SIGMOD Officers, Committees, and Awardees

Chair
Juliana Freire
Computer Science & Engineering
New York University
Brooklyn, New York
USA
+1 646 997 4128
juliana.freire <at> nyu.edu

Vice-Chair
Ihab Francis Ilyas
Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario
CANADA
+1 519 888 4567 ext. 33145
ilyas <at> uwaterloo.ca

Secretary/Treasurer
Fatma Ozcan
IBM Research
Almaden Research Center
San Jose, California
USA
+1 408 927 2737
fozcan <at> us.ibm.com

SIGMOD Executive Committee:
Juliana Freire (Chair), Ihab Francis Ilyas (Vice-Chair), Fatma Ozcan (Treasurer), K. Selçuk Candan, Yanlei Diao, Curtis Dyreson, Yannis Ioannidis, Christian Jensen, and Jan Van den Bussche.

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, Tim Kraska

SIGMOD Information Director:
Curtis Dyreson, Utah State University

Associate Information Directors:
Huiping Cao, Manfred Jeusfeld, Asterios Katsifodimos, Georgia Koutrika, Wim Martens

SIGMOD Record Editor-in-Chief:
Yanlei Diao, University of Massachusetts Amherst

SIGMOD Record Associate Editors:
Vanessa Braganholo, Marco Brambilla, Chee Yong Chan, Rada Chirkova, Zachary Ives, Anastasios Kementsietsidis, Jeffrey Naughton, Frank Neven, Olga Papaemmanouil, Aditya Parameswaran, Alkis Simitsis, Wang-Chiew Tan, Nesime Tatbul, Marianne Winslett, and Jun Yang

SIGMOD Conference Coordinator:
K. Selçuk Candan, Arizona State University

PODS Executive Committee:
Jan Van den Bussche (Chair), Tova Milo, Diego Calvanse, Wang-Chiew Tan, Rick Hull, Floris Geerts

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

Awards Committee:
Surajit Chaudhuri (Chair), David Dewitt, Martin Kersten, Maurizio Lenzerini, Jennifer Widom

Jim Gray Doctoral Dissertation Award Committee:
Ashraf Aboulnaga (co-Chair), Chris Jermaine (co-Chair), Paris Koutris, Feifei Li, Qiong Luo, Ioana Manolescu, Lucian Popa, Renée Miller

SIGMOD Systems Award Committee:
Mike Stonebraker (Chair), Make Cafarella, Mike Carey, Yanlei Diao, Paul Larson

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:

Michael Stonebraker (1992)
David DeWitt (1995)
Serge Abiteboul (1998)
Rudolf Bayer (2001)
Ronald Fagin (2004)
Jennifer Widom (2007)
Umeshwar Dayal (2010)
Stefano Ceri (2013)
Gerhard Weikum (2016)
Jim Gray (1993)
C. Mohan (1996)
Hector Garcia-Molina (1999)
Patricia Selinger (2002)
Michael Carey (2005)
Moshe Y. Vardi (2008)
Surajit Chaudhuri (2011)
Martin Kersten (2014)
Goetz Graefe (2017)
Philip Bernstein (1994)
David Maier (1997)
Rakesh Agrawal (2000)
Don Chamberlin (2003)
Jeffrey D. Ullman (2006)
Masaru Kitsuregawa (2009)
Bruce Lindsay (2012)
Laura Haas (2015)

SIGMOD Systems Award

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

Michael Stonebraker and Lawrence Rowe (2015)
Martin Kersten (2016)
Richard Hipp (2017)

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:

Maria Zemankova (1992)
Jeffrey Ullman (1996)
Raghu Ramakrishnan (1999)
Daniel Rosenkrantz (2001)
Surajit Chaudhuri (2004)
Hans-Jörg Schek (2007)
David Lomet (2010)
H.V. Jagadish (2013)
Samuel Madden (2016)
Gio Wiederhold (1995)
Avi Silberschatz (1997)
Michael Carey (2000)
Richard Snodgrass (2002)
Hongjun Lu (2005)
Klaus R. Dittrich (2008)
Gerhard Weikum (2011)
Kyu-Young Whang (2014)
Yannis E. Ioannidis (2017)
Yahiko Kambayashi (1995)
Won Kim (1998)
Laura Haas (2000)
Michael Ley (2003)
Tamer Özsu (2006)
Beng Chin Ooi (2009)
Marianne Winslett (2012)
Curtis Dyreson (2015)

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

A complete list of all SIGMOD Awards is available at: **https://sigmod.org/sigmod-awards/**

[Last updated : June 30, 2017]

Editor's Notes

Welcome to the September 2017 issue of the ACM SIGMOD Record!

First of all, we welcome Pinar Tözün to join the editorial board of the SIGMOD Record as the new associate editor of the Surveys column.

The first column of this issue is the Database Principles column, featuring an article by Guagliardo and Libkin on correctness of SQL Queries on databases with nulls. Motivated by experimental evidence that null values in a database introduce false positive answers, this article surveys efficient approximation techniques for running SQL queries on data with nulls which come with correctness guarantees. It presents two recent approximation schemes and provides theoretical guarantees for both. For the latter scheme, it also presents experimental results showing that its real-life behavior in terms of the price of correctness falls in several major categories: among them, the first, and largest, group of queries incur a small price for correctness guarantees, while the last group incurs a significant performance penalty, which has to do with how commercial optimizers handle disjunctions in queries. These results point to a real opportunity to fix many of the issues related to the handling of nulls in RDBMSs, at a reasonable cost in terms of query evaluation.

The Research column features an article by Pham et al. on uninterruptible migration of continuous queries. The elasticity brought by cloud infrastructure provides a solution for a data stream management system to handle variable workloads through scale out when heavily loaded, or scale in otherwise. Key to such a solution is an efficient mechanism that can migrate a query from one node to another with zero downtime and minimum overheads on the compute nodes. The article by Pham et al. presents a migration protocol, named UniMiCo, that satisfies those requirements and extends the state of the art with multiple stateful operators per continuous query and a variety of window definitions.

The Systems and Prototypes column features an article on stream processing for edge clouds by Esteves et al. An edge cloud is a network fabric that resides between the core network and the access network for the end user and consists of a geographically distributed network of small datacenters serving a limited number of end users. Common stream processing systems (SPS) such as Spark are designed to operate a stream processing cluster within a single datacenter, but not to span multiple data-centers (in different geographic areas). To support stream processing in edge clouds, this article presents a system based on an earlier prototype, CHive, which orchestrates multiple SPS clusters, one for each datacenter, to collectively compute a query plan. An important feature of the system is to decouple the CHive query planner and optimizer from its underlying runtime environment and support multiple runtime engines that may suit different applications.

The Distinguished Profiles column features Ron Fagin from the IBM Almaden Research Center. Ron is an IBM Fellow, Fellow of ACM and IEEE, and member of the National Academy of Engineering and the American Academy of Arts and Sciences. He has won the IEEE McDowell Award (the highest award of the IEEE Computer Society), the SIGMOD Edgar F. Codd Innovations Award, and most recently, the Gödel Prize (the highest award for a paper in Theoretical Computer Science). In this interview, Ron discusses the most important scientific results achieved in his career, including Fagin's Algorithm, Threshold Algorithm, and Fagin's 0-1 Law. He also discusses his mission as an IBM Fellow, that is, to convince theoreticians that they will prove better theorems and they will do more interesting work if they keep talking to practitioners.

The Centers column features an article by Wolfgang Lehner on the Database Systems Group at Technische Universitat Dresden, Germany. The Dresden Database Systems Group focuses on the advancement of data management techniques from a system's perspective as well as information management's perspective. The group is involved in research projects ranging from activities to exploit modern hardware for scalable storage engines to advancing statistical methods for large-scale time series management.

The issue finally closes with a call for nomination of ICDT 2018 Test-of-Time Award, which is due on November 1, 2017.

On behalf of the SIGMOD Record Editorial board, I hope that you enjoy reading the September 2017 issue of the SIGMOD Record!

Your submissions to the SIGMOD Record are welcome via the submission site:
http://sigmod.hosting.acm.org/record

Prior to submission, please read the Editorial Policy on the SIGMOD Record's website:
https://sigmodrecord.org

Yanlei Diao

September 2017

Past SIGMOD Record Editors:

Ioana Manolescu (2009-2013)
Ling Liu (2000–2004)
Arie Segev (1989–1995)
Thomas J. Cook (1981–1983)
Daniel O'Connell (1971–1973)
Alexandros Labrinidis (2007–2009)
Michael Franklin (1996–2000)
Margaret H. Dunham (1986–1988)
Douglas S. Kerr (1976-1978)
Harrison R. Morse (1969)
Mario Nascimento (2005–2007)
Jennifer Widom (1995–1996)
Jon D. Clark (1984–1985)
Randall Rustin (1974-1975)

Correctness of SQL Queries on Databases with Nulls

Paolo Guagliardo
School of Informatics
The University of Edinburgh
pguaglia@inf.ed.ac.uk

Leonid Libkin
School of Informatics
The University of Edinburgh
libkin@inf.ed.ac.uk

ABSTRACT

Multiple issues with SQL's handling of nulls have been well documented. Having efficiency as its main goal, SQL disregards the standard notion of correctness on incomplete databases – certain answers – due to its high complexity. As a result, the evaluation of SQL queries on databases with nulls may produce answers that are just plain wrong. However, SQL evaluation can be modified, at least for relational algebra queries, to approximate certain answers, i.e., return only correct answers. We examine recently proposed approximation schemes for certain answers and analyze their complexity, both theoretical bounds and real-life behavior.

1. INTRODUCTION

The way incomplete information is handled in commercial DBMSs, specifically by SQL, has been heavily criticized for producing counter-intuitive and just plain incorrect answers [4, 9]. This is often blamed on SQL's 3-valued logic (3VL), and there are multiple discussions in the literature on the relative merits of SQL's 3VL and some alternatives; see, e.g., [11, 33, 6]. They often try to justify a logic within itself, without having an external yardstick definition of correctness. Given the futility of such an approach, we first need to settle on what constitutes the notion of correctness.

For this, we adapt the standard approach found in the database literature: correct answers are those that we are *certain* about. Intuitively, this means that such answers will be true no matter how we interpret incomplete information that is present in the database. This approach, first proposed in the late 1970s [13, 26], is now dominant in the literature and it is standard in all applications where incomplete information appears (data integration, data exchange, ontology-based data access, data cleaning, etc.).

Why cannot SQL then just compute certain answers? The reason is that SQL's designers had first and foremost efficient evaluation in mind, but correctness and efficiency do not always get along. Computing certain answers is coNP-hard for most reasonable semantics, if we deal with relational calculus/algebra queries [2]. On the other hand, SQL evaluation is very efficient; it is in AC^0 (a small parallel complexity class) for the same class of queries, and so it provably cannot compute certain answers.

If SQL provably cannot produce what is assumed to be *the* correct answers, then what kinds of errors can it generate? To understand this, consider the simple database in Figure 1. It shows orders for books, information about customers paying for them, and basic information about customers themselves.

Decision support queries against such a database may include finding *unpaid orders*:

```
SELECT O.order_id FROM Orders O
WHERE  O.order_id NOT IN
     ( SELECT order_id FROM Payments )
```

or finding customers *who have not placed an order*:

```
SELECT C.cust_id FROM Customers C
WHERE  NOT EXISTS
     ( SELECT * FROM Orders O, Payments P
       WHERE  C.cust_id = P.cust_id
         AND  P.order_id = O.order_id )
```

As expected, the first query produces a single answer Ord3, while the second returns the empty table. But now assume that just a single entry in these tables is replaced by `NULL`: specifically, the value of `order_id` in the second tuple of `Payments` changes from Ord2 to `NULL`. Then the answers to queries change drastically, and in *different ways*: now the unpaid orders query returns the empty table, and the customers without an order query returns Cust2. That is, due to the presence of nulls, we can both miss answers, and invent new answers!

Let us analyze this in more detail. If we consider certain answers as the correct behavior of query answering over incomplete databases, then SQL evaluation can differ from it in two ways:

- SQL can miss some of the tuples that belong to certain answers, thus producing *false negatives*; or
- it can return some tuples that do not belong to certain answers, that is, *false positives*.

In the previous example, Cust2 returned by the second

ORDERS

order_id	*title*	*price*
Ord1	Big Data	30
Ord2	SQL	35
Ord3	Logic	50

PAYMENTS

cust_id	*order_id*
Cust1	Ord1
Cust2	Ord2

CUSTOMERS

cust_id	*name*
Cust1	John
Cust2	Mary

Figure 1: A database of orders, payments, and customers.

query is a false positive. The unpaid orders query does not generate any false negatives: certain answers are actually empty since we cannot know which order was unpaid. But a simple query

```
SELECT cust_id FROM Payments
WHERE order_id = 'Ord2' OR order_id <> 'Ord2'
```

returns only Cust1 in the database with null as described above, while the certain answer is $\{$ Cust1, Cust2 $\}$.

To sum up, SQL cannot compute certain answers due to the complexity gap. Furthermore, it can produce *both* false positives and false negatives. However, the gap in complexity does not yet justify such a behavior: it leaves open the possibility that a query evaluation scheme produces *only one type of undesirable results*.

For now we take the view that false positives are the worst of the two: after all, they produce an outright lie as opposed to hiding some of the truth. We admit that an alternative point of view has merits too [6], and in fact we shall address it later. One can accept one type of errors – false negatives – as the price to be paid for lowering complexity.

This idea is not new: it was first explored more than 30 years ago [29, 32]. Those papers assumed the model of databases as logical theories and could not lead to implementations that would handle familiar relational databases with nulls. Some ad hoc translations of SQL queries were studied later but without any formally proved correctness guarantees [19].

The first approach to fixing SQL's evaluation scheme that provides provable correctness guarantees was presented surprisingly recently, in [25] (with the conference version appearing in 2015). It showed how to translate a relational algebra query Q into a query $Q^{\mathbf{t}}$ of **t**rue answers such that:

- false positives never occur: $Q^{\mathbf{t}}$ returns a subset of certain answers to Q;
- data complexity of $Q^{\mathbf{t}}$ is still AC^0; and
- on databases without nulls, Q and $Q^{\mathbf{t}}$ coincide.

Given the attractive theoretical properties of the approach, it was natural to ask two questions. First, do we address a real problem, that is, do false positives occur in real queries? And second, do theoretical guarantees of the approach translate into good behavior in practice? What is the price to pay, in terms of query evaluation performance, for correctness guarantees?

These questions were addressed in [15]. It provided experimental evidence that false positives are indeed a real-life problem. It then noticed that the translation of [25] cannot be implemented as-is: queries in translations tend to build very large Cartesian products and are thus impractical, despite very good theoretical complexity bounds.

To remedy this, [15] proposed a new translation $Q \mapsto (Q^+, Q^?)$ of relational algebra queries. The query Q^+ shares the desirable properties of $Q^{\mathbf{t}}$, and $Q^?$ addresses the alternative point of view that false negatives are evil: it eliminates false negatives but can produce false positives instead. The translations are by mutual recursion, hence one cannot define Q^+ without $Q^?$ and vice versa.

Algorithms in [15] introduced extra steps to restore correctness. We do not, therefore, expect them to outperform native SQL evaluation, which was designed to optimize execution time. We can hope, however, that the overhead is sufficiently small. If this is so, one can envision two modes of evaluation: the standard one, where efficiency is the only concern, and an alternative, perhaps slightly more expensive one, that provides correctness guarantees. The difference between the two is the *price of correctness*.

The goal of this short survey is to report recent developments in finding efficient approximations of SQL queries on databases with nulls that come with correctness guarantees. Experimental evidence shows that false positives are a real issue. We present the approximation schemes of [25] and [15] and provide theoretical guarantees for both. For the latter, we also present an experimental evaluation showing that its real-life behavior in terms of the price of correctness falls into three major categories:

- for the first, and largest, group of queries, the price of correctness is small, as was indeed hoped (the overhead ranges between 1% and 4%);
- for another group, somewhat surprisingly, there is a significant improvement in performance despite the query performing additional checks;
- for the last group, performance becomes an issue, but it has to do with the well documented issues in the way commercial optimizers handle disjunctions in queries [5]; depending on the size of the data-

base, the approximating query Q^+ runs at between one quarter to half the speed of Q.

These results point to a real opportunity to fix many of the issues related to the handling of nulls in RDBMSs, at a reasonable cost in terms of query evaluation.

