T_2, T_3 \in \mathcal{T}$ (where $T_1$ and $T_3$ are not necessarily different) form a dangerous structure $T_1 \rightarrow T_2 \rightarrow T_3$ in $s$ if:

- there is a rw-antidependency from $T_1$ to $T_2$ and from $T_2$ to $T_3$ in $s$;
- $T_1$ and $T_2$ are concurrent in $s$;
- $T_2$ and $T_3$ are concurrent in $s$; and,
- $C_3 <_{s} C_1$ and $C_3 <_{s} C_2$.

**Definition 4.3.** We say that the schedule $s$ is allowed under SSI if every transaction is allowed under SI in $s$, and there is no dangerous structure in $s$.

The latter definitions correspond to the ones in the literature (see, e.g., [12, 18]).

**Example 4.1.** Consider the set of transactions $\mathcal{T} = \{T_1, T_2, T_3\}$ from Example 2.1. For a schedule in which the transactions of $\mathcal{T}$ are allowed under SI, consider $s_6$ over $\mathcal{T}$.

It can be verified that all three transactions of $\mathcal{T}$ are indeed allowed under SI in $s_6$, but not under SSI, since $T_2 \rightarrow T_1 \rightarrow T_3$ is a dangerous structure.

We remark that the transactions in $s_6$ are also allowed under RC. For a schedule over $\mathcal{T}$ in which all transactions are allowed under RC but not under SI, consider consider schedule $s_7$.

It can be verified that all transactions of $\mathcal{T}$ are allowed under RC in $s_7$ but not under SI, because of the concurrent writes $w_2[q]$ and $w_3[q]$. ""
Definition 5.2 (Mv split schedule). Let $T$ be a set of transactions and $C = (T_1, b_1, a_2, T_2), (T_2, b_2, a_3, T_3), \ldots, (T_m, b_m, a_1, T_1)$ a sequence of conflicting quadruples for $T$ such that each transaction in $T$ occurs in at most two different quadruples. A multiversion split schedule for $T$ based on $C$ is a multiversion schedule that has the following form:

$$\text{prefix}_{b_1}(T_1) \cdot T_2 \cdot \ldots \cdot T_m \cdot \text{postfix}_{b_1}(T_1) \cdot T_{m+1} \cdot \ldots \cdot T_n,$$

where

1. there is no operation in $T_1$ conflicting with an operation in any of the transactions $T_3, \ldots, T_{m-1}$;
2. there is no write operation in $\text{prefix}_{b_1}(T_1)$ ww-conflicting with a write operation in $T_2$ or $T_m$;
3. $b_1$ is rw-conflicting with $a_2$.

Furthermore, $T_{m+1}, \ldots, T_n$ are the remaining transactions in $T$ (those not mentioned in $C$) in an arbitrary order.

5.1 Snapshot Isolation

We say that a multiversion split schedule $s$ for some set $T$ of transactions satisfies the SI requirements if there is no write operation in $\text{postfix}_{b_1}(T_1)$ ww-conflicting with a write operation in $T_2$ or $T_m$; and $b_m$ is rw-conflicting with $a_1$.

Proposition 5.1. For a set of transactions $T$, the following are equivalent:

1. $T$ is not robust against SI;
2. there is a multiversion split schedule $s$ for $T$ based on some $C$ that satisfies the SI requirements.

Proof Sketch. ($2 \rightarrow 1$) This direction is straightforward, as it can be verified that such a schedule is allowed under SI and is not conflict-serializable.

($1 \rightarrow 2$) Since $T$ is not robust against SI, a schedule $s$ for $T$ exists that is allowed under SI but not conflict-serializable. Let $\Gamma = (T_1, b_1, a_2, T_2), (T_2, b_2, a_3, T_3), \ldots, (T_m, b_m, a_1, T_1)$ be a cycle in $\text{SeG}(s)$. W.l.o.g., we assume $\Gamma$ is minimal and $T_3$ is the first transaction (among those occurring in $\Gamma$) to commit in $s$. That is, $C_2 <_s C_1$ for every other transaction $T_i$ in $\Gamma$.

Next, let $C = (T_1, b_1, a_2, T_2), (T_2, b_2, a_3, T_3), \ldots, (T_m, b_m, a_1, T_1)$ be the sequence of conflicting quadruples derived from $\Gamma$. In the remainder of the proof, we argue that the multiversion split schedule $s'$ for $T$ based on $C$ is indeed valid and satisfies the SI requirements. Condition 1 of Definition 5.2 is immediate by our assumption that $\Gamma$ is a minimal cycle in $\text{SeG}(s)$. Since $C_2 <_s C_1$, the edge $(T_1, b_1, a_2, T_2)$ in $\Gamma$ must be based on a rw-antidependency in $s$, thereby proving Condition 3 of Definition 5.2. Indeed, by definition of SI, if $b_1 \rightarrow a_2$ would be a ww-dependency or a wr-dependency, then $C_1 <_s \text{first}(T_2)$. This rw-antidependency $b_1 \rightarrow a_2$ furthermore implies that $T_1$ and $T_2$ are concurrent in $s$, as otherwise these two operations would imply a wr-dependency in the opposite direction instead.

We next argue that there is no write operation in $T_1$ ww-conflicting with a write operation in $T_2$ or $T_m$. Since $T_1$ and $T_2$ are concurrent, and since SI does not allow concurrent writes, the result is immediate for $T_2$. If $T_2 = T_m$, the result is immediate for $T_m$ as well. Otherwise, such a pair of conflicting write operations between $T_1$ and $T_m$ would imply a ww-dependency from $T_m$ to $T_1$ (as the opposite direction would contradict our assumption that $\Gamma$ is minimal). But then the definition of SI implies that $C_m <_s \text{first}(T_1) <_s b_m$, thereby contradicting our assumption that $T_2$ commits first.

To conclude the proof, we argue that $b_m$ is rw-conflicting with $a_1$. Since $b_m \rightarrow a_1$ is a dependency in $s$, $b_m$ wr- or ww-conflicting with $a_1$ would imply by definition of SI that $C_m <_s \text{first}(T_1) <_s b_2$, again leading to the desired contradiction. $\square$

Algorithm 1 provides a direct decision procedure for robustness against SI based on the previous characterization. There, for a transaction $T_i$ and a set of transactions $T$, we refer by $\text{si-graph}(T_i, T)$ to the graph containing as nodes all transactions in $T$ that do not have an operation conflicting with an operation in $T_i$, and with an edge between transactions $T_i$ and $T_j$ if $T_i$ has an operation conflicting with an operation in $T_j$.

The following theorem then readily follows:

Theorem 5.1. [12] Deciding whether a set of transactions is robust against SI is in PTIME.

An immediate corollary of the main result by Fekete [12] is that for a set of transactions $T$ robustness against SI can be characterized by the absence of a specific structure, called pivots, in the interference graph $\text{IG}(T)$. In this graph, each transaction in $T$ is represented by a node and edges summarize the possible dependencies between transactions. That is, if there exists a schedule $s$ with a dependency between two transactions $T_i$ and $T_j$, then $T_i \rightarrow T_j$ is an edge in $\text{IG}(T)$. An edge is furthermore referred to as an exposed edge if the dependency is a rw-antidependency and the two transactions are concurrent in $s$. A pivot is a transaction $T_i$ part of a chord-free cycle in $\text{IG}(T)$ with two adjacent
Algorithm 1: Deciding robustness against SI.

Input: Set of transactions $\mathcal{T}$
Output: True iff $\mathcal{T}$ is robust against SI

def reachable($T_2, T_m, T_1$):
    if $T_2 = T_m$ then
        return True;
    for $b_2 \in T_2, a_m \in T_m$ do
        if $b_2$ conflicts with $a_m$ then
            return True;
    $G := \text{si-graph}(T_1, \mathcal{T} \setminus \{T_1, T_2, T_m\})$;
    $TC := \text{reflexive-transitive-closure of } G$;
    for $(T_3, T_{m-1}) \in TC$ do
        for $b_2 \in T_2, a_3 \in T_3, b_{m-1} \in T_{m-1}, a_m \in T_m$ do
            if $(b_2$ conflicts with $a_3$ and $b_{m-1}$ conflicts with $a_m)$ then
                return False;
    return True;

for $T_1 \in \mathcal{T}, T_2 \in \mathcal{T} \setminus \{T_1\}, T_m \in \mathcal{T} \setminus \{T_1\}$ do
    if reachable($T_2, T_m, T_1$) then
        for $b_1 \in T_1$ do
            if $T_1, T_2,$ and $T_m$ have no
                ww-conflicting operations then
                for $a_1 \in T_1, a_2 \in T_2, b_m \in T_m$ do
                    if $b_m$ conflicts with $a_1$ and
                        $b_1$ is rw-conflicting with $a_2$
                        and $b_m$ is rw-conflicting
                            with $a_1$ then
                        return False;
        return True

Proof Sketch. (2 $\rightarrow$ 1) This direction is straightforward, as it can be verified that such a schedule is allowed under RC and is not conflict-serializable.

(1 $\rightarrow$ 2) The proof strategy is analogous to the proof of Proposition 5.1. In particular, let $\Gamma$ and $C$ be as in the proof of Proposition 5.1. We now argue that the multiversion split schedule based on $C$ is valid and satisfies the RC requirements. Condition 1 of Definition 5.2 is again immediate by our assumption that $\Gamma$ is a minimal cycle in $SeG(s)$. Towards Condition 3 of Definition 5.2, note that if $b_1 \rightarrow_s a_2$ is a ww-dependency or a wr-dependency, then by definition of RC, we have $c_1 <_s a_2$, thereby contradicting our assumption that $T_2$ commits first. Furthermore, this rw-antidependency $b_1 \rightarrow_s a_2$ implies that $b_1 <_s C_2$, as otherwise these operations would imply a wr-dependency in the opposite direction instead. Because of this, there can not be a write operation in $\text{pref}_{b_2}(T_1)$ conflicting with a write operation in $T_2$, as this would create a dirty write in $s$. If $T_m = T_2$, the result is immediate for $T_m$ as well. Otherwise, if $T_m \neq T_2$, such a pair of ww-conflicting operations in $T_1$ and $T_m$ would imply a ww-dependency from $T_m$ to $T_1$ (as the opposite direction contradicts our assumption that $\Gamma$ is a minimal cycle), and hence $c_m <_s b_1 <_s C_2$, again leading to the desired contradiction and thereby proving Condition 2 of Definition 5.2.

It remains to argue that the multiversion split schedule satisfies the RC requirements. Towards a contradiction, assume $b_m$ is wr- or ww-conflicting with $a_1$ and $a_1 \leq_{T_1} b_1$. Then, by the definition of RC, we have $c_m <_s b_1 <_s C_2$, which contradicts our assumption that $T_2$ commits first. \qed

Algorithm 1 can easily be adapted for RC, leading to the following result:

Theorem 5.2. [18] Deciding whether a set of transactions is robust against RC is in PTIME.

Ketsman et al. [14] provide full characterizations for robustness against read committed and read uncommitted under lock-based semantics as opposed to the multiversion semantics that is used here. In addition, it is shown that the corresponding decision problems are complete for coNP and LOGSPACE, respectively. The coNP-hardness stems from the fact that counterexample schedules no longer take the simple form of a split schedule.

5.3 Robust allocations

In practice, an isolation level is not set uniformly on the level of the database or even on the level of exposed edges $T_{i-1} \rightarrow T_i$ and $T_i \rightarrow T_{i+1}$. It can be shown that every pivot implies a multiversion split schedule satisfying the SI requirements and vice versa. Intuitively, the rw-antidependencies $b_m \rightarrow_s a_1$ and $b_1 \rightarrow_s a_2$ in such a multiversion split schedule $s$ correspond to exposed edges $T_m \rightarrow T_1$ and $T_1 \rightarrow T_2$ in $IG(T)$, thereby witnessing that $T_1$ is a pivot.

5.2 Read Committed

We say that a multiversion split schedule $s$ for some set $\mathcal{T}$ of transactions satisfies the RC requirements if either $b_m$ is rw-conflicting with $a_1$ or $b_1 <_{T_1} a_1$.

Proposition 5.2. [18] For a set of transactions $\mathcal{T}$, the following are equivalent:

1. $\mathcal{T}$ is not robust against RC;
2. there is a multiversion split schedule $s$ for $\mathcal{T}$ based on some $C$ that satisfies the RC requirements.
the application but can be specified on the level of an individual transaction. Let $I \subseteq \{ \text{RC, SI, SSI} \}$. An $I$-allocation $A$ for a set of transactions $T$ is a function mapping each transaction $T \in T$ onto an isolation level $A(T) \in I$.

A schedule $s$ over a set of transactions $T$ is allowed under an $I$-allocation $A$ over $T$ if:

- for every transaction $T_i \in T$ with $A(T_i) = \text{RC}$, $T_i$ is allowed under RC;
- for every transaction $T_i \in T$ with $A(T_i) \in \{ \text{SI, SSI} \}$, $T_i$ is allowed under SI; and
- there is no dangerous structure $T_i \rightarrow T_j \rightarrow T_k$ in $s$ formed by three (not necessarily different) transactions $T_i, T_j, T_k \in \{ T \in T \mid A(T) = \text{SSI} \}$.

We say that a set of transactions $T$ is robust against an allocation $A$ when every schedule that is allowed under $A$ is conflict-serializable. The allocation problem then consists of deciding whether a robust allocation exists and if so to find an optimal one.

In [21] it is shown that it can be decided in polynomial time whether a set of transactions is robust against a given $\{ \text{RC, SI, SSI} \}$-allocation. That result is based on a notion of split schedules for allocations. Furthermore, it is shown that a unique optimal\(^3\) allocation always exists and can be found in polynomial time as well. Fekete [12] provided a characterization for robust allocations when every transaction runs under either snapshot isolation or strict two-phase locking (S2PL). He also obtained a polynomial time algorithm to compute the optimal allocation.

6 Transaction programs and templates

Transaction programs Previous work on static robustness testing [13, 2] for transaction programs is based on the following key insight: when a schedule is not serializable, then the dependency graph constructed from that schedule contains a cycle satisfying a condition specific to the isolation level at hand (dangerous structure for snapshot isolation and the presence of a counterflow edge for RC). That insight is extended to a workload of transaction programs through the construction of a so-called static dependency graph where each program is represented by a node, and there is a conflict edge from one program to another if there can be a schedule that gives rise to that conflict. The absence of a cycle satisfying the condition specific to that isolation level then guarantees robustness while the presence of a cycle does not necessarily imply non-robustness.

Other work studies robustness within a framework for uniformly specifying different isolation levels in a declarative way [8, 4, 9]. A key assumption here is atomic visibility requiring that either all or none of the updates of each transaction are visible to other transactions. These approaches aim at higher isolation levels and cannot be used for RC, as RC does not admit atomic visibility.

Transaction Templates The static robustness approach based on transaction templates [18] differs in two ways. First, it makes more underlying assumptions explicit within the formalism of transaction templates (whereas previous work departs from the static dependency graph that should be constructed in some way by the dba). Second, it allows for a decision procedure that is sound and complete for robustness testing against RC, allowing to detect larger subsets of transactions to be robust [18].

Example 6.1. Figure 4 shows the transaction template for DepositChecking, which is a part of the SmallBank benchmark. The template consists of two operations. The first operation is a read operation over variable X of type Account. In particular, the attributes Name (N) and CustomerID (C) are read. The second operation is an update operation over variable Z of type Checking. Such an update operation should be interpreted as a read immediately followed by a write that cannot be interleaved with other operations. In particular, the attributes CustomerID (C) and Balance (B) are read, immediately followed by a write to attribute Balance. We can now instantiate transactions from these templates by assigning tuples of the corresponding type to variables. For example, transaction $R[t]U[v]C$ is a valid instantiation, but $R[t]U[t]C$ is not, since object $t$ cannot be of type Account and Checking at the same time.

The formalization of transactions and conflict serializability in [18] and this paper is based on [12], generalized to operations over attributes of tuples and extended with U-operations that combine R- and U-operations into one atomic operation. These definitions are closely related to the formalization presented by Adya et al. [1], but we assume a total

DepositChecking:

$R[X : \text{Account}\{N, C\}]$
$U[Z : \text{Checking}\{C, B\}\{B\}]$

Figure 4: Transaction template for DepositChecking.
rather than a partial order over the operations in a schedule. There are also a few restrictions to the model: there needs to be a fixed set of read-only attributes that cannot be updated and which are used to select tuples for update. The most typical example of this are primary key values passed to transaction templates as parameters. The inability to update primary keys is not an important restriction in many workloads, where keys, once assigned, never get changed, for regulatory or data integrity reasons.

In [18], a PTIME decision procedure is obtained for robustness against RC for templates without functional constraints and [20] improves that result to NLOGSPACE. In addition, an experimental study was performed showing how an approach based on robustness and making transactions robust through promotion can improve transaction throughput. In particular, we show on the SmallBank and TPC-C-Kv (based on TPC-C) benchmarks that in case of increasing contention our approach leads to practical performance improvements compared to when executed under SI or SSI. It should be noted that both benchmarks in their original form are not identified as robust. By promoting a small number of read operations such that they write the observed value back to the database, robustness is obtained without altering the semantics of these benchmark programs. By more accurately modeling transaction programs, it becomes possible to recognize larger sets of workloads as robust.

7 Conclusion

Despite its practical relevance and challenging problems, concurrency control has only attracted limited attention from the database theory community. We hope that the present paper eases the barrier of entrance to this exciting topic. To spark further interest, we mention some open problems that we consider interesting.

Research on robustness for example has mostly focused on guaranteeing conflict-serializability. But as we explained in Section 3, there exist alternative definitions of serializability that can be used for a robustness analysis, like view-serializability and final-state serializability. It would be interesting to see if the robustness problem cast with one of the alternative definitions has similar characterizations as the ones obtained for conflict-serializability. Furthermore, while in theory one could expect that a more liberal definition of serializability leads to larger classes of transactions and templates that are robust, it is not clear if such a difference can be observed in practice.

Another direction for further research lies in the consideration of systems with a higher-degree of parallelism. The considered isolation levels, RC, SI, are mostly designed for conventional database systems utilizing a limited degree of parallelization. High isolation levels in many-core systems are known to be particularly challenging and therefore robustness analysis against lower-isolation levels that are meaningful in a highly parallel context would be particularly relevant.

Acknowledgments

This work is partly funded by FWO-grant G019921N.

8 References


Management of Machine Learning Lifecycle Artifacts: A Survey

Marius Schlegel
TU Ilmenau, Germany
marius.schlegel@tu-ilmenau.de

Kai-Uwe Sattler
TU Ilmenau, Germany
kus@tu-ilmenau.de

ABSTRACT
The explorative and iterative nature of developing and operating ML applications leads to a variety of artifacts, such as datasets, features, models, hyperparameters, metrics, software, configurations, and logs. In order to enable comparability, reproducibility, and traceability of these artifacts across the ML lifecycle steps and iterations, systems and tools have been developed to support their collection, storage, and management. It is often not obvious what precise functional scope such systems offer so that the comparison and the estimation of synergy effects between candidates are quite challenging. In this paper, we aim to give an overview of systems and platforms which support the management of ML lifecycle artifacts. Based on a systematic literature review, we derive assessment criteria and apply them to a representative selection of more than 60 systems and platforms.

1. INTRODUCTION
Machine learning (ML) approaches are well established in a wide range of application domains. In contrast to engineering traditional software, the development of ML systems is different: data and feature preparation, model development, and model operation tasks are integrated into a unified lifecycle which is often iterated several times. Although there are systems and tools that provide support for a broad range of tasks within the ML lifecycle, such as data cleaning and labeling, feature engineering, model design and training, experiment management, hyperparameter optimization, and orchestration, achieving comparability, traceability, and reproducibility of model and data artifacts across all lifecycle steps and iterations is still challenging.

To meet these requirements, it is necessary to capture the input and output artifacts of each lifecycle step and iteration. That includes model artifacts and data-related artifacts, such as datasets, labels, and features. Reproducibility also requires capturing software-related artifacts, such as code, configurations, and environmental dependencies. By additionally considering metadata, such as model parameters, hyperparameters, quality metrics, and execution statistics, comparability of artifacts is enabled.

Since the manual management of ML artifacts is simply not efficient, systems and platforms provide support for the systematic collection, storage, and management of ML lifecycle artifacts, which we collectively referred to as ML artifact management systems (ML AMSs). Since ML AMSs are often integrated into general ML development platforms or frameworks for a subset of the ML lifecycle tasks, it is typically not obvious what the precise functional and non-functional scope of an AMS is, how an AMS compares to others, and to what extent possible synergy effects can be exploited through tool-chaining.

The objective of this paper is to provide a comprehensive overview of AMSs from academia and industry. We address the following research questions: (RQ1) What are criteria to describe, assess, and compare AMSs? (RQ2) Which AMSs exist in academia and industry, and what are their functional and non-functional properties according to the assessment criteria? To answer these questions, we conducted a systematic literature review.

The paper is organized as follows: §2 gives an overview of related work. §3 describes the ML lifecycle and concretizes the tasks of ML lifecycle management. Based on the conducted systematic literature review, §4 discusses criteria for assessing AMSs w.r.t. their functional and non-functional scope of features. §5 applies the criteria to the 64 identified AMSs and discusses the results.

2. RELATED WORK
In recent years, both academia and industry have produced a variety of systems that provide artifact collection and management support for individual steps of ML lifecycles. Authors often compare with related works in the scope of the particular system, which, however,

1Whenever we use just “AMS”, we refer to an ML AMS.
does not enable the comparability with a broad range of systems and criteria.

This problem has been addressed by a few surveys [69, 168, 67]. In the context of reproducibility of empirical results, Isdael et al. [69] have investigated what support is provided by existing experiment management systems. However, these systems cover only a subset of the ML lifecycle. Weißgerber et al. [168] develop an open-science-centered process model for ML research as a common ground and investigate 40 ML platforms and tools. However, the authors analyze only 11 platforms w.r.t. ML workflow support capabilities and their properties.

In contrast to the aforementioned studies and surveys, Idowu et al. [67] adopt a more fine-grained understanding of artifacts and system capabilities which is most closely to our work. Based on the selection of 17 experiment management systems and tools, the authors develop a feature model for assessing their capabilities. Although this survey shows parallels to our work, the authors only consider a limited selection of systems which is, again, only focused on the area of experiment tracking and management.

3. MACHINE LEARNING LIFECYCLE ARTIFACT MANAGEMENT

In this section, we discuss the steps of ML lifecycles based on typical ML workflows (§3.1), derive the tasks of ML artifact management, and outline the support ML AMSs should provide (§3.2).

3.1 ML Lifecycle

In contrast to traditional software engineering, the development of ML-powered applications is more iterative and explorative. Thus, developers have adapted their processes and practices for ML: Following methodologies in the context of data science, data analytics and data mining, such as TDS/ [102], KDD [45], CRISP-DM [137, 169], or ASUM-DM [64], workflows specialized for ML have been established [26, 9, 53, 128]. Despite minor differences, ML workflows contain both data-centric and model-centric steps and often multiple feedback loops among the different stages, which leads to a lifecycle. Fig. 1 depicts a common view on the ML lifecycle.

The ML lifecycle consists of four stages: Requirements Stage, Data-oriented Stage, Model-oriented Stage, and Operations Stage. Starting with the Requirements Stage, the requirements for the model to be developed are derived based on the application requirements [163]. This stage is dedicated to three major decisions: (1) to decide which functionality and interfaces to realize, (2) to decide which types of models are best suited for the given problem, and (3) to decide which types of data to work on.