2. PRELIMINARIES

We consider incomplete databases with nulls interpreted as missing values. Much of the following is standard in the literature on databases with incomplete information; see, e.g., [1, 18, 31]. The usual way of modeling missing values in a database is to use *marked* (or labeled) nulls, which often appear in applications such as data integration and exchange [3, 22]. In this model, databases are populated by two types of elements: *constants* and *nulls*, coming from countably infinite sets denoted by Const and Null, respectively. Nulls are denoted by $\bot$, sometimes with sub- or superscripts. For the purpose of the general model we follow the textbook approach assuming one domain Const for all non-null elements appearing in databases. In real life, such elements can be of many different types, and those appearing in the same column must be of the same type. Adjusting results and translations of queries for this setting is completely straightforward.

A relational schema is a set of relation names with associated arities (numbers of attributes). With each k-ary relation symbol S from the vocabulary, an incomplete relational instance D associates a k-ary relation S^D over $\mathsf{Const} \cup \mathsf{Null}$, that is, a finite subset of $(\mathsf{Const} \cup \mathsf{Null})^k$. When the instance is clear from the context, we write S instead of S^D for the relation itself. We denote the arity of S by $\mathsf{ar}(S)$, and use the same notation for queries. Note that we now assume *set* semantics of queries; we shall comment on *bag* semantics, which is used in real-life DBMSs, in Section 5.

The sets of constants and nulls that occur in a database D are denoted by $\mathsf{Const}(D)$ and $\mathsf{Null}(D)$, respectively. The *active domain* of D is the set $\mathsf{adom}(D)$ of all elements occurring in it, that is, $\mathsf{Const}(D) \cup \mathsf{Null}(D)$. If D has no nulls, we say that it is *complete*. A *valuation* v on a database D is a map $v : \mathsf{Null}(D) \to \mathsf{Const}$. We denote by $v(D)$ the result of replacing each null $\bot$ with $v(\bot)$ in D. An incomplete database represents the collection of complete databases $\{v(D) \mid v \text{ is a valuation}\}$; this is often referred to as the closed-world semantics of incompleteness [28].

Query languages. As our query language, we consider relational algebra with the standard operations of selection σ, projection π, Cartesian product $\times$ (or join $\bowtie$), union $\cup$, and difference $-$. This corresponds to the basic fragment of SQL – which we use in the experiments of Section 4 – consisting of the usual `SELECT-FROM-WHERE` queries, with (correlated) subqueries preceded by `IN` and `EXISTS`, as well as their negations. We shall comment in more detail about the correspondence between SQL and relation algebra in Section 5.

We assume that selection conditions are positive Boolean combinations of equalities of the form $A = B$ and $A = c$, where A and B are attributes and c is a constant value, and disequalities $A \neq B$ and $A \neq c$. Note that these conditions are closed under negation, which can simply be propagated to atoms: e.g., $\neg\big((A = B) \vee (B \neq 1)\big)$ is equivalent to $(A \neq B) \wedge (B = 1)$.

We also use conditions $\mathsf{const}(A)$ and $\mathsf{null}(A)$ in selections, indicating whether the value of an attribute is a constant or a null. These correspond to `A IS NOT NULL` and `A IS NULL` in SQL.

Correctness guarantees. The standard notion of correct query answering on incomplete databases is *certain answers*, that is, tuples that are present in the answer to a query regardless of the interpretation of nulls. For a query Q and a database D, they are typically defined as tuples $\bar{a}$ that are present in $Q(v(D))$ for all valuations v; see [1, 18].

This definition has a serious drawback, though, as tuples with nulls cannot be returned, while standard query evaluation may well produce such tuples. For instance, if we have a relation $R = \{(1, \bot), (2, 3)\}$, and a query returning R, then the only certain answer according to the above definition is $(2, 3)$, while intuitively we would expect the entire relation.

In light of this, we use a closely-related but more general notion from [27], called *certain answers with nulls* in [25]. Formally, for a query Q and a database D, these are tuples $\bar{a}$ over $\mathsf{adom}(D)$ such that $v(\bar{a}) \in Q\big(v(D)\big)$ for every valuation v on D. The set of all such tuples is denoted by $\mathsf{cert}(Q, D)$. In the above example, the certain answers with nulls are $(1, \bot)$ and $(2, 3)$. The standard certain answers are exactly the null-free tuples in $\mathsf{cert}(Q, D)$ [25].

Definition 1. A query evaluation algorithm has *correctness guarantees* for query Q if for every database D it returns a subset of $\mathsf{cert}(Q, D)$.

In other words, with correctness guarantees, false positives are not allowed: all returned tuples must be certain answers.

Often our evaluation algorithms will be of the following form: translate a query Q into another query Q', and then run Q' on D. If $Q'(D) \subseteq \mathsf{cert}(Q, D)$ for every D, then we say that Q' has correctness guarantees for Q.

Some results concerning correctness guarantees are known. By *naïve evaluation* for a fragment of relational algebra we mean the algorithm that treats elements of Null as if they were the usual database entries, i.e., each

evaluation $\bot = c$ for $c \in \mathsf{Const}$ is false and $\bot = \bot'$ is true iff $\bot$ and $\bot'$ are the same element in Null.

Recall that the *positive* fragment of relational algebra is the fragment without the difference operator and without disequalities in selection conditions. It corresponds to the fragment of SQL in which negation does not appear in any form, i.e., **EXCEPT** is not allowed, there are no negations in **WHERE** conditions and the use of **NOT IN** and **NOT EXISTS** for subqueries is prohibited.

FACT 1 ([12, 18, 25]). *For positive relational algebra queries, naïve evaluation computes exactly certain answers with nulls, and thus it has correctness guarantees. This remains true even if we extend the language with the division operator as long as its second argument is a relation in the database.*

Recall that division is a derived relational algebra operation; it computes tuples in a projection of a relation appearing in all possible combinations with tuples from another relation (e.g., 'find students taking all courses').

SQL evaluation. The query evaluation procedure in SQL is different from naïve evaluation: it is based on a 3-valued logic. Comparisons such as $\bot = c$, or $\bot = \bot'$, evaluate to *unknown*, which is then propagated through conditions using the rules of 3VL.

More precisely, selection conditions can evaluate to true (**t**), false (**f**), or unknown (**u**). If at least one attribute in a comparison is null, the result of the comparison is **u**. The interaction of **u** with Boolean connectives follows the rules of SQL's 3VL (which is Kleene's 3-valued logic) shown below for the cases when **u** is involved:

$$\neg \mathbf{u} = \mathbf{u} \qquad \mathbf{u} \wedge \mathbf{f} = \mathbf{f} \qquad \mathbf{u} \vee \mathbf{t} = \mathbf{t}$$
$$\mathbf{u} \wedge \mathbf{t} = \mathbf{u} \qquad \mathbf{u} \wedge \mathbf{u} = \mathbf{u}$$
$$\mathbf{u} \vee \mathbf{f} = \mathbf{u} \qquad \mathbf{u} \vee \mathbf{u} = \mathbf{u}$$

Then, σ_θ selects tuples on which θ evaluates to **t** (that is, **f** and **u** tuples are not selected). We refer to the result of evaluating a query Q in this way as $\mathsf{Eval}_{\mathsf{SQL}}(Q, D)$.

FACT 2 ([25]). $\mathsf{Eval}_{\mathsf{SQL}}$ *has correctness guarantees for positive relational algebra.*

Thus, it is the negation in queries – that may appear in various forms – that causes SQL's behavior to deviate from correct answers. Not surprisingly, all the example queries in the introduction used some form of negation.

3. APPROXIMATION SCHEMES WITH CORRECTNESS GUARANTEES

Due to the high complexity of certain answers, we must settle for approximations that can be computed efficiently. As we have seen, although efficient, standard SQL evaluation may produce answers that are not certain, so we need alternative evaluation schemes that have correctness guarantees and tractable complexity.

One such scheme was first devised in [25], but despite its promising complexity bounds it was not effectively applicable in practice. For this reason, [15] proposed a new evaluation scheme with correctness guarantees and the same theoretical complexity of the previous one, but that can also be implemented efficiently.

We will now present and discuss these two schemes in more detail for queries expressed in relational algebra.

3.1 A simple translation

The key idea of the approximation scheme of [25] is to translate a query Q into a pair $(Q^{\mathbf{t}}, Q^{\mathbf{f}})$ of queries that have correctness guarantees for Q and its complement $\overline{Q}$, respectively. That is, tuples in $Q^{\mathbf{t}}(D)$ are certainly true, and tuples in $Q^{\mathbf{f}}(D)$ are certainly false:

$$Q^{\mathbf{t}}(D) \subseteq \mathsf{cert}(Q, D) \tag{1}$$
$$Q^{\mathbf{f}}(D) \subseteq \mathsf{cert}(\overline{Q}, D) \tag{2}$$

To describe the translation, we need the following.

Definition 2. Two tuples $\bar{r}$ and $\bar{s}$ of the same length over $\mathsf{Const} \cup \mathsf{Null}$ are *unifiable*, written as $\bar{r} \Uparrow \bar{s}$, if there exists a valuation v of nulls such that $v(\bar{r}) = v(\bar{s})$.

For example, $(\bot, 2, \bot') \Uparrow (2, \bot, 3)$ with the valuation $v(\bot) = 2$ and $v(\bot') = 3$, but $(\bot, 3, \bot')$ and $(2, \bot, 3)$ do not unify. Checking whether tuples unify is very efficient: it can be done in linear time, and in fact can be expressed by a condition in **WHERE**.

The translations of [25] are shown in Figure 2, where adom refers to the query computing the active domain. For a single relation R with attributes $A_1, \ldots, A_n$, this is $\mathsf{adom}(R) = \pi_{A_1}(R) \cup \ldots \cup \pi_{A_n}(R)$, and for a database D with relations $R_1, \ldots, R_m$, it is $\mathsf{adom}(D) = \mathsf{adom}(R_1) \cup \ldots \cup \mathsf{adom}(R_m)$. Recall that $\mathsf{ar}(Q)$ is the arity of Q, so $\mathsf{adom}^{\mathsf{ar}(Q)}$ refers to the Cartesian product $\mathsf{adom} \times \ldots \times \mathsf{adom}$ taken $\mathsf{ar}(Q)$ times.

The translation also uses conditions θ^* which are obtained by translating selection conditions θ as defined inductively by the following rules:

$$\begin{aligned}
(A = B)^* &= (A = B) \\
(A = c)^* &= (A = c) \quad \text{if } c \text{ is a constant} \\
(A \neq B)^* &= (A \neq B) \wedge \mathsf{const}(A) \wedge \mathsf{const}(B) \\
(A \neq c)^* &= (A \neq c) \wedge \mathsf{const}(A) \\
(\theta_1 \vee \theta_2)^* &= \theta_1^* \vee \theta_2^* \\
(\theta_1 \wedge \theta_2)^* &= \theta_1^* \wedge \theta_2^*
\end{aligned}$$

THEOREM 1 ([25]). *The translations of Figure 2 have correctness guarantees: (1) and (2) hold. Moreover, both queries $Q^{\mathbf{t}}$ and $Q^{\mathbf{f}}$ have AC^0 data complexity, and $Q^{\mathbf{t}}(D) = Q(D)$ for complete databases.*

$$
\begin{array}{rcl@{\qquad}rcl}
R^{\mathbf{t}} & = & R & R^{\mathbf{f}} & = & \{\, \bar{s} \in \mathsf{adom}^{\mathsf{ar}(R)} \mid \nexists\, \bar{r} \in R\colon \bar{r} \Uparrow \bar{s} \,\} \\
(Q_1 \cup Q_2)^{\mathbf{t}} & = & Q_1^{\mathbf{t}} \cup Q_2^{\mathbf{t}} & (Q_1 \cup Q_2)^{\mathbf{f}} & = & Q_1^{\mathbf{f}} \cap Q_2^{\mathbf{f}} \\
(Q_1 - Q_2)^{\mathbf{t}} & = & Q_1^{\mathbf{t}} \cap Q_2^{\mathbf{f}} & (Q_1 - Q_2)^{\mathbf{f}} & = & Q_1^{\mathbf{f}} \cup Q_2^{\mathbf{t}} \\
\big(\sigma_\theta(Q)\big)^{\mathbf{t}} & = & \sigma_{\theta^*}(Q^{\mathbf{t}}) & (\sigma_\theta(Q))^{\mathbf{f}} & = & Q^{\mathbf{f}} \cup \sigma_{(\neg\theta)^*}\big(\mathsf{adom}^{\mathsf{ar}(Q)}\big) \\
(Q_1 \times Q_2)^{\mathbf{t}} & = & Q_1^{\mathbf{t}} \times Q_2^{\mathbf{t}} & (Q_1 \times Q_2)^{\mathbf{f}} & = & Q_1^{\mathbf{f}} \times \mathsf{adom}^{\mathsf{ar}(Q_2)} \cup \mathsf{adom}^{\mathsf{ar}(Q_1)} \times Q_2^{\mathbf{f}} \\
\big(\pi_\alpha(Q)\big)^{\mathbf{t}} & = & \pi_\alpha(Q^{\mathbf{t}}) & \big(\pi_\alpha(Q)\big)^{\mathbf{f}} & = & \pi_\alpha(Q^{\mathbf{f}}) - \pi_\alpha\big(\mathsf{adom}^{\mathsf{ar}(Q)} - Q^{\mathbf{f}}\big)
\end{array}
$$

Figure 2: Relational algebra translations of [25].

While (1) and (2) ensure correctness guarantees for all relational algebra queries, and queries $Q^{\mathbf{t}}$ and $Q^{\mathbf{f}}$ have good theoretical complexity, they suffer from a number of problems that severely hinder their practical implementation. Crucially, they require the computation of active domains and, even worse, their Cartesian products. While expressible in relational algebra, the $Q^{\mathbf{f}}$ translations for selections, products, projections, and even base relations become prohibitively expensive. Several optimizations have been suggested in [25] (at the price of missing some certain answers), but the cases of projection and base relations do not appear to have any reasonable alternatives. Yet another problem is the complicated structure of the queries $Q^{\mathbf{f}}$. When translations are applied recursively, this leads to very complex queries $Q^{\mathbf{t}}$ if Q used difference.

In fact we tried a simple experiment with the translations in Figure 2, and found that they are already infeasible for very small databases: some of the queries start running out of memory on instances with fewer than 10^3 tuples.

All this tells us that good theoretical complexity is not yet a guarantee of real-life efficiency, and we need an implementable alternative, which we present next.

3.2 An implementation-friendly translation

To overcome the practical difficulties posed by the translation in Figure 2, [15] proposed an alternative translation that is implementation-friendly and comes with sufficient correctness guarantees. This translation does not produce a second query $Q^{\mathbf{f}}$ that underapproximates certain answers to the negation of the query, which was the main source of complexity. To see what we can replace it with, note that, in the $Q^{\mathbf{t}}$ translation, $Q^{\mathbf{f}}$ was only used in the rule for difference: a tuple $\bar{a}$ is a certain answer to $Q_1 - Q_2$ if

1. $\bar{a}$ is a certain answer to Q_1, and
2. $\bar{a}$ is a certain answer to the complement of Q_2.

That necessitated working with the complex $Q^{\mathbf{f}}$ translation.

But we can use a slightly different rule: a tuple $\bar{a}$ is a certain answer to $Q_1 - Q_2$ if

1. $\bar{a}$ is a certain answer to Q_1, and
2. $\bar{a}$ does not match any tuple that could possibly be an answer to Q_2.

The advantage of this is that the query that approximates possible answers can be built in a much simpler way than $Q^{\mathbf{f}}$. For instance, for a base relation R, it will be just R itself, as opposed to the complex expression involving adom we used before. Then the rule for $Q_1 - Q_2$ involves a left anti-semijoin (to be defined soon) of the approximation of certain answers to Q_1 and possible answers to Q_2.

We need to formally say what "(not) matching possible answers" means. To this end, we define approximations of possible answers and two matching-based semijoin operators. There already exists a notion of *maybe-answers* [2, 31] – answers that appear in $Q\big(v(D)\big)$ for at least one valuation v – but those can be infinite, and include arbitrary elements outside of $\mathsf{adom}(D)$. What we need instead is a compact representation.

Definition 3. Given a k-ary query Q and an incomplete database D, we say that a set $A \subseteq \mathsf{adom}(D)^k$ *represents potential answers* to Q on D if $Q\big(v(D)\big) \subseteq v(A)$ for every valuation v. A query Q' *represents potential answers* to Q if $Q'(D)$ represents potential answers to Q on D, for every D.

Obviously, there are trivial ways of representing potential answers: take, e.g., $\mathsf{adom}(D)^k$. But we shall be looking for good approximations, just as we are looking for good approximations of $\mathsf{cert}(Q, D)$, for which bad ones can also be found easily (e.g., the empty set). In general, testing if a set A represents potential answers to a query is computationally hard:

PROPOSITION 1 ([15]). *There is a fixed relational algebra query Q such that the following problem is* coNP*-complete: given a database D and a set A of tuples over $\mathsf{adom}(D)$, does A represent potential answers to Q on D?*

$$R^+ = R \tag{3.1}$$
$$(Q_1 \cup Q_2)^+ = Q_1^+ \cup Q_2^+ \tag{3.2}$$
$$(Q_1 - Q_2)^+ = Q_1^+ \mathbin{\overline{\ltimes}_{\Uparrow}} Q_2^? \tag{3.3}$$
$$\big(\sigma_\theta(Q)\big)^+ = \sigma_{\theta^*}(Q^+) \tag{3.4}$$
$$(Q_1 \times Q_2)^+ = Q_1^+ \times Q_2^+ \tag{3.5}$$
$$\big(\pi_\alpha(Q)\big)^+ = \pi_\alpha(Q^+) \tag{3.6}$$

$$R^? = R \tag{4.1}$$
$$(Q_1 \cup Q_2)^? = Q_1^? \cup Q_2^? \tag{4.2}$$
$$(Q_1 - Q_2)^? = Q_1^? - Q_2^+ \tag{4.3}$$
$$\big(\sigma_\theta(Q)\big)^? = \sigma_{\theta^{**}}\big(Q^?\big) \tag{4.4}$$
$$(Q_1 \times Q_2)^? = Q_1^? \times Q_2^? \tag{4.5}$$
$$\big(\pi_\alpha(Q)\big)^? = \pi_\alpha\big(Q^?\big) \tag{4.6}$$

Figure 3: Improved relational algebra translations of [15].

However, we shall see that potential answers can be efficiently approximated, which is what we need for the translation.

To express conditions involving matching, we shall need two semijoin operations based on unifiable tuples (see Definition 2).

Definition 4. For relations R, S over $\mathsf{Const} \cup \mathsf{Null}$, with the same set of attributes, the *left unification semijoin* is

$$R \ltimes_{\Uparrow} S = \big\{\, \bar{r} \in R \mid \exists\, \bar{s} \in S \colon \bar{r} \Uparrow \bar{s} \,\big\}$$

and the *left unification anti-semijoin* is

$$R \mathbin{\overline{\ltimes}_{\Uparrow}} S = R - (R \ltimes_{\Uparrow} S) = \big\{\, \bar{r} \in R \mid \nexists\, \bar{s} \in S \colon \bar{r} \Uparrow \bar{s} \,\big\}$$

These are similar to the standard definition of (anti) semijoin; we simply use unifiability of tuples as the join condition. They are definable operations: we have that $R \ltimes_{\Uparrow} S = \pi_R\big(\sigma_{\theta_{\Uparrow}}(R \times S)\big)$, where the projection is on all attributes of R and condition $\theta_{\Uparrow}$ is true for a tuple $\bar{r}\bar{s} \in R \times S$ iff $\bar{r} \Uparrow \bar{s}$. The unification condition $\theta_{\Uparrow}$ is expressible as a selection condition using predicates const and null [25]. Note that, in this notation, $R^{\mathbf{f}}$ of Figure 2 is $\mathsf{adom}^{\mathsf{ar}(R)} \mathbin{\overline{\ltimes}_{\Uparrow}} R$.

We now define the translation $Q \mapsto (Q^+, Q^?)$. For Q^+ with correctness guarantees, all of the rules are the same as in Figure 2, except the one for difference, which becomes

$$(Q_1 - Q_2)^+ \;=\; Q_1^+ \mathbin{\overline{\ltimes}_{\Uparrow}} Q_2^?$$

This is precisely the set of tuples certainly in Q_1 that do not match potential answers to Q_2.

For queries $Q^?$, the translation follows the structure of the query closely, but it needs a different translation of selection conditions: $\theta \mapsto \theta^{**}$ is given by $\theta^{**} = \neg(\neg\theta)^*$. Recall that negating selection conditions means propagating negations through them, and interchanging $=$ and $\neq$, and const and null. For completeness, we provide it below:

$$\begin{aligned}
(A \neq B)^{**} &= (A \neq B) \\
(A \neq c)^{**} &= (A \neq c) \quad \text{if } c \text{ is a constant} \\
(A = B)^{**} &= (A = B) \vee \mathsf{null}(A) \vee \mathsf{null}(B) \\
(A = c)^{**} &= (A = c) \vee \mathsf{null}(A) \\
(\theta_1 \vee \theta_2)^{**} &= \theta_1^{**} \vee \theta_2^{**} \\
(\theta_1 \wedge \theta_2)^{**} &= \theta_1^{**} \wedge \theta_2^{**}
\end{aligned}$$

The full translation is given in Figure 3.

THEOREM 2 ([15]). *For the translation $Q \mapsto (Q^+, Q^?)$ in Figure 3, the query Q^+ has correctness guarantees for Q, and $Q^?$ represents potential answers to Q.*

In particular, $Q^+(D) \subseteq \mathsf{cert}(Q, D)$ and

$$v\big(Q^+(D)\big) \;\subseteq\; Q\big(v(D)\big) \;\subseteq\; v\big(Q^?(D)\big) \tag{5}$$

for every database D and every valuation v.

The theoretical complexity bounds for queries Q^+ and $Q^{\mathbf{t}}$ are the same: both have the low AC^0 data complexity. However, the real world performance of Q^+ will be significantly better, as it completely avoids large Cartesian products.

We conclude this section with a few remarks. First, the translation of Figure 3 is really a family of translations: our result is more general.

COROLLARY 1. *If in the translation in Figure 3 one replaces the right sides of rules by queries*

- *contained in those listed in* (3.1)–(3.6)*, and*
- *containing those listed in* (4.1)–(4.6)*,*

then the resulting translation continues to satisfy the claim of Theorem 2.

This opens up the possibility of optimizing translations (at the expense of potentially returning fewer tuples). For instance, if we modify the translations of selection conditions so that θ^* is a stronger condition than the original and θ^{**} is a weaker one, we retain overall correctness guarantees. In particular, the unification condition $\theta_{\Uparrow}$ is expressed by a case analysis that may become onerous for tuples with many attributes; the above observation can be used to simplify the case analysis while retaining correctness.

Next, we turn to the comparison of Q^+ with the result of SQL evaluation, i.e., $\mathsf{Eval}_{\mathsf{SQL}}(Q, D)$. Given that the

latter can produce both types of errors – false positives and false negatives – it is not surprising that the two are in general incomparable. To see this, consider first a database D_1 where $R = \{(1,2),(2,\bot)\}$, $S = \{(1,2),(\bot,2)\}$ and $T = \{(1,2)\}$, and the query $Q_1 = R - (S \cap T)$. The tuple $(2,\bot)$ belongs to $\mathsf{Eval}_{\mathsf{SQL}}(Q_1, D)$ and it is a certain answer, while $Q_1^+(D) = \varnothing$. On the other hand, for D_2 with $R = \{(\bot,\bot)\}$ over attributes A, B, and $Q_2 = \sigma_{A=B}(R)$, the tuple $(\bot,\bot)$ belongs to $Q_2^+(D_2)$, but $\mathsf{Eval}_{\mathsf{SQL}}(Q_2, D_2) = \varnothing$.

4. EXPERIMENTAL EVALUATION

We now report on the experiments carried out in [15] that answer two questions posed in the introduction: Do false positives occur in real-life queries? Does the approximation scheme $(Q^+, Q^?)$ perform well in practice?

We have seen that what breaks correctness guarantees is queries with negation; the example in the introduction was based on a `NOT EXISTS` subquery. To choose concrete SQL queries for our experiments, we consider the well established TPC-H benchmark that models a business application scenario and typical decision support queries [30]. Its schema contains information about customers who place orders consisting of several items, and suppliers who supply parts for those orders.

Only few TPC-H queries use `NOT EXISTS`, so we supplement them with very typical database textbook [10] queries (slightly modified to fit the TPC-H schema) that are designed to teach subqueries.

Another issue is that the standard TPC-H data generator, DBGen, only produces instances without nulls, so we need to insert nulls to make them fit for our purpose. To this end, we separate attributes into nullable and non-nullable ones; the latter are those where nulls cannot occur (due to primary key constraints, or `NOT NULL` declarations). For nullable attributes, we choose a probability, referred to as the *null rate* of the resulting instance, and simply flip a coin to decide whether the corresponding value is to be replaced by a null. The resulting instances contain a percentage of nulls in nullable attributes that is roughly equal to the null rate with which nulls are generated. We consider null rates in the range 0.5%–10%.

The smallest instance DBGen generates is about 1GB in size, containing just under $9 \cdot 10^6$ tuples. We measured the relative performance of our translated queries w.r.t. the original ones on instances of size comprised between 1GB and 10GB.

Estimating the amount of false positives in query answers in queries is trickier, since finding certain answers is computationally hard. We overcome this difficulty by using ad hoc algorithms for the specific queries we experiment with, and by using smaller instances generated by a configurable data generator, DataFiller [7]. These instances are compliant with the TPC-H specification in everything but size, which we scale down by a factor of 10^3. For additional details of the experimental setup, we refer to [15].

4.1 How many false positives?

A false positive answer is a tuple that is returned by the SQL evaluation and yet is not certain; that is, the set of false positives produced by a query Q on a database D is $Q(D) - \mathsf{cert}(Q, D)$. They only occur on databases with nulls; on complete databases, $Q(D) = \mathsf{cert}(Q, D)$. A simple example was given in the introduction; our goal now is to see whether real-life queries indeed produce false positives. For this, we shall run our test queries on generated instances with nulls and compare their output with certain answers. As explained above, for each test query we designed a specialized algorithm to detect (some of the) false positives. This will tell us that at least some percentage of SQL answers are false positives.

Recall that null values in instances are randomly generated: each nullable attribute can be null with the same fixed probability, referred to as the null rate. To get good estimates, we generated 100 instances for each null rate in the range 0.5%–10%, and we ran each query 5 times, with randomly generated values for its parameters. At each execution, a lower bound on the percentage of false positives is calculated by means of the algorithms mentioned above.

The outcome of the experiment showed that the problem of incorrect query answers in SQL is not just theoretical but it may well occur in practical settings: every single query we tested produced false positives on incomplete databases with as low as 0.5% of null values. In extreme cases, false positives constitute almost the totality of answers, even when few nulls are present. Other queries appear to be more robust (as we only find a lower bound on the number of false positives), but the overall conclusion is clear: *false positives do occur in answers to very common queries with negation, and account for a significant portion of the answers*.

4.2 The price of correctness

Our goal was to test whether the translation $Q \mapsto Q^+$ works in practice. For this, we executed our test queries and their translations with correctness guarantees on randomly generated incomplete TPC-H instances to compare their performance.

The translation $Q \mapsto Q^+$ was given at the level of relational algebra. While there are multiple relational algebra simulators freely available, we carried out our experiments using a real DBMS on instances of realistic size (which rules out relational algebra simulators). Thus, we took test SQL queries, applied the translation $Q \mapsto Q^+$ to their relational algebra equivalents, and

then ran the results of the translation as SQL queries.

Note that we measured the *relative performance* of the correct translations Q^+s, i.e., the ratio between the running times of Q^+ and of the original queries Q. We used the DBGen tool to generate instances and populated them with nulls, depending on the prescribed null rate. For each null rate in the range 1%–5%, in steps of 1%, we generate multiple incomplete instances, and ran queries multiple times for randomly generated values of their parameters. The reported results were averages of those runs.

Regarding the size of instances, it seems, intuitively, that the *ratio* of execution times of Q^+ and Q should not significantly depend on the size of the generated instances. With this hypothesis in mind, we first did a detailed study for the smallest allowed size of TPC-H instances (roughly 1GB). After that, we tested our hypothesis using instances up to 10GB. For the majority of queries relative performances indeed remained about the same for all instance sizes as we expected, although we did find an exception (we shall discuss this later).

One of the key changes that our translation introduces is to convert conditions of the form `A=B` to

```
A=B OR A IS NULL OR B IS NULL
```

inside correlated **NOT EXISTS** subqueries. The reason for this should be clear when one looks at the translation $\theta \mapsto \theta^{**}$ of conditions in queries $Q^?$. This is the translation that is applied to negated subqueries, due to the rule $(Q_1 - Q_2)^+ = Q_1^+ \,\overline{\ltimes}_{\Uparrow}\, Q_2^?$, thus resulting in such disjunctions.

In general, and this has nothing to do with our translation, when several such disjunctions occur in a subquery, they may not be handled well by the optimizer [5]. One could in fact observe that for a query of the form

```
SELECT * FROM R WHERE NOT EXISTS
  ( SELECT * FROM   S, ..., T
    WHERE  ( A=B OR A IS NULL OR B IS NULL )
             AND ··· AND
           ( X=Y OR X IS NULL OR Y IS NULL ) )
```

the estimated cost of the query plan can be thousands of times higher than for the same query from which the **IS NULL** conditions are removed.

One way to overcome this is quite simple and takes advantage of the fact that such disjunctions will occur inside **NOT EXISTS** subqueries. We can then propagate disjunctions in the subquery, which results in a **NOT EXISTS** condition of the form $\neg\exists\bar{x} \bigvee \phi_i(\bar{x})$, where each ϕ_i now is a conjunction of atoms. This in turn can be split into conjunctions of $\neg\exists\bar{x}\,\phi_i(\bar{x})$, ending up with a query of the form

```
SELECT * FROM R WHERE NOT EXISTS
    ( SELECT * FROM S_i, i ∈ I_1 WHERE ⋀_j ψ_j^1 )
AND ··· AND NOT EXISTS
    ( SELECT * FROM S_i, i ∈ I_k WHERE ⋀_j ψ_j^k )
```

where formulae ψ_j^l are comparisons of attributes and statements that an attribute is or is not null, and relations S_i for $i \in I_l$ are those that contain attributes mentioned in the ψ_j^ls.

Based on the experiments we conduct, we observe three types of behavior, discussed below.

Small overhead. In half of the queries, the price of correctness is *negligible* for most applications, under 4%. The **IS NULL** disjunctions introduced by our translation are well handled by the optimizer, resulting in small overheads. In some cases, these overheads get lower as the null rate gets higher. This is most likely due to the fact that with a higher null rate it is easier to satisfy the **IS NULL** conditions in the **WHERE** clause of the **NOT EXISTS** subquery. As a result, a counterexample to the **NOT EXISTS** subquery can be found earlier, resulting in an overall faster evaluation.

Significant speedup. The translation with correctness guarantees is *much faster* than the original query; in fact we observed that it could be more than 3 orders of magnitude faster on average. This behavior arises when the translation with correctness guarantees results in decorrelated subqueries, which allows one to quickly detect that the correct answer is empty and terminate execution early, while the original query, on the other hand, spends most of its time looking for incorrect answers. In fact, this behavior was observed for queries with a rate of false positive answers close to 100%. As instances grow larger, the speedup of the translated query increases, since the original query is forced to spend more time looking for incorrect answers.

Moderate slowdown. The translated queries with correctness guarantees run at roughly half the speed of the original ones on 1GB databases. The slowdown is worse for bigger instances, increasing to about a quarter of the speed on 10GB databases, but it may still be tolerable if correctness of results is very important.

This behavior may arise when there are complex multi-way joins with large tables in **NOT EXISTS** subqueries. Without splitting the **IS NULL** disjunctions introduced by our translation, PostgreSQL produces query plans with astronomical costs, as it resorts to nested-loop joins even for large tables. This is due to the fact that it underestimates the size of joins, which is a known issue for major DBMSs [21]. In order to make the optimizer produce better estimates and a reasonable query plan, the direct translation of these queries may also require some additional hand-tuning involving common table expressions.

We conclude our experimental evaluation by addressing the standard measures for assessing the quality of approximation algorithms, namely precision and recall.

The first refers to the percentage of correct answers given. With the correctness guarantees proven in Section 3, we can state that the precision of our algorithms is 100%. Recall refers to the fraction of relevant answers returned. In our case, we can look at the certain answers returned by the standard SQL evaluation of a query Q, and see how many of them are returned by Q^+. The ratio of those is what we mean by recall in this scenario.

In some artificial examples, Q^+ may miss several, or even all, certain answers returned by Q. Thus, we cannot state a theoretical bound on the recall, but we can see what it is in the scenarios represented by our test queries. For this, we could use algorithms for identifying false positives, as explained in Section 4.1, on smaller TPC-H instances generated by DataFiller. In all those cases, the behavior we observed was that the translated queries returned precisely the answers to the original queries except false positive tuples. That is, for those instances, the recall rate was 100%, and no certain answers were missed.