The Data-oriented Stage starts with the Data Collection and Selection step. Datasets, either internal or publicly available, are searched, or individual ones are collected and the data most suitable for the subsequent steps is selected (e.g. dependent on data quality, bias, etc.). By using available generic datasets, models may be (pre-)trained (e.g. ImageNet for object recognition), and later, by using transfer learning [24, 111] along with more specific data, trained to a more specific model. Then, in the Data Cleaning step, datasets are prepared, removing inaccurate or noisy records. As required by most of the supervised learning techniques to be able to induce a model, data labeling is used to assign a ground truth label to each dataset record. Subsequently, feature engineering and selection is performed to extract and select features for ML models. For some models, such as convolutional neural networks, this step is directly intertwined with model training.

The Model-oriented Stage starts with the Model Design step. Often, existing model designs and neural network architectures are used and tailored towards specific requirements. During model training, the selected models are trained on the collected and preprocessed datasets using the selected features and their respective labels. Subsequently, in the Model Evaluation step, developers evaluate a trained model on test datasets using predefined metrics, such as ac-
accuracy or F1-score. In critical application domains, this step also includes extensive human evaluation. The subsequent Model Optimization step is used to fine-tune the model, especially its hyperparameters. In the context of the model development steps, we refer to an experiment as a sequence of model development activities that result in a trained model but do not include cycles to previous steps.

Finally, in the Operations Stage, the model is distributed to the target systems and devices, either as an on-demand (online) service or in batch (offline) mode (Model Deployment), as well as continuously monitored for metrics and errors during execution and use (Model Monitoring). In particular, CI/CD practices from software engineering are adapted.

As illustrated by Fig. 1, multiple feedback loops from steps of the Model-oriented Stage or the Operations Stage to any step before may be triggered by insufficient accuracy or new data. Moreover, Sculley et al. [131] point out, that the model development often takes only a fraction of the time required to complete ML projects. Usually, a large amount of tooling and infrastructure is required to support data extract, transform, and load (ETL) pipelines, efficient training and inference, reproducible experiments, versioning of datasets and models, model analysis, and model monitoring at scale. The creation and management of services and tools can ultimately account for a large portion of the workload of ML engineers, researchers, and data scientists.

3.2 Management of ML Lifecycle Artifacts

Within the steps of the ML lifecycle, a variety of artifacts is created, used, and modified: Datasets, labels and annotations, and feature sets are inputs and outputs of steps in the Data-oriented Stage. Moreover, data processing source code, logs, and environmental dependencies are created and/or used. In the Model-oriented Stage, results from the Data-oriented Stage are used to develop and train models. In addition, metadata such as parameters, hyperparameters, and captured metrics as well as model processing source code, logs, and environment dependencies are artifacts that are created and/or used in this stage. The Operations Stage requires trained models and corresponding dependencies such as libraries and runtimes (e.g. via Docker container), uses model deployment and monitoring source code which is typically wrapped into a web service with a REST API for on-demand (online) service or scheduled for batch (offline) execution, and captures execution logs and statistics. To achieve comparability, traceability, and reproducibility of produced data and model artifacts across multiple lifecycle iterations, it is essential to also capture metadata artifacts that can be easily inspected afterwards (e.g. model parameters, hyperparameters, lineage traces, performance metrics) as well as software artifacts.

Manual management of artifacts is simply not efficient due to the complexity and the required time. To meet the above requirements, it is necessary to systematically capture any input and output artifacts and to provide them via appropriate interfaces. **ML artifact management** includes any methods and tools for managing ML artifacts that are created and used in the development, deployment, and operation of ML-based systems. Systems supporting ML artifact management, collectively referred to as **ML artifact management systems (ML AMSs)**, provide the functionality and interfaces to adequately record, store, and manage ML lifecycle artifacts.

4. ASSESSMENT CRITERIA

The goal of this section is to define criteria for the description and assessment of AMSs. Based on a priori assumptions, we first list functional and non-functional requirements. We then conduct a systematic literature review according to Kitchenham et al. [81]: Using well-defined keywords, we search ACM DL, DBLP, IEEE Xplore, and SpringerLink for academic publications as well as Google and Google Scholar for web pages, articles, white papers, technical reports, reference lists, source code repositories, and documentations. Next, we perform the publication selection based on the relevance for answering our research questions. To avoid overlooking relevant literature, we perform one iteration of backward snowballing [170]. Finally, we iteratively extract assessment criteria and subcriteria, criteria categories, as well as the functional and non-functional properties of concrete systems and platforms based on concept matrices. The results are shown in Tab. 1, which outlines categories, criteria (italicized), subcriteria (in square brackets).

**Lifecycle Integration.**

This category describes for which parts of the ML lifecycle a system provides artifact collection and management capabilities. The four stages form the criteria, with the steps assigned to each stage forming the subcriteria (cf. § 3.1).

**Artifact Support.**

Orthogonal to the previous category, this category indicates which types of artifacts are supported and managed by an AMS. Based on the discussion in § 3.2, we distinguish between the criteria **Data-related, Model, Metadata, and Software Artifacts.**

The criteria **Data-related Artifacts and Model Ar-**
<table>
<thead>
<tr>
<th>Category</th>
<th>Criteria and Subcriteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lifecycle</td>
<td><strong>Requirements Stage</strong> [Model Requirements Analysis]</td>
</tr>
<tr>
<td></td>
<td><strong>Data-oriented Stage</strong> [Data Collection &amp; Selection, Data Preparation &amp; Cleaning, Data Labeling, Feature Engineering &amp; Selection]</td>
</tr>
<tr>
<td></td>
<td><strong>Model-oriented Stage</strong> [Model Design, Model Training, Model Evaluation, Model Optimization]</td>
</tr>
<tr>
<td></td>
<td><strong>Operations Stage</strong> [Model Deployment, Model Monitoring]</td>
</tr>
<tr>
<td>Integration</td>
<td><strong>Artifact Support</strong> [Data-related Artifacts [Dataset, Annotations &amp; Labels, Features]]</td>
</tr>
<tr>
<td></td>
<td>Model Artifacts [Model]</td>
</tr>
<tr>
<td></td>
<td>Metadata Artifacts [Identification, Model Parameters, Model Hyperparameters, Model Metrics, Experiments &amp; Projects, Pipelines, Execution Logs &amp; Statistics]</td>
</tr>
<tr>
<td></td>
<td>Software Artifacts [Source Code, Notebooks, Configurations, Environment]</td>
</tr>
<tr>
<td>Operations</td>
<td><strong>Logging &amp; Versioning</strong> [Log/Capture, Commit, Revert/Rollback]</td>
</tr>
<tr>
<td></td>
<td>Exploration [Query, Compare, Lineage, Provenance, Visualize]</td>
</tr>
<tr>
<td></td>
<td>Management [Modify, Delete, Execute &amp; Run, Deploy]</td>
</tr>
<tr>
<td></td>
<td>Collaboration [Export &amp; Import, Share]</td>
</tr>
<tr>
<td>Collection &amp; Storage</td>
<td><strong>Collection Automation</strong> [Intrusive, Non-intrusive]</td>
</tr>
<tr>
<td></td>
<td>Storage Type [Filesystem, Database, Object/BLOB Storage, Repository]</td>
</tr>
<tr>
<td></td>
<td>Versioning [Repository, Snapshot]</td>
</tr>
<tr>
<td>Interfaces &amp; Integration</td>
<td><strong>Interface</strong> [API/SDK, CLI, Web UI]</td>
</tr>
<tr>
<td></td>
<td>Language Support &amp; Integration [Languages, Frameworks, Notebook]</td>
</tr>
<tr>
<td>Operation &amp; Licensing</td>
<td><strong>Operation</strong> [Local, On-premise, Cloud]</td>
</tr>
<tr>
<td></td>
<td>License [Free, Non-free]</td>
</tr>
</tbody>
</table>

Table 1: Assessment categories, criteria, and subcriteria.

tifacts represent core resources that are either input, output, or both for a lifecycle step. Data-related Artifacts are datasets (used for training, validation, and testing), annotations and labels, and features (cf. corresponding subcriteria). Model Artifacts are represented by trained models (subcriterion Model).

The criteria Metadata Artifacts and Software Artifacts represent the corresponding artifact types, that enable the reproducibility and traceability of individual ML lifecycle steps and their results. The criterion Metadata Artifacts covers different types of metadata: (i) identification metadata (e.g. identifier, name, type of dataset or model, association with groups, experiments, pipelines, etc.); (ii) data-related metadata; (iii) model-related metadata, such as inspectable model parameters (e.g. weights and biases), model hyperparameters (e.g. number of hidden layers, learning rate, batch size, or dropout), and model quality & performance metrics (e.g. accuracy, F1-score, or AUC score); (iv) experiments and projects, which are abstractions to capture data processing or model training runs and to group related artifacts in a reproducible and comparable way; (v) pipelines, which are abstractions to execute entire ML workflows in an automated fashion and relates the input and output artifacts required per step as well as the glue code required for processing; (vi) execution-related logs & statistics.

The criterion Software Artifacts comprises source code and notebooks, e.g. for data processing, experimentation and model training, and serving, as well as configurations and execution-related environment dependencies and containers, e.g. Conda environments, Docker containers, or virtual machines.

**Operations.**

This category indicates the operations provided by an AMS for handling and managing ML artifacts. It comprises the criteria Logging & Versioning, Exploration, Management, and Collaboration.

The criterion Logging & Versioning represents any operations that enable logging or capturing single artifacts (subcriterion Log/Capture), creating checkpoints of a project or an experiment comprising several artifacts (subcriterion Commit), and reverting or rolling back to an earlier committed or snapshot version (subcriterion Revert/Rollback).

The criterion Exploration includes any operations that help to gain concrete insights into the results of data processing pipelines, experiments, model training results, or monitoring statistics. These operations are differentiated by the subcriteria Query, Compare, Lineage, Provenance, and Visualize. Query operations may be represented by simple searching and listing functionality, more advanced filtering functionality (e.g. based on model performance metrics), or a comprehensive query language. Compare indicates the presence of operations for the comparison between two or more versions of artifacts. In terms of model artifacts, this operation may be used to select the most promising model from a set of can-
didates (model selection), either in model training and development [122] or in model serving (e.g. best performing predictor) [33]. Lineage represents any operations for tracing the lineage of artifacts, i.e. which input artifacts led to which output artifacts, and thus provide information about the history of a model, dataset, or project. Provenance represents any operations, which in addition provide information about which concrete transformations and processes converted inputs into an output. Visualize indicates the presence of functionality for graphical representation of model architectures, pipelines, model metrics, or experimentation results.

The criterion Management characterizes operations for handling and using stored artifacts. The subcriteria Modify and Delete indicate operations for modifying or deleting logged and already stored artifacts. Execute & Run comprises operations that are interfaces for the execution and orchestration of data processing or model training experiments and pipelines. The subcriterion Deploy refers to deployment operations for offline (batch) and online (on-demand) model serving.

The criterion Collaboration indicates the presence of operations for collaboration which enable simple export/import functionality (subcriterion Export & Import) as well as sharing of artifacts among internal company and team members or publishing of artifacts for external instances (subcriterion Share).

Collection & Storage.

This category describes the model for artifact collection and storage based on the criteria Collection Automation, Storage Type, and Versioning.

The criterion Collection Automation represents the degree of manual effort required to collect and capture ML artifacts. The collection of artifacts is either intrusive, which requires engineers to explicitly add instructions or API calls within the source code, non-intrusive, which means that no explicit manual actions are required and the collection is performed automatically, or both.

Storage Type describes the type of storage used and supported by an AMS. We identified the subcriteria Filesystem, Database, Object/BLOB Storage and Repository. An AMS can support multiple types of storage, and also hybrid variants are possible.

While local filesystems are often the means of choice to store artifacts for small and manageable experiments, such as smaller textual datasets (e.g. in .csv or .parquet files) or metadata (e.g. in .yaml files), distributed file systems (e.g. HDFS) are rather used for larger projects and permanently scalable solutions. These are suitable for both large numbers and large files, as is the case for image, video, and text datasets as well as trained models in serialized file formats (e.g. .pb, .onnx, .pkl, .pt, or .pmml).

Often, these are also stored on separate storage servers or clusters, and provided with an API to fulfill availability and replication requirements.

Databases are established for storing a wide range of different types of data. In the context of artifact management, large tabular, sequential, and text datasets may be stored in (object-)relational or time-series databases, metadata in relational databases, and logs in key-value databases. While modern and widely used DBMSs (e.g. PostgreSQL [120]) provide BLOB data types, these often have limitations regarding maximum file sizes, which is why also object/BLOB stores are often used.

Repositories are version-controlled and typically suited best for source code and text. It is also possible to add version control on top of other storage types, so that the storage of large files in a distributed file system is combined with version control. For example, Git LFS (Large File Storage) replaces large files such as image, audio, and video datasets, with text pointers inside Git, while storing the file contents on a remote file server. This preserves and accelerates typical workflows, while enabling versioning of large amounts of data or large files.

The criterion Versioning characterizes the way versioning of artifacts is done. Either versioning is delegated to a traditional version control system (e.g. Git or Mercurial) and managed by means of a repository (see also corresponding storage type). A repository contains several to all artifacts associated with a project. In contrast, versioning may be done by following a snapshot-based approach: Snapshots are created manually for each individual artifact and possibly independently of other artifacts.

Interfaces & Integration.

This category characterizes an AMS’s user interfaces and integration capabilities. The criterion Interface states the type of provided interfaces that may be based on an API (e.g. a REST interface) or a higher-level SDK, based on a command line interface (CLI), or based on a web application. Language Support & Integration distinguishes between the integration into programming languages (e.g. into Python via provided libraries), integration into well-known frameworks providing functional integration for the steps of the ML lifecycle (e.g. data processing with Apache Spark [13], model training with TensorFlow [56], or model orchestration with Metafflow [110]), and notebook support (e.g. Jupyter [121], Apache Zeppelin [12], or TensorBoard [55]).
**Operation & Licensing.**

The last category covers two non-functional, usage-related criteria: *Operation* and *License*. The criterion *Operation* defines whether the operation of a system or tool is local (e.g., the case for Python libraries, subcriterion *Local*), on-premise (e.g., the case for server-based systems, subcriterion *On-premise*) or by a dedicated cloud provider (e.g., for hosted services, subcriterion *Cloud*). The criterion *License* describes the type of license, which is either classified as free (e.g., public domain, permissive, or copyleft licenses) or non-free (e.g., non-commercial or proprietary licenses), and which may be further concretized by the concrete license.

5. **ASSESSMENT**

This section presents the assessment of concrete ML AMSs. As part of the systematic literature review (see §4), we identified a total of 64 systems and platforms from research and industry. We assessed these based on our criteria and subcriteria (cf. §4). An additional result of this assessment is the derivation of the classes which aim to group systems with high similarity regarding lifecycle integration, artifact support, and functionality. Fig. 2 visualizes the results at criteria level (depending on a criterion’s semantic, we consider either its fulfillment or presence) and the AMS classes. The following two subsections discuss the assessment results on these orthogonal dimensions: Based on the derived classes, we first provide an overview of the identified AMSs and their core characteristics (§5.1). Subsequently, we discuss to what extent the criteria and subcriteria are fulfilled by the systems within their classes (§5.2).

5.1 **Discussion Along Classes**

A general observation is that the focus of ML-supporting systems is often not obvious: The boundaries are blurred between systems that purely provide functionality for the development of ML-based systems to those that purely focus on the management, storage, and deployment of ML artifacts (and typically complement the former). Therefore, we classified the assessed systems based on the characteristics of the criteria within the categories Lifecycle Integration, Artifact Support, and Operations into five classes: Lifecycle Management, Pipeline Management, Experiment Management, Model Management, and Dataset & Feature Management.

**Lifecycle Management.**

This class includes systems and platforms that focus on the entire ML lifecycle and, typically beyond broad functional support for the different tasks within the ML lifecycle, provide artifact management capabilities for nearly all ML lifecycle steps. Microsoft Azure ML [103], Amazon SageMaker [130, 8], Google Vertex AI [57], IBM Watson Studio [63, 123, 65], Comet [31], DataRobot AI Cloud Platform [142, 36], and Cloudera Data Science Workbench [35] are “all-in-one” ML as a service (MLaaS) platforms which are comparable in objective and functional scope. Although the direct integration into a provider’s cloud infrastructure usually offers rich processing possibilities, such as the scaling of computing and memory resources, the usage is subject to the fees and pricing of the provider. Furthermore, the usage of MLaaS may not be possible in certain application scenarios due to data protection requirements and legal regulations.

In contrast, open-source AMSs, such as MLflow [174, 28, 83, 84], ClearML [7, 6], Polyaxon [118, 119], and Hopsworks [70, 113, 112, 88, 89], can be deployed both in the cloud and on-premise. MLflow focuses on capturing, storing, managing, and deploying ML artifacts: MLflow Tracking is an API for logging experiment runs, including code and data dependencies, via automatic or manual instrumenting application code. These runs can be viewed, compared, and searched through an API or the UI. MLflow Models are a convention for packaging models and their dependencies, that is compatible with diverse serving environments. MLflow Projects provides a standard format for packaging reusable and reproducible project code. MLflow Model Registry is a hub for storing models and managing their deployment lifecycles. ClearML and Polyaxon also aim at simplifying the entire ML lifecycle. Both provide a comparable range of functions, plus model monitoring and resource management. The Hopsworks platform also has a comparable range of functions but additionally includes big data and GPU support for highly scalable learning, HDFS extended by a distributed hierarchical metadata service using a NewSQL database (HopsFS), Github-like project management, and an integrated feature store.

Valohai [153] is a platform for managing and versioning ML pipelines from data extraction to model deployment. Its objective is comparable to the previous AMSs. The three layers of the platform, (Web) Application Layer, Computer Layer, and Data Layer, can be flexibly deployed in the cloud (e.g., Valohai or own AWS account) or on-premise.

In relation to the previously described AMSs, DVC [18, 74, 73] is a complementary ML artifact management tool for ML pipelines. DVC’s versioning is built upon Git and provides experiment branching semantics, push and pull processing of
Figure 2: The heatmap visualizes the assessment results. For each criterion, the number of subcriteria fulfilled/present was determined, related to the total number of subcriteria, and normalized to the value range [0,1]. The degree of fulfillment/presence of a criterion (y-axis) by the investigated systems (x-axis) is represented by the hue of a cell ranging from dark red (i.e. not fulfilled/present) to dark green (i.e. completely fulfilled/present). If the fulfillment/presence of all subcriteria of a criterion is unclear or not exactly known, the corresponding cell’s hue is white. The last criterion “License” is an exception due to its binary character: the hue is either dark red (“non-free”) or dark green (“free”).
bundles of models, data, and code, as well as automatic metric tracking. Recently, the company behind, Iterative, built a tool ecosystem around DVC to achieve ML lifecycle management [76]; a library for implementing CI/CD in ML projects (CML) [71, 72], a library for logging ML metrics and metadata (DVCLive) [75], a web application for seamless data and model management, experiment tracking, visualization, and collaboration (Studio) [78, 77], a model registry and deployment tool (MLEM) [79].

In contrast, Ease.ML is an ML lifecycle management system specifically targeting non-ML experts [4, 42, 125, 80]. Built on top of existing data ecosystems and techniques, Ease.ML guides users step-by-step: Starting by automatic data ingestion and augmentation, automatic feasibility studies, data noise debugging, data acquisition, scalable multi-tenant automatic training, continuous integration, and ending with continuous quality optimization.

Additionally, we identified proprietary and internally deployed ML lifecycle management platforms from the major tech companies Airbnb, Uber, and LinkedIn. Bighead is Airbnb’s framework-agnostic ML platform tailored to their use cases and environment [25]. It includes a feature management framework based on the lambda architecture principle [94, 95], a model development and execution management toolkit, a lifecycle management service, an offline training and inference engine, an online inference service with containerization and cloud native architecture, and a container management tool.

Michelangelo is Uber’s ML platform which enables internal teams to build, deploy, and operate uniform and reproducible ML pipelines for applications in a microservices-based production environment [86]. Michelangelo consists of a mix of open-source systems and components built in-house, such as a centralized feature store, a domain-specific language (DSL) for feature selection and transformation, the distributed deep learning framework Horovod [135] and the model management system Gallery [143].

LinkedIn’s Pro-ML system specifically aims to meet internal scalability requirements [87]. Pro-ML supports a key set of ML lifecycle steps: data exploration and model authoring with an own DSL for feature and model representations and a central feature marketplace, real time and batch model training, model deployment, and model monitoring.

Pipeline Management.
ML pipelines are abstractions to enable a holistic view on data processing, model development, and model deployment workflows. This class includes any AMMSs that support the management of ML pipelines and related artifacts. In contrast to the previous class, these systems typically do not comprise either support for the Operations Stage (in a few cases), the management of software artifacts, or both.

Although the pipeline management systems Velox [32, 10], Vamsa [108], and ArangoML Pipeline [129, 14, 15] differ to some extent in their goals, design principles, and intended uses, they overall provide a comparable basic range of artifact management capabilities, with a focus on models and model metadata. Velox provides management of training pipeline orchestration for a set of predeclared models, model performance evaluation, retraining as necessary, and low-latency model inference. Vamsa is a tool for automated provenance tracking of ML pipelines based on static analyses of Python programs. ArangoML Pipeline is an artifact and metadata storage layer for ML pipeline lineage tracking, auditing, reproducibility, and monitoring built around ArangoDB.

In comparison, Apache SystemDS [23, 147, 148] is a declarative ML pipeline system. It provides a DSL with declarative language abstractions for different ML lifecycle tasks. High-level scripts are compiled into hybrid execution plans of local, in-memory CPU and GPU operations, as well as distributed operations on Spark based on a tensor data model. Based on the LIMA framework, SystemDS provides fine-grained, multi-level lineage tracing as well as compiler-assisted, full and partial reuse during runtime removing redundancy at different levels of hierarchically composed ML pipelines [116].

Mittrace [136, 158] and ProvDB [97, 98, 96] also have a strong focus on lineage and provenance tracing. Mitrace is a Python tool for lineage and tracing of artifacts in ML pipelines. It integrates into existing codebases without requiring redesigning pipelines or rewriting pipeline code. ProvDB is a unified provenance, model lineage, and metadata management system founded on a graph-based provenance representation model that generalizes the W3C PROV data model. It features the Neo4j Cypher and Apache Gremlin graph query languages and two query operators for graph segmentation and summarization geared towards the characteristics of provenance information.

Furthermore, the TensorFlow Extended (TFX) framework provides libraries for creating ML pipelines for Data-oriented, Model-oriented and Operations Stages, as well as the ML Metadata (MLMD) library for metadata management and version control [20, 54, 52]. Embedded in this technical ecosystem, Kubeflow enables coordinated deployments of workflows on Kubernetes clusters [149, 150].

Although providing a platform for distributed in-
memory ML and predictive analysis on big data, H2O's [59, 60] ML lifecycle integration and ML artifact management capabilities are mostly comparable. Additionally, H2O aims at easy productionalization of ML models in enterprise environments.

Neptune [109], FB Learner Flow & Predictor [11, 43], MLCask [90], and Disdat [173, 172] additionally support software artifacts and further promote reproducibility. Neptune provides dataset, model, and metadata artifact storage for pipelines, experiment tracking, and workspaces for coordination of projects and collaborating users. FB Learner Flow & Predictor is Facebook's proprietary ML pipeline development and processing system. It comprises Flow, a DAG-based pipeline management system that facilitates experimentation, training, and comparison of models, and Predictor, an online inference framework based on an own model format. MLCask is a pipeline-oriented AMS. It builds upon Git-inspired versioning of pipeline components with nonlinear version control semantics for collaborative environments. Disdat manages ML pipelines and related ML artifacts by building upon two core abstractions: bundles and contexts. A bundle is a versioned, typed, immutable collection of data and files. A context is a view abstraction gathering a sharable set of bundles and assisting with managing bundles across multiple locations such as local and cloud storage environments.

**Experiment Management.**

Systems and platforms of this class aim to achieve comparability and reproducibility of exploratory ML experiments for model development, training, and optimization. They typically complement model training frameworks and AMSs of the previous class, since the results often serve as a starting point for subsequent pipeline creation and execution. Because of this, there is typically no or only limited support for the Model Operations Stage, but support for metadata and especially software artifacts.

The Deep-water Framework (DWF) [48, 39] has a basic set of functionality that enables tracking of experiments and training runs, involved artifacts’ identifiers, as well as configurations and performance metrics. DWF supports only a predefined set of models provided by TensorFlow and scikit-learn and no storage or versioning of trained models. StudioML [133, 134] additionally captures model artifacts without the necessity of modifying experiment code.

Focusing on reproducibility for the Model Development and Model Operations Stages, MLCube [105, 104] and MLPM [171, 17] both provide model packaging capabilities: MLCube is a library for packaging ML tasks and models, which enables sharing and consistent reproduction of models, experiments, and benchmarks. MLPM enables users to adopt existing ML algorithms and libraries, resolving dependencies, and deploying as HTTP services.

An even higher degree of reproducibility is provided by Guild AI [144, 145], Datmo [1, 2], Deepkit [37, 38], and Keepsake [126, 127], which additionally capture any necessary software including source code, dependencies, execution environment, and logs. Runway [152], Sacred [58, 68], and Weights & Biases (W&B) [22, 167] furthermore provide capabilities for the management of data-related artifacts.

**Apache Submarine** [29, 146] and Determined [41, 40] both provide functional interfaces and integration for popular ML training, experimentation, artifact management frameworks (e.g. TensorFlow, PyTorch, MLflow, and TensorBoard) and Python SDKs for different stages of model development without requiring extra infrastructure knowledge for orchestration. Submarine supports both on-premise clusters managed by Kubernetes or YARN, and clouds to ensure portability and resource-efficiency.

**Model Management.**

AMSs of this class treat models and model metadata as central abstractions. While typically limited integration with the Data-oriented Stage and capabilities for managing data-related artifacts are provided, the focus is on the lifecycle steps for model development and operations as well as the management of models and their metadata. Although there are some functionality-related intersections with the class of experiment management systems, AMSs of this class often provide support for the Operations Stage.

As two of the first model-oriented AMSs, ModelDB [157, 154, 156, 159, 16] and ModelHub [99, 100, 101] both focus on supporting model development, deployment, and monitoring. While ModelDB versions models and their metadata in a relational database, ModelHub incorporates an ML artifact versioning system enriching and extending Git, and a read-optimized parameter archival storage that minimizes storage footprint using deltas and accelerates query workloads with minimal loss of accuracy.

ModelKB is an AMS with a focus on model management, experimentation, deployment, and monitoring [49, 51, 50]. It uses custom callbacks in native ML frameworks to collect metadata about each experiment and automatically generates source code for deployment, sharing, and reproducibility.

The MMP is a model management platform tailored to Industry 4.0 environments by associating ML models with business and domain metadata.
It provides a model metadata extractor, a model registry, and a context manager to store model metadata in a central metadata store.

Compared to the previously discussed systems, MISTIQUE is specialized for the storage and management of model intermediates (e.g. input data, learned hidden representations) to accelerate model evaluation, performance diagnosis, and interpretability [155]. It decides for each diagnostic query whether to re-run the model or to read a previously stored intermediate and reduces storage footprint of model intermediates with storage optimizations such as quantization, summarization, and data de-duplication.

Motivated by fragmented ML workflows which require juggling between different programming paradigms and software systems, ML model training and inference algorithms as well as model management capabilities are increasingly integrated directly into DBMSs. Sql/ml enables expressing supervised ML models in SQL and translating them into Python code for training in TensorFlow [92, 93].

Vertico-ML is an ML extension on top of the distributed and parallelized RDBMS Vertica Analytic Database [44, 161, 82], which aims for eliminating the transfer of big volumes of data, avoiding the maintenance of a separate analytical system, and addressing concerns of data security and provenance by combining a full-fledged DBMS with the scalability and performance of in-database ML algorithms.

MLModelCI [176, 175, 30], ModelCI-e [62], Clipper [33, 34], Rafiki [164, 165], and Overton [124] are AMSs focusing on the Model Operations stage. MLModelCI is an MLOps platform for the automated deployment of pre-trained ML models and online model serving. Profiling under different settings (e.g. batch size and hardware) provides guidelines for balancing the trade-off between performance and cost. For the deployment to cloud environments, MLModelCI uses Docker. ModelCI-e is a plugin system for continual learning and deployment, enabling model updating and validation without model serving engine adaption. Clipper is a general-purpose low-latency model serving system. By exploiting caching, batching, and adaptive model selection techniques, Clipper reduces prediction latency and improves prediction throughput, accuracy, and robustness without modifying underlying ML frameworks. Rafiki provides a model training service supporting distributed hyperparameter tuning and a model inference service with online model ensembling that is amenable to the trade-off between latency and accuracy. Overton is an AMS focusing on building, deploying, and monitoring production models. It aims to support ML engineers in maintaining and improving model quality in the face of changes to the input distribution and new production features. CMS is a container-based continuous learning and serving platform designed for industrial monitoring and analysis use cases [85]. Its primary goal is to simplify and automate the process of model generation, deployment, and switching. Building on top of the Kubernetes management platform Rancher, CMS provides resource-efficient orchestration of model training tasks and seamless model switching and serving without interruption of online operations and with minimal human interference.

ModelHub.AI is a community-driven platform for the dissemination of deep learning models [61, 106, 107]. It is founded on a container-based software engine that provides a standard template for models and exposes interfaces for model-specific functions as well as data pre- and post-processing. ModelHub.AI is domain-, data-, and framework-agnostic, catering to different workflows and contributor’s preferences.

**Dataset & Feature Management.**

Complementary to the previously described class, this class focuses on support for the Data-oriented Stage by providing dataset, label, and feature storage and management capabilities as well as functionality and interfaces for data (pre)processing, feature selection and engineering, and provenance tasks.

MLdp is Apple’s platform for managing ML data artifacts [3]. It provides a minimalist and flexible data model for integrating different varieties of data, a hybrid storage approach for large volumes of raw data and high concurrent updates on volatile data, version and dependency management, data provenance, and integration with major ML frameworks.

In comparison, ExDra provides an infrastructure for data acquisition, integration, and preprocessing from federated and heterogeneous raw data sources [19]. It uses SystemDS for federated linear algebra programs, parameter servers, and data processing pipelines. Trained models and their provenance are stored in a model management database.

Pachyderm is a data pipeline management platform [114, 115]. It provides automated data versioning, containerized pipeline execution, as well as immutable data lineage and provenance. Also motivated by enabling explainability, ProvLake is a data management system capable of capturing, integrating, and querying data across multiple distributed services, programs, databases, stores, and computational workflows by leveraging provenance data [141, 140, 139, 66, 117].

Unlike the previous systems, Data Provenance Library [27, 138] and Shuffler [151, 132] operate at the
library level. Data Provenance Library is a Python library for capturing and querying fine-grained provenance of data preprocessing pipelines. It is based on a formal model comprising data reduction, augmentation and transformation operators, as well as a MongoDB database as provenance store. Shuffler is a toolbox for data preparation workflows of computer vision tasks. It employs relational databases and SQL for storing and manipulating annotations.

*Feast* is a feature store for managing and serving ML features to models in production [47, 46]. Feast aims for enabling DevOps-like practices for the lifecycle of features. As a single source of truth, Feast serves feature data either from a low-latency online store for real-time prediction, or from an offline store for scale-out batch scoring or model training.

### 5.2 Discussion Along Criteria

This section discusses the assessment results comparatively along the criteria and subcriteria.

**Lifecycle Integration.**

Requirements engineering in the context of ML is a young field of research [5, 162, 163]. Many of the methods and approaches known from traditional software engineering have yet to be adapted for ML systems [16, 21, 91]. Additionally, the specification of non-trivial requirements often necessitates domain expert knowledge. Consequently, the tool support is still poor and requirements engineering functionality is not yet covered by the assessed AMSs.

In contrast, many of the systems and platforms studied do at least provide partial functional support or interfaces for the Data-oriented Stage: While most systems and platforms lack support for the Data Collection step, probably due to the individuality of data type, volume, sources, and collection approaches, many integrate functionality or provide functional interfaces for at least one of the Data Preparation & Cleaning, Data Labeling, and Feature Engineering & Selection steps (ca. 75%).

Systems and platforms for lifecycle management offer the widest range of functions: Cloud platforms with integrated artifact management often provide their own tools with a graphical UI. Open-source systems such as ClearML or MLflow, on the other hand, offer interfaces for integrating user code, which can then be used within pipelines for automation. In particular, AMSs with a narrow focus stand out here: For example, Feast and Hopsworks provide feature stores that are designed specifically for ML feature selection, storage, processing, and distribution.

A large proportion of the systems and platforms assessed provide wide or complete support for the Model-oriented Stage (86%), including both integrated functionality and interfaces to typical ML frameworks (TensorFlow, PyTorch, etc.), which provide functional support for model building, training, evaluation, and optimization. Nevertheless, some systems deviate from this due to their goals: for example, MMP and Vertica-ML do not support *Model Design* but have an integrated set of ML models.

The functional support for deployment and monitoring of models (*Operations Stage*) is quite heterogeneous: 27 of 64 (ca. 42%) provide full support and 14 of 64 (ca. 22%) partial support. A majority of the lifecycle management systems and platforms provide functionality for deploying models as web services (e.g. via REST interfaces) as well as continuous collection and monitoring of performance and quality metrics. In 6 out of 13 systems from the pipeline management class this is also the case, 3 further systems only provide support for deploying model serving environments. In the remaining 3 classes, the support is much lower due to the objective of the corresponding systems and platforms.

**Artifact Support.**

With more than 92% on average, a large proportion of the systems and platforms takes model artifacts into account. The proportion for data-related artifacts is lower at ca. 80% (support for at least one type). Model-specific metadata such as hyperparameters or metrics are collected and processed by more than 80% of the systems. Experiment/project metadata and pipelines are supported by ca. 67% respectively 55% of all systems. Only every second system takes software artifacts into account. The kind of support is strongly dependent on the objective and system class and is very heterogeneous across all systems and platforms; the relationship to the supported operations must be considered.

**Operations.**

With more than 90%, the majority of the systems and platforms offers artifact capturing and logging functionality. Depending on the use of repositories and comparable techniques, only less than a half enable snapshots and intermediate states of artifacts to be checked in and rolled out again. Also, over 90% provide operations to query and retrieve stored artifacts. More than two-thirds (ca. 67%) have comparison functionality, ca. 52% provide artifact lineage, and only ca. 17% offer provenance functionality. Visualization operations are present in ca. 62% of the systems. While the presence of typical management operations such as modify, delete, and execute & run is quite common, deployment operations are only present in about half of all systems.
While the platforms and systems of the lifecycle, experiment, and model management classes mostly provide complete functionality for export and import as well as sharing with other collaborators, these functions are less prominent or not available at all for the other classes.

**Collection & Storage.**

The collection of artifacts can either require explicitly added instructions (subcriterion *Intrusive*), such as Python functions or callbacks, or be (semi-)automatic (subcriterion *Non-intrusive*). While exactly half of the systems provide both intrusive and non-intrusive collection of artifacts, primarily of the lifecycle, pipeline, and experiment management classes, almost two-thirds of all systems (ca. 64 %) at least support automatic collection.

The types of storage used are highly dependent on the goals and focus of a system or platform. For example, lifecycle management systems provide an appropriate type of storage for each type of artifact (see §4 for related discussion). It is also recognizable for the other system classes that the supported storage types are related to the supported artifacts themselves. Exceptions to this are systems and platforms such as Velox or MMP, which are tailored to specific domains and for this reason only support limited number of dataset and model types, as well as MISTIQUE or sql4ml, in which deep learning models are storage based on individual data models or supervised ML models in relational table structures.

In total 27 of the 64 systems and platforms complementarity support both the complete versioning of a project or lifecycle state including any artifacts and the versioning of individual artifact snapshots. About half of all systems and platforms support at least repositories for versioning, whereby in addition to the general-purpose version control systems, variants tailored specifically to ML are increasingly being used, which, for example, perform effective model versioning using deltas and provide special commands for model, dataset, pipeline, and experiment comparison and lineage information, as demonstrated by ModelHub’s dlv and DVC.

**Interfaces & Integration.**

Overall, many of the examined systems and platforms provide a wide range of interfaces and integration. While Python SDKs or REST APIs are provided by almost 9 out of 10 systems and platforms (ca. 88 %), CLI tools and web UIs are available in over two-thirds (ca. 66 % respectively ca. 73 %). Here, especially the lifecycle management and experiment management classes stand out.

The programming language support (ca. 94 %) primarily Python as the data science quasi-standard –, the integration for ML and data science frameworks (ca. 92 %) as well as the direct or indirect support for interactive and collaborative notebooks (ca. 92 %) is pronounced across all classes of systems and platforms. In particular, because of their focus, the systems and platforms of the lifecycle, pipeline, and experiment management classes have the highest degree of functional integration, which is related to the first category Lifecycle Integration.

**Operation & Licensing.**

The capabilities for system and platform operation are highly dependent on the corresponding software architecture. 26 systems allow only one, 14 two, and 24 all three modes of operation. Over half of all systems can be used locally (ca. 54 %), either as a library, local server application, Docker container, or locally executable Kubernetes variant such as Minikube or Kind. Two-thirds of the systems (ca. 66 %) can be deployed on-premise (e.g. on a dedicated server or cluster) and three-quarters are capable of running in a cloud (ca. 75 %).

Among the systems and platforms studied, a total of 35 systems have some kind of free license, typically with source code freely available, and the remaining 32 have a non-free, proprietary license.

**6. CONCLUSION**

This paper discusses system support for ML artifact management as an essential building block to achieve comparability, reproducibility, and traceability of artifacts created and used within the ML lifecycle. Objectives, fields of application, and functional ranges are heterogeneous and the selection of AMSs is quite difficult. Based on a systematic literature review, we derive functional and non-functional criteria that enable the systematic assessment of AMSs. Using the criteria, we assess and discuss a comprehensive selection of 64 systems and platforms from academia and industry.

As complementary and future work, we aim to investigate system support for automating ML tasks, e.g. AutoML techniques such as automated hyperparameter optimization, neural architecture search, and meta-learning, as well as for establishing ML-related security properties, e.g. techniques for hardening against and preventing model exploratory, data poisoning, and evasion attacks.

**7. ACKNOWLEDGMENTS**

This work was partially funded by the Thuringian Ministry of Economic Affairs, Science and Digital Society (grant 5575/10-3).
8. REFERENCES


[30] Cloud Application and Platform Lab, NTU. cap-ntu/ML-Model-CI: MLModelCI is a complete MLOps platform for managing, converting,


machine-learning/team-data-science-process/.


[140] R. Souza, M. Mattoso, L. Azevedo, R. Thiago,


[174] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi,


ABSTRACT

Reasoning-based query planning has been explored in many contexts, including relational data integration, the Semantic Web, and query reformulation. But infrastructure to build reasoning-based optimization in the relational context has been slow to develop. We overview PDQ 2.0, a platform supporting a number of reasoning-enhanced querying tasks. We focus on a major goal of PDQ 2.0: obtaining a more modular and flexible architecture for reasoning-based query optimization.

1. INTRODUCTION

For many decades database researchers have explored the interaction of reasoning in query planning. Particularly relevant to our work are: 1. Optimizing queries with respect to constraints. We are given integrity constraints $\Sigma$, which we know that our input data satisfies. Given a query $Q$, our goal is to rewrite $Q$ to a lower cost query, or directly to a physical plan, taking advantage of the constraints in $\Sigma$. 2. Querying over limited interfaces with constraints: We have constraints as above, but access to our dataset restricted. A model for access restrictions is to associate each relation $R$ with a set of function calls or access methods, each requiring a subset of attributes of $R$, and performing a lookup on $R$. Access restrictions clearly complicate the query planning problem, even in the absence of constraints. The presence of access restrictions can make a naïve implementation of $Q$ impractical or even infeasible even when $Q$ is in principle answerable with the available data. The goal is to convert $Q$ to a physical plan that makes use of the given access methods, again with the equivalence being with respect to constraints $\Sigma$.

These topics have received extensive attention in the database literature. For query optimization with respect to constraints, there is a well-studied approach, the “chase and backchase” (C&B) [9], which can be seen as a special case of interpolation [13]. For querying with respect to access restrictions, there is considerable theory [8, 7], with the most common approaches for dependencies being closely-related to the C&B. While the theory of these topics has continued to evolve, software support for the three topics above has not matured the way one would hope. The authors are aware of no open source system for querying with respect to constraints, access methods, or (as a standalone tool) views, even though algorithms have been available since the 1990s [2]. The lack of mature systems relates to an even broader issue. Although individual problems have been studied in the literature, a vision of how components for the problem might interrelate is lacking.

In this work, we try to make a small step in the direction of software maturity. We introduce an open source evolution of the PDQ system, originally targeted towards the second task above [5, 6]. Although the evolved 2.0 system focuses on the two tasks above, it can also be used as a standalone reasoning tool supporting “open world query answering” or chasing [3]. The goals of the software are to: 1. support the tasks above with performance comparable to PDQ 1.0, which is to our knowledge, the state of the art, 2. provide modularity, allowing a developer to mix and match different components of a solution, including components related to how reasoning is performed, how source data is stored and accessed, and how reasoning within query optimization is combined with non-logic factors like cost, 3. provide a base of well-documented code [1] to build on.

EXAMPLE 1. Consider a schema listing relations Actor, Movie, and MovieActor. The intended semantics of the tables would be captured with integrity constraints, including constraints capturing the semantics of MovieActor: $\text{Movie}(\text{mid}, \text{mname}, \text{aid}) \land \text{Actor}(\text{aid}, \text{aname}) \rightarrow \text{MovieActor}(\text{mname}, \text{aname}),\text{MovieActor}(\text{mname}, \text{aname}) \rightarrow \exists \text{aid aname}$ $\text{Movie}(\text{mid}, \text{mname}, \text{aid}) \land \text{Actor}(\text{aid}, \text{aname}),\text{and Movie}(\text{mid}, \text{mname}, \text{aid}) \land \text{Actor}(\text{aid}, \text{aname}) \rightarrow \text{MovieActor}(\text{mid}, \text{aid}, \text{aname}).$

Consider the query asking for the names of all actors in the movie “Inside Man”. Depending on which tables the user is familiar with, this could be expressed...
in different ways, for example SELECT aname FROM MovieActor WHERE mname='Inside Man'. However, there are a number of different ways to answer the query, in addition to the naive join plan; for example, it is possible to first access MovieActor and then filter. PDQ will explore all equivalent plans, looking for the one with the lowest cost.

Finally, assume these tables are available only as web services that require particular inputs, and the MovieActor table requires an aid as input. Then some of the plans will no longer be valid, and PDQ will restrict the search accordingly. PDQ can also evaluate the plans produced by the search.

We focus on the architecture of PDQ 2.0: how it would be used to solve a task, what flexibility it supports, and what issues arise. Algorithmic and implementation issues are discussed elsewhere in the context of PDQ 1.0 [5, 6], and for space reasons we defer a discussion of the 1.0 to 2.0 delta to online material [1].

2. PDQ CORE FUNCTIONALITY

The motivating application for PDQ is enhancing query planning via reasoning. Given query \( Q \) and \( \Sigma \), we seek a query plan \( P \) equivalent to \( P \) for all instances satisfying \( \Sigma \), where \( P \) has lower cost according to some cost function. A variation of the problem allows access restrictions in the schema, where the restrictions are specified by each relation being associated with a collection of access methods, with each access method implementing a lookup with some requirements: the inputs to the method. The goal is now to get a plan where access to the data is only allowed through these access methods. The abstraction of access methods serves to model a number of application scenarios [2].

The query planning problem can, in principle, be reduced to the well-known problem of computing certain answers to query \( Q \) on database \( D \) with respect to constraints \( \Sigma \). These are the tuples \( \vec{t} \) such that in every database \( D' \) that contains \( D \) and satisfies \( \Sigma \), \( \vec{t} \) is an answer to \( Q \) in \( D' \) in the usual sense. The decidability and complexity of this task depends on the properties of \( \Sigma \). The version of PDQ described here deals only with one of the most well-studied cases, where \( \Sigma \) consists of tuple-generating dependencies (TGDs). These are logical sentences of the form: \( \forall \vec{x} \lambda(\vec{x}) \rightarrow \exists y \rho(\vec{x}, y) \) where \( \lambda \) and \( \rho \) are conjunctions of atoms, or equality generating dependencies (EGDs), sentences of the form \( \forall \vec{x} \lambda(\vec{x}) \rightarrow x_i = x_j \) where \( \lambda \) is as above. See Example 1 for examples of TGDs: in the examples, the outer universal quantifiers have been omitted. For such sentences the certain query computation problem can be tackled via the well-known chase algorithm [10], in which \( D = D_0 \) is iteratively enhanced to database \( D_1 \ldots D_n \) via applying chase steps. A chase step for a TGD \( \tau \) extends \( D_i \) by adding witnesses for the conclusion \( \rho \) of \( \tau \), while for an EGD it identifies two elements of \( D_i \) to make the conclusion hold.

We state the reduction only in the case where \( Q \) is a Boolean Conjunctive Query (CQ), but the reduction generalizes to non-Boolean CQs. Given \( Q, \Sigma \), and the access methods, one can derive a set of constraints \( \Sigma^* \) and another query \( Q' \) such that there is a plan for \( Q \) using the access patterns modulo \( \Sigma \) exactly when \( Q' \) is certain for the database \( CanonDB(Q) \) with respect to \( \Sigma^* \). Here \( CanonDB(Q) \) is the canonical database of \( Q \), whose elements are the variables and constants mentioned in \( Q \), containing facts for each atom of \( Q \). Furthermore, one can find plans for \( Q \) by exploring chase proofs proving \( Q' \) from \( \Sigma^* \) and \( CanonDB(Q) \). This process was developed in the context of views and integrity constraints by Popa, Tannen, and Deutsch [9]. The variant for access patterns was developed in [8, 5, 6]. The idea is that one first chases \( CanonDB(Q) \) with the constraints in \( \Sigma \), producing a set of facts \( I_0 \). One can then begin to apply certain axioms capturing the interface restrictions. In [7] these latter axioms are called accessibility axioms. Firing accessibility axioms at the chase level corresponds to exploring a plan with certain accesses. While exploring these plans, one also checks whether they are equivalent to the original query \( Q \), by chasing again. Following [9], this process of iteratively exploring plans is called back-chasing.

Most of the details of this reduction of planning to certain answer computation will not concern us, but there are several aspects that we want to highlight: 1. The certain answer process that is generated for planning is initiated with the incomplete dataset \( CanonDB(Q) \): that is, a dataset only as large as the user’s original query. While the chase process will generate new facts, generally the amount of data involved is very small, in contrast to standalone certain answer computation, typically directed towards large-scale data. 2. In order to find one plan, we need simply to find one chase proof. But to find good plans we will need to explore multiple chase proofs. Thus, we need to follow prior work on constraint-based query planning [12] in finding efficient ways to explore multiple proofs. 3. Determining which plans are low cost requires costing plans. When the process is run on top of a DBMS, this may be done via a function call to an \textsc{Analyze} method. In a data integration setting the costing will have to be done in middleware, which will usually involve metadata, such as histograms or catalogs.