5. THEORETICAL MODELS VS. REAL LIFE

We saw that good theoretical complexity bounds do not guarantee efficiency in real systems: the evaluation schemes with correctness guarantees presented in Section 3 are both very efficient in theory, yet only one of them performs well in practice. The mismatch between theoretical results and their practicality is not limited to efficiency. Before our approach [15] could be successfully applied in real life scenarios, several other important factors must be taken into account. We discuss them below.

5.1 Bag semantics and certain answers

As prescribed by the SQL Standard, relational database management systems use bag semantics in query evaluation. With bags, a tuple $\bar{a}$ can have a *multiplicity* (number of occurrences) $\#(\bar{a}, R)$ in a table R, which is a number in $\mathbb{N}$. Thus, instead of saying that a tuple is certainly in the answer, we have more detailed information: namely, the range of the numbers of occurrences of the tuple in query answers. This is captured by the following definitions, that extend the notion of certain answers with nulls:

$$\min_Q(D, \bar{a}) = \min_v \#\big(v(\bar{a}), Q(v(D))\big) \quad (6a)$$

$$\max_Q(D, \bar{a}) = \max_v \#\big(v(\bar{a}), Q(v(D))\big) \quad (6b)$$

where v ranges over valuations. Note that, if $\bar{a}$ has no nulls, $\min_Q(D, \bar{a})$ and $\max_Q(D, \bar{a})$ are simply the minimum and the maximum numbers of occurrences of $\bar{a}$ in the answer to Q over all databases $v(D)$ represented by D. Then we know with certainty that every query answer must contain at least $\min_Q(\bar{a}, D)$ occurrences of $\bar{a}$, and no answer will contain more than $\max_Q(\bar{a}, D)$ of them. When a query is evaluated under set semantics, $\min_Q(D, \bar{a}) = 1$ means that $\bar{a} \in \mathsf{cert}(Q, D)$.

Relational algebra operations under bag semantics are interpreted in a way that is consistent with SQL evaluation: union, for example, adds up occurrences and, for difference, $\#(\bar{a}, R-S) = \max\big(\#(\bar{a}, R)-\#(\bar{a}, S), 0\big)$. We refer to [14] for a survey on the subject and the full definition of all operations of relational algebra $(\sigma, \pi, \times, \cup, -)$ under bag semantics.

The complexity of the bounds (6a) and (6b) mimics analogous results for set semantics: for every relational algebra query Q interpreted under bag semantics, and for every $m \in \mathbb{N}$, checking whether $\min_Q(D, \bar{c}) > m$ or whether $\max_Q(D, \bar{c}) < m$ can be done in coNP with respect to data complexity, and the problems could be coNP-hard already without duplicates.

The difference with the set case comes when we look at *positive* relational algebra which, as before, excludes difference.[1]

THEOREM 3 ([8]). *For each positive relational algebra query Q, under bag semantics, $\min_Q(D, \bar{c})$ can be computed in polynomial time (in fact,* DLOGSPACE*) with respect to data complexity.*

However, there is a positive relational algebra query Q such that checking, for given D, $\bar{a}$, and m, whether $\max_Q(D, \bar{a}) < m$ is coNP*-complete.*

In fact, coNP-hardness is witnessed by an extremely simple query that returns a relation in a database, that is, `SELECT * FROM R`.

Next, we look at possible extensions of the approximation schemes of Section 3 to bag semantics. A simple analysis of the definition of queries $Q^{\mathbf{t}}, Q^{\mathbf{f}}$ shows that for every tuple $\bar{a}$,

$$\begin{aligned} \#\big(\bar{a}, Q^{\mathbf{t}}(D)\big) &\leq \min_Q(D, \bar{a}) \\ \#\big(\bar{a}, Q^{\mathbf{f}}(D)\big) &\leq \big(1 + \max_Q(D, \bar{a})\big) \mod 2 \end{aligned}$$

This suggests a natural extension of the translation scheme $(Q^{\mathbf{t}}, Q^{\mathbf{f}})$ to bags: we simply omit modulo 2 from addition, since it was only needed to force multiplicities to be either 0 or 1. But this is suddenly very problematic, as $\max_Q(D, \bar{a})$ is hard computationally, for *all* queries, since we cannot even compute it efficiently for base relations! Thus, implementing this approximation scheme in a real-life RDBMS (which is bag-based) is infeasible not only practically but also *theoretically* when we use bag semantics.

[1]Please note that Theorems 3 and 4, as stated in [8], referred to languages that also erroneously included duplicate elimination. However, the claims hold only when this operation is not part of the language.

On the other hand, (5) suggests a natural extension of the correctness criterion for the translation scheme $(Q^+, Q^?)$, namely:

$$\#(\bar{a}, Q^+(D)) \;\leq\; \min_Q(\bar{a}, D) \leq\; \#(\bar{a}, Q^?(D)) \quad (7)$$

for every database D and every tuple $\bar{a}$ of elements of D. Indeed, for bags B_1 and B_2, we have that $B_1 \subseteq B_2$ iff $\#(b, B_1) \leq \#(b, B_2)$ for every element b.

THEOREM 4 ([8]). *The translation $Q \mapsto (Q^+, Q^?)$ in Figure 3 satisfies (7) when queries are interpreted under bag semantics.*

In summary, the translation of Figure 2 loses its good theoretical complexity bounds and becomes intractable under bag semantics, while the approximation scheme of Figure 3 remains provably feasible also under bag semantics, thus strengthening the claim of its efficiency, practicality, and robustness.

5.2 Relational algebra vs SQL

The translations [15] in Section 3 work at the level of relational algebra, while the experimental evaluation in Section 4 was carried out with concrete SQL queries on a real DBMS. This was achieved by first translating an SQL query Q to relational algebra, applying the translation with correctness guarantees, and then translating the resulting RA query Q^+ back to SQL.

Unfortunately, database textbooks provide only a few examples of translations between SQL and RA, and detailed translations that appeared in the literature made simplifying assumptions that deviate significantly from the behavior of SQL specified by the Standard, such as the use of set semantics and the omission of nulls along with the associated three-valued logic.

Recently, [16] proposed a formal semantics of SQL that captures the core of the real language and that was experimentally validated on a very large number of randomly generated queries and databases. The semantics was applied to provide precise translations between the core fragment of SQL and relational algebra, yielding the first formal proof that they have the same expressive power. Using this formal semantics, [16] also showed that the three-valued logic of SQL is not really necessary for query evaluation, despite what is commonly believed, and that the usual Boolean logic with only true and false suffices.

The test queries we used in Section 4 go slightly beyond relational algebra as used in the translations of Figure 3. Given their decision support nature, many TPC-H queries involve aggregation, but this is not important for our purposes: if a tuple without an aggregate value is a false positive, it remains so even when an extra attribute value is added. Thus, since we only need to measure the ratio of false positives, and the *relative* change of speed in query evaluation, we can safely drop aggregates from the output of those queries. As for aggregate subqueries, we just treated them as a black box, that is, we viewed the result of such a subquery as a constant value c.

5.3 Marked nulls vs SQL nulls

The approximation schemes of [25] and [15] rely on the standard theoretical model of incompleteness where missing values in a database are represented by marked nulls. In SQL, however, we only have a single syntactic object for this purpose: `NULL`. Marked nulls are more expressive than SQL nulls, in that two unknown values can be asserted to be the same simply by denoting them with the same null. Indeed, $\bot_1 = \bot_1$ is *true* independently of which concrete value is assigned to $\bot_1$. On the other hand, the comparison `NULL` $=$ `NULL` in SQL evaluates to *unknown*, because we do not know whether the two occurrences of `NULL` refer to the same value.

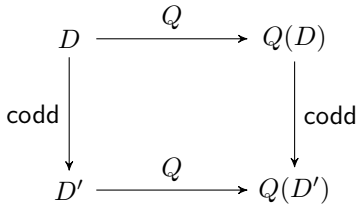
Due to the coarseness of SQL nulls, the translations Q^+ and $Q^?$ must be slightly adjusted to work correctly when evaluated as SQL queries. As expected, the adjustment occurs in selection conditions. For the θ^* translation in Q^+, we need to make sure that attributes compared for equality are not nulls (the existing translation already does that for disequality). For the θ^{**} translation in $Q^?$, the situation is symmetric: we need to include the possibility of attributes being nulls for disequality comparisons (the existing translation already does that for equality). That is, we change the translations as follows:

$$\begin{aligned}(A = B)^* &= (A = B) \wedge \mathsf{const}(A) \wedge \mathsf{const}(B)\\ (A \neq B)^{**} &= (A \neq B) \vee \mathsf{null}(A) \vee \mathsf{null}(B)\end{aligned}$$

and likewise for $(A = c)^*$ and $(A \neq c)^{**}$. Observe that, as stronger conditions are used for equality rules in θ^* and weaker ones for disequality rules in θ^{**}, by Corollary 1 the adjusted translations ensure that Q^+ continues to underapproximate certain answers and $Q^?$ continues to represent potential answers on databases with marked nulls, but now we also take into account SQL's behavior in comparisons with nulls.

There is one more issue we need to address. Usually, at least in the theoretical literature, SQL nulls are identified with *Codd nulls*, that is, marked nulls that do not repeat. The idea is to interpret each occurrence of `NULL` as a fresh marked null that does not appear anywhere else in the database. However, [17] recently showed that this way of modeling SQL nulls does not always work. If SQL nulls are to be interpreted as Codd nulls, this interpretation should apply to input databases as well as query answers, which are incomplete databases themselves. To explain this point, let $\mathsf{codd}(D)$ be the result of replacing SQL nulls in D with distinct marked nulls; as this choice is arbitrary, technically $\mathsf{codd}(D)$ is a set of databases, but these are all isomorphic. To ensure that

Codd nulls faithfully represent SQL nulls for a query Q, we need to enforce the condition in the diagram below:



Intuitively, it says the following: take an SQL database D, and compute the answer to Q on it, i.e., $Q(D)$. Now take some D' in $\mathsf{codd}(D)$, and compute $Q(D')$. Then $Q(D')$ must be in $\mathsf{codd}\big(Q(D)\big)$, that is, there must be a way of assigning Codd nulls to SQL nulls in $Q(D)$ that will result in $Q(D')$.

Unfortunately, [17] showed that this condition does not hold already for simple queries computing the Cartesian product of two relations. Furthermore, the class of relational algebra queries that transform SQL databases into Codd databases is not recursively enumerable, and therefore it is impossible to capture it by a syntactic fragment of the language. Exploiting `NOT NULL` constraints declared on the schema, [17] then proposes mild syntactic restrictions on queries that can be checked efficiently and are sufficient to guarantee the condition in the above diagram (i.e., that SQL nulls behave like Codd nulls).

We remark that the queries – and their translations – used for the experimental evaluation in Section 4 satisfy these restrictions and therefore they work correctly with the SQL implementation of nulls. However, the results of [17] tell us that in full generality we cannot guarantee correctness for all queries unless a proper implementation of marked nulls is available.

6. OUTLOOK & OPEN PROBLEMS

The main conclusion is that it is practically feasible to modify SQL query evaluation over databases with nulls to guarantee correctness of its results. This applies to the setting where nulls mean that a value is missing, and the fragment of SQL corresponds to first-order, or relational algebra, queries. We saw that the modified queries with correctness guarantees run at roughly a quarter of the speed in the worst case, to almost 10^4 times faster in the best case. For several queries, the overhead was small and completely tolerable, under 4%. With these translations, we also did not miss any of the correct answers that the standard SQL evaluation returned.

Given our conclusions that wrong answers to SQL queries in the presence of nulls are not just a theoretical myth – there are real world scenarios where this happens – and correctness can be restored with syntactic changes to queries at a price that is often tolerable, it is natural to look into the next steps that will lift our solution from the first-order fragment of SQL to cover more queries and more possible interpretations of incompleteness. We shall now discuss those.

Aggregate functions. An important feature of real-life queries is aggregation which, in fact, is present in most of the TPC-H queries. However, here our understanding of correctness of answers is quite poor; SQL's rules for aggregation and nulls are rather ad-hoc and have been persistently criticized [4, 9]. Therefore, much theoretical work is needed in this direction before practical algorithms emerge.

Incorporating constraints. In the definition of certain answers we disregarded constraints, even though every real-life database will satisfy some, typically keys and foreign keys. While a constraint ψ can be incorporated into a query ϕ by finding certain answers to $\psi \to \phi$, for common classes of constraints we would like to see how to make direct adjustments to rewritings. One example of this that we actually used in query rewriting is that the presence of a key constraint let us replace $R \,\overline{\ltimes}_{\Uparrow}\, S$ by $R - S$. Ideally such query transformations need to be automated for common classes of constraints.

Other types of incomplete information. We dealt with nulls representing missing values, but there are other interpretations. For instance, non-applicable nulls [23, 34] arise commonly as the result of outer joins. We need to extend the notion of correct query answering and translations of queries to them. One possibility is to adapt the approach of [24] that shows how to define certainty based on the semantics of inputs and outputs of queries. At the level of missing information, we would like to see whether our translations could help with deriving partial answers to SQL queries, when parts of a database are missing, as in [20].

Direct SQL rewriting. We have rewritten SQL queries by a detour via relational algebra. With the assistance of the formal semantics of [16], we should look into direct rewritings from SQL to SQL, without an intermediate language. This would also allow us to run queries with correctness guarantees directly on a DBMS.

Marked nulls in SQL. The results of [17] show us that with standard SQL nulls we can only guarantee correctness for a restricted class of queries, which cannot even be captured syntactically. To overcome this limitation, we are currently working towards extending SQL with a proper implementation of marked nulls.

Acknowledgments

This survey is based on the work originally published in [15, 25], which greatly benefited from discussions with

Marco Console, Chris Date, Hugh Darwen, Ron Fagin, Chris Ré, and Cristina Sirangelo. Work partly supported by EPSRC grants N023056 and M025268.

7. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] S. Abiteboul, P. C. Kanellakis, and G. Grahne. On the representation and querying of sets of possible worlds. *Theoretical Computer Science*, 78(1):158–187, 1991.

[3] M. Arenas, P. Barceló, L. Libkin, and F. Murlak. *Foundations of Data Exchange*. Cambridge University Press, 2014.

[4] J. Celko. *SQL for Smarties: Advanced SQL Programming*. Morgan Kaufmann, 1995.

[5] J. Claußen, A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimization and evaluation of disjunctive queries. *IEEE Trans. Knowl. Data Eng.*, 12(2):238–260, 2000.

[6] E. F. Codd and C. J. Date. Much ado about nothing. In C. J. Date, editor, *Relational Database Writings 1991–1994*. 1995.

[7] F. Coelho. DataFiller – generate random data from database schema. `https://www.cri.ensmp.fr/people/coelho/datafiller.html`.

[8] M. Console, P. Guagliardo, and L. Libkin. On querying incomplete information in databases under bag semantics. In *IJCAI*, pages 993–999. ijcai.org, 2017.

[9] C. Date and H. Darwen. *A Guide to the SQL Standard*. Addison-Wesley, 1996.

[10] C. J. Date. *An Introduction to Database Systems*. Pearson, 2003.

[11] G. H. Gessert. Four valued logic for relational database systems. *SIGMOD Record*, 19(1):29–35, 1990.

[12] A. Gheerbrant, L. Libkin, and C. Sirangelo. Naïve evaluation of queries over incomplete databases. *ACM Trans. Database Syst.*, 39(4):31:1–31:42, 2014.

[13] J. Grant. Null values in a relational data base. *Inf. Process. Lett.*, 6(5):156–157, 1977.

[14] S. Grumbach, L. Libkin, T. Milo, and L. Wong. Query languages for bags: expressive power and complexity. *SIGACT News*, 27(2):30–44, 1996.

[15] P. Guagliardo and L. Libkin. Making SQL queries correct on incomplete databases: A feasibility study. In *PODS*, pages 211–223. ACM, 2016.

[16] P. Guagliardo and L. Libkin. A formal semantics of SQL queries, its validation, and applications. *PVLDB*, 11(1), 2017.

[17] P. Guagliardo and L. Libkin. On the Codd semantics of SQL nulls. In *AMW*, 2017.

[18] T. Imielinski and W. Lipski. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[19] H. Klein. How to modify SQL queries in order to guarantee sure answers. *SIGMOD Record*, 23(3):14–20, 1994.

[20] W. Lang, R. V. Nehme, E. Robinson, and J. F. Naughton. Partial results in database systems. In *SIGMOD*, pages 1275–1286, 2014.

[21] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[22] M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, pages 233–246, 2002.

[23] N. Lerat and W. Lipski. Nonapplicable nulls. *Theor. Comput. Sci.*, 46(3):67–82, 1986.

[24] L. Libkin. Certain answers as objects and knowledge. *Artificial Intelligence*, 232:1–19, 2016.