3. SOFTWARE ARCHITECTURE

The PDQ 2.0 architecture, released recently on GitHub [1], is built around 6 major packages: Com-
mon, Datasources, Reasoning, Planner, Cost, and Runtime. The dependencies are shown in Figure 1.

The Common package has base classes for database schemas, database queries, database query plans, and first-order logic. This includes PDQ’s internal data management infrastructure: the management classes which store the intermediate results of reasoning. PDQ allows internal data management to be deferred to an external DBMS, and some of these database classes simply translate query evaluation for logical formulas into SQL evaluation, passing them off to an external engine. But PDQ also has a bare-bones main-memory DBMS that can evaluate basic SQL SELECT queries.

The Reasoning package supports reasoning with common classes of database constraints. It exposes an abstraction called a chase state, which makes use of the database infrastructure in the Common package. A reasoner object can transform a chase state by either making a single chase step or chasing until termination (or timeout). The package provides an entry point allowing PDQ to be used for standalone reasoning tasks: computing the chase of an instance, determining the certain answers of a query under constraints, or the related task of determining whether one conjunctive query follows from another in the presence of constraints (query containment with constraints).

The Datasources package represents datasource descriptions and their translations to PDQ’s internal representation. It includes code for extracting metadata from other formats – e.g., from the catalog of a DBMS. It also includes PDQ’s framework for wrapping web services as access patterns.

The Cost package represents cost estimators for physical plans. PDQ can defer the estimation to a DBMS; simply translating the plan to SQL and sending it to the ANALYZE method of a DBMS. This technique is appropriate if using PDQ to add constraint-based optimization on top of an existing DBMS. PDQ also includes hand-rolled cost estimation techniques; for example, using catalog estimates and heuristics.

The Planner package is the most complex. The main abstraction within it is an explorer, which is given an input query $Q$ as well as a schema $S$ describing a collection of abstract access methods and constraints $\Sigma$, and searches for the lowest cost physical plan making use of the accesses that is equivalent to $Q$ modulo constraints $\Sigma$. The planner makes use of the certain answer computation provided by the Reasoning package and the cost calculations provided by the Cost package. An explorer can restrict the search to the traditional left-deep plans that are often the focus in traditional query planners; it can also search for bushy plans, imposing additional restrictions to make the search more manageable.

Finally, the Runtime package is middleware for evaluating physical plans. It provides executors that represent implementation strategies for plans. At the leaf level of a plan tree, each executor makes use of the access methods specified by the Datasources package. The strategy for evaluating other plan operators is encapsulated in the executor, which implements traditional and dependent joins. Note that the Runtime package has no dependence on the Planner package: it can be used to evaluate any plan, even one constructed operator-by-operator via the plan constructors.

In addition to these core packages, PDQ 2.0 has packages for front-ends. A web-based GUI serves as a demonstration of the planning functionality available for a fixed schema. A console-based GUI allows for both planning and some administration of the schema. It also serves as purely Java-based infrastructure for building front ends to the system. There are also packages (omitted from Figure 1) for demos, regression testing, and web services wrapping system functionality.

We show how some of the scenarios in Example 1 would play out using the PDQ 2.0 architecture. In setting up an application, a DBA would create metadata that describes the web services: for example they would describe a REST web service that takes as input an aid and returns other MovieActor info. PDQ offers a simple specification language that allows DBAs to describe how inputs are inserted into a service call and how the outputs are extracted, using XPath with placeholders. This specification bridges between the format of the concrete web services and the more abstract view of the services as taking in a parameter and returning a table. The parsing of these services as well as the marshalling and unmarshalling code generated by them is done via
the Datasources package. The DBA must also supply a schema that includes integrity constraints, those listed in Example 1. PDQ provides a number of syntaxes for these, a native XML format as well as the "ASCII" format supported by the ChaseBench project [3]. The processing of these inputs is done within the IO subpackage of the Common package. The DBA may also make available catalog information about the underlying web services: e.g., how selective an access on MovieActor with a given aid is. This can be described using a catalog file processed by the Cost package.

We describe how a querier would interact with the system. One option is to utilize the command line interface, where various parameters (described below) will need to be passed. Alternatively one of PDQ’s GUIs can be utilized. Both GUIs allow a user to select a schema, bring up an existing query or write a query from scratch, validate the query against the schema, and then perform planning. The Web GUI in Figure 2 shows a query written in SQL referencing several data sources for geographic information, available via Yahoo web services. The query can be answered via data from other services (e.g., Google): PDQ will detect multiple plans and select the one with the lowest cost. For the Web GUI, a fixed schema, planning method and cost function is available. In the console GUI the user can configure internal parameters as in the command line; it can be run on a set of metadata files available on a local machine. In both GUIs, a user can see the best plan, while within the console GUI they can see the entire search space of plans. The user can choose any available plan and run it, using the Runtime package. A small number of output tuples are displayed in the GUI, while the entire output can be downloaded as a CSV file.

We now turn to the parameters and their relation to the architecture. One set of parameters controls the type of reasoning to use: basically, how to perform multiple chases, with each chase corresponding to a potential plan. One parameter controls whether this is stored in an external database or PDQs in-memory DBMS.

Note that PDQ’s internal database is distinct from the datasources with which PDQ interfaces. A second reasoning parameter specifies what “saturating with consequences” means: it can mean that reasoning is done until complete, or using a fixed number of steps. A reasoner object within the Reasoning package encapsulates these decisions. A second set of parameters concerns planning search heuristics. They are encapsulated in the explorer classes that the Planner package provides. For example, the linear planner searches only for tree-shaped plans, while the dag planner allows bushy plans to be considered. A third set of parameters determine which cost function provided by the Cost package is used. Some cost functions that make use only of the syntax of the plans (e.g., based on size of the plan or number of joins), while others make use of catalog information, like the selectivity of the MovieActor access method mentioned above. PDQ provides a default executor from the Runtime package for running the plans, but a plan can also be saved for later running by other means (e.g., translation to SQL and execution via a DBMS).

4. ARCHITECTURAL ISSUES

In the process of constructing PDQ, we encountered a number of design decisions that are worth highlighting. They give a hint at the complexity of query reformulation/planning software, which we believe has been one of the major obstacles in creating a codebase for this problem.

Interfacing reasoning and data management. The most basic issue we needed to deal with was how the reasoner would use a DBMS. We considered this problem in the case where the constraints are dependencies where the chase algorithm terminates. In implementing the chase there are a number of design decisions, including whether one interfaces to a data management system with SQL, whether one works natively in main memory or on top of an external DBMS, and whether one implements nulls via strings or via integers. Many of the trade-offs are discussed in [3]. For the purposes of PDQ, the question was not so much which design is better, but to what extent we can insulate the reasoner from these decisions.

In our original implementation, we attempted to abstract away completely from database management issues, creating a homomorphism detection class that could be implemented either in bulk within SQL – to find all triggers at once and then use SQL update commands to do a full round of chasing, or tuple-at-a-time, allowing one to perform the restricted chase. The resulting architecture was complex, and since our abstractions were not standard, it was difficult for a developer to maintain the structure. We thus simplified the architecture to assume an SQL interface to a DBMS. The rea-
soning project of PDQ provides an implementation of the terminating chase on top of an SQL abstraction. We developed two implementations of the abstraction, one using the open source DBMS PostgreSQL, and another using our own main-memory implementation, supporting only conjunctive queries with inequalities.

Insulating the reasoner from the encoding of nulls in the DBMS is challenging. In performing the chase it is necessary to generate fresh values for unbounded datatype. One could use different value-generation strategies for each datatype, but a common solution is to make use of strings, where creating a fresh value and encoding the parameters it depends on (e.g., the dependency and the homomorphism) can be done using string operations. PDQ takes a variant of this approach: we have a special class of values corresponding to null, and they are associated with string. In our own main memory implementation, tables are not strongly-typed, so in executing a chase step we are free to add tuples containing nulls in positions where the schema specifies a datatable other than string. When we use an external DBMS, we simply pre-process to encode the entire database as strings and perform the entire reasoning process with the encoded database. The encoding/decoding steps are given in a single place within the code, and thus the heart of the chase implementation code need not be aware of whether the encoding has occurred.

**Interfacing planning and reasoning.** The reduction of planning to reasoning mentioned in Section 2 is simple, but a few caveats must be noted. For finding some plan, it gives a very general reduction for an arbitrary first-order theory. But for exploring low-cost plans it works only for restricted cases, like dependencies and disjunctive dependencies, and there are modifications to the planning algorithms needed for each case. Even when restricting to dependencies with terminating chase, there are some design decisions relating to the fact that we need to explore many proofs. We need to maintain multiple chase states of the exploration, all of which are subsets of the facts in \( \text{Chase}_\Sigma(\text{CanonDB}(Q)) \). This required us to broaden the DBMS abstraction used for query answering, introducing a multi-database (omitted from Figure 1) consisting of multiple virtual instances of the same schema. In our main-memory DBMS this is implemented by just creating multiple Java object instances. In our external DBMS implementation, a multi-database is realized by adding an additional id field, and there is then some additional translation of queries and updates.

Another issue involves the extent of parallelization. While many aspects of chasing are inherently sequential, much of the exploration of plans making use of the chase – the “back-chasing” – can be done in parallel. PDQ supports this, but the cost is difficulty in system testing, since with parallelization enabled, different runs will often produce different plans.

The interface to a reasoner was a key design decision. We oriented PDQ towards chase-based reasoners that work “in place” and expose the evolving chase state – that is, supporting the \texttt{MakeChaseStep} and \texttt{Saturate} methods. This simplified the design, but did restrict reformulation options. In the course of the PDQ project we have experimented with a number of more advanced reasoning methods: 1. The “infinite chase” or “blocking chase”, applicable to tuple-generating dependencies that satisfy conditions such as guardedness; 2. The disjunctive chase, which generalizes the classical chase to rules with disjunction in the head; 3. Resolution-based theorem provers [4], applicable to arbitrary first-order integrity constraints. All of these are available in the PDQ 1.0 codebase. While the first could be re-implemented in PDQ 2.0 (by enhancing the ChaseState), the second and third require changes to the interface.

**Interfacing planning, cost, and runtime.** Query reformulation systems based on the C&B were geared towards “high-level planning” – logical reformulation with constraints. The notion of cost supported was minimal size of the reformulation [12]. In PDQ a key goal was to allow more general notions of cost, including cost functions that can make use of database statistics and the ordering of operations. On the other hand, it is desirable for planning to be decoupled from runtime processing. One should be able to take a plan and utilize it on any runtime that implemented that accesses in the plan.

Our solution is to support high-level planning, but with some abstraction for the notion of “better plan”. The PDQ planner assumes access to a cost function. We provide a number of cost functions, some that just use count or weighted sum of certain operators. For example, one cost function takes a weighted sum of the subterms corresponding to data access. The weight of an access method defaults to 1, but it can be assigned a numerical weight in a catalog file which is read in at planning time. More sophisticated cost functions work recursively on an access plan, making use of catalog information in a standard way. The cost functions work on “logical plans”, expressions in a variant of relational algebra. The runtime package converts these expressions to physical plans – for instance, by choosing a join implementation. Cost information is not taken into account in the “low-level plan creation” given by the runtime process. PDQ also allows costing to be deferred to a DBMS, using an SQL ANALYZE command. This is appropriate if PDQ is used only for planning.

**Interfacing planning and data APIs.** A unique feature of PDQ is that it performs query reformulation over access methods. An access method is abstractly represented by an identifier, a relation, and a subset of at-
tributes of the relation (the *input attributes* for the relation). Plans based on the access methods are built up from access primitives via the usual relational algebra commands along with dependent joins [11], in which the ordering of the joined terms is fixed. The high-level description of access methods, along with the integrity constraints, are sufficient for the purposes of determining what the valid plans are for a given query. Assuming a cost function on such plans, we can begin to search for low-cost plans implementing the query.

But to run the plan we need an implementation of each access method on relation $R$ with input attributes $a_1, \ldots, a_k$; this is a function call that takes values for these attributes and returns a set of tuples matching $R$. PDQ provides a Java interface for defining implementations of access methods, so in principle a developer can implement an access method using an arbitrary Java function. A thornier issue, and a longstanding one in the database community, is how to ease the burden on DBAs in wrapping datasources for PDQ. Automated support is provided for a few kinds of access methods. Access to data in main memory is supported in a straightforward way. For database lookups, the developer just needs to declare that the method is implemented via an SQL DBMS, and associate it with JDBC-style connection information. A more involved type of access method implementation is via web services. Even sticking to REST-based services, we unsurprisingly found a great diversity in the encoding schemes used to wrap real-world services. After experimenting with a number of more ambitious specification formalisms, PDQ currently allows only a simple kind of wrapper, where the developer needs to specify regular expressions that determine how values are encoded in a URL. Returning to Example 1, the wrapper implementing a lookup of Movie on aid might include the “template” expression http://www.imdb/actor={1}. When a service call is made with a particular actorid, that value will be inserted in place of $1$ in the template.

5. DISCUSSION

We presented the open source version of PDQ, focusing on introducing the functionality offered and presenting some issues — many of them admittedly still open — in creating a simple but flexible architecture for reasoning-based optimization.

We consider the system in light of the goals outlined in Section 1. Despite the challenges noted in Section 4, PDQ 2.0 offers much in the way of re-use. Developers can re-use our reasoner and runtime independent of planning. They can use our cost estimators and data-source infrastructure independent of both reasoning or planning. It is also possible to swap in another chase engine in place of our reasoner, if the engine provides

the *ChaseState* interface from Figure 1. The codebase is well-documented, and we are using it as a foundation for our ongoing projects in integrating querying and reasoning. We hope others can build on it as well.

Turning to the final goal, for space reasons we could not discuss the algorithms, implementation, and performance of PDQ 2.0 in depth. The search algorithms are those from PDQ 1.0: see [6]. The performance of PDQ 1.0 for standalone reasoning is evaluated in [3]. We do note that PDQ 2.0 configured with the internal in-memory database improves substantially on the reasoning performance of 1.0; this translates into better performance on the ChaseBench benchmark and the ability to perform complex planning tasks in seconds. We refer the reader to the Wiki in [1] for details of the performance on both standalone reasoning and planning.

6. REFERENCES

Reminiscences on Influential Papers

When I started my PhD, I wanted to do something related to systems but I wasn’t sure exactly what. I didn’t consider data management systems initially, because I was unaware of the richness of the systems work that data management systems were built on. I thought the field was mainly about SQL. Luckily, that view changed quickly.

This issue’s contributors cover a variety of topics targeting different layers of the systems work in data management: data management on specialized and resource-constrained hardware, design of storage managers, and synergies across data management, data processing, and machine learning systems. They all underline the richness of the field. Enjoy reading!

While I will keep inviting members of the data management community, and neighboring communities, to contribute to this column, I also welcome unsolicited contributions. Please contact me if you are interested.

Pınar Töüzüm, editor
IT University of Copenhagen, Denmark
pito@itu.dk

Tilmann Rabl
Hasso Plattner Institute, University of Potsdam,
Potsdam, Germany
tilmann.rabl@hpi.de

Christophe Bobineau, Luc Bouganim, Philippe Pucheral, and Patrick Valduriez.

PicoDBMS: Scaling down Database Techniques for the Smartcard


I was always fascinated by complex software systems and advances in hardware. When I started my PhD, in the search for a topic, I enjoyed many database systems papers and while I generally followed the trend of thinking about larger and larger databases, I also liked work that considered specialized and constrained environments.

One of the papers that was an inspiration during my PhD and continues to influence my work is “PicoDBMS: Scaling down Database Techniques for the Smartcard” by Bobineau et al. It is a very nice instance of a prototypical database systems paper of the time: it is easy to follow, but contains many interesting and thought-provoking problems and solutions. I especially like the focus on special but widespread hardware and the pragmatical solution, which trades off performance and efficiency in a practical setup.

Based on the preposition that structured data should always be managed in a database system, the authors design a database management system for smart cards. The first thing I learned from the paper is that smart cards not only come in the form of memory cards but can contain fully programmable microprocessors that can execute arbitrary code, but only while powered through a reader device. Back then, and even now, smart cards have very constrained resources, which requires rethinking database architecture.

The authors did so in an extreme way. They design custom storage formats that optimize for space efficiency by selecting between classical flat storage, domain storage, which encodes attributes as pointers to a separate domain table, and ring storage. Ring storage stores pointers between tables (e.g., key relations) and between tuples and attribute domains in a circular fashion, to enable bidirectional lookups with very limited space overhead. Just as interesting, the authors discuss the problem of query processing with a very limited amount of memory. The authors explain how to process queries using extreme right deep query plans, which pipeline
all operators, including joins and aggregations, thus not materializing any intermediate results. They show that their design is feasible for tables with thousands of records.

Besides being interesting technically, the paper also is reminiscent of an approach of privacy and security, which is almost unthinkable today. The idea of giving the user full physical control over their data has been traded in for promises of convenience and better applications in times of cloud-based services, which yet have to demonstrate to be an improvement for the user in privacy-relevant use cases.

The other aspect that the paper addresses and which, in my mind, gets too little appreciation in database research today, is the constrained environment and small applications. Running a DBMS on a smart card means dealing with very limited memory and relatively small amounts of data. Thinking about efficiency at scales below gigabytes and tens of cores, even today makes sense and helps to improve the long tail of applications outside of Internet companies.

Avrilia Floratou
Microsoft Gray Systems Lab, USA
avrilia.floratou@microsoft.com

Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O’Neil, Pat O’Neil, Alex Rasin, Nga Tran, and Stan Zdonik.

C-Store: A Column-oriented DBMS.

When I was invited by Pinar to write for the “Reminiscences on Influential Papers” column, I initially thought the task would be straightforward. However, upon further contemplation, I realized it was quite challenging. There are many papers that have influenced not only the areas and problems I have been working on, but also my perception of what constitutes true innovation in a field that has been established for over four decades. A truly exceptional paper not only alters the direction of progress in research and industry, but also affects how aspiring researchers approach problems. After much consideration, I ultimately chose to discuss the C-Store paper, which I believe was not only influential for the database community, but also had a significant impact on my understanding of what constitutes high-quality research.

As a fresh graduate student in data management, I did not fully understand the impact that simple ideas can have. Many students, including myself at the time, tended to create complex problem statements and develop complicated solutions, disregarding simpler ideas as being too obvious or unoriginal. The C-Store paper helped me to overcome this tendency. The problem statement was quite straightforward: “How can we design databases to optimize for read-mostly workloads?” The solution and approach were also simple: “Let’s store the data in a columnar format and explore how the rest of the DB engine should be designed based on that.” Although it is certainly challenging to design and build such a system, the problem addressed and the fundamental design choices behind the system were all quite simple and resonated well with the database community. Another important lesson I took from this paper is that combining ideas from previous work (e.g., the decomposition storage model proposal from the 80s and the substantial work done in the context of the Monet system) with novel ones to build a functioning system and then demonstrating its efficiency can be a significant research contribution. Focusing solely on how to differentiate from others rather than finding the best solution to a problem may actually hinder potential for transformative change.

C-Store was a highly influential development in the database community. As one of the first columnar database systems, it has inspired further research in various areas such as columnar formats, compression techniques, operating on compressed data, and interactions with hardware. The system had a significant impact on industry as well: it serves as the foundation for the Vertica analytic database. Multiple other database technologies such as DB2 BLU, SQL Server Columnstore Indexes, BigQuery Capacitor are all based on fundamental concepts behind C-Store. Furthermore, many of the ideas behind C-Store and columnar databases in general have influenced Big Data analytics platforms such as Hadoop and Spark. For example, in my own work, we studied how to create efficient columnar storage formats for Hadoop and HDFS. Today, Apache Parquet and Apache Arrow are two columnar formats that are leveraged by many modern analytics engines. I believe that the research presented in this paper is a prime example of the kind of work that is needed to truly push the boundaries of the field and advance the state-of-the-art in data management.
This is one of the foundational papers of the “data mining” area, bridging the areas of databases and ML. Nowadays, since “data mining” is used mainly for algorithmic aspects of applied ML/AI, this paper can be seen as canon in the new “ML systems” area. Its impact, both immediate and transitive, has been massive. On the former, its “cluster feature” idea revolutionized how clustering algorithms work, especially on larger-than-memory data. On the latter, it showed that by taking a joint view of the abstract math involved in ML programs and data systems principles such as decomposing computations on data and staging data movement across the memory hierarchy, one can build better ML systems. That also means we must be mindful of both system metrics such as runtime and ML metrics such as accuracy.

That basic philosophy of reimagining ML workloads as DB-style workloads—decoupling the what of the math from the how of the execution—really inspired me as a PhD student. For instance, one of my papers that introduced the paradigm of “learning over joins” resembles BIRCH in many ways, e.g., unpacking the ML math to explain data access patterns, carefully counting I/O and CPU costs, formally analyzing its efficiency, and extensive empirical evaluation, all in service of making end-to-end ML/AI workloads faster, more scalable, and easier for users. That philosophy has guided my subsequent research as an academic in this now popular sub-area of “data management/systems for ML.”

As the BIRCH paper showed a quarter century ago, the synergy between the “DB systems” area and the “ML systems” area is high. After all, DB and ML/AI are two sides of the same coin of data-intensive computing, both at the heart of the software foundations of Data Science. So, I hope the DB community steps up to fulfill its intellectual responsibility to both science and wider society by helping democratize ML/AI and Data Science.

---

Danica Porobic
Oracle, Zürich, Switzerland
danica.porobic@oracle.com

Surajit Chaudhuri and Gerhard Weikum.
Rethinking Database System Architecture: Towards a Self-tuning RISC-style Database System.

In Proceedings of the 26th International Conference on Very Large Databases, pages 1-10, 2000.

I started learning about computer architecture only at the beginning of my PhD studies and I was immediately impressed by the elegance of RISC designs, even though CISC was much more popular at the time. Before starting my PhD, I have worked on the edge of a complex database system codebase and have experienced some of the pains Chaudhuri and Weikum discuss in this paper. When I first read this paper, I appreciated the succinct representation of the problems, but the analysis and recommendations seemed a bit abstract and vague.

Few years later, towards the end of my PhD, I was thinking about the designs of transaction processing systems for clusters of multicores and “rediscovered” this paper and its clear and well presented discussion of complexities of data management systems and their causes that I was starting to understand better. However, the application design considerations and research agenda recommendations were still vague to me.

After spending a few years in the industry, gaining better understanding of the real world application requirements and working with a mature and well structured data management system, I think I finally understood the analysis and recommendations Chaudhuri and Weikum presented in this visionary paper. It is certainly possible to design clearly separated system components with well defined interfaces and behavior and pick and choose these components for each application and deployment scenario. This paper goes a step further by laying out a vision for tuning and interoperability that would enable different teams to advance the state of the art much faster than anyone can develop a complex, monolithic, system. While many of the problems authors have identified are still there, the proposals have stood the test of time and it is high time they are adopted by the broader community.
ADVICE TO MID-CAREER RESEARCHERS

We are starting a new series to provide advice to mid-career researchers. There are a number of programs that SIGMOD organizes for researchers at the beginning of their careers (PhD Symposium and the like) and senior people do not (or should not) need much help. There are considerable challenges for those who are about to transition from an early researcher to a more senior role. In academia, these are people who are about to get tenured that comes with starting to think of moving from shorter-term research objectives to longer-term ones. In industrial research, this corresponds to the transition from participating in projects to initiating and leading them. As a community we don’t seem to talk about these challenges much. That is the gap this series attempts to fill. We will get the views of senior researchers from diverse backgrounds and diverse geographies. We will continue as long as we find original advice and the views are not repetitions.

M. Tamer Özsu
University of Waterloo

Mid-Career Researcher, huh? What just Changed?

Laks V.S. Lakshmanan, University of British Columbia, Vancouver, BC, Canada

You just got promoted to Associate Professor. Like most things in life, whether joys or sorrows, the joy of this accomplishment will not last forever. However, that doesn’t mean that you should not look back and reflect on years of hard work and tenacity that you have put in which have earned you this promotion, so first of all, congratulations! Take a moment to savor this accomplishment. On the other hand, it would be a mistake to not ask the question, what just changed about me. Let’s see. You now have tenure and you have been promoted to a senior rank. In one sense, that translates to less stress, but in another, you do have to wonder whether it necessarily does mean less stress. On the flip side, you should also take advantage of the opportunity to ask, what are some new freedoms I have just earned. The stress component is driven by partly knowing, but also partly being unsure of, the expectations from a newly minted Associate Professor. The freedom component stems from knowing that you are now tenured, which hopefully means that you can embark on more daring, high risk projects, even if you don’t feel like you know quite how to negotiate the trade-off between risk and impact.

Research Strategy

Depth over Quantity

One of the questions to consider is whether you should change your research strategy. How critical is it for you to choose bite-sized problems that lend themselves to clean technical solutions with a relatively quick turnaround that is appropriate for the next SIGMOD or VLDB deadline? You have no doubt heard that with becoming a senior researcher comes an expectation that your choice of problems and projects place depth and impact over quantity of publications. But what does it mean concretely, in terms of how to align your choice of projects and problems with the major deadlines that you value? It can be daunting to make a sudden transition where you seemingly don’t care about the next deadline and are instead focusing on a “long term vision”. The truth is you cannot, and in fact, don’t want to, ignore those deadlines, not for too long anyway. What I have found worked well for me is diversification of projects – some focusing on short-term research output and some on longer term goals. You end up sending fewer submissions to immediate deadlines, strategically splitting your time between your immediate and long-term priorities. This may not necessarily work for everyone. For example, you may have a long stream of pending projects from your pre-tenure efforts (and this is a good thing!) and taking them to fruition may be your top priority. In that case, you may want to ease into a diversification-based solution more gradually over a period of time. It is important to be not overly caught up with a “publish a lot or perish” mindset and instead ask what are some exciting things you learned from the new project (which may be completed or in progress) and how that can make a difference to something or some area. Be patient with yourself and have faith in your ability. After all, you didn’t make it this far by accident. It can be helpful to
talk to people – including your senior colleagues in the department, your letter writers, as well as senior researchers you met through conferences and other events. Talking with people with various levels of experience can not only be a source of ideas for how to manage the transition to senior researcher but even open new doors for collaboration.

**How many areas at a time?**

A second, but related question has to do with focus and patience. Achieving impact does not happen quickly and certainly not with one or even a small number of publications, unless you are lucky. It often takes persistent, focused, and sustained effort lasting several years to generate impact and visibility for a theme of research you have been devoting your energies to. Does this mean you want to begin by focusing on exactly one area, for let’s face it, time is finite. And in that finite window, as a senior faculty, your department may have just started to reward you with additional responsibilities, including leadership positions in certain areas of teaching and/or service. In addition, new reviewing and/or editorial responsibilities may have started to land on your lap or soon will. As such, your time is seriously constrained. You have to make a tough decision between choosing one area to focus on and choosing several, possibly at the expense of depth and impact. Or so it seems. For some, working on one area with laser focus for the next $k$ years is perhaps the right choice. I have found once again that diversification with some finetuning worked well for me. Specifically, choosing a small (e.g., 2-3) number of areas, keeping one of them as the “central” area and devoting a majority of my research time to that area with enough left over which can be meaningfully invested in the other areas has been a rewarding experience. For me, when I got tenured and promoted, this meant keeping Query Optimization for Datalog (aka deductive databases) as the central area with some non-trivial investment in other areas including logic programming, object-oriented databases, and managing data with various forms of imperfection such as incompleteness and uncertainty. The initial part of such journey can feel like one is visiting different islands on different days (or weeks, depending on the frequency with which you meet with your graduate students) as it did for me. I have always been fascinated with mathematical logic: logic is what drew me to computer science in the first place. Before long this led me down the path of seeking extensions to Datalog endowed with features capable of addressing the aforementioned functionalities. In sum, during the initial part of the transition to senior researcher, the model that worked well for me was to deliberately divvy up my time between one clear central area and a few other carefully chosen “satellite” areas.

If you are one of those who enjoys diversity, not just in the set of people we interact and transact with, advise, and mentor, but in the set of areas that you focus on, then the next natural question is how do you choose the set? I will not presume to provide you with a prescription but would like to share with you the principles that have guided my choice. There are at least the following pressures acting on this question and not everyone will care about each of these questions equally. Which areas are “hot” in the major venues you care about? Somewhat correlated with that, which areas and skills are valued most by the industry? Which areas really appeal to you on a personal level? The answer to the first question can be helpful in strategizing your choice in such a way that you have a reasonable chance of success at publishing your (and your students’) research in the short term. I am not arguing that you should choose trendy topics. From time to time going against the grain can be rewarding, if somewhat challenging. However, ignoring current trends can be a costly mistake. You certainly want to be well informed of trends in current research and align at least aspects of your research in your chosen research areas with current trends to the extent possible and meaningful. An example can be helpful. In the nineties, as a “young” senior researcher, when I was looking for areas to diversify into, one of the “hot” areas that caught my attention was federated databases (variously referred to as multidatabases, database interoperability, and heterogeneous databases). Given my fascination for logic, I was looking for a set of problems in an area that would lend themselves to elegant and effective solution via logic abstractions. This led me down the path of developing algebraic and logical query languages capable of querying collections of databases with autonomously developed, and therefore heterogeneous schemas. Patience, hard work, wonderful graduate students, and collaboration with awesome colleagues were instrumental in shaping a multi-year journey within this area which led to several impactful publications. I did not always make the right decision.
As a case in point, thanks to a fun collaboration, I was fortunate enough to publish, what in hindsight, is an influential paper on probabilistic data management (ProbView) in 1997. As it turned out, we were well ahead of the pack that rode the subsequent wave of interest in probabilistic data management. There was limited immediate uptake of our research and in fact whenever I explained this research to our colleagues, many reacted it with “Why do we need to worry about probabilistic data? Which industry strength applications warrant it?” I wish I had had some patience and persisted with that area to develop the applications more fully. As such, we ended up working instead on theoretical aspects of query optimization over probabilistic (and deductive) databases, which was not a bad outcome from that investment. However, our contributions could have been broader, richer, and more impactful had I persisted with our original vision for probabilistic data management. An even stronger example of a bad decision where I moved on too quickly for there to be any impact is data management of spreadsheets. We had a paper on declarative querying of spreadsheets based on abstractions we had developed, which was published, again, well before its time came! I wish I had had the clairvoyance to anticipate just how important that topic would become in the future. The bottom line is, do not worry about making such mistakes. You will survive them. I certainly did. There are plenty of opportunities to achieve depth and impact throughout your career.

Supervision and Mentoring

By now, you are a seasoned supervisor of students. Pre-tenure, you may have been mostly focused on accepting graduate students primarily on the basis of excellence and match, perhaps paying little attention to other aspects such as diversity. Diversity not just with respect to underrepresented groups but also differences in abilities. I am of course not arguing against going after excellence, but from time to time, helping students at various levels of accomplishment excel and meet their full potential can be a rewarding experience. To give two examples, I once recruited someone with relatively poor grades in their Bachelors. In fairness, I had one strong recommendation from someone I knew and trusted. That student went on to do an outstanding PhD with several strong publications. In the second example, a student who spoke halting and hard to understand English approached me for doing his Masters thesis. For those who don’t know, in Canada, we have thesis Masters, which is by far the most popular Masters option. I reluctantly agreed. It was the time when I had just moved to UBC and was in the process of building up my group. In hindsight, I almost missed that student. He went on to do an outstanding Masters thesis: not only did he publish a well cited VLDB paper, he was one of those rare cases to be hired directly into an industry research position without a PhD. He has continued to flourish in research. Having made the case for diversity, let me say, don’t go after diversity because you can brag about it or because of political correctness or its appeal to funding agencies. Do it if you are convinced that it’s the right thing to do. For the longest time, I have been actively recruiting and fostering more minorities and women among my advisees. I feel that I (and we) still have a long way to go. One of the joys of doing research necessarily comes from the social process of conducting research with colleagues and students. What better way to make it even more fun by boosting your group’s diversity and inclusiveness?

In addition to just supervising students and postdocs, you will be expected to mentor junior colleagues, both in your department and elsewhere. Try to be giving and generous with your time as much as possible. Mentoring a research colleague (faculty or industry researcher) can be just as rewarding as graduate supervision, but at a whole new level. Rather than look upon this as part of your new “duties”, look at it as an opportunity to give back. Remember the great advisors and mentors who played an influential role in your career and share the knowledge, enriched by your own experience. Don’t forget that mentoring a junior faculty colleague includes not just research but teaching too.

Service

You won’t be surprised to hear that your department, university, and community will have higher expectations of service from you than they did before you got promoted. While most of us may not have a great appetite for administration, you cannot avoid it. Besides, we all have to step up to the plate at some point. Try to find a portfolio that you feel passionate about. I have always deeply cared about faculty recruiting and graduate affairs. Administrative service does take up
time, however the impact you will have had on your chosen portfolio may well make it worthwhile.

Expectations from your research community can take the form of serving in leadership roles in conference PCs, journal editorial boards, organizing new workshops, and launching new initiatives within well-established conferences, to name a few. As a rule, do not accept any invitation or start an initiative for which you are unsure about either your time or expertise, both of which are critical for successful delivery. It’s yet another instance of quality over quantity: fewer well delivered responsibilities go farther than many unremarkable stints.

Closing Remarks

Find ways to expose your graduate students (and postdocs) to new and exciting opportunities, be they competitions, fellowships, or high-risk collaborative projects spanning multiple disciplines. Always look for ways to recognize their talent and accomplishments. Try to move students and junior colleagues up author lists, with a higher priority for students whenever possible and reasonable. Equally important, be there for them when they face setbacks in research or other fronts, even more in your senior avatar than before you got tenured. The value of your emotional support is immeasurable. Groom students and postdocs for writing high quality reviews, report their contributions to PCs, and later recommend them to PCs. Think you want to turn each of them into a version of you. Hopefully better. ☺
ABSTRACT
This report summarizes the outcomes of the first international workshop on Data Systems Education: Bridging Education Practice with Education Research (DataEd ‘22). The workshop was held in conjunction with the SIGMOD ‘22 conference in Philadelphia, USA on June 17, 2022. The aim of the workshop was to provide a dedicated venue for presenting and discussing data management systems education experiences and research by bringing together the database and the computing education research communities to share findings, to cross-pollinate perspectives and methods, and to shed light on opportunities for mutual progress in data systems education. The program featured two keynote talks, ten research paper presentations, a discussion session, and an industry panel discussion. In this report, we present the workshop’s main results, observations, and emerging research directions.

1. INTRODUCTION
Interest in data systems education (DSE) is increasing, especially with the rise in demand for well trained and re-trained data scientists. Data systems education is foundational in programs such as computer science, data science, and information systems and science. A continual focus since the 1970’s in the database research community is the place in curricula and best practices for teaching data systems concepts. There is also a long tradition in both the computing education (CSE) and computer science education (CDEd) community on research into how students learn data systems concepts. For example, a recent research space in the community is understanding the difficulties students face when learning query languages such as SQL and how teachers might improve query language educational practices [12, 17]. Both the DSE and CSE communities, and adjacent communities, e.g., in statistics education, have complementary perspectives and experiences to share with each other, and there is much to be gained by bringing them together: to share findings, to cross-pollinate perspectives and methods, and to shed light on opportunities for mutual progress.

The DataEd workshop was organized as a dedicated venue for the presentation and discussion of data systems education research. DataEd focused on the broad area of data systems education: the teaching and learning of databases, data management, and data systems topics, ranging across the whole field, from classical topics, such as physical design, query optimization, data modeling, data integration, visual analytics, and query languages) to contemporary topics, such as ML & AI for data management systems, data management for ML & AI, very large data science applications/pipelines, and responsible data management.

DataEd ‘22 had a strong focus on encouraging interaction among the participants. It took place as a full day workshop consisting of:

1. a keynote talk Data-Centricity: Rethinking Introductory Computing to Support Data Science by Kathi Fisler (Brown University)
2. a keynote talk Teaching Responsible Data Science by Julia Stoyanovich (NYU Tandon School of Engineering)
3. ten research paper presentations & discussions
4. an industry panel on industry perspectives on education and training for emerging roles in data organized by Juan Sequeda (data.world), with panelists Sarah Krasnik (independent) and Emilie Schario (Amplify)
5. a discussion session on topics prioritized by the attendees, including curriculum placement & content of data systems topics and assessment types

In the following section, we present the themes that emerged from the various workshop activities.

2. WORKSHOP THEMES

2.1 Course and Curriculum Design

The design and curriculum integration of courses related to data, data systems, and data management was one of the main themes that came up in the workshop. Those topics were addressed in Kathi Fisler’s keynote [6], three papers, plus the discussion session on curriculum placement and content of data management topics.

In Data-Centricity: Rethinking Introductory Computing to Support Data Science [6], Kathi Fisler presented a novel introductory computing course which combines data science, data structures, and socially-responsible computing. The course is data-centric, focusing on data science and engineering topics, while covering the necessary content for an introductory computing course. The course is designed to support students across majors, introducing computing concepts via data in familiar formats, such as images and two-dimensional tables, before moving to more advanced data types. It starts with Pyret (a functional programming language with Python-esque syntax which has been developed for education) before introducing Python.

In Piloting Data Engineering at Berkeley [8], Joseph M. Hellerstein and Aditya G. Parameswaran present the Data Engineering course that was designed targeting the Data Science major at Berkeley. The course focuses on fluency of data models and transformation tasks, using SQL as the primary language. Moving from course design to specific aspects, paper Instructional Design for Teaching Relational Query Optimization to Undergraduates by Karen C. Davis [4] presents a module designed to fit within database systems courses to teach query optimization. The module includes both logical optimization concepts and physical optimization ones, and includes quizzes and a project covering computation of database statistics, and performance of selection and join algorithms. A timely intervention is proposed by Alan G. Labouseur in Managing Data... and Covid – An Experience Report [10], which discusses experiences with integrating real-world practice in a data management course by utilizing a Covid-19 screening database for practicing with queries, aggregates, joins, stored procedures, and reports.

The integration of data management topics in the curriculum also came up in the discussion session. Commenting on the placement of the databases or data systems courses, most participants indicated that these are towards the middle or second half of the CS Bachelor’s studies in their institutions. While this may limit reuse and practice opportunities via subsequent projects, other advanced courses such as web systems are sometimes ran in parallel to allow for such opportunities. The core elements of the databases courses were found to be mostly uniform across institutions, with outlying topics being Datalog, database security and SQL injections, especially if they are not covered by other courses in the curriculum. Discussing the concepts and topics that should be considered as core, even if their difficulty is high, participants mentioned declarativeness and conceptual modelling.

2.2 Learning Instruments, Tools, and Practices

A fundamental component to database systems, more specifically DSE, is the necessity to utilise novel learning instruments and tools to aid in the delivery of key material concepts but also better the student experience in data systems and database courses. Over the last decade there have been substantial improvements in the DSE space, much about the curriculum, methods, and tools [9]. It is clear that we need to modernize our courses, as suggested by Kathi Fisler in her keynote [6], but also there is the general need to boost engagement in computer science through novel teaching approaches [2, 7, 13], as well as the shifting change and training needs for students to succeed in industry (see Subsection 2.5).

The research presented at DataEd ‘22 is a direct extension to this progress, as several DSE learning instruments were presented: a gamified experience [14] for students to learn about SQL injection attacks, engaging learners with a graphical user interface [1] to teach data models, and leveraging the community at large for datathons [11] to allow students to practice their data science skills with real use cases and datasets. There is even research being conducted into student reflections and their impact on students’ ability to learn, retain, and apply knowledge after being prompted to reflect on material [15], as well as experience reports and improvements into the curricula [5].

The overarching motive of all the papers in this theme is the need for student engagement. This has been undertaken in many different ways by the authors presenting at the workshop, such as novel tools and creative assignments. However, there are many more avenues of engaging database education research which are open to take on, and the need to continue to collaborate with the community at-large remains important to ensure we stay on track to best educate our students.

2.3 Ethics and Responsibility

Responsible computing, data science, and AI in data education was another major theme of the workshop. The highlight of this discussion was Julia Stoyanovich’s keynote Teaching Responsible Data Science [16]. In
this talk, an overview of the critical and widespread ethical, legal, data quality, fairness, transparency, privacy, and data protection challenges of contemporary data science was presented. In response to these vital challenges, two new courses developed by Julia Stoyanovich at NYU were presented in depth. The first is a technical course for undergraduate and graduate students on “Responsible Data Science”, which introduces these challenges through theory and hands on work, striking a balance between techno-optimism (solutionism) and technocriticism. The second is a public education course “We are AI: Taking Control of Technology”, based on a peer-learning format for a non-technical general audience. A special feature of the course is their open format, freely available online to the public, with ample rich materials to be adopted and extended in a wide variety of settings

2.4 Formative and Summative Assessment

The final theme that came up in the workshop was that of assessment approaches for data systems education. The main three avenues for this theme were two papers [3, 18], and the discussion session on teaching.

In Analyzing Student SQL Solutions via Hierarchical Clustering and Sequence Alignment Scores [18], the authors aim to explore the problem-solving behavior of their students. They do this by 1) calculating the alignment between the nth solution and the final solution and 2) clustering solutions to determine different approaches used by students to solve the problem. Their system offers these metrics to the instructors of the course to visualize their students’ learning progress. The authors hope that this can help the instructors identify SQL concepts that warrant more in-depth instruction.

In Collaborative Learning in an Introductory Database Course: A Study with Think-Pair-Share and Team Peer Review [3], the authors aim to evaluate whether the application of collaborative learning techniques can be beneficial in a data systems course. They selected Think-Pair-Share to test in lectures and lab sessions, and team peer-review for projects. Participation in these collaborative activities was optional, which meant that the authors could compare the course results of both a test and control group. They found that students who passed the course and had participated in collaborative activities had more homogeneous results than students who had passed the course but not participated. Furthermore, the students rated usefulness of the activities on a scale from 1 (definitely useless) to 4 (definitely useful). All three activities were, on average, ranked higher than 3, showing that students appreciated the activities. The authors hypothesize based on the results, that collaborative activities (and the summative assessment within it) lead to more balanced learning efforts compared to self-regulated learning.

In the discussion session, the two main points of discussion were assessment types and assessment creation.

Typical assessments in data systems include the creation of Entity Relationship Diagrams from a textual description and writing queries (in various languages) based on set requirements. However, as one participant noted, students make lots of mistakes in SQL query formulation. As such, they decided to first include it in a small project such that the student can be scaffolded into the more standard exercises mentioned above. Another supporting mechanism is to use group work, such that students can discuss what parts of their answers might be (in)correct.

Finally, assessment creation is seen as a challenge. Coming up with new questions on interesting topics is a drain on creative resources of lecturers and teaching assistants. However, some of our participants came up with ingenious ways of finding subjects for their questions. Some ideas include: asking your kid(s) to come up with a topic, choosing a random Wikipedia category, or using pop culture references such as Disney and Marvel. One way in which this issue could be abated is to create a shared repository of questions, where many teachers add theirs, such that we end up with a resource of thousands of questions. However, students will be able to find it, which might be a problem in case of online exams.

Overall, it seems that most innovations in data systems assessment are based on trial-and-error. From the teacher perspective, we are looking for efficient creation of assignments. Here the findings from Yang et al. [18] may help to identify the (SQL specific) topics that students need more practice in. On the student side, we are looking for in-depth understanding. The findings from Catania et al. [3] suggest that applying collaborative active learning techniques may help students learn in a more balanced way.

2.5 Industry Perspectives on Data Management Knowledge and Skills

The workshop closed with a panel discussion on industry perspectives on education and training for emerg-
ing roles in data, organized by Juan Sequeda (Principal Scientist, data.world), with panelists Sarah Krasnik (Data Engineering and Analytics Advisor, independent) and Emilie Schario (Data Strategist in Residence, Amplify). George Fletcher moderated the discussion.

Several themes arose during the opening position presentations by the panelists and the ensuing discussion with the workshop attendees.

A central theme was gaps between practice and university curricula. Example topics in this gap highlighted were: relatively little coverage of topics in dynamic and streaming data management; principled methods and frameworks for choosing which solutions and technologies to use in a given practical data engineering task; data integration solutions; (re)aligning emerging and established data roles (such as data engineer, analytics engineer, data analyst, machine learning engineer) with university curricula; and, mapping between academic education and industry norms around data workflows and the “modern” data stack. A second related theme was that of balancing generality and ideas which transcend current practice, on the one hand, and mapping these general concepts and perspectives to current practice, on the other hand. What are the perennial ideas with practical impact which are currently poorly covered in curricula? A final theme which arose was that of the role of education and academic curricula with respect to professional and non-technical skills. What should be covered in education? Which topics are better learned outside the classroom? How do we bridge training and practice? This lively discussion was a perfect way to close a very productive and stimulating day.

3. CONCLUSIONS AND EMERGING RESEARCH DIRECTIONS

We identified five main themes which arose during the workshop: course and curriculum design, learning instruments, ethics and responsibility, assessment, and industry requirements. Each of the subsections describing these themes can be seen as illustration of an upcoming research direction within data science education. Clearly, much more work is needed in each of these areas. We hope that DataEd will continue to inspire research efforts on data systems education, in both the aforementioned themes, as well as in new directions.

We aim to continue DataEd as a workshop under SIGMOD, with an ultimate goal to create a space in the data management research community for both computing and computer science education research. Furthermore, under the DataEdInitiative umbrella, we aim to bridge the gap between CSE/CSED and Data Systems researchers with “sister” activities to DataEd. As such, we will be organizing activities in the CSED community in the future, with one possible avenue being a working group (for instance, those that occur at ITiCSE4).

Acknowledgement

The workshop’s advisory board members supported setting the scope of the workshop and provided valuable feedback at all stages of its preparation. We also appreciate the efforts of DataEd’s program committee in reviewing the submissions. We would like to thank all of them for their valuable support and encouragement. In addition, we would like to thank all participants for their valuable contributions, discussions, and feedback to DataEd ‘22.

4. REFERENCES


3. https://dataedinitiative.github.io


Motivation: Data science is increasingly collaborative. On the one hand, results need to be distributed, e.g., as interactive visualizations. On the other, collaboration in the data development process improves quality and timeliness. This can take many forms: partitioning a problem and working on aspects in parallel, exploring different solutions or reviewing someone else’s work.

While the benefits of tool support for collaborative software development are established, the existing tools do not meet the demands of data scientists. They struggle to manage complex data processing pipelines without a clear notion of correctness on large and dirty datasets. While classic data management systems have limited support for “collaboration” in the form of concurrent clients, they present the same view of the data to all users (most of the time). Data science has different requirements: first, users may operate “offline,” i.e., on a local copy of the dataset. Second, even when online, developing a data processing pipeline on a dataset concurrently modified by others hurts productivity.