[25] L. Libkin. SQL's three-valued logic and certain answers. *ACM TODS*, 41(1):1:1–1:28, 2016.

[26] W. Lipski. On semantic issues connected with incomplete information databases. *ACM Transactions on Database Systems*, 4(3):262–296, 1979.

[27] W. Lipski. On relational algebra with marked nulls. In *PODS*, pages 201–203, 1984.

[28] R. Reiter. On closed world data bases. In *Logic and Data Bases*, pages 55–76, 1977.

[29] R. Reiter. A sound and sometimes complete query evaluation algorithm for relational databases with null values. *Journal of the ACM*, 33(2):349–347, 1986.

[30] Transaction Processing Performance Council. *TPC Benchmark™ H Standard Specification*, Nov. 2014. Revision 2.17.1.

[31] R. van der Meyden. Logical approaches to incomplete information: A survey. In *Logics for Databases and Information Systems*, pages 307–356, 1998.

[32] M. Vardi. Querying logical databases. *Journal of Computer and System Sciences*, 33(2):142–160, 1986.

[33] K. Yue. A more general model for handling missing information in relational databases using a 3-valued logic. *SIGMOD Record*, 20(3):43–49, 1991.

[34] C. Zaniolo. Database relations with null values. *J. Comput. Syst. Sci.*, 28(1):142–166, 1984.

Uninterruptible Migration of Continuous Queries without Operator State Migration

Thao N. Pham, Nikos R. Katsipoulakis, Panos K. Chrysanthis, Alexandros Labrinidis
Department of Computer Science, University of Pittsburgh, USA
{thao, katsip, panos, labrinid}@cs.pitt.edu

ABSTRACT

The elasticity brought by cloud infrastructure provides a promising solution for a data stream management system to handle its incoming workload, which can be highly variable: the system can scale out when heavily loaded, and scale in otherwise. In such a solution, the efficiency of the mechanism used to migrate a query from one node to another is very important. Generally, a stream application requires real-time outputs for its continuous queries, and downtime is not acceptable. Moreover, the migration should not add considerable processing cost to a node that could have been already overloaded. In this paper, we present our migration protocol, named UniMiCo, which satisfies those requirements. We implemented UniMiCo in a *DSMS* prototype and experimentally show that the protocol preserves correctness, while introducing no noticeable changes in the response time of the continuous query being migrated.

1. INTRODUCTION

Today, the ubiquity of sensing devices as well as mobile and web applications leads to the generation of huge amounts of data, which take the form of streams. Those data streams are typically high-volume, high-velocity (fast) and have high-variability (bursty). Data stream management systems (*DSMS*s) have become the popular solution to handle data streams, by efficiently supporting continuous queries (*CQ*s), which process data as they arrive *on the fly*.

The bursty incoming workload can overload the *DSMS* during its peaks. As a result, output is delayed and fails to meet the real-time requirements of monitoring applications and of emerging "Big Data" applications [8]. Most modern cloud infrastructures provide *elasticity*, which can be used to handle overloading situations [9]. Flux [12] was one of the early attempts to introduce a monitoring and load detection operator in a query network, and provided a state migration protocol to move *CQ*s across different machines. Another solution uses backup Virtual Machines (*VM*s) for periodically storing state [3]. In the event of load imbalance, the migrated *CQ*s restore the state from the backup *VM*s and apply incremental changes before resuming execution. Similarly, the operation migration mechanism in [10] follows the state migration paradigm. The efficiency of the migration mechanism is crucial, and no system downtime is acceptable since it translates to loss of data (hence the term "live" in previous work).

As part of the effort to scale-up/-down AQSIOS [4], our *DSMS* prototype, we implemented our own query migration protocol, named UniMiCo (**Uni**nterruptible **Mi**gration of **Co**ntinuous Queries). UniMiCo has the ability to (i) migrate stateful *CQ*s without the need to transfer any state, and (ii) do the migration in a "live" fashion (i.e., no downtime).

Our approach on *CQ* migration generalizes the idea of the *Window Recreation Protocol* (*WRP*) presented in [7] in two functional ways: First, while *WRP* was proposed to handle the migration of a sub-query with only one stateful operator, the UniMiCo protocol allows migrating a query with multiple stateful operators, each of which could have a different window specification. Second, in contrast to *WRP* that considers only time-based windows, UniMiCo's protocol has been designed in a general way to handle both time-based and tuple-based windows. A minor difference between *WRP* and UniMiCo is that UniMiCo does not involve the upstream data source in synchronizing the migration point, otherwise the two protocols share the same performance advantages and limitations. Both migration protocols are equally effective in migrating operators without state migration and query downtime, yet they might not be suitable when the window is too large (e.g., 24 hours [7]) since they may prolong an overloaded situation at the originating node.

We make the following *contributions* in this paper:

- We present the complete UniMiCo protocol that migrates a CQ with multiple stateful operators from one node to another.
- We experimentally show that UniMiCo migrates a *CQ* correctly without incurring any noticeable "hiccups" in its response time.

2. SYSTEM MODEL

We assume a system consisting of multiple shared-nothing nodes, connected by a reliable, high-speed network. One node serves the role of the coordinator, while the others are peers and each one of them runs one instance of AQSIOS. AQSIOS is our experimental *DSMS*, extended from the STREAM source code [2]. Our extensions include new operator implementation [6], optimization schemes [5], new scheduling policies [13], load shedders [11], and UniMiCo, our protocol to transfer a *CQ* from one node to another.

Based on the workload of each node reported by AQSIOS's load manager, the coordinator initializes a *CQ* migration when necessary. For a specific *CQ* migration between two nodes, we refer to the node which is running the *CQ* as *originating node*, and the node which is going to receive the *CQ* as *target node*. The migration can be materialized either through direct communication between the *originating* and *target node*s, or through indirect communication via the coordinator. In this paper we assume the former, but UniMiCo can work equally well with the latter.

AQSIOS supports a *CQ* execution model similar to Borealis and Apache Flink. Each AQSIOS node keeps a copy of the whole query network, but, only a subset of it is active on the node. A node only connects to the stream sources that are necessary for the active queries in the node. Data streams, coming from (possibly) different sources, are received by the *source operators*, which are the most upstream operators in a *CQ*. Figure 1 is an example of our system model with two AQSIOS nodes. The *CQ*s comprised of dark operators are those active at the node. The dashed lines represent network connections among the nodes.

In this paper we consider the whole query as the migration unit. However, the protocol can also be used to migrate only a segment of a *CQ*: the operator(s) right before the migrated segment becomes the stream source(s) for that segment. and their downstream operators act as source(s) in the corresponding *CQ*s.

Window-based operators

There are two types of operators in a *CQ*: *stateless* and *stateful* operators. A stateless operator, such as selection (σ), produces an output tuple based solely on the current input tuple. Conversely, a stateful operator, such as join or aggregation, needs to refer to values from previous input tuples. Due to the fact that input streams are infinite, *DSMS*s use either *tumbling* or *sliding windows*, to limit the state of operators. Sliding windows allow the output to be continuously computed based on the most recent "portion" of the stream data. In addition, a sliding window is specified through a length (or range) l, and a slide s, which can be either time interval or tuple count.

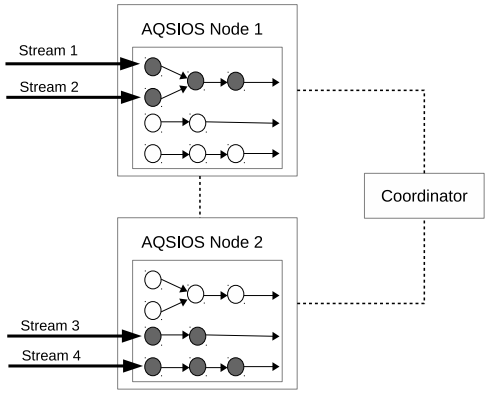


Figure 1: System model

These two types of windows are called *time-based* and *tuple-based windows*, respectively [2].

While most *DSMS*s embed the window definition into the stateful operator, some systems treat it as a separated operator (e.g., [2]). In this paper, when the semantics of the stateful operator are not important, we refer only to the window aspect of it as if the window is a separate operator. UniMiCo works the same way no matter whether the window operator is physically merged to the corresponding aggregate/join operator or not.

3. UNIMICO

The key goal of UniMiCo is to avoid transferring state during the migration of a *CQ* containing stateful operators. To achieve this, UniMiCo migrates a *CQ* at a window boundary, meaning that the *originating node* continues processing until it completes the last in-progress window, while the *target node* starts processing from the first tuple of the next window. Given that two consecutive sliding windows overlap, the tuples belonging to the overlap of the two windows are processed by both the originating and the target nodes. This way, the state of the operator is reconstructed at the *target node* so there is no need to migrate it.

We illustrate this strategy in Figure 2. In this example, the sliding window of a stateful operator (e.g., aggregate) has a size of 4 seconds and a slide of 2 seconds, with input rate 1 tuple/second. The number in

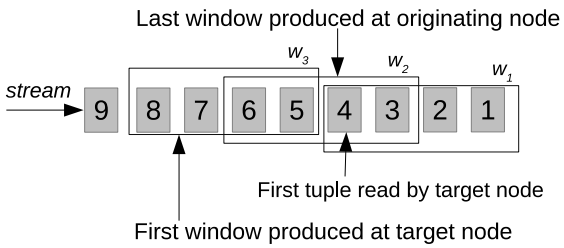


Figure 2: UniMiCo's migration strategy

each stream tuple is its timestamp, which is assumed to monotonically increase over time (i.e. *in-order* processing of tuples). By the time the migration process starts, the most recent window produced is w_1, whose start timestamp is 1. In addition, the first tuple received by the target node after it connects to the stream has a timestamp of 4. UniMiCo determines that (1) the originating node will continue processing until w_2 expires, which happens to be the last window with start timestamp less than 4, and (2) the corresponding *CQ* at the *target node* will start processing tuples with timestamp greater or equal to 5 (w_3).

3.1 Migration timestamp

The migration timestamp marks a *CQ* hand-off from the *originating* to the *target node*. That timestamp is used to synchronize the stop of the last window at the *originating node* and the start of the next window at the *target node*.

Definition The *migration timestamp* is the start timestamp of the last window to be processed at the *originating node*.

In the example in Figure 2, the start timestamp of w_2, which is 3, is the migration timestamp.

3.1.1 Calculating the migration timestamp

The exact calculation of the migration timestamp depends on the implementation details of the window operation. In this section we present how to calculate the migration timestamp on both time-based and tuple-based cases. In all the equations below, s denotes the slide of the window.

Time-based, single-input window: Assuming a time-based window of length l and slide s, let ts_{start} denote the timestamp of the first input tuple the stream source at *target node* was able to read after connecting to the stream. Furthermore, ts_{last_w} is the timestamp of the most recent window processed. The migration timestamp, denoted ts_{mi} is calculated as follows (note that now s is in number of tuples):

$$ts_{mi} = \begin{cases} ts_{last\_w} & \text{if } ts_{start} \leq ts_{last\_w} \\ ts_{start} - \delta & \text{otherwise} \end{cases}$$
$$\text{where } \delta = \begin{cases} s & \text{if } (ts_{start} - ts_{last\_w})\%s = 0 \\ (ts_{start} - ts_{last\_w})\%s & \text{otherwise} \end{cases} \quad (1)$$

Tuple-based, single-input window: For tuple-based windows, the calculation is the same in the case when $ts_{start} \leq ts_{last\_w}$. When $ts_{start} > ts_{last\_w}$, UniMiCo needs to wait until a tuple t comes to the window operator, whose timestamp is equal to or greater than ts_{start}. This way, UniMiCo is aware of the number of tuples with timestamps between ts_{last_w} and ts_{start} (let that number be N). The migration timestamp can be calculated by the following equation:

$$ts_{mi} = timestamp(\delta^{th} \text{ tuple preceding } t)$$
$$\text{where } \delta = \begin{cases} s \text{ if } (N+1)\%s = 0 \\ (N+1)\%s \text{ otherwise} \end{cases} \quad (2)$$

Multiple-input window: The most common example of window-based operator with multiple inputs is a binary join. For time-based windows, Equation 1 can be used, with $ts_{start} = max(ts_{start_i})$, where ts_{start_i} is the timestamp of the first input tuple the stream source i at *target node* was able to read. For tuple-based window, the number of tuples N_i coming between ts_{start_i} and ts_{last_w} is calculated separately for each input i. Afterwards, Equation 2 is applied with $N = max(N_i)$.

Multiple window operators: A *CQ* can have multiple window-based operators with different window specifications (i.e., length and slide), such as a query with an aggregation on top of a join. For these cases, we introduce the concept of the *controlling window operator*.

Definition The *controlling window operator* is the last window operator of the *CQ*. The *controlling window operator* handles the calculation of the migration timestamp, as well as controlling the start and stop of the migrated query at the *target* and *originating node*s.

For simplicity, we assume that the timestamp of an output tuple of a window-based operator is the *earliest* timestamp of input tuples involved in the calculation of that output tuple (we discuss later how this assumption is relaxed). When the aforementioned condition holds, we know that all the original input tuples, contributing to the result produced by the farthest window of start timestamp ts, have timestamps greater than or equal to ts. Therefore, only the farthest window operator (i.e., the *controlling window operator*) in the *CQ* needs to be involved, and the calculation is the same as in the case of single window. Note that the previous assumption is not required for the *controlling window operator*.

Figure 3 shows an example of a *CQ* consisting of two window-based operators: a binary join, whose window has length of 4 seconds and slide 2 seconds, followed by an aggregation, whose window has length of 3 tuples and size of 2 tuples. For each tuple its timestamp is shown on the upper and its join key on the bottom part. For the controlling window, the most recent window being produced is w_{21}, whose start timestamp is 1 (i.e., $ts_{last\_w} = 1$). In addition, assume that out of the two first tuples read from S and T by the target node, the latest timestamp ts_{start} equals 5. In this case, the migration timestamp is calculated as if there is only the *controlling window operator* (i.e, the aggregation) with two inputs S and T. Because the *controlling window operator* is tuple-based, UniMiCo has to wait until tuple

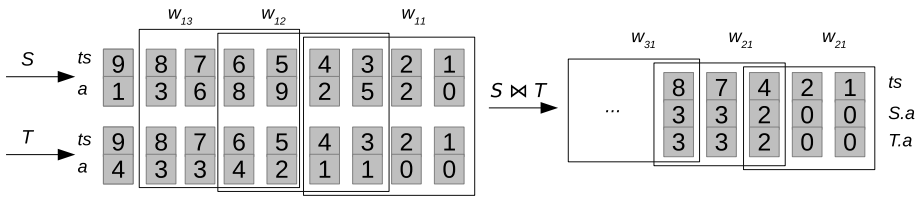


Figure 3: Calculating migration timestamp with two consecutive windows

t of timestamp 7 arrives to know that there are 3 tuples whose timestamps are between 1 and 5, i.e., $N = 3$. Applying the calculation from Equation 2 for the case of tuple-based window, UniMiCo decides that the migration timestamp is that of the tuple preceding t, which is 4. That is, the last window produced at target is w_{21}.

When the previous condition on output tuples' timestamps of preceding window operators does not hold, ts_{start} is measured as the timestamp of the first tuple arriving at the *controlling window operator* on the *target node*. Note than when this condition holds, ts_{start} is the timestamp of the first tuple coming to the source operator, i.e., it can be captured earlier. With the new ts_{start}, all of the above calculations of the migration timestamp are still applicable. Note that in this case if ts_{start} is smaller than ts_{last_w}, there will be some wasted processing at the target to process tuples from source up to the controlling window between ts_{start} and ts_{last_w}. Since migration happens when the target is lightly loaded, it is expected that processing at the *target node* will be at least as fast as that at the *originating node*, hence the wasted processing, if any, would be small.

3.2 Stoping and resuming *CQs*

3.2.1 *Stopping the query at the* originating node

Once the migration timestamp is determined, stopping the query at the *originating node* is relatively straightforward: all operators in the *CQ* continue to process normally until they receive the signal from the *controlling window operator* to deactivate themselves, unless they are shared with other *CQ*s. This happens when the *controlling window operator* has consumed its last window, i.e., the window started with the migration timestamp. Shared operators are not deactivated at the originating node and continue to process normally to serve the remaining queries. Upon stopping, an operator cleans up all its queues.

When the *controlling window operator* is associated with a join, a minor adjustment is needed in order to avoid duplicate outputs between the *originating* and *target node*s. Normally, when there is a match between a tuple t of one input and t' of the other, the join tuple tt' is produced only once, even if both t and t' fall in the overlap of two (or more) consecutive windows. If we start migrating from one of the windows, the join tuple tt' will be produced once at the *originating node*, and again at the *target node*. In the latter case, the production of a duplicate tuple is avoided by suppressing the production of the join result at the *originating node*. Note that when two matching tuples have their timestamps in the window overlap, the previous adjustment is needed only if the join is the last window-based operator in the query. In the event that a join is followed by another window operator, the duplicated intermediate output tt' is needed, as it is an input for the subsequent window at the *target node*.