State of the Art projects like OrpheusDB [13], Dolt [2], DVC [1] and Pachyderm [9] have recognized the need for versioning in data science and provide the infrastructure to store multiple versions of a dataset. Where they fall short is the handling of branching versions, in particular resolving conflicts during merges. To be practical, a collaborative data science system must make trivial merges fast, automate as many merges as possible and provide tool support when merges need manual intervention. Existing systems do not fulfill these requirements: they have no notion of concurrent versions, present diffs at tuple-granularity and require mostly manual conflict-resolution. Sophisticated merge strategies require operational information, e.g., in the form of provenance [6]. Unfortunately, tuple-granularity provenance does not scale to the size of contemporary datasets, restricting provenance systems to coarse-grained information [8, 4].

While merging code without considering its effect on data is an option, it is error-prone. Thus, most merge conflicts need to be resolved manually [5]. In software development, manual conflict resolution regularly involves careful analysis, some guesswork and, crucially, good test coverage to ensure correctness [11]. Resolving conflicts in this fashion is infeasible for complex data-intensive pipelines with non-obvious semantics.

The problem is that existing tools separate code and data: merging data without the code is tedious, while merging code without data is risky. What is required is an approach that considers code and data when merging.

My group at Imperial College has started working on BOSS (Bulk-Oriented Symbol Store), a system that manages code and data in a single format — a concept known as Homoiconicity (popularized by Lisp [7]). Lisp-style Homoiconicity has two components: a structured view of the program code (nested lists of instructions in Lisp) and a means to manipulate it using compile-time functions (called macros in Lisp). This approach is useful, e.g., to implement domain-specific languages [3, 12]. In BOSS, we turn this concept around: the system allows storing value-producing “Expressions” (pieces of code in a Lisp-like language) in the database and evaluates them at query time. This idea generalizes User-Defined Functions, and also has many potential applications for data integration, cleaning, model management, and many more. For now, we focus on collaborative data science: BOSS stores the “operation graph” (depending on research field, one might call it the provenance or the version graph) of every data item directly in the tuple, enabling complex merges.

Three key challenges arise. For fully automatic merges, scalability is an unsolved problem. State-of-the-art tools (using either tuple-granularity provenance or the transaction log) can easily take hours or run out of memory, even at a moderate scale. To limit the effort of semi-automatic merging, users need diff/merge tools that operate in bulk, e.g., by exploiting provenance like “outliers, i.e., prices above £99 were clipped” or “prices were normalized to adjust for inflation” and letting the user select their order. Finally, a collaborative data science system needs to account for data manipulation by third-party libraries (Python, R, Tensorflow, etc.). Extracting provenance information from those will require whole-program analysis.
REFERENCES

Chenggang Wu Speaks Out on his ACM SIGMOD Jim Gray Dissertation Award, Rejection, Believing in Your Work, and More

Marianne Winslett and Vanessa Braganholo

Welcome to this installment of ACM SIGMOD Record’s series of interviews with distinguished members of the database community. I’m Marianne Winslett, and today we are on Zoom with Chenggang Wu, co-founder and CTO of Aqueduct. Chenggang received the 2022 ACM SIGMOD Jim Gray Dissertation Award for his thesis entitled The Design of Any-scale Serverless Infrastructure with Rich Consistency Guarantees. His PhD is from UC Berkeley. So, Chenggang, welcome!
Thank you for hosting me. It was a great honor to receive this award.

Our runners-up for the award this year were PingCheng Ruan (National University of Singapore) and Kexin Rong (Stanford University).

So, Chenggang, what is the thesis of your thesis?

My dissertation was primarily centered around the topic of serverless. I actually can’t believe it’s already been two years since I submitted my dissertation. Although these days serverless computing has become a widely used technique, back then, in 2016, it was still a fairly newish concept and became a hot topic in the research community around 2018. So, just to give a little bit of background, serverless is a software design pattern whose main advantage is to offer a higher level of abstraction to program the cloud.

Basically, in the pre-serverless world, to run an application in the cloud, we first need to provision a virtual machine, specify some resource requirements, like how many CPUs, GPUS, RAM, and disc you need and the operating system you want to use, launch that VM, and copy your code into the VM and run it. For seasoned backend engineers, this might be fine, but for folks without deep systems expertise, this is actually a tall order. But, with serverless computing, programmers can now just simply upload the code to the cloud, issue a request to run the code and get the results back, without having to worry about any of these that I mentioned beforehand, which is very convenient.

Can you tell me more about the challenges that you had to tackle?

My first project, called Anna, was centered around exploring the tradeoff between scalability and consistency. The main motivation behind Anna is that these days there are a bunch of highly scalable distributed computing infrastructures being built. But the interesting thing is that even within a single machine, we actually have access to very rich, very beefy compute resources, like lots of CPU cores and a bunch of RAM. And we noticed that people actually aren’t making the most use out of these, even for a single instance of compute node.

So, the research question becomes how do we design a unified architecture that scales well in the distributed setting, of course, but also takes the maximum benefit out of a single core. This architecture can then make full use of every CPU core and available RAM to deliver the best performance, even within a single-node system. The solution that we landed on is something called a “Coordination Free Execution Model.” Because we noticed the bottleneck that inhibits the scalability usually lies in the coordination between different compute threads when they access shared state.

So, the coordination free model basically means every thread has access to its own local memory without communicating and just does its own work. So, in that way, everybody can proceed in parallel so that it minimizes the coordination between threads. But of course, this is the ideal world, because if everybody just keeps doing their own thing, then eventually, they have to communicate and exchange information to offer a consistent view of the world.

The corollary challenge that emerges from that is how do we design a suitable consistency mechanism such that although different threads are accessing its local copy, eventually, they will find some way to reach an agreement to offer the application a consistent view of the world. So, we introduced a technology called “Lattice-based Conflict Resolution Strategy.” We found that carefully composing these different lattices allows us to guarantee that, although everybody may be doing their own things, and although these messages may arrive at different threads in different orders, eventually they will reach a consensus. So, it’s a decent design that offers maximum scalability, both within a single node and across multiple nodes while achieving different levels of consistency such that an application can safely run.

[...] how do we design a unified architecture that scales well in the distributed setting [...], but also takes the maximum benefit out of a single core [?]
That sounds like a great start to your thesis. What was the topic of the rest of it?

So, basically, in the first chapter of my thesis, Anna, we designed a system architecture that performed well at each scale point, be it the single node context or the geodistributed deployment. But the key promise that serverless computing wanted to deliver is “pay as you go”, which means that it must be elastic. As your workload requirement goes up, the system needs to be able to detect your workload shift and then dynamically add nodes to satisfy your compute demand. And also, if your workload is going to shrink, it should be able to reduce the resource allocation to save costs for you.

The second chapter of Anna is to take this architecture that’s performing well at each scale and then implement our own scaling mechanism so that we can dynamically adapt to these workload shifts and adjust the resource allocation accordingly. So, fundamentally, it’s exploring the design of the underneath autoscaling mechanism, and exploring the tradeoff between performance and cost efficiency.

The first two chapters focus on the storage side of things, which actually lays out a very good foundation for the compute side of things because you can imagine, during the compute, inevitably, each compute agent is going to access all the shared state that’s maintained in the underlying storage system. So, the third chapter is basically bringing in all of these core design principles: coordination free execution model, lattice-based conflict resolution scheme, and autoscaling, and applying these principles to the compute layer.

So, now, there are two layers, there’s this compute layer, and there’s a storage layer, and we may have workloads that exert different amount of tension to each layer: Maybe there’s a workload that requires a tremendous amount of compute but only little storage. In this case, the compute layer can be very beefy, and the storage layer could be pretty lean. And you can imagine a workflow that’s the other way around. So, this advocates for the design of resource disaggregation – allowing the two layers to scale independently, which is very economical.

But then the challenge is that as we progress through the compute, at some point, it needs to access the storage; it needs to issue a network request to the storage tier to request the data, which is then sent back to the compute layer. This process can introduce high network latency, which is a significant performance challenge. So, the third chapter is more on exploring this concept of logical disaggregation with physical colocation (LDPC). Logical disaggregation allows the compute tier and the storage tier to scale independently. Physical colocation means that in our implementation, we can carefully cache some of the data from the storage tier up one level to the compute layer.

In most of the cases, because we are accessing the cache from the compute layer, it minimizes the latency drastically, and inside each cache, we extend Anna’s design principle, so it still offers a suitable level of consistency while eliminating the network latency. This way, the LDPC design allows us to achieve the best of both worlds (performance and consistency). This summarizes the three chapters of my thesis.

That sounds very appropriate for a startup.

Yeah. Exactly. So, that’s why after my PhD research, both my colleague (Vikram Sreekanti) and I, and both of our advisors, Joe Hellerstein and Joseph Gonzalez, were very interested in packaging our research into something that the industry can use. The company we founded, Aqueduct, is doing exactly this. I mentioned before that serverless computing is very suitable for folks without deep systems expertise, and that’s why our current target audience is more on the data teams, especially data science folks.

These groups of people are domain experts in various fields, like biology, some scientific computing fields, and the financial industry. They have deep domain knowledge but only have some basic programming knowledge in Python and SQL, and they don’t have deep expertise on how Kubernetes or Docker containerization works, so they’re the perfect audience. So, on the API level, we want to offer them a way to easily program their workflows (which involves some machine learning to predict churns, the weather, or financial trends), and they can construct the workflow inside their familiar Python or Jupyter notebook environment. Then, underneath the hood, when they submit this Python workflow definition, we package that into Docker containers and do all of these performance optimizations outlined in the thesis. So, from our user’s perspective, they get both ease of use and peace of mind in the sense that all of the scalability, consistency, and fault tolerance aspects are automated and abstracted away from them. That’s the ultimate promise that we wanted to deliver to industry folks.

Do you have any words of advice for today’s graduate students? Things that you wish you had known when you were a new PhD student?

I think the most important one is to be a believer of your
own research and stick with your belief. There are two parts to this. The first part is that you have to believe in your own research, which means you need to know that what you’re working on is important and you need to be the No.1 fan of your own research. Actually, I’ve been through this during my first year. When I was an undergrad, I was working on something that was unrelated to my dissertation field; I was working on interactive data visualization, and I continued this line of research during the first year of my graduate study.

But then, along the way, I realized that’s just not really where my core passion was, and I was interested in more “hardcore” systems-oriented topics. So that’s why I made a switch from interactive visualization to distributed storage and distributed systems area, and I found myself to be very much enjoying that. I feel this is the prerequisite of me being a believer of my own research.

The second part is to stick with your belief because I found that, along the way, not everybody is going to be believing in your research. Especially when you submit papers, you’re probably going to encounter some rejections. In my case, I think my first paper on Anna only got published when I was a third-year PhD student. That paper actually got rejected three times consecutively. So, that was a “dark age” for me. It was definitely a frustrating experience, but I don’t actually think I was depressed by that mainly because I was confident that I was doing good work, and there’s an old saying, “good work eventually gets published.” So, the fundamental belief that I am doing great work, that it will get publicized and recognized by people, is ultimately the source of power that’s driving me through these dark ages.

The second lesson is that I found collaborating with your peers is usually a little bit more enjoyable than working alone. When I first joined Berkeley, it just happened that all of my advisors’ grad students either already graduated or were about to graduate, so I had to explore the research on my own. That was fine, but later on, I found my peer collaborator, Vikram Sreekanti (actually, now we’re running the company together). Once the two of us started working together, there were a few things that I discovered. I think, first of all, it’s just more fun to work with people. You have other folks you can talk to, either research-wise or just complaining about things together is always better than dealing with everything by yourself. Also, although I was communicating very frequently with zero issues with my advisors, they tend to give advice at a higher level: the idea generation level or the design level. But regarding the implementation details, it’s still very nice to get some feedback from your peers, who are working together with you and know the details of your codebase.

Also, I think collaborating with folks, in the end, leads to a net gain on productivity. Usually, I was leading a project that my collaborator was co-leading, and I was also participating in the project that he was leading, and I was the co-lead of that project. So, in the end, it was a net gain for both our individual growth and the team’s growth.

Great advice. Thank you very much for talking to me today.

No problem. It was my great pleasure, Marianne.
The World of Graph Databases from An Industry Perspective

Yuanyuan Tian
Gray Systems Lab, Microsoft
yuanyuantian@microsoft.com

ABSTRACT
Rapidly growing social networks and other graph data have created a high demand for graph technologies in the market. A plethora of graph databases, systems, and solutions have emerged, as a result. On the other hand, graph has long been a well studied area in the database research community. Despite the numerous surveys on various graph research topics, there is a lack of survey on graph technologies from an industry perspective. The purpose of this paper is to provide the research community with an industrial perspective on the graph database landscape, so that graph researcher can better understand the industry trend and the challenges that the industry is facing, and work on solutions to help address these problems.

1. INTRODUCTION
Rapidly growing social networks and other graph data have created a high demand for graph technologies. No wonder Gartner ranked graph technologies among the top 10 data and analytics trends in 2021 [40]. According to Gartner, up to 50% of their client inquiries around the topic of AI involve a discussion about the use of graph technology [40], and by 2025, graph technologies will be used in 80% of data and analytics innovations [12]. Inkwood Research projected that the global market for graph databases will grow at 21.7% from 2019 to 2027, and reach $4.6 billion by 2027 [30]. The industry has responded to the high demand of graph technologies with a boom of graph companies, systems, and solutions, as depicted in [45]. The venture capital investment has also been very active in graphs in the last couple of years. Not only new startups, like Katana graph ($28.5 million in Series A), but even seasoned graph database companies, like Neo4j and TigerGraph, received a lot of funding (Neo4j raised $325 million in Series F and TigerGraph received $105 million in Series C).

On the research side, graph has long been a well studied area in the database research community. In his VLDB 2019 keynote [58], Professor Özsu provided a good summary of the various subareas of graph research. Professor Boncz delivered a keynote in EDBT 2022 about the state of graph database systems [17], touching on graph models, graph languages, the common pitfalls in designing graph database systems, and the blueprint of a competent graph database system. Professor Fan's keynote in VLDB 2022 [23] discussed the challenges and progress made on processing big graphs, including parallel scalability, incremental computation, and semantic joins between relations and graphs. There have also been numerous research surveys on topics such as graph database models [16], graph query languages [15], graph stream algorithms [37], knowledge graphs [29], distributed graph pattern matching [18], large-scale graph processing [57], etc. Back in 2014, Professor Deshpande blogged his views on graph data management and pointed out some open problems [13]. The VLDB 2018 best paper [42] and its extension [43] conducted a comprehensive user survey about how graphs are used in practice, and revealed many interesting insights, including the ubiquity of large graphs, variety of entities represented by graphs, the scalability challenges faced by many graph systems, the importance of visualization tools, and the continued popularity of RDBMSs in managing and processing graphs. The recent community publication [44] painted a picture of what the next-decade big-graph processing systems look like in the aspects of abstractions, ecosystems, and performance. However, none of the above work discussed in detail the solution space or architecture of existing graph databases in the market. Despite the recent surge in graph technology innovation in the industry, there is still a lack of survey on graph technologies from an industry perspective.

The database research community, as a whole, has been having very strong ties to and impact on the industry, witnessed by the fleet of database products (e.g. PostgreSQL and Flink) and startups (e.g. Vertica and Databricks) originated from research. In the area of graph databases, the research community has also influenced heavily on graph benchmarking [10] and graph query languages [2]. But still, some of the major problems that the graph database industry cares about are not well known to the research community. The purpose of this paper is to provide the research community with an industrial perspective on the graph database landscape, in the hope of helping researchers better understand the current industry status quo and the challenges they are facing, and ultimately increasing the impact of the graph database research community.

2. USE CASES AND WORKLOADS
In terms of customer use cases, graph databases have been used in many vertical industries, including finance,
insurance, healthcare, retail, energy, power, manufacturing, government, marketing, supply chain, transportation, etc. This diverse and wide applicability of graphs in many domains is also observed in [42]. Some of the concrete use cases of graph databases have been provided in [51, 39, 48, 46]. Perhaps, the most common example of graph database usage is fraud detection. For example, [47] demonstrated a detailed example scenario of traversing through a graph containing insurance claims information and patients medical records to detect fraudulent claims.

Similar to the different types of workloads in relational databases, there are also two different types of graph database workloads. The first type focuses on low-latency graph traversal and pattern matching. They are often called graph queries. These queries only touch small local regions of a graph, for example, finding 2-hop neighbors of a vertex, or the shortest path between two vertices. Due to the low-latency requirement and the interactive nature of the graph queries, people also call them graph OLTP. Graph OLTP is often used in exploratory analysis and case studies. The second type of graph workload is graph algorithms, which usually involve iterative, long running processing on the entire graph. Good examples are Pagerank and community detection algorithms. Graph algorithms are often used for BI-ish applications. Because of this reason, people also call them graph OLAP. Recently, a new trend emerges that combines graph and machine learning together, called graph ML. For example, graph embedding or vertex embedding are used to transform graph structures into vector space which are then included as features for ML model training. Graph neural network (GNN) is another example of graph ML. Quite often graph ML is lumped together with the graph OLAP workload.

3. GRAPH MODELS

![RDF Model](image1)

*(a) RDF Model*

![Property Graph Model](image2)

*(b) Property Graph Model*

Figure 1: RDF and property graph models

Whenever talking about a graph database, we need to first talk about the graph model(s) that it supports. The two prominent graph models supported by most commercial graph databases are the RDF model and the property graph model.

**RDF Model.** RDF is among the suite of W3C standards to support Linked Data and Knowledge Graphs [52]. An RDF graph is a directed edge-labeled graph, represented by the subject-predicate-object triples. Figure 1(a) shows an example graph represented in the RDF model. This graph captures the following information: A patient, named Alice Brown, with patient ID 19806, is diagnosed with Type 2 Diabetes which has disease ID 6472345 on March 24, 2020; and Type 2 Diabetes is sub-type of Diabetes which has disease ID 6472345. For example, in the (Patient 1) − [hasName] → (Alice Brown) triple, Patient 1 is the subject, hasName is the predicate, and Alice Brown is the object. The RDF model is often used in knowledge representation and inference as well as semantic web applications. For example, DBpedia [21] and YAGO [56] both utilize RDF to represent their knowledge graphs and support queries on the knowledge bases using SPARQL [53].

**Property Graph Model.** In comparison, a property graph is a direct graph where each vertex and edge can have arbitrary number of properties. Vertices/edges can also be tagged with labels to distinguish the different types of objects/relationships in the graph. Figure 1(b) shows how the same information captured in the RDF graph in Figure 1(a) is represented in the property graph model. Here, instead of representing the ID and the name of a patient or disease as separate nodes, the property graph model can fold them in as the properties of the patient and the disease nodes. Similarly, the diagnosis time can be represented as a property of the diagnosedWith edge, eliminating the need to create a separate diagnosis node and its connecting edges to the patient and disease nodes. In general, the property graph model can capture the same information with fewer nodes and edges than the RDF model, as illustrated by this example. This is because a piece of information can only be represented either as a node or an edge in the RDF model, whereas the property graph model can also define it as an attribute of an existing node or edge, thus leading to fewer number of nodes and edges in the graph. The property graph model is often used for applications that require graph traversal, pattern matching, path and graph analysis.

Today, although both models are supported in the graph database industry, as we will show in Section 5, the property graph model has overwhelming endorsement, despite the fact that RDF is a much older model. All the major offerings we surveyed in the paper support the property graph model, and two of them also support the RDF model. In [27], Hartig proposed a formal transformations between the RDF and property graph models, in the hope to reconcile both models.

4. GRAPH QUERY LANGUAGES

On the graph OLTP side, for RDF graphs, there is the standard SPARQL query language [53]. For property graphs, there are many languages being used and proposed, but no clear winner. One of the top contenders is Tinkerpop Gremlin [1] which is supported
by around 30 graph vendors today, also probably the
most widely used graph query language today. Another
strong contender is openCypher [8]. Cypher [24] was
originally Neo4j’s proprietary declarative graph query
language, and it was open-sourced in 2015. Around 10
graph vendors support openCypher. Besides these two
more widely adopted languages, many vendors proposed
their own graph query languages. Oracle proposed a
declarative language based on SQL, called PGQL [9].
GSQL [4] is the SQL-like graph query language adopted
by TigerGraph. Microsoft SQL Graph extended SQL
with MATCH clause for graph pattern matching [38].
The LDBC [11] Graph Query Language Task Force (with
members from both academia and industry) has pro-
posed G-Core [14]. In an attempt to reduce the chaos
on graph query languages, in 2019, the Joint Technical
Committee 1 of ISO/IEC, approved a project to create
a standard graph query language, called GQL [2]. This
effort is also complemented by another project that ex-
tends SQL with graph view definition and graph query
constructs, called SQL/PGQ. GQL and SQL/PGQ share
a common declarative graph pattern matching language
component. This common component integrates ideas
from openCypher, Oracle’s PGQL, TigerGraph’s GSQL,
and LDBC G-CORE. The standardization effort on GQL
and SQL/PGQ has strong participation from academia,
and it is one of the areas where the graph research com-
pany has heavily impacted the graph industry. How-
ever, given the current state of the graph query lan-
guages, even when GQL and SQL/PGQ standards are
published, it is going to take time for the vendors to
adopt it, since a large number of graph applications are
already written in these existing languages. It will still
take many years for the standardization to settle down.

In terms of language properties, Gremlin is more of
an imperative graph traversal language (although the
recent version of Gremlin also has some declarative lan-
guage features), while the others are declarative. As a
result, Gremlin is relatively more low-level and less user-
friendly. But in terms of expressiveness, Gremlin is Tur-
ing Complete [41], while most of the declarative counter-
parts, including openCypher, are not. This means there
are graph algorithms or operations not expressible in
these non-Turing-Complete languages. Out of all the
declarative languages, TigerGraph’s GSQL is the only
one that is Turing Complete [55].

On graph OLAP side, there is also no standard lan-
guage or API, but most vendors support a variation of
the Pregel-like API [36]. Like for ML, a library of build-
in graph algorithms are more useful to users, so the lack
of standard is not so much an issue for graph OLAP.

5. GRAPH DATABASE OFFERINGS

The graph database area is a very crowded space in
the industry, with new projects and startups popping
out every moment. It is impossible to enumerate all the
current graph database offerings. So, this section only
highlights some of the major offerings in the three cat-
egories: graph-database-only vendors, data companies
with graph support, and enterprise cloud vendors with
built-in graph database support. The different features
of their graph products are summarized in Figure 2. In
Section 6, we will discuss the different architecture sol-
lutions adopted by the various vendors.

In the pure-play space, Neo4j and TigerGraph are the
two strongest contenders. They provide solutions both
on premise and on all major clouds, AWS, Azure, and
GCP. They have good support for both graph OLTP
and OLAP workloads. The pure players have also per-
fected the art of visualization and tooling, as well as the
support of a large number of built-in graph algorithms.

DataStax and Databricks are two data companies with
a wide range of data portfolio. The graph component is
also integrated well with the rest of the system compo-
ents. For example, DataStax Enterprise Graph (DSG)
is built on top of the DataStax’s main NoSQL data
engine Cassandra. And for Databricks graph support,
GraphX is built on top of Spark’s RDD, and Graph-
Frames is based on DataFrames. Since both companies
aim at more general data systems, their support on
graphs is not as comprehensive as the graph-only ven-
dors. DataStax’s support on graph OLAP is very rudim-
entary (only relying on SparkGraphComputer API in
Gremlin with just 3 built-in graph algorithms). Databricks’
graph OLTP support comes only from the simple mo-
tif finding support in GraphFrames. This support is not
only limited by the very simple motif finding DSL, but
also unlikely to perform well, since the graph OLTP
query processing utilizes DataFrames underneath, which
is originally designed for analytics purpose.