3.2.2 *Starting the query at* target node

The operators of a migrated *CQ* can be activated at the *target node*, as soon as the migration is initialized. However, full activation is attained by controlling the flow of tuples based on the migration timestamp. That process is different for time- and tuple-based windows, as we describe below.

Time-based *controlling window operator*: If the *CQ* has a time-based *controlling window operator*, the stream source operator(s) calculate(s) the activation timestamp as migration timestamp increased with the slide of the window. Then, the stream source operator discards any input tuples, which carry timestamps less than the activation timestamp. In addition, it starts producing tuples with timestamp equal to or greater than the activation timestamp. With tuples being outputted from the stream source(s), the query is fully activated.

Tuple-based *controlling window operator*: In this case, the stream source operator(s) start(s) producing results from tuples with timestamps greater than the migration timestamp. But, the *controlling window operator* will discard all first $(s - \delta)$ tuples, where s is the slide of the window and δ is calculated from Equation 2 by the *originating node*.

For both types of windows, if the output timestamp of the preceding window-based operator is not the window's start timestamp, the *controlling window operator* has the only authority to decide when to output tuples. Thus, the source operator cannot do any early filtering.

Algorithms 1 and 2 outline the UniMiCo protocol executed at *target* and *originating node*, respectively.

Algorithm 1 UniMiCo protocol at *target node*

1: **BEGIN**
2: Receive(originating_node, *migrate*(Q))
3: **for** $i = 0; i <$ Q.num_streams; $i++$ **do**
4: connect(Q.streams[i])
5: ts_{start}[i] = read(Q.streams[i])
6: **end for**
7: Send(originating_node, ts_{start})
8: Receive(originating_node,ts_{mi})
9: Resume Q based on ts_{mi}
10: **END**

Algorithm 2 UniMiCo protocol at *originating node*

1: **INPUT**: Query Q to be migrated
2: **BEGIN**
3: Send(target_node, *migrate*(Q)
4: Receive(target_node, ts_{start})
5: ts_{mi} = calculate_migration_timestamp
6: Send(target_node, ts_{mi})
7: Finish_processing(Q, ts_{mi})
8: **END**

4. EXPERIMENTAL EVALUATION

While UniMiCo enhances *WRP*'s functionality, at the same time it inherits from *WRP* both its performance advantages and its limitations as stated in Introduction. Since these were experimentally shown in [7], our experimental evaluation focused on showing that UniMiCo migrates *CQ*s with single and multiple stateful operators correctly without impacting their response time.

We implemented and evaluated UniMiCo in a distributed setup of AQSIOS. As mentioned earlier, the window operator in AQSIOS is a separate operator, which receives stream tuples as input, and injects *minus* tuples to the stream to mark the boundary of a window [1]. Windows can have either time-based or tuple-based length, but the window slide is always 1 tuple. Therefore, window-based operators, such as join or aggregation, will rely on those minus tuples to perform their window-based processing. With the separation of the window operator, each input to a join operator can have a window of different length and type. In this paper, we assume that join inputs have windows of the same length and the same type, however, our design can be extended to heterogeneous window environments.

We ran two types of experiments: (1) simple *CQ*s with a single window operator and (2) a complex *CQ* consisting of two window operators. Given our focus on correctness and not performance, window size is not an important parameter in our experiments. We ran each *CQ* twice, under the same settings, and changed only if a migration took place. Then, we compared *CQ*s' outputs and response times around the migration point.

4.1 Simple *CQ* migration (Figures 4 & 5)

We used UniMiCo to migrate a *CQ* with a join operator (Q1), and another one with an aggregate operator

Output with migration

```
[10051579000]:+:location03, 3, 98, location03, 3, 98
[10051589000]:-:location03, 3, 98, location03, 3, 98
[10052632000]:+:location09, 30, 56, location09, 30, 56
[10052642000]:-:location09, 30, 56, location09, 30, 56
----------------------------------
[10053685000]:+:location16, 2, 77, location16, 2, 77
[10053695000]:-:location16, 2, 77, location16, 2, 77
[10054737000]:+:location20, 43, 21, location20, 43, 21
[10054747000]:-:location20, 43, 21, location20, 43, 21
```

Output without migration

```
[10051579000]:+:location03, 3, 98, location03, 3, 98
[10051589000]:-:location03, 3, 98, location03, 3, 98
[10052632000]:+:location09, 30, 56, location09, 30, 56
[10052642000]:-:location09, 30, 56, location09, 30, 56
[10053685000]:+:location16, 2, 77, location16, 2, 77
[10053695000]:-:location16, 2, 77, location16, 2, 77
[10054737000]:+:location20, 43, 21, location20, 43, 21
[10054747000]:-:location20, 43, 21, location20, 43, 21
```

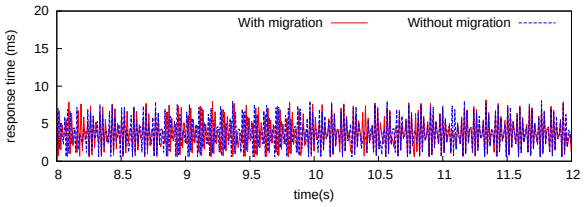


Figure 4: Results and response time of Q1 around the migration point at 10^{th} sec. The response time lines corresponding to "with migration" and "without migration" are indistinguishable as the migration adds no noticeable delay

(Q2). These two queries written in CQL [2] are:

Q1

```
SELECT *
FROM S [Range 10 seconds],
     T [Range 10 seconds]
WHERE S.1 = T.1;
```

Q2

```
SELECT sum(m)
FROM S [Rows 5];
```

where `S` and `T` are input streams. Q1 is associated with time-based windows with size 10 seconds (i.e., [Range 10 seconds]) whereas Q2 is associated with a tuple-based window of size 5 (i.e., [ROWS 5]).

Figures 4 and 5 show the results of Q1 and Q2 around the migration point, respectively. In Figure 4, the top plot is the result under migration, in which the rows above the dashed line are the last output tuples at the *originating node*, and those below are the first output tuples at the *target node*. The middle plot shows the result without migration, which is exactly the same as the concatenation of the two parts of the top plot. Similar observations can be made in Figure 5 for Q2. As one can see, the correctness of the output is maintained by using UniMiCo, and its protocol succeeds in performing the hand-off without losing any data.

The bottom plots in Figures 4 and 5 show the response time of queries Q1 and Q2 two seconds before and after the migration point of about the 10^{th} second. As can be seen in both figures, there are no noticeable "hiccups" in the response time of the queries throughout the

Output with migration

[10054323000]:+:92
[10054323000]:-:46
[10054323000]:+:87
[10054323000]:-:92

[10055771000]:+:102
[10055771000]:-:87
[10055771000]:+:99
[10055771000]:-:102

Output without migration

[10054323000]:+:92
[10054323000]:-:46
[10054323000]:+:87
[10054323000]:-:92
[10055771000]:+:102
[10055771000]:-:87
[10055771000]:+:99
[10055771000]:-:102

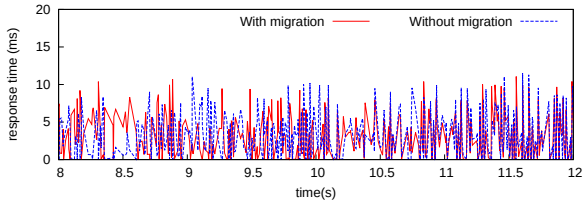


Figure 5: Results and esponse time of Q2 around the migration point at 10^{th} second. The response time lines corresponding to "with migration" and "without migration" are indistinguishable as the migration adds no noticeable delay

Output with migration

[10022574000]:+:109
[10022574000]:-:104
[10022574000]:+:104
[10022574000]:-:109

[10028529000]:+:107
[10028529000]:-:104
[10028529000]:+:70
[10028529000]:-:107

Output without migration

[10022574000]:+:109
[10022574000]:-:104
[10022574000]:+:104
[10022574000]:-:109
[10028529000]:+:107
[10028529000]:-:104
[10028529000]:+:70
[10028529000]:-:107

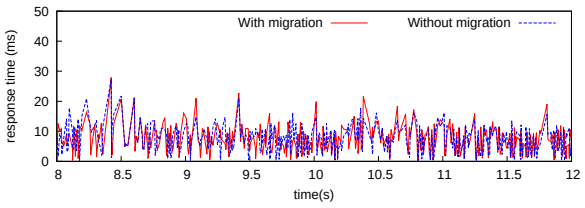


Figure 6: Results and response time of the complex query Q3 around the migration point at 10^{th} second. The response time lines corresponding to "with migration" and "without migration" are indistinguishable as the migration adds no noticeable delay

migration. For Q1, the average and standard deviation of the response time in this period without migration is 3.751 ms and 3.99 ms, respectively, while under migration they are 3.750 ms and 3.97 ms. For Q2, the corresponding numbers are 3.155 ms and 3.923 ms without migration, and 3.101 ms and 3.836 ms with migration. In both cases, the difference is negligible.

4.2 Complex *CQ* migration (Figure 6)

In this experiment we migrated a more complex query Q3, consisting of a join and an aggregate operator, each using a different window definition as below:

```
Q3: SELECT sum(S.m)
    FROM ISTREAM (SELECT *
                  FROM S [Range 10 seconds],
                       T [Range 10 secon
                  WHERE S.l = T.l ) [ROWS 5];
```

In this case, the last window, which is the tuple-based window of size 5 (i.e., [ROWS 5]) associated with the aggregation, plays the role of the *controlling window*.

Figure 6 shows the output tuples and the response time of the query Q3 around the migration point, compared with the run when there is no migration. Similar to the cases of the simple queries, the query output is preserved and the cost of migration is not noticeable. The average and standard deviation of the response time without migration are 6.568 ms and 6.133 ms respectively, while those with migration are 6.658 ms and 6.217 ms.

5. CONCLUSIONS

We presented UniMiCo, a general migration protocol for *CQ*s, used in distributed *DSMS*s. UniMiCo achieves migration without the need to transfer state or stop processing input tuples during *CQ* hand-off. UniMiCo is more general than previous work by being applicable to *CQ*s with different window semantics and with multiple stateful operations. Our experimental evaluation demonstrated UniMiCo's feasibility, by implementing it in a full-fledged prototype *DSMS* (AQSIOS). Our experiments showed its correctness and that it does not incur any noticeable delays in the *CQ*'s response time.

6. REFERENCES

[1] A. Arasu et al. Stream: The stanford data stream management system. Technical report, Stanford InfoLab, 2004.

[2] B. Babcock, S. Babu, et al. Models and issues in data stream systems. In *PODS '02*.

[3] R. Castro Fernandez et al. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD '13*.

[4] P. K. Chrysanthis. AQSIOS - Next Generation Data Stream Management System. *CONET Newsletter*, 2010.

[5] S. Guirguis et al. Optimized processing of multiple aggregate continuous queries. In *CIKM '11*.

[6] S. Guirguis et al. Three-level processing of multiple aggregate continuous queries. In *ICDE'12*.

[7] V. Gulisano et al. Streamcloud: An elastic and scalable data streaming system. *IEEE TPDS*, 2012.

[8] H. V. Jagadish et al. Big data and its technical challenges. *CACM*, Jul 2014.

[9] N. R. Katsipoulakis et al. Ce-storm: Confidential elastic processing of data streams. In *SIGMOD*, 2015.

[10] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu. Scalable distributed stream join processing. In *SIGMOD '15*.

[11] T. N. Pham, P. K. Chrysanthis, and A. Labrinidis. Avoiding class warfare: Managing continuous queries with differentiated classes of service. *VLDBJ*, 2016.

[12] M. A. Shah et al. Flux: an adaptive partitioning operator for continuous query systems. In *ICDE'03*.

[13] M. Sharaf et al. Algorithms and metrics for processing multiple heterogeneous continuous queries. *ACM TODS*, 2008.

Empowering Stream Processing through Edge Clouds

Sergio Esteves[1]*, Nico Janssens[2], Bart Theeten[2], and Luis Veiga[1]
[1]INESC-ID, Instituto Superior Tecnico, Universidade de Lisboa
[2]Bell Labs, Nokia, Antwerp

ABSTRACT

CHive is a new streaming analytics platform to run distributed SQL-style queries on edge clouds. However, CHive is currently tightly coupled to a specific stream processing system (SPS), Apache Storm. In this paper we address the decoupling of the CHive query planner and optimizer from the runtime environment, and also extend the latter to support pluggable runtimes through a common API. As runtimes, we currently support Apache Spark and Flink streaming. The fundamental contribution of this paper is to assess the cost of employing inter-stream parallelism in SPS. Experimental evaluation indicates that we can enable popular SPS to be distributed on edge clouds with stable overhead in terms of throughput.

1. INTRODUCTION

Stream Processing Systems (SPS) are vastly used by companies and organizations to extract insights and value from continuous streams of user data in near real-time. Storm [5], Spark [4], and Flink [2] are popular examples of such systems. At present date, there has been an accentuated demand for these systems to 1) fully support geo-distributed scenarios and 2) support SQL-like queries at interactive speeds. The former implies that partial computation graphs can be computed at possibly distant geographic locations and connected by a same cluster. As for the latter, there have been some recent efforts like Catalyst [7] for Spark. However, this native support for structured queries is yet limited to data sets of fixed size, and precludes continuous streams of data.

In our previous work, CHive [15] is a streaming analytics platform tailored for distributed edge clouds.[1]It enables SQL-like queries, that are executed over continuous streams of data, to be partitioned and distributed over constelations of micro-datacenters (i.e., following an edge computing model). CHive's fundamental contribution is that it optimizes query plans in such a way that the overall bandwidth consumption is minimal.

CHive targets a new scenario of edge clouds that is yet uncommon and not supported by major SPS. Typical widely-deployed SPS, such as Spark, are designed to operate on large clusters within a single datacenter, and assume nodes to be interconnected through high-throughput, low latency, Local Area Networks with full bandwidth availability. Further, CHive is tightly coupled with its runtime environment, Storm, which hinders the adoption of CHive by users of different SPS.

In this paper, we report our experience while addressing this problem of decoupling the CHive query planner and optimizer from its underlying runtime environment. We also propose a middleware layer, named CHive Deployer, that supports the plugging of different runtimes into CHive through a common API. Currently, we provide support for two major and recent SPS, Spark and Flink streaming.

Since commonly used SPS (e.g., Spark) are not designed to allow a cluster to span multiple datacenters (in different geographic areas), CHive relies on orchestrating multiple SPS clusters: each of the clusters typically corresponds to a datacenter, and CHive is responsible for connecting them according to a query plan. This, not commonly explored scenario, follows a edge computing model in which multiple datacenters are combined to execute different parts of a single distributed query. For example, a cluster might handle the first part of a query,

*This work was carried out while the author was an intern at Bell Labs

[1]A typical telecommunications network is built up into multiple layers: the core network, the edge network and the access network. The core network is located at the central office of the operator, while the access network is where the end-user communication lines are terminated. In between sits the edge layer, which is a geographically distributed network of smaller datacenters serving only a limited number of end users. Over the recent years, general purpose compute resources have been added to these distributed datacenters, effectively building out a distributed cloud, also called an edge cloud.

which relies on performing a project and a filter over data coming from a nearby source (in order to reduce the data stream volume), and another cluster might compute the rest of the query which relies on counting the tuples, within a temporal window, grouped by some key.

Supporting multiple runtimes is challenging because it involves using different programming APIs and models that have their own specificities. In addition, the distributed deployment of client applications, that interact directly with the SPS, varies immensely across different systems (e.g., complete Scala application without restrictions, or just a specification of the job computation graph). We aim at making CHive Deployer neutral, with respect to overhead introduced on the underlying SPS, and transparent for applications.

The fundamental contribution of this paper is to generalize CHive so it can be used with widely-deployed SPS. By doing so, we empower commonly used SPS to be distributed on edge clouds and enjoy major bandwidth reductions.

In the next section we survey related work. Section 3 describes the architecture design of the CHive Deployer, and Section 4 its evaluation in a distributed scenario. Finally, Section 5 concludes the paper and points out future work directions.

2. RELATED WORK

Performing streaming data analytics on the edge of networks follows a computing model that has been gaining significant traction lately, specially after the advent of IoT. This model permits to a great extent reducing the amount of data that needs to be transmitted and stored in a central system to perform analytics.