The last category of graph database vendors are big
cloud companies, including Amazon, Microsoft, Oracle,
and IBM. They all provide a large number of data ser-
ices on their cloud platform, and the built-in graph
database service is one of them. Microsoft, Oracle, and
IBM are previously big relational database shops, so it
is not surprising that their graph database solutions
are based on their relational databases: Microsoft SQL
Graph on top of SQL Server (on premise) and Azure
SQL Database (on cloud), Oracle Spatial and Graph on
top of Oracle databases (on premise and cloud), and
IBM Db2 Graph on top of Db2 (on premise and Cloud
Pak for Data). Microsoft also provides another graph
database solution, Cosmos DB Graph, built on top of the
NoSQL database Azure Cosmos DB. Amazon, on the
other hand, builds Neptune on the same back end
storage as other AWS platforms, such as Aurora and Dyn-
namoDB. Amazon, since a pure cloud company, doesn’t
provide an on-premise graph database solution. Most
of the graph databases from this category focus on the
graph OLTP workload, except that Oracle Spatial and
Graph has a very good graph OLAP support with a
large number of built-in algorithms.

Now, let’s look at the different dimensions of the ta-
ble shown in Figure 2. Most vendors support both on-
premise and cloud deployment, except that Amazon Ne-
puncture and Microsoft Cosmos DB Graph are on cloud
only. In terms of graph models, all the vendors sup-
port the property graph model. Amazon Neptune along
with Oracle Spatial and Graph additionally support the
RDF model. For graph OLTP workload, the languages
that different vendors use reflect the language chaos
discussed in Section 4, but Gremlin appears to be the most supported. Due to the exploratory nature of graph OLTP workload, visualization is especially important for customers. Most graph vendors do provide visualization support. Compared to relational databases, transaction support has been a sore spot for graph databases. An update to a single node often affects its edges and connected nodes, e.g., deleting a node requires the deletion of all the edges it is connected to. So, a transaction is often more complex in a graph database, especially in a distributed setting. Some graph databases manage to provide full ACID support, but others either have no support or week support for transactions. Compared to graph OLTP, the graph OLAP support is relatively weak in general, but TigerGraph, Neo4j, and Oracle stand out due to their large number of built-in algorithms. The VLDB survey [42] has observed the ubiquity of large graphs with over a billion edges and pointed out that scalability is a challenge that many users face. As a result, major graph vendors strive to address this challenge. All the graph database solutions can scale up nicely to a certain extent, which can satisfy a lot of customers, and most also provide scale-out solutions for those customers working on huge graphs that cannot fit in a single node. As rightly pointed out by [23], while distributed parallelization can handle larger graphs, it does not always provide desirable performance. Due to the connected nature of graphs, it is almost impossible to achieve access locality in a distributed setting. As a result, distributed graph computation often accesses many partitions of a graph, which incurs a lot of communication cost. If a large graph can fit in a single node, the scale-up solution might provide better performance than the scale-out version of the same system. As shown in [46], a single node system on a decent machine configuration can comfortably handle large graphs with billions of edges. However, converging with [42], efficiently querying and processing large-scale graphs (way beyond billions of edges) remains a challenge.

6. GRAPH DATABASE SOLUTION SPACE

6.1 Native vs Hybrid Graph Databases

One way to categorize the solution space is native graph databases vs hybrid graph databases, as shown in Figure 3. As the name suggested, native graph databases are built with specialized query and storage engines from scratch just for graphs. Neo4j and Tigergraph are two prime examples of native graph databases. This type of graph databases are highly optimized to the supported graph workloads. But the drawback is the high engineering cost, since they have to reinvent the wheels for the support of transactions, access control, scalability, high availability (HA), disaster recovery (DR) and so on. In contrast, a hybrid graph database has a specialized graph query engine, but resorts to an existing data store, a key value store, or a document store. As shown in Figure 3, more graph databases fall into this camp. Since a hybrid graph database delegates its storage engine to an existing data store, it has much faster development time. In addition, it can also get many things for free from the backend store, such as transaction support, access control, scalability, HA and DR, etc. But the potential downside is that the per-
formance of a hybrid graph database may not match a highly optimized native graph database. Of course, the performance of an individual graph database also highly depends on the implementation details.

### 6.2 Graph-Only vs Converged Databases

Another way to categorize the solution space is graph-only databases vs converged databases, or also called multi-model databases. As shown in Figure 3, all native graph databases are graph-only databases, and most hybrid graph databases are converged databases, but some are graph-only. Graph-only databases support graph workload only. This fact can also be a fundamental limitation of these databases. In fact, Neo4j and TigerGraph both have dedicated chapters on data import and export in their user manuals. In contrast, a converged database or a multi-model database supports poly query languages/APIs on the shared data. This fact can also be an advantage of the converged database architecture. We elaborate on some of the advantages below.

Fundamentally, the converged database solution solves the fragmented database problem. Real applications seldom have only homogenous workload that just contains graph analysis. Often graph analysis is mixed with SQL, ML, and other analytics. In order to support the heterogenous workload, developers have to move data around different systems, in the fragmented database world. By supporting multiple languages/APIs on the shared data, the converged database solution essentially allows users to view the data in the way that is needed! SQL, graph, and ML can work on the same data synergistically. There is also no data transfer or transformation needed. This is a huge saving. Even though native graph databases are highly optimized for graph workloads, but if we consider the performance of the end-to-end pipeline of a heterogenous workload, the converged graph databases may actually have an edge.

Moreover, some converged database solutions, such as IBM Db2 Graph, even allow graph queries to be performed on the original data in the operational databases. The extra advantage it brings is to have the graph query capability without disturbing the large number of existing relational applications and that transaction updates to the operational data can be visible to the graph queries in real time.

Other advantages of converged database solution come from the existing backend data store, for example transaction support, access control, compliance to audits and regulations, temporal support, scalability support, HA and DR support, etc.

As discussed above, each type of graph solution has its own pros and cons. Choosing the right architecture depends highly on the actual application requirements, such as whether the workload is graph-only or heterogeneous, the latency and throughput requirement, the frequency of updates, the recency requirement of the results, etc.

### 7. GRAPH BENCHMARKS

Benchmarks are very important in evaluating different database systems. Since graph databases are a relatively new area compared to relational databases, there is no standard benchmarks, like TPCC, TPCH, and TPCDS, yet for graph databases. There has been some community efforts in establishing graph benchmarks, such as the Linked Data Benchmark Council (LDBC) [11] benchmarks, Linkbench from Facebook [5], Graph500 [3], HPC Scalable Graph Analysis Benchmark [26], and Open Graph Benchmark [7] for graph ML specifically. Notably, LDBC benchmarks are the most widely adopted and hence the closest to a standard. It is also the most comprehensive benchmark and currently has three benchmark suites. The LDBC Social Network Benchmark (LDBC-SNB) contains tests for interactive workload, which corresponds to graph OLTP workload, and business intelligent workload, which is more relational (aggregation and join heavy) queries on graph data and is still under development. The LDBC Graphalytics benchmark targets graph OLAP workload. And the LDBC Semantic Publishing Benchmark (SPB) is an RDF-based benchmark for semantic databases. Since LDBC is a non-profit organization with members from both industry and academia, graph benchmarking is another area where the graph research community has influenced the industry. Both TigerGraph and Neo4j has published white papers or blogs about their test results on LDBC-SNB benchmark [25, 49], as well as the responses to each other’s results [54].

This has been a lot of work in the research community that compares various graph databases [22, 32, 34, 35]. This line of research is highly useful, however, most of these efforts proposed their own benchmark frameworks, and the results from different studies often generated different conclusions. It will be more valuable if future studies adopt an existing widely used benchmark that the industry embraces, such as the LDBC benchmark.

### 8. DISCUSSION

#### 8.1 Graph Database Users

There are different types of graph users from the most sophisticated to the most novice. The first category of graph users are the few companies such as LinkedIn or Facebook for whom graph is the business! These users usually have an army of in-house engineers to build customized systems for their bread and butter [6, 19]. They will not shop for a general graph database. The second category is the power users, where their core business has a strong dependence on graphs, such as companies specialized in fraud detection, anti-money laundering, security, and intelligence. They most likely will adopt the best graph technologies in the market that fit their needs. The third category is a larger group of standard experienced users. Many of them were the traditional database customers who now build new applications enabled by graph technologies. Some example success stories of such customers were showcased in [48]. Then there is a large of number of novice users and potential users, who are interested in trying out graph technologies. The last three categories of users are the ones whom graph database vendors typically go after. And if they come from the traditional database side, converged...
database solutions may provide them with an easier entry point into graphs. To better serve the experienced graph users and convert novice or potential users, customers need education on what they can do with graph technologies and how to apply them. Pure-play companies like Neo4j and TigerGraph are particularly strong in customer education. They publish books, organize workshops, summits and conferences where they showcase the technologies as well as demonstrate use cases.

Another important point is that graph problems are greater than graphs! Yes, customers need graph solutions, but not just graph solutions. They need an end-to-end solution which involves data ingestion, preprocessing, graph analytics, maybe also other types of analytics, and result rendering. When we think about the big picture, graph performance is not the only factor that customers consider when choosing graph solutions.

Finally, both graph OLTP and OLAP are important. Some customers primarily use one type vs the other, but others use both. For graph OLTP, due to the exploratory nature of the queries, graph visualization is a must-have. For graph OLAP, the set of built-in algorithms will be the winning factor that attracts customers.

### 8.2 In-House Graph Systems

As mentioned before, big tech companies, like LinkedIn and Facebook, have developed their own specialized graph systems to serve their business needs. Although these in-house systems are not yet in the commercial space, these companies have a good track record of open-sourcing their in-house systems with great industrial impact. So, some of these graph systems are worth watching for.

Liquid [6] is LinkedIn’s in-house graph database for real-time querying of its economic graph. Liquid adopts a subject-predicate-object triple model to store edges similar to RDF, employs a declarative language based on Datalog, and achieves nanosecond-level query efficiency via dynamic query optimization on wait-free shared-memory index structures [20]. TAO [19] is Facebook’s geographically distributed graph system for serving efficient access to its social graph. It supports a property-graph-like model, provides simple APIs to access nodes and edges, and is built with an efficient caching layer on top of the MySQL storage layer.

### 8.3 Research Graph Database Prototypes

Although a comprehensive survey of graph databases from academia is beyond the scope of this paper, it is still interesting to briefly discuss some recent related systems. Research graph systems are also divided into native and hybrid graph databases. Graphflow [33] and AvantGraph [50] are two examples of native graph systems. Both systems implemented the worst case optimal (wco) joins and factorization of intermediate results. Although not yet widely adopted in industry, these techniques have been proven to be promising in speeding up graph query processing and present great potential for future adoption. GRFusion [28] and GRainDB [31] are two examples of hybrid graph databases. GRFusion [28] extends VoltDB to define graph views on relational tables, and to materialize graph structures in memory for the graph queries to execute on. This architecture resembles Oracle Spatial and Graph. GRainDB extends the internals of DuckDB to support predefined pointer-based joins, and plans to also incorporate wco joins and factorization into the query engine. Most commercial hybrid graph databases today have conservatively chosen to avoid modifying the core engines when supporting graph queries. GRainDB points out interesting directions on how graph-specific techniques can be incorporated into an existing database engine.

### 8.4 Opportunities and Directions

There is a growing global market for graph databases. Right now, the graph-only vendors are leading in query performance and algorithm support. But there is still a lot of opportunities for new vendors or other existing vendors to catch up. Especially, when looking at end-to-end scenarios, data import and export may increasingly become a bottleneck for graph-only databases. Making data movement easier and faster will be a crucial investment for these pure-play vendors. Major cloud companies are also investing in the graph space. Their major advantage is that they own the whole stack of data services, including operational data services (OLTP and NoSQL engines), which is a major source of graph data. It makes sense that most of these vendors adopt converged database solutions. Taking the advantage of the whole stack and focusing on end-to-end solutions will be a winning recipe for them.

### 8.5 Recommendation for Researchers

The graph database research community has been focusing heavily on sophisticated algorithms and query performance so far. These are very important, but there are other equally important but also practical problems that the graph database industry cares about and needs more help on. Due to the connected nature of graphs, transaction support, especially in a distributed setting, is hard to achieve and even harder to perform efficiently in graph databases. Visualization is crucial for graph OLTP workloads, but laying out graphs in a way that users can clearly understand the relationship of entities in the graph is generally a hard problem. Compliance to regulations (e.g. GDPR) requires keeping track of versions of graph data and potentially supporting point-in-time queries, but this feature is unfortunately missing in most existing graph databases. Multi-tenancy and access control are not supported efficiently in existing graph databases (most just assume there is a single tenant for a graph database server), so helps are definitely needed on this front. Graph queries and analytics are seldom executed in isolation, so integrating them with non-graph workloads efficiently deserves a lot of attention. For native graph databases, optimizing data import and export will be crucial.

In addition, existing graph research largely assumes read-only workloads. But real-life graphs do change over time. Future research would better support the dynamic nature of graphs. Some of the existing research also define their own graph models (e.g. simple graphs and attributed graphs) or propose new query languages. Focusing on widely adopted graph models and languages from industry might lead to more practical impact.
9. REFERENCES

Customers Choose TigerGraph.
https://www.tigergraph.com/customers/.


[56] YAGO. YAGO. https://yago-knowledge.org/.


ABSTRACT

The Hasso Plattner Institute (HPI), academically structured as the independent Faculty of Digital Engineering at the University of Potsdam, unites computer science research and teaching with the advantages of a privately financed institute and a tuition-free study program. Founder and namesake of the institute is the SAP co-founder Hasso Plattner, who also heads the Enterprise Platform and Integration Concepts (EPIC) research center which focuses on the technical aspects of business software with a vision to provide the fastest way to get insights out of enterprise data. Founded in 2006, the EPIC combines three research groups comprising autonomous data management, enterprise software engineering, and data-driven decision support.

1. INTRODUCTION

In the past, data was kept in separate database systems — specializing either in transaction processing (OLTP) or analytical query execution (OLAP) — for performance reasons. As a result, maintaining tedious materialized aggregates and time-consuming, error-prone ETL processes underpinned business applications and made them complex, slow, inflexible as well as difficult to scale and extend. In particular, it was not possible to ask ad-hoc analytical questions about the current business state without lengthy preparation and data collection steps, and obtaining results required hours instead of seconds.

In 2006, these shortcomings, together with recent advances in hardware such as multi-core CPUs and large main-memory capacities, motivated Hasso Plattner, co-founder of SAP, to question the status quo and to rethink database architectures. He founded the Enterprise Platform and Integration Concepts (EPIC) research center at the HPI and, together with his PhD students and SAP, he created a novel class of database systems that exclusively store and process all data in main-memory [34]. At the core of this idea is the adoption of the columnar data layout allowing for extremely fast scans that are capable of processing billions of records per second even on a single CPU core; cutting down storage requirements with highly efficient in-memory compression; and reducing the need for additional database constructs such as materialized aggregates [35]. With this in-memory database technology, the traditional separation of data processing systems into transactional and analytical could be overcome and the benefits are a simplified system landscape, removal of data replication, increased transactional throughput and business applications that perform real-time reporting on fresh data [36].

Based on the EPIC’s research and its open-source, in-memory database prototype — called Hyrise — SAP started to develop a professional version with columnar storage for OLTP and OLAP workloads — SAP HANA — which is now the foundation of nearly all SAP solutions. Furthermore, the joint research work led to more than 200 peer-reviewed publications and fundamentally changed the approach to data management in enterprises. Today, EPIC is still researching the next-generation of database management and enterprise software with the goals of maximizing performance as well as improving cost efficiency and business value. Within the three research groups, we focus on (i) autonomous data management on modern in-memory and cloud hardware, (ii) rethinking development of business applications, and (iii) models and quantitative methods for data-driven decision support. For all of our research, we apply our methods to real-world scenarios and showcase future enterprise applications by developing and evaluating prototypes closely together with industrial partners.

In this work, we further motivate the aforementioned research topics, highlight our key contributions, and conclude with collaboration activities across our research projects; for more details see https://epic.hpi.de.

2. AUTONOMOUS DATA MANAGEMENT

The trend toward storing more and more data in the cloud requires database providers to focus on self-optimization and cost-efficiency. At the same time, the limited capacity and high costs of main memory are the driving force behind storing less frequently accessed data on cheaper disk-storage, e.g., in so called data lakes.
We address these trends in two connected lines of research, by sharing certain base concepts. First, we research a self-driving (i.e., autonomous / unsupervised) in-memory database management system, called Hyrise, that finds (near-)optimal configurations for important in-memory database concepts like (i) data compression and tiering, (ii) replication and scale-out, (iii) index selection, and (iv) data dependencies for query optimizations, as well as (v) approaches for joint optimizations (e.g., for spatio-temporal data management).

Second, a more recent line of research addresses the efficient in-situ processing of massive amounts of infrequently accessed data stored in data lakes for interactive, analytical workloads. For this, we are developing Skyrise, which leverages advances in modern serverless technologies like Function-as-a-Service (FaaS). We develop Skyrise along pure serverless / FaaS principles, making it a seminal, cost-efficient, and fully functional data processing-as-a-service system.

Subsequently, we discuss our research contributions in the areas of autonomous databases and cloud data processing systems, which includes Hyrise and Skyrise.

2.1 Autonomous In-Memory Database Hyrise

The initial version of Hyrise [12] was publicly presented in 2010, featuring the concept of hybrid row-and-column data layouts for in-memory databases that was coincidentally at the core of other database systems at that time (e.g., SAP HANA [11]). Through subsequent research efforts in the aforementioned areas of autonomous databases, Hyrise was re-engineered starting from 2016 [9] and is available open source\(^1\) as an in-memory database system for reproducible research and education. The foundation of Hyrise is comprised of typical storage and query processing components like SQL parser, logical (LQP) and physical query plans (PQP) that are cached for optimization, and an executor that runs PQP operators. The Hyrise system is accessible via PostgreSQL wire protocol, making it compatible with other database systems. Together with a plugin concept that allows for rapidly building prototypes and performing experiments, the competitive performance of Hyrise (cf. [2, 10]), compared to state-of-the-art in-memory systems like HyPer [26] and analytical databases like DuckDB [37], makes it suitable for systems research.

To that foundation, we added self-driving components as plugins (e.g., data tiering), which leverage runtime parameters (e.g., data, workload) for decision making, to predict the impact of potential configuration changes and automatically apply those changes that are deemed beneficial. The changes are associated with tuning options (e.g., creation of secondary indexes, (re-)encoding of columns), which we describe subsequently. Many of the tuning options have shared requirements (e.g., access counters for encoding, tiering).

Finally, Hyrise is also relevant for teaching upcoming generations of database developers about the intricacies of in-memory computing. In our In-Memory Research Laboratory, students find a suitable environment to experiment and conduct studies on novel main-memory, multi-core and GPU processor technology. Since 2017, over 130 HPI master students have contributed as part of our database seminar, master projects, master theses, or as student assistants to Hyrise.

2.1.1 Data Compression & Tiering

Data compression and tiering are powerful methods to address the memory bottleneck and cost inefficiencies for in-memory databases. The automatic decision on which data compression technique to use in in-memory column stores is challenging due to trade-offs and non-obvious impacts on large workloads. In [2], we propose a solution for an automatic selection of a budget-constraint encoding in Hyrise, based on linear programming (LP) and greedy heuristics. The encoding configurations are robust with respect to runtime performance, adaptable and workload-aware. To ensure performance robustness, LP techniques are applied to achieve equally distributed performance gains over all queries. The results show the potential of significant memory budget reductions without a deterioration of runtime performance.

Similarly, data tiering promises to reduce the amount of data in main memory by moving infrequently used data to cheaper and more elastic lower memory and secondary storage tiers. The challenge is to find an optimal balance for the trade-off between performance and costs. We propose an automatic tiering for Hyrise in [4], using LP, that addresses this challenge. Our approach tracks frequency and pattern of data accesses to identify rarely used data, which are moved to secondary memory tiers (e.g., NVM / SSDs). This method is applicable to column selection problems in general and ensures Pareto-efficiency for varying memory budgets. Since, aspects like selectivity, size and frequency of queries are taken into account, the resulting performance is optimized and outperforms other heuristics.

2.1.2 Replication & Scale-out

Database replication and query load-balancing are important mechanisms to scale query throughput. The analysis of workloads allows load-balancing queries to replica nodes according to their accessed data. As a result, replica nodes must only store and synchronize subsets of the data. However, evenly balancing the load of large-scale workloads while minimizing the memory footprint is complex and challenging. Moreover, state-of-the-art allocation approaches are either time consuming or the resulting allocations are not memory-efficient.

\(^1\)Hyrise, visited 04/2022: https://github.com/hyrise/
In our work, we used LP-based decomposition techniques to determine optimized data placements and workload distributions [15, 16]. We extended these solutions considering potential node failures [18, 17]. Further, we derived a heuristic solution to compute robust solutions for large, real-life workload instances providing a competitive performance for different potential as well as uncertain workload scenarios [50].

2.1.3 Index Selection

For complex workloads, secondary indexes become highly relevant for database performance, but current index selection algorithms are either not fast or not highly competitive, as we found in our survey which evaluates state-of-the-art approaches [27]. To overcome the observed limitations of existing approaches, we developed two new index selection algorithms, serving different purposes: EXTEND [51] determines (near-)optimal solutions with an iterative heuristic. The produced solutions outperform others in most evaluated cases while the selection runtime is up to 10x lower. SWIRL [30] is based on reinforcement learning (RL) and — after training — delivers solutions instantaneously. SWIRL decreases selection runtimes by orders of magnitude, while the solution quality is within 2% of the best solutions. While EXTEND is universally applicable with a high solution quality, SWIRL requires training, but reduces runtimes. In further approaches, we consider different stochastic potential workload scenarios and aim at finding index selections under risk-averse criteria [49].

2.1.4 Data Dependencies

Efficient query optimization is usually based on metadata such as cardinalities and other basic statistics. More advanced techniques consider data dependency types such as functional, uniqueness, order, or inclusion constraints / dependencies. In our recent survey [28], we identified 60 query optimization techniques for application areas like join, selection, sorting and set operations in the literature that are based on data dependencies.

Toward an efficient implementation and integration into commercial database systems, we laid out a vision in [29] for a workload-driven discovery system for query optimization. The dependency discovery is considered “lazy” since only those data dependency candidates are considered that are relevant for the observed workload. Our prototypical implementation in Hyrise identifies relevant data dependency candidates based on executed query plans and dynamically validates the candidates against the database, leading to performance improvements.

2.1.5 Joint Tuning & Spatio-Temporal Decisions

Challenges for self-driving database systems, which tune their physical design and configuration autonomously, are manifold: such systems have to anticipate future workloads, find robust configurations efficiently, and incorporate knowledge gained by previous actions into later decisions. We present a theoretical, component-based framework for self-driving database systems that enables database integration and development of self-managing functionality with low overhead, by relying on separation of concerns [31]. In [38], we started to implement joint tuning approaches in Hyrise, accounting for combined indexing, sorting, and compression configurations for spatio-temporal applications.

2.2 Serverless Data Analysis in Skyrise

Enterprises are moving their applications to public cloud environments to benefit from the resource elasticity, and cost efficiency that their infrastructures provide. The resulting collocation of applications provides an opportunity to explore application data within and across organizations with integrated analytics. Current database systems, however, lack either cost-efficiency, or in-memory performance for real-time analytical data processing inside an organization on large data.

With Skyrise [1], we explore a data analysis architecture that exploits modern, serverless cloud infrastructure such as Function-as-a-Service (FaaS) to achieve cost-efficiency and low latency through massive, ad-hoc resource elasticity and auto-scaling. While the technical aspects of FaaS have to further evolve and mature for efficient data processing [19], we started employing FaaS for data processing in the cloud [1] by targeting the current sweet spot for sporadic, interactive, analytical, in-situ processing of large amounts of infrequently used data (e.g., stored in data lakes). Skyrise builds on a number of components used in Hyrise (e.g., SQL parsing, query plan translation and optimization), and introduces new solutions for query plan optimization and distributed plan execution on FaaS infrastructure.