In addition, supporting SQL-style queries over continuous streams of data is a significantly trending topic, especially in an industrial setting. The advantages of using SQL-style are many, including short development cycles and lower maintaince costs when compared to low-level general purpose languages, such as Java and C++, to express analytic-based computations [14]. As aforementioned, the work in this paper builds upon and extends CHive [15], which enables such support for structured queries with windowing-based operators over a edge computing model.

In the research literature, SPSs like Aurora and Medusa [9] have corroborated our vision that stream-based systems can be inherently geographically distributed. They propose a distributed federation of participating nodes (e.g., datacenters) in different administrative domains, that can be scattered in different locations around the globe. Global applications include market data analytics, network monitoring, global surveillance, and e-fraud detection. Despite the author's described intentions regarding declarative query support, it is not clear whether Aurora supports in practice such SQL-style queries.

Apache Edgent [1] is a new project tailored to IoT that allows analyzing data on distributed edge devices. It consists of a programming model and runtime for edge devices. Analytics can be performed locally or in a back-end system according to their complexity. Unlike CHive, devices do not coordinate and share data among themselves, thereby always requiring a centralized system to answer queries involving more than 1 device. Edgent highlights however the necessity of running analytics on the edge of network (premise shared with CHive).

To the best of our knowledge, there is a considerable gap between theory and practice in what concerns to SQL support over streams of data (examples include [10, 8]). Other projects, although claiming to have SQL-like queries support implemented [11, 16, 12], have remained as research projects mainly used by academics, and not available for the general public nor companies. Following, we focus on open-source available solutions.

In the open-source domain, SparkSQL [7] is a new module that enables relational processing, and SQL queries, in Apache Spark. It introduces a highly extensible optimizer, Catalyst, that makes it easy to add new optimization techniques. Although it is part of the authors' future plans, SparkSQL with Catalyst currently do not support the streaming component of Spark. To overcome this, StreamingSQL [6] attempts to extend SparkSQL with windowing based capabilities. However, StreamingSQL functionality is still limited and inefficient, since queries can only be executed over data frames (like a table in a traditional DBMS) that are obtained by converting the results of stream transformations.

MRQL [3] is the closest project to ours: it shares our goals of providing a query processing and optimization system that can be plugged to different underlying data processing systems. Nevertheless, the streaming support is still a work in progress. Despite that, the query optimization techniques are unaware of any network topology, unlike CHive which follows an edge computing model that distributes operators across different datacenters. To the best of our knowledge, none of the available and popular open-source project allows a query to be partitioned and distributed across more than one cluster/datacenter.

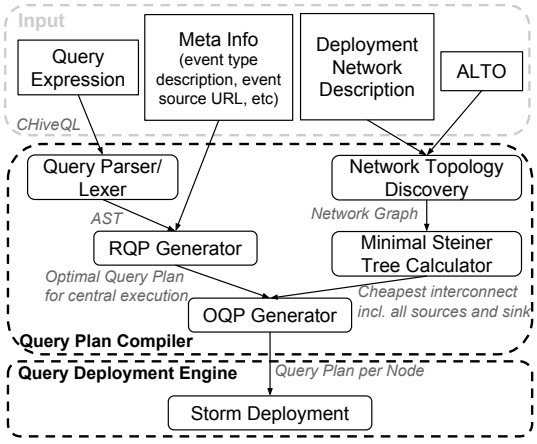


Figure 1: CHive architecture and work flow

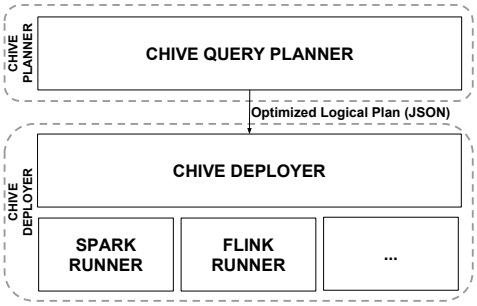


Figure 2: CHive Deployer architecture

3. DESIGN AND IMPLEMENTATION

Figure 1 depicts the general architecture and work flow of the original CHive. Aside from the input layer on top, we can see that there are two main layers in this architecture, represented by the Query Plan Compiler and the Query Deployment Engine components.

Briefly, the query plan compiler generates a reference query plan by using a CHive query expression along with meta information describing event types and event source URLs (among other parameters). Soon after, this reference query plan is combined with a Network topology description and an Optimized Query Plan (OQP) is generated. This OQP that is outputted from the query compiler specifies which chain of operators (or query primitives) should run on which datacenters. This mapping between operators and datacenters is made in a way such that the overall bandwidth consumption is minimal. Afterwards, the deployment engine takes the OQP, which in fact contains a local query plan per datacenter, and deploys them to run on top of Storm, the SPS used with CHive. For more details we refer to [15].

In this work we focus on decoupling the Query Plan Compiler from the Query Deployment Engine, which is specific to Storm in the original CHive. We also change and extend the deployment engine so that different and widely-deployed SPS can be plugged into CHive. Figure 2 clearly depicts the separation that we want to achieve and the architecture overview of the CHive Deployer (Query Deployment Engine in the original CHive).

3.1 Work flow

Figure 3 illustrates the distributed architecture that we get with Spark for a simple pipeline job with 3 stages. In darker grey, we have the components addressed in this paper. The general work flow works as follows. First, the CHive Deployer receives an OQP from the the CHive Query Planner and translates it into a common specification language (more details are given in Section 3.2). Then, this specification is sent to the selected runner and SPS computation graphs are generated.

Soon after, the runner launches the client applications (Spark Driver) to run onto (geo-distributed) remote clusters. These applications, in their turn, submit jobs to the spark executors and collect the corresponding results. Also, these applications, except the one on the last cluster, execute connectors, which serve their corresponding output data to the next downstream cluster.

Finally, a source injects data into the first cluster, which performs some initial computation on it, and sends the results to the next cluster in the middle. This cycle is repeated for the other 2 clusters, using as sources the output of the previous cluster, until the final computation results are sunk to the client.

3.2 CHive Deployer

The CHive Deployer, which takes an OQP from the CHive Planner, is responsible for preparing the local plans therein contained to be deployed on an SPS. This preparation involves translating and adapting the local plans to a common abstract SPS specification. For example, CHive query plans refer to schema attributes, such as to perform a project over *name* and *age* of a stream of data containing people's information. Since most SPS are schemaless, CHive Deployer replaces the attribute's names (e.g., *age*) by the position by which they appear in the stream of data against a string separator; if our stream of data is composed of text lines containing *name*|*address*|*age*|*telephone*, then *name* and *age* would be replaced by *0* and *2* respectively (against the separator |). Hence, this abstract specification is an attempt to find the greatest common denom-

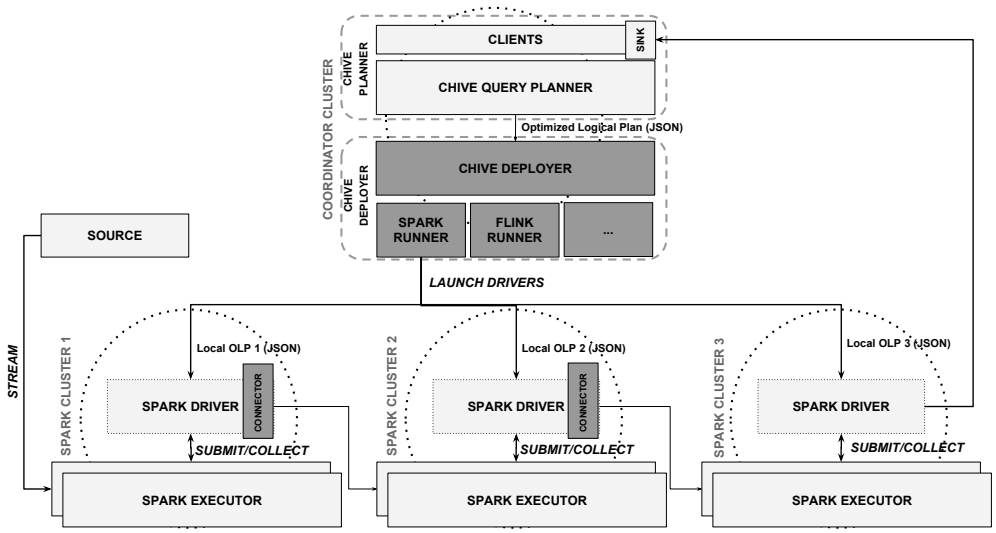


Figure 3: Distributed architecture for simplified pipeline job spanning 3 clusters

inator between the majority of SPS.

In its turn, a runner is responsible for translating this common SPS specification to a target runtime, which includes using the programming model and APIs that are specific to a given SPS. Different runners can be plugged into the CHive Deployer through a common API, which mainly communicates the common SPS specification. Further, this specification must be as fine grained as possible, since some operators of some SPS can perform all-in-one actions; e.g., *reduceByKeyAndWindow* in Spark versus *keyBy().timeWindow().reduce()* in Flink.

By design, it is not currently possible to execute a query plan over heterogeneous runners (e.g., executing a plan using clusters of distinct SPSs). In theory, however, this is possible, since client SPS applications can implement custom code to handle all different types of data and data sources.

3.3 Runner

A runner is responsible for building the computation graph for each local plan (which has been translated into the common specification) that will run on each cluster or datacenter, thereby making use of the underlying programming model that is specific to a given SPS. A runner is also responsible for launching the SPS client applications onto remote clusters. These client applications (one per cluster) interact directly with the SPS, namely submitting jobs, corresponding to the local plans, and collecting results to be shipped out to the next downstream clusters.

Finally, a runner should also take care of connecting the stream of data across clusters. It basically needs to add sources and sinks to all local execution graphs that do not have them, so that all clusters get connected into a single distributed execution graph.

Developing a runner takes a considerable effort: since each SPS has a unique programming model and API, it is necessary to implement all functions and operators that concretize the common specification in a target SPS runtime. Currently, the common abstract specification generated by the CHive Deployer is fully compatible with Spark and Flink, but as we add new features to CHive, it might be possible that not all runners support them (e.g., session windows are available in Flink but not in Spark).

3.4 Connectors

A connector is a component that is responsible for connecting the stream of data across intermediate clusters. For example, having a pipeline job comprising 3 stages spanned across clusters, we would have one cluster to get the input from a given external source; one cluster to sink the final results of the entire computation; and one intermediate cluster that would use connectors to: i) receive its input from the output of the first cluster; and ii) send its output to the input of the last cluster.

A connector can be embedded in SPS client applications or launched as an external process on the same cluster nodes as of the client applications. Whenever possible, connectors should be embedded in SPS client applications, since it is slightly more efficient to send records directly from the application than piping them to an external process. How-

ever, for some SPS this is not possible: Flink, for instance, only allows job graphs to be launched onto remote clusters, and not general full client applications (like it is allowed by Spark).

At its core, a connector maintains an open server connection (e.g., socket based) so that records can be shipped out to the next downstream clusters. In practice, the next downstream clusters fetch input data from the servers opened by connectors.

In case of embedded connectors, they are dependent of the SPS, and thus they need to be developed for each runner. Otherwise, external connectors can be used by different runners, and thus the development effort is reduced at the expense of a slightly slower inter-cluster connection (as aforementioned).

4. EXPERIMENTAL EVALUATION

All benefits of CHive, especially in terms of bandwidth, were already demonstrated in our previous work [15]. In this paper we evaluate the neutrality of the CHive Deployer; i.e., we assess the impact that the CHive Deployer has in terms of overhead on the underlying SPS. Specifically, we show that the CHive Deployer runners are coherent with their corresponding single cluster versions (i.e., without being distributed in a edge clouds model). The objective of this evaluation is to understand whether the overhead of the CHive Deployer is stable and not highly influenced by the specific underlying SPS.

All tests were conducted using 6 machines with an Intel Core i7-2600K CPU at 3.40GHz, 11926MB of RAM memory, and HDD 7200RPM SATA 6Gb/s 32MB cache, connected by 1 Gbps LAN (which ensures a fair reference comparison, given that the network latency between all machines was the same). Also, we used Spark Streaming 1.5.2 and Flink 0.10.2 in our two provided runners.

To evaluate the CHive Deployer we relied on a scenario to calculate the top 20 websites generating the highest download volumes in the last 10 seconds, from a stream of real-world traces of a large mobile operator. Figure 4 depicts the Optimized Query Plan that we get from the CHive Planner for this considered scenario. We can see that the respective query is distributed across 6 different datacenters, corresponding to 6 different machines in our experiment. The normal triangles, ellipses, and inverted triangle represent data sources, operators, and the sink respectively.

In a first experiment, we measured the throughput obtained for the considered scenario in terms of total number of records processed, per window of 10 seconds, for the entire computation. We measured this throughput for our two implemented runners, Spark and Flink, and compared CHive with the baseline system. In CHive mode (i.e., using a edge computing model), there are 6 Spark/Flink separate clusters where each executes part of the computation graph (like depicted in Figure 4). As for the baseline mode, which represents a classic situation, it comprises a single Spark/Flink cluster spanned across 6 machines where each one runs the entire computation graph on different data partitions. Note that these systems are not designed to run across different geographic locations that are distant from one another; i.e., there are strong limitations in spanning a cluster across multiple datacenters [13] (unlike the CHive approach).

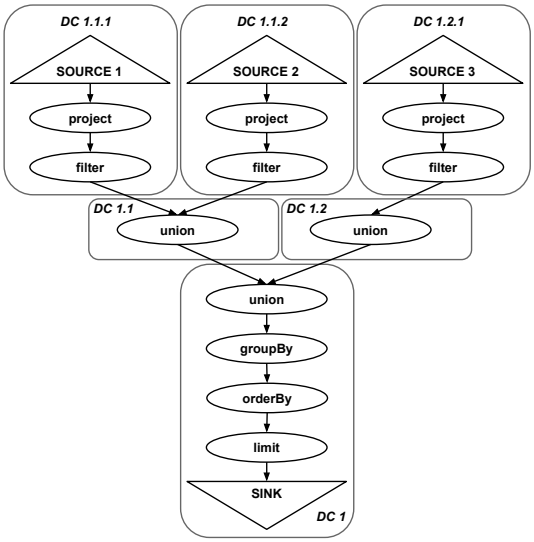


Figure 4: OQP for the top 20 websites with the highest download volumes

We observed that the baseline mode yields higher throughput than the CHive mode for both considered SPS. That is to be expected, since input data in the baseline system is given to all of the 6 machines, whereas in CHive only 3 machines (corresponding to 3 clusters) are fed with data from the sources. (Refer to our previous work to see the advantages of CHive against other systems, especially in terms of bandwidth.) The important point to note here is the neutrality of the CHive Deployer: on each considered SPS, the throughput difference remains in (almost) the same proportion between baseline and CHive modes.

Table 1 shows the differences of throughput in proportion (i.e., we divide the throughput obtained with baseline by the one of CHive for Spark and Flink). In both SPS, the ratio between baseline and CHive is the same within a deviation of less than 5%, which indicates that CHive Deployer is neutral

Table 1: Deviation (as percentage change) of throughput ratios between baseline and CHive with Spark and Flink runners

Spark	Flink	Deviation
1.32	**1.37**	**4%**
(2832/2150)	(1184/863)	(1.37/1.32-1)

Table 2: Deviation (as percentage change) of total number of bytes ratios between baseline and CHive with Spark and Flink runners

Spark	Flink	Deviation
1.50	**1.38**	**9%**
(148631/99159)	(62372/45175)	(1.50/1.38-1)

and not intrusive in relation to the baseline system.

In a second experiment, and using the same setup and query as of the first experiment, we assessed the deviation in the results, originated by different throughputs (within a window), with the objective of seeing how far results are between baseline and CHive. Specifically, we have counted the total number of bytes that we obtain from the output, which corresponds to the sum of the bytes of all top 20 websites, in a 10 second window for Spark and Flink, while comparing the CHive mode against the baseline system. This comparison consisted of the division between the total number of bytes in baseline and CHive, for Spark and Flink, as shown in Table 2. The result accuracy difference that we obtain for both SPS is the same within a deviation of less that 10% (cf. table below), which indicates CHive Deployer is significantly coherent across runners.

As a side effect, we have also shown a comparison between Spark and Flink themselves. For our specific workload, Spark outperformed Flink. This was mainly due to some operators/tasks that have higher parallelism levels and are more optimized in Spark (such as sorting tuples within a window).

5. CONCLUSION

In our previous work, CHive [15] enables structured interactive queries to run distributed on continuous streams of data, over edge clouds, in a bandwidth efficient manner. This computing model, that has been gaining significant traction lately (specially with the advent of IoT), is made available to popular SPS through the generalization of CHive (that is addressed in this paper).