3. ENTERPRISE SOFTWARE ENGINEERING

Built on a fast database layer, it is time to question how enterprise applications are developed on top. Therefore, we explore current shortcomings in enterprise software engineering to create methods and tools for new development practices and software architectures. While our current research focuses on the process side of software development, we are shifting our focus to the cloud-native engineering side.

3.1 Software Development Processes

Enterprise software development usually requires several collaborating teams working on a shared codebase. In popular agile process frameworks, such as Scrum, work processes are maintained through iterative process improvement cycles and retrospection meetings. In our
research, we show how existing agile methods can be supplemented with process improvement steps based on software engineering team data. Our approach includes gathering empirical data on the perceptions of team members, as well as deriving insights from teams’ project data [33]. The underlying artifacts, such as source code, commits, or work documentation, are already being produced during regular development work. By aggregating, linking, analyzing and visualizing the available data, we enable teams to gain actionable insights into their own development processes [32].

Traditionally, software development teams collaborate with other disciplines such as design, management, or sales. As multiple disciplines work differently, such teams may experience problems in their collaboration. Additionally, companies often implement different process frameworks for these disciplines, leaving collaborating teams with a variety of options and tools without guidance on how to integrate them. In our research, we develop InnoDev [7], a comprehensive framework for such teams to work jointly on innovative software products from idea to implementation and market growth. InnoDev integrates three modern frameworks from complementary disciplines: (i) Design Thinking from the field of Design focuses on understanding the users’ problems and developing desirable solutions, (ii) Lean Startup from the field of Entrepreneurship focuses on customer and business development, and (iii) Agile practices from the field of Software Development focus on organizing software development collaboratively and flexibly. InnoDev thus serves as a common language and improves teamwork. We provide empirical evidence that Agile scaling and project management activities are well suited to support the conceptualization phases of the software development process. We also show that Design Thinking supports various activities during the development process and increases the team’s product understanding, empathy within the team, and empathy towards users [6]. With our toolbox and our workshop [8], we offer practical guidance for teams which want to integrate Design Thinking, Lean Startup, and Agile software development for creating business applications.

3.2 Cloud-native Enterprise Architecture

To fully leverage the potential of fast databases and modern cloud-native development approaches, our research group is moving its focus up one level in the software stack onto the architecture of Enterprise Resource Planning (ERP) systems. ERP systems play a vital role for enterprises by providing comprehensive standard solutions. However, they offer only limited opportunities to customize the standard processes according to companies’ individual needs. Additionally, unsatisfying integration with third-party software leads to numerous data silos on-premise and the cloud. For these reasons, after tackling the database foundations, we rethink the general architecture of ERP systems. Future enterprise applications should enable companies to build and support their growing business processes, while maintaining a future-oriented data model that allows for adaptation to the evolving needs of an upcoming enterprise.

We start research in that direction by analyzing the requirements, the feasibility, and the effects of implementing a new generation of cloud-native enterprise systems based on executable business processes on top of their underlying business data. Thereby, we address the resulting engineering challenges, especially concerning the responsibility of services and the scalability of inter-process communication.

4. DATA-DRIVEN DECISION SUPPORT

Data-driven decision support (DDDS) for enterprise applications has become highly relevant in recent years, as firms face the challenge of integrating data-driven automation in their processes. Based on the ability to effectively store, process, and handle data, the DDDS research group investigates how quantitative methods of operations research and data science can be used to improve automated decision-making. Our scope involves (i) the identification of causal relations (Sec. 4.1), (ii) specific revenue management problems (Sec. 4.2), and (iii) resource allocation and database tuning problems in close collaboration with the ADM group, see Sec. 2.

4.1 Causal Structure Learning (CSL)

The key for effective decision support is to understand and to be able to measure the causal interplay of main variables involved in a specific use case. We address challenges in CSL, from data to learned causal structures, by improvements in both the application of statistical concepts and the computational acceleration.

Knowledge of the causal structures within complex systems is crucial to many domains. For example, in manufacturing [23, 13], where causal structural knowledge constitutes the basis of error avoidance in a production process. While domain experts within the company have enough expertise to identify the most common relationships, they will require support in the context of both an increasing amount of observational data and the complexity of large systems. This gap can be closed by algorithms of CSL that derive the underlying causal structures from observational data, e.g., error messages and inline measurements of a production process.

Our research of data-driven causal inference concentrates on several workstreams which, combined, aim to allow an efficient application of CSL. In a first stream, we work on information-theoretic approaches [20] to improve CSL in the context of heterogeneous data char-
characteristics, which are typical for real-world scenarios. Second, we built a modular pipeline (MPCSL [22]) for experimental evaluation of CSL from observational data accompanied with a first benchmark framework (MANMCS [21]) to generate test data allowing us to compare the strengths and weaknesses of CSL algorithms. Third, we exploit hard- and software acceleration to speed up CSL algorithms, which are numerically intensive. In particular, we leverage GPUs to develop efficient implementations in heterogeneous computing systems [14, 53].

4.2 Revenue Management (RM)

Over the years, RM applications have become increasingly difficult to administrate. The number of decisions to control business processes have become too complex to be managed manually. As currently established rule-based solutions are far from optimal, automated decision-making has enormous potential. However, it is challenging to derive optimized decisions, as most RM applications lead to complex stochastic dynamic problems. Nevertheless, the overall vision is to develop self-driving decision support systems, which automatically analyze market data and optimize operational business decisions.

One stream of our work aims at analytical solutions of dynamic pricing problems. In contrast to solely numerical solutions, such approaches allow us to gain structural results and general insights regarding optimal decision-making, which allow us to infer managerial recommendations. Results have been derived for pricing applications with time-dependent demand [39], risk aversion [40], joint advertising [41], oligo-poly competition [43], online vs. offline markets [47], and towards a circular economy [5] (cf. sustainability).

Besides studying analytical solutions, we also look for improved and extended methodologies, especially in the area solving Markov decision processes (MDP). For example, we developed approaches to solve MDPs under risk aversion [44] and designed approximate dynamic programming (ADP) techniques to solve, e.g., challenging multi-product pricing problems [42, 45].

In a further project, we partnered with a top 10 seller of used books on Amazon in Germany. The firm has an inventory of over 100,000 distinct books. The challenge was to optimize (expected) profits and the number of sales. The established pricing strategy of our project partner was a rule-based system that has been developed over the past years by carefully adjusting rules to lessons learned from selling books on Amazon. We developed a data-driven pricing strategy based on a relaxed Markov model with estimated sales probabilities. In a real-world comparison, our strategy clearly outperformed the merchant’s benchmark strategy with respect to both profitability and speed of sales [46].

Another stream of the DDDS group is to study the interplay of self-adaptive pricing strategies in simulated environments. While RL-based strategies require less knowledge and can be applied in highly complex environments, they are also extremely data hungry. Hence, before applying them in practice, they have to be tested in simulated markets. Our group is working on both frontiers, i.e., RL-based pricing strategies [24, 25, 52] as well as flexible simulation frameworks [3, 48, 54].

5. COOPERATION OF THE GROUPS

To reach the vision of building the next generation of enterprise software, the EPIC’s research groups are mutually supportive: At the technical bottom, the ADM group works on the fastest data processing approaches, the ESE builds on top of this and rethinks business applications in a cloud-native world, and, finally, the DDDS group benefits from the work of both groups by being able to conveniently access and handle business data. The DDDS group also supports the other groups by providing models, quantitative methods, and solution techniques for their various kinds of resource allocation problems. Automated database tuning problems, in particular, require not only sophisticated mathematical concepts but are also clearly developing in a direction where dynamic aspects, stochastic or uncertain workloads as well as robust solutions — to be able to guarantee good performances in different potential scenarios — are key.

Acknowledgements

We would like to thank the EPIC’s current and former, very talented PhD students and PostDoc researchers. Further, we acknowledge our various industry partners. First of all, we thank SAP for funding us and interesting research questions, especially SAP HANA, New Ventures and Technologies, and the entire Technology and Innovation board area, for bringing research results to innovative products. Besides SAP, we are collaborating with Seagate, Porsche, Heidelberg Druckmaschinen, Hilti, AWS, Zalando, and many others on leveraging technologies for enterprise software. Thanks for providing tough research challenges, believing in us, and transferring the results back to business.

6. REFERENCES

How Connected Are Our Conference Review Boards?

Sourav S Bhowmick
Nanyang Technological University, Singapore
assourav@ntu.edu.sg

ABSTRACT
Dense co-authorship network formed by the review board members of a conference may adversely impact the quality and integrity of the review process. In this report, we shed light on the topological characteristics of such networks for three major data management conference venues. Our results show all these venues give rise to dense networks with a large giant component. We advocate to rethink the traditional way review boards are formed to mitigate the emergence of dense networks.

1. INTRODUCTION
Conference review process enables us to rapidly vet our research results through peer review and then quickly share them. Our professional societies (e.g., ACM) strive to ensure that our conference review boards are of high quality and continue to serve the review process effectively. Specifically, program committee (PC) chairs of our review boards have endeavoured to improve the quality by selecting experts to ensure high coverage of all topics of interest, diversifying members along various dimensions (e.g., gender, location, experience), monitoring reviews, among others. Despite these efforts, there has been anecdotal evidence on the existence of collusion rings and violation of a venue’s conflicts-of-interest (COI) policy [1,11,12] that undermine the quality, fairness, and integrity of the review process. Hence, it is paramount to look beyond these traditional strategies (topic coverage, diversity) to enhance the quality of review processes.

Co-authorship (i.e., collaboration) relationship is one of the key pillars of COI policies for all major venues. Intuitively, given two sets of review board members with similar topic coverage and diversity, $R_1$ and $R_2$, it is superior to choose $R_1$ over $R_2$ if the co-authorship network formed by $R_1$ is significantly less dense than that of $R_2$. This is because a dense network may adversely impact the review process in at least three ways. First, it increases the likelihood of (even inadvertent) undeclared COI and COI violations. An author (can be a review board member) may have higher chance of prior co-authorship with some review board members of a dense subnetwork that may potentially give rise to COIs that are either unreported by the author or existing duration-based COI policies fail to capture them. To elaborate on the latter case, consider an author $a$ and two reviewers $m_1$ and $m_2$ with strong co-authorship tie (i.e., there is an edge $(m_1,m_2)$ in the network). All are located in the same region. Suppose $a$ has strong co-authorship tie with $m_1$. The likelihood of $m_2$ to have ties with $a$ (i.e., a wedge co-authorship pattern) may be higher in this scenario compared to the case where $m_1$ and $m_2$ are isolated nodes in the network. Furthermore, in some cases they may be part of a collusion ring [11]. A recent anecdotal evidence of such possible collusion involving co-authors is mentioned in [1].

Second, a denser network increases the likelihood of a set of reviewer board members reviewing a submission to be connected. For instance, in one of the submission cycle of a major data management venue, around 40% of the submissions have at least one co-authorship edge between assigned reviewers and 36% of these submissions have strong ties (10 or more papers). For some cases, this may not have any adverse impact on the review process. But for other cases, a group of connected reviewers may either collude together to reach a favourable decision for a submission authored by close ties (e.g., [1]) or a junior member may be unduly influenced by an influential reviewer who has been their co-author.

Third, a dense network makes it challenging for a PC chair to assign unbiased reviewers for submissions where authors either have close ties with members of a subnetwork or they themselves are also review board members having high degree cen-

---

1The number of review board members who are also authors of submissions is often significant. For example, at least 58% of them are found to be authors in a recent major venue.

2This behaviour can be explained by the social impact theory in social psychology [8].
trality or clustering coefficient in the network.

Despite the potential impact of co-authorship network topology on the review process, to date there has not been any systematic study that shed light on the characteristics of these networks in data management venues. In this report, we take a concrete step to this end. Our study revealed the existence of dense networks with small-world characteristics in all major data management venues. We report various topological features of these networks and their implications on the review process. We conclude by advocating the need to depart from the traditional approach of review board formation to a data-driven, system-based approach to mitigate the emergence of dense co-authorship networks.

2. DATASET

We consider the review boards of recent editions of three major data management venues, SIGMOD 2022, VLDB 2022, and ICDE 2022, for our study. The lists of members (meta-reviewers and reviewers) were received from the PC chairs of respective venues. Table 1 reports the statistics. In practice, a review board can be dynamic in nature with the addition and removal of board members during the review process for various reasons. Hence, we consider the aggregated review boards of VLDB and ICDE after the final submission cycles (i.e., March 2022 and November 2021, respectively). For SIGMOD we use the review board of the first submission cycle (July 2021) since we did not receive any subsequent updates.

<table>
<thead>
<tr>
<th>Venue</th>
<th># of meta-reviewers</th>
<th># of reviewers</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGMOD 2022</td>
<td>26</td>
<td>177</td>
</tr>
<tr>
<td>VLDB 2022</td>
<td>22</td>
<td>200</td>
</tr>
<tr>
<td>ICDE 2022</td>
<td>22</td>
<td>194</td>
</tr>
</tbody>
</table>

We manually retrieved the DBLP addresses of all review board members using Google search. Each review board member for a given venue is uniquely identified by their email address or DBLP name which is unique in DBLP3.

3. NETWORK GENERATION

The co-authorship network \( C = (V, E) \) of a review board is an undirected, labeled, weighted graph where \( V \) is a set of review board members and \( E \) is a set of co-authorship edges between them. Given a pair of review board members \( u, v \in V \), \((u, v) \in E\) if \( u \) and \( v \) have co-authored one or more articles. A node \( u \in V \) is labeled with an unique identifier of the review board member and \((u, v) \in E\) is labeled with a weight \( w \) representing the number of co-authored articles by \( u \) and \( v \). We automatically generate the co-authorship network of a review board from DBLP by leveraging the co-authorship network generation component of CLEVER [5], a state-of-the-art COI detection and management system. Specifically, for each review board member it retrieves the corresponding XML version of their DBLP page and extracts all co-authors who are members of the review board and computes the frequencies of co-authorship. Then the network is constructed from it. Each node is labeled with the corresponding DBLP name of the member.

For each review board, we generate three types of co-authorship networks: meta-reviewer network, reviewer network, and review board network. In a meta-review network \( C_M = (V_M, E_M) \), \( V_M \) represents the set of meta-reviewers. On the other hand, in a reviewer network \( C_R = (V_R, E_R) \), \( V_R \) represents the set of reviewers. The review board network \( C = (V, E) \) is the aggregated network of meta-reviewers and reviewers, i.e., \( V = V_M \cup V_R \).

4. NETWORK PROPERTIES

In this section, we analyse various properties of the co-authorship networks of the three venues.

Global properties. We first report the global topological properties of the networks. Specifically, we compute the network size, average degree (denoted by \( \langle k \rangle \)), density (denoted by \( \rho \)), average clustering coefficient (denoted by \( \langle c \rangle \)), number of connected components (denoted by \( M \)), the size of the largest connected component (i.e., giant component) (denoted by \( L \)), and the local network efficiency \( [10] \) (denoted by \( \Phi_L \)) of the three types of co-

<table>
<thead>
<tr>
<th>Prop.</th>
<th>Venue</th>
<th>( C_M )</th>
<th>( C_R )</th>
<th>( C )</th>
<th>ER</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGMOD 2022</td>
<td>26</td>
<td>177</td>
<td>200</td>
<td>194</td>
<td>0.019</td>
</tr>
<tr>
<td>VLDB 2022</td>
<td>22</td>
<td>200</td>
<td>219</td>
<td>194</td>
<td>0.019</td>
</tr>
<tr>
<td>ICDE 2022</td>
<td>22</td>
<td>194</td>
<td>216</td>
<td>194</td>
<td>0.019</td>
</tr>
<tr>
<td>SIGMOD 2022</td>
<td>26</td>
<td>177</td>
<td>200</td>
<td>194</td>
<td>0.019</td>
</tr>
<tr>
<td>VLDB 2022</td>
<td>22</td>
<td>200</td>
<td>219</td>
<td>194</td>
<td>0.019</td>
</tr>
<tr>
<td>ICDE 2022</td>
<td>22</td>
<td>194</td>
<td>216</td>
<td>194</td>
<td>0.019</td>
</tr>
<tr>
<td>SIGMOD 2022</td>
<td>26</td>
<td>177</td>
<td>200</td>
<td>194</td>
<td>0.019</td>
</tr>
<tr>
<td>VLDB 2022</td>
<td>22</td>
<td>200</td>
<td>219</td>
<td>194</td>
<td>0.019</td>
</tr>
<tr>
<td>ICDE 2022</td>
<td>22</td>
<td>194</td>
<td>216</td>
<td>194</td>
<td>0.019</td>
</tr>
<tr>
<td>SIGMOD 2022</td>
<td>26</td>
<td>177</td>
<td>200</td>
<td>194</td>
<td>0.019</td>
</tr>
<tr>
<td>VLDB 2022</td>
<td>22</td>
<td>200</td>
<td>219</td>
<td>194</td>
<td>0.019</td>
</tr>
<tr>
<td>ICDE 2022</td>
<td>22</td>
<td>194</td>
<td>216</td>
<td>194</td>
<td>0.019</td>
</tr>
</tbody>
</table>

3In DBLP, homonyms are distinguished from one another by a unique numerical suffix to their name.

4Efficiency of a network measures how efficiently it exchanges...
authorship networks for each venue. Table 2 reports the results. We can make the following observations. First, although the size of the review boards of all venues is similar, the number of edges in the three types of co-authorship networks of VLDB and ICDE are significantly higher than that of SIGMOD. That is, the review boards of the former have more collaborative links compared to the latter. Second, the densities are high with ICDE networks being the highest among all. Third, the average clustering coefficients of all venues are significantly higher than the corresponding \( \langle k \rangle / N \) values, demonstrating small-world properties of these networks [9]. For instance, \( \langle k \rangle / N = 0.019 \) for \( C_{VLDB} \) of VLDB2022 which is significantly lower than \( c \) = 0.2142. Specifically, the high values of \( c \) indicate well-connectedness of the neighborhood of a reviewer in these networks.

Observe that ICDE and VLDB networks have higher \( c \) than that of SIGMOD. Fourth, the number of connected components in all networks is low hovering between 14-35 in \( C_{RL} \) and \( C \). Interestingly, the size of the giant components in \( C_{RL} \) and \( C \) is large for all venues. For example, around 94\% of the nodes in \( C \) of VLDB are part of the giant component! Lastly, \( C_{ICDE} \) of ICDE2022 has the highest local efficiency. Figure 1 depicts the review board networks of these venues. Note that the goal here is not visual clarity as graphs with more than 100 nodes look like a hair-ball [13]. Instead, the intention here is to visually appreciate the denseness of these networks and the existence of a large giant component.

**Comparison with random network.** The above results demonstrate high average clustering coefficient (resp. local network efficiency) in \( C_{RL} \) and \( C \) highlighting strong co-authorship ties and information exchange between the neighborhoods of review board members. *Can these properties emerge by chance?* We utilize the reviewer network \( C_R \) to answer this question. We construct a random network of similar size and average degree of \( C_{RL} \) and compute these measures of the network. Specifically, we construct an Erdos-Renyi (ER) network using the \( G(N, p) \) model [6]. We set \( N = |V_R| \) and \( p = \langle k_R \rangle / N - 1 \) [6] where \( \langle k_R \rangle \) is the average degree of \( C_R \). This enables us to generate an ER network with size and density similar to \( C_R \). We generate 100 instances of the ER network and compute the average values of the seven topological properties. The last column in Table 2 reports the results. Observe that the avg. clustering coefficients and local efficiency of the ER networks are around an order of magnitude smaller than the corresponding values in \( C_{RL} \). Hence, randomness cannot explain them and it may represent some “signature of order, requiring a deeper explanation” [4]. The ER network also demonstrates the co-existence of a giant component and isolates. The emergence of giant component is expected as \( 1 < \langle k \rangle < ln|V| \) in these networks [4,7]. However, its size is larger than the ones in \( C_{RL} \).

**Degree distribution.** Figure 2 depicts the degree histograms of reviewer networks (\( C_{RL} \)) of the three venues. Observe that all venues have several high-degree reviewers (i.e., degree more than 7).

**PC overlap.** Next, we look at the amount of overlap between the review boards. Since VLDB has the most onerous multi-cycle review process, we compute the overlap between the nodes in the reviewer networks of (VLDB 2022, SIGMOD 2022) and (VLDB 2022, ICDE 2022). We focus on the reviewer network since reviewing fatigue is encountered most by the reviewers. The number of common reviewers are 40 and 34, respectively. To get an understanding of the trend of the overlap, we also compute the overlap between the 2021 and 2023 editions of VLDB and SIGMOD. There are 38 and 60 common reviewers in 2021 and 2023, respectively, showing an upward trend in the size of common reviewers.

**Weight thresholding.** A co-authorship network may contain strong and weak ties where the

---

Footnotes:

1. Information [10]. Local efficiency is the average efficiency of the local subgraphs. Small-world networks have high global and local efficiency.
strength of a tie is measured using frequency of co-authorship (i.e., edge weight). In recent times, the frequency of co-authorship has been incorporated in the COI policies of several venues (e.g., ICDE 2020, ICDE 2021, VLDB 2023). Hence, in this set of experiments we study the topological properties of the subnetworks with strong ties in a co-authorship network using weight thresholding. That is, we sparsify the network by using an edge weight threshold $\theta$ and observe how the topological properties of subnetworks with strong ties evolve. Given $\theta$ and a co-authorship network $C$, we remove all edges with $w < \theta$ along with nodes that have no edge with $w \geq \theta$ resulting in a sparsified network $C_\theta = (V_\theta, E_\theta)$ where $V_\theta \subseteq V$ and $E_\theta \subseteq E$. Then, for each property $P$ we compute $P_\theta / P$ where $P_\theta$ is the value of the property $P$ in $C_\theta$ and $P$ is the corresponding value in $C$. We use the reviewer network $C_R$ of each venue as the original network and vary $\theta$ from 3 to 7.

Figure 3 plots the results. Observe that the densities of $C_\theta$ remain relatively robust. Importantly, for VLDB 2022 and ICDE 2022 a significant part of the networks (around 60% of the nodes) maintain strong ties when $\theta = 7$. Furthermore, $\langle c \rangle$ and $\Phi_l$ of $C_R$ of 2021 edition of these venues vary between 0.02-0.033, 0.23-0.24, and 0.28-0.29, respectively. Similarly, for SIGMOD 2023 and VLDB 2023 these values vary between 0.021-0.023, 0.22-0.23, and 0.280-0.283, respectively. While such features are highly desirable for social networks, as remarked earlier, in a peer review setting they may potentially create challenges for PC chairs to assign unbiased reviewers to submissions.

We believe that the manifestation of dense co-authorship networks of our review boards is mainly due to the traditional “manual” approach of review board formation where a candidate reviewer is invited independent of other candidates primarily based on the recommendations from meta-reviewers and PC chairs. We advocate that it is important to advance towards a system-based approach where the goal is to select a set of review board members that is diverse, covers the topics of a venue, but forms a sparse co-authorship network with low average clustering coefficient and a smaller giant component and network efficiency. Naturally, this is infeasible to achieve purely manually (by PC chairs) or purely automatically in practice. Hence, it is paramount to build data-driven, PC chair-in-the-loop tools that can facilitate it. A possible approach can be to randomly choose reviewers from a large pool of candidates, while satisfying the topic constraints. Exploring this and related options are avenues of future work.

Acknowledgement: We thank Divesh Srivastava for his insightful comments on the article and the paradigm of data-driven PC formation.
6. REFERENCES