In particular, this paper addressed the decoupling of the CHive query planner from its underlying runtime environment. We have built CHive Deployer, a middleware layer that makes possible to use different and widely-deployed SPS with CHive through a common API. We have also demonstrated the feasibility of plugging runners and SPS to CHive Deployer by developing two for popular SPS: Spark and Flink. Experimental evaluation, with real-world data, indicates that CHive Deployer is neutral (not affected by the underlying technology) and does not introduce any additional overhead.

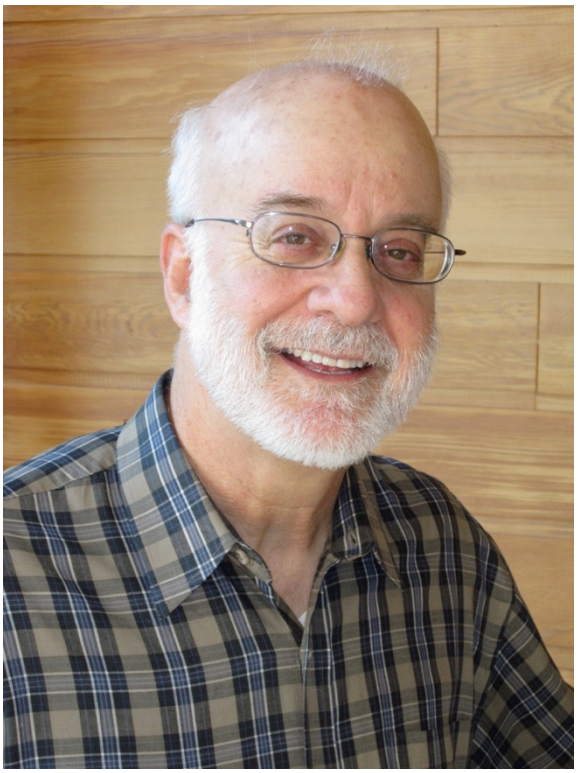
Acknowledgements: This work was supported by national funds through Fundação para a Ciência e a Tecnologia with reference UID/CEC/50021/2013.

6. REFERENCES

[1] Apache Edgent. `http://edgent.incubator.apache.org/`.
[2] Apache Flink. `http://flink.apache.org/`.
[3] Apache MRQL. `https://mrql.incubator.apache.org/`.
[4] Apache Spark. `http://spark.apache.org/`.
[5] Apache Storm. `http://storm.apache.org/`.
[6] Streaming SQL for Apache Spark. `https://github.com/Intel-bigdata/spark-streamingsql`.
[7] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.
[8] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30(3):109–120, Sept. 2001.
[9] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable Distributed Stream Processing. In *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.
[10] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: The system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1123–1134, New York, NY, USA, 2008. ACM.
[11] M. A. Hammad, M. F. Mokbel, M. H. Ali, W. G. Aref, A. C. Catlin, A. K. Elmagarmid, M. Eltabakh, M. G. Elfeky, T. M. Ghanem, R. Gwadera, I. F. Ilyas, M. Marzouk, and X. Xiong. Nile: a query processing engine for data streams. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 851–, March 2004.
[12] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 245–256, Asilomar, California, Jan. 2003.
[13] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica. Low latency geo-distributed data analytics. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 421–434, New York, NY, USA, 2015. ACM.
[14] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, Dec. 2005.
[15] B. Theeten and N. Janssens. Chive: Bandwidth optimized continuous querying in distributed clouds. *Cloud Computing, IEEE Transactions on*, 3(2):219–232, April 2015.
[16] Y. Wei, S. H. Son, and J. A. Stankovic. Rtstream: real-time query processing for data streams. In *Object and Component-Oriented Real-Time Distributed Computing, 2006. ISORC 2006. Ninth IEEE International Symposium on*, pages 10 pp.–, April 2006.

Ron Fagin Speaks Out on His Trajectory as a Database Theoretician

Marianne Winslett and Vanessa Braganholo



Ron Fagin
http://researcher.ibm.com/person/us-fagin

Welcome ACM SIGMOD Record's series of interviews with distinguished members of the database community. I'm Marianne Winslett, and today we are in Snowbird, Utah, USA, site of the 2014 SIGMOD and PODS conference. I have here with me Ron Fagin, who has spent many years as a researcher at IBM. He is an IBM Fellow. He is a Fellow of ACM, IEEE, and the American Association for the Advancement of Science. He was elected to the National Academy of Engineering and the American Academy of Arts and Sciences. He has won the IEEE McDowell Award (the highest award of the IEEE Computer Society), the IEEE Technical Achievement Award, and the SIGMOD Edgar F. Codd Innovations Award, and he has won a bunch of Best Paper and Test-of-Time Awards. He was named Docteur Honoris Causa by the University of Paris. Most recently, he won the Gödel Prize in 2014. Ron's Ph.D. is in mathematics, from Berkeley.

So, Ron, welcome!

Thank you, Marianne.

Tell me about the work that you received the Gödel Prize for.

Well, actually it was a bit of a long story. It arose when Laura Haas knocked on my door one day and said, "Okay Mr. Database Theoretician, we've got a problem." So I said, "Laura, what's the problem?" and she said, "Well we have this middleware database system called Garlic, and it is on top of pure database systems like DB2, and it's also on top of QBIC." QBIC ("Query by Image Content") is a system where you can query by image content: you search for objects based on color, shape, or texture. She said, "The trouble is that there are mixed data types. The answer to a query in a normal database system is a set (or a bag), and the answer to these multimedia queries is a sorted list." She said, "So what do we do? How do we combine the results together?"

I thought about it and came up with a solution involving fuzzy logic (where a proposition can be not just true or false, but somewhere in between). I was very excited. I went to see Laura and said, "Laura, I got you an answer. Use fuzzy logic." She said, "That's good, Ron. But we don't have time to look at every single item in the database and assign some kind of fuzzy score to it. We need to get our answers fast. I need an efficient algorithm." So I said, "Okay, fine."

I went back to my office and a day or two later I came back and said, "Laura, good news, I got a square root of n algorithm for you (where n is the number of objects in the database)." She said, "Great! Square root of n beats linear, but you know what Ron, we database people are spoiled. We are used to log n algorithms like in B-trees." I'll never forget what she said to me next. She said, "Ron, be smarter. Go back to your office and get me a log n algorithm."

So I went back to my office and came back a day or so later and said, "Laura, I can prove square root of n is the best you can do. It's a matching upper and lower bound. " She said, "Fine; we'll take it." And it was implemented in Garlic. Then a few years later (and here's where the Gödel Prize winning work came in), I was doing some work with Moni Naor and Amnon Lotem, and we miraculously came up with a new algorithm called the Threshold Algorithm, which beat Fagin's Algorithm ("Fagin's Algorithm" is the name of the algorithm that I originally gave to Laura -- she named it that, and it appeared in papers that way). The Threshold Algorithm is optimal but in a stronger sense than Fagin's Algorithm. Fagin's Algorithm is optimal in a certain worst-case sense, which is the usual standard for optimality of an algorithm. The Threshold Algorithm is optimal not just in the worst case, or in the average case, but in every case! We called this property "instance optimality". Thus, the adversary can design his own database and his own algorithm fine-tuned to that database, and our algorithm can perform just as well on the adversary's database as the adversary's algorithm performs on the adversary's database. Even though the algorithm is only about ten lines long this paper won the Gödel Prize, which is the highest award for a paper in Theoretical Computer Science! It was hard to find that algorithm, but once you have it, it is easy to verify. Our paper is the only database paper ever to win the Gödel Prize. Our definition of instance optimality is a strong notion -- it was an exciting notion to the people in the Computer Science Community.

My goal is to convince theoreticians that they will prove better theorems and they'll do more interesting work if they just talk to practitioners.

Okay, you have won two Test-of-Time Awards from PODS and one from ICDT. What were those pieces of work about?

Well, the first Test-of-Time Award from PODS[1] was for the work that eventually won the Gödel Prize. It also won the Best Paper Award for that conference. The other two Test-of-Time Awards, the one from ICDT, which we got last year[2], and the one from PODS that we are getting this year[3], both had to do with data exchange. Data exchange deals with converting data from one format (the source) to another (the target). In data exchange, there are certain first-order logic formulas called "tuple-generating dependencies" (or TGDs) that specify a relationship between the source and the target, but do not

[1] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal Aggregation Algorithms for Middleware. In: PODS, 2001.

[2] Ronald Fagin, Phokion Kolaitis, Renee Miller, and Lucian Popa. Data Exchange: Semantics and Query Answering. In: ICDT, 2003.

[3] Ronald Fagin, Phokion G. Kolaitis, Lucian Popa, and Wang Chiew Tan. Composing schema mappings: Second-order dependencies to the rescue. In: PODS, 2004.

completely specify the target. In the first paper, which won the Test-of-Time Award last year, we described a particular family of choices for the target (which we called "universal solutions") with a number of desirable properties. The concept of universal solutions became widely accepted.

This year's Test-of-Time Award for PODS was for composition. There, the question is "If you convert data from format A to format B and then convert from format B to format C, how do you convert directly from format A to format C?" To our amazement, it turned out that even if the methods that go from A to B and from B to C are both specified by these simple TGDs, going from A directly to C not only could take you away from TGDs but even out of first-order logic! We had to go to second-order logic. We invented something called second-order TGDs, and we proved that those were exactly the right ones for the task. Specifically, every composition where each component is specified by first-order TGDs can be specified by a second-order TGD, and for every second-order TGD, there is some sequence of compositions where each component is specified by a first-order TGD that gives that second-order TGD.

You just got named to the American Academy of Arts and Sciences too!

True! I'm really very proud of that because, for example, Wikipedia calls it one of the nation's highest honors. Something really cool about it is that it was founded during the American Revolution, and the first class included George Washington and Benjamin Franklin. Each year the Academy selects something like 7 or 8 people from each discipline. Like 7 or 8 computer scientists, 7 or 8 mathematicians, 7 or 8 physicists, and (since it's arts and sciences) they even pick people from movies and TV. So it's exciting to be invited to become a member of that select group.

What will you guys do?

Well, I think one of the main purposes of all these National Academies is to figure out whom we elect for the next year (laughs). It seems to be! (More laughs).

I know from the National Academy of Engineering that this selection process seems to be the immediate task. In my short time of my being a member, we spent a lot of time figuring out who is going to be next year's candidates. I know that an important role of the National Academies is to conduct policy studies. I have not been involved in that, and I'm not sure how much I could contribute to policy. I'm currently excited that the National Academy of Engineering is going to have a black tie inauguration. I have to actually go in a tuxedo, since they take this very seriously.

It's like the prom all over again!

Exactly like the prom!

Well, congratulations on that too. Now it must be pretty cool to have a theorem named after you. What is Fagin's Theorem?

What it does is to tie together complexity theory on the one hand with logic on the other hand. There's the important complexity class NP, and there's also something called existential second-order logic. What Fagin's Theorem says is that they are really equivalent (for example, the class of 3-colorable graphs is both in NP and expressible in existential second-order logic). On the face of it, NP and existential second-order logic look very different; they're from different disciplines (one from complexity theory and one from mathematical logic). Because of this equivalence, you can use tools from one area to help you in the other area. It's always cool when you get a connection between two very different fields, and that's what Fagin's Theorem does. There has been much follow-up work.

And that was done back in your dissertation.

Correct. Incidentally, my Ph.D. thesis seemed to be completely unnoticed for a number of years. There is probably a moral in there about keeping the faith.

I feel like much of your career was the Golden Age of Logic for Computer Science? It seems like things are switching over to the Golden Age of Statistics in Computer Science. Have you seen that too?

Yeah, I do see some of the statistics, but logic is still going strong. In fact, Jeff Ullman and I were two of the founding fathers of relational database theory, where Jeff focused on the algorithms, and I focused on the logic. Both tracks are still very active. But you're right Marianne, there are other things that are entering in the picture, but logic is an important track and is still very much there.

Okay, you've already mentioned Fagin's Algorithm. We've got Fagin's Theorem, Fagin's Algorithm, Fagin's 0-1 Law, Fagin games, Ajtai-Fagin Games, and the Fagin-inverse. It's good that you don't have a really long name I guess. So what's Fagin's 0-1 Law?

It's one of my favorite theorems ever. It says the following: take any sentence in first-order logic involving only relational symbols, and it's either going to be almost always true or almost always false in the asymptotic sense. That is, as the size of the finite structures get larger and larger, the fraction of structures that obey the sentence will converge, and it will converge to either 0 or 1. So first-order sentences are either almost always true or almost always false. That's what the 0-1 Law says.

IBM has been around for more than 100 years now, and you're not around for more than 100 years by just continuing to do what you have always done, because what you do eventually becomes obsolete and then you have to move on.

Is there an impact on the practical side of Computer Science from that?

Well, some people say they use the 0-1 Law to help them understand the average case behavior of things. I'm not quite sure if I buy into that. I mean, it's a motivation I've heard people give because there's been a lot of work on the 0-1 Law, extending it to other logics and so on. People argue that it has this practical impact. To me, it's this beauty. To me, it's just very neat that things perform in such a very simple, natural way. You'd think that either the probabilities might not converge, or they might converge to a half or two-thirds or something, but no, they always converge, and always to 0 or 1. To me, it's just mathematically beautiful. That's why I love it so much. I have to say that I love my proof too. In fact, of all of my results over the years, my proof of the 0-1 law is probably my favorite.

What about Fagin games and Ajtai-Fagin Games?

So-called "Ehrenfaucht-Fraisse games" are used to prove inexpressibility results in logic. In an Ehrenfaucht-Fraisse game, there are two players, called the Spoiler and the Duplicator, and they take turns picking points. There are two structures, and the Spoiler picks point 1 in one structure (either the first structure or the second structure), and then the Duplicator picks point 1 in the other structure. Then the Spoiler picks point 2 in one structure (again, either the first structure or the second structure), and then the Duplicator picks point 2 in the other structure. This continues for a fixed number of rounds. Consider the mapping between the two structures, where for each k, the kth point selected in the first structure maps to the kth point selected in the second structure. The Duplicator wins if this mapping is an isomorphism, and otherwise the Spoiler wins. Proving that the Duplicator has a winning strategy gives an inexpressibility result for first-order logic.

There are a lot of variations to that game. Fagin games arise when you try to prove inexpressibility results in a logic called existential monadic second-order logic. Then the rules get a little more complicated. In Fagin games, you again have the Spoiler and the Duplicator, and they first color the points. Thus, the Spoiler colors the points in the first structure and then the Duplicator colors the points in the second structure. Then they play the Ehrenfaucht-Fraisse game I described earlier, but now for the Duplicator to win, the isomorphism must respect colors. Miki Ajtai and I came up with new games (now called "Ajtai-Fagin games") in which we changed the rules of the Fagin game in an interesting way to make it much easier for the Duplicator to win, which gives a much easier proof of inexpressibility results.

Okay, and that leaves the Fagin-inverse.

That's something from data exchange. I talked earlier about converting from format A to format B, but what if you say, "I want to go back from B to A. How do I do that?" The Fagin-inverse is all about going backwards. There are a lot of very subtle issues that arise: the inverse may not exist, and even if it exists it may not be unique. So I defined this thing that is now called the Fagin-inverse and described how you take a mapping that goes from A to B and when and how you can invert it, using the Fagin-inverse, to go from B to A. It's not obvious. It's not obtained by simply reversing the arrows. Since then, there have been a number of other flavors of inverses that have been studied.

Okay, being an IBM Fellow gives you a bit of hope for evangelizing for your favorite technical causes inside IBM. How have you used that?

I'm glad you asked. My mission as an IBM Fellow has been to convince theoreticians and practitioners to work together. My goal is to convince theoreticians that they will prove better theorems and they'll do more interesting work if they just talk to practitioners.

By talking to practitioners, they will discover new exciting problems that no one else has considered before, and then other people will jump on the bandwagon. You will be creating a new field, and you'll have a real impact. The best example I can give of this is that resolving the very practical problem that Laura Haas posed to me led to the Gödel Prize.

I also have to convince practitioners they should work with theoreticians to make their products better: they'll get new algorithms, they'll get performance guarantees, and they'll have a much more solid system with features that other systems don't have. So as an IBM Fellow, my mission has been going around to IBM's worldwide research labs, giving lectures on this, talking to the young people to mentor them and to spread my gospel on applying theory to practice. I have recently expanded my mission by giving my speech on applying theory to practice at a number of major universities. My goal is to get theoreticians and practitioners to interact more with each other.

Do you think they believe you?

Well, they seem to. It is much easier for theoreticians to work only with other theoretician, to just talk to people who speak their language. It's a real effort to speak to someone outside your field. There are two ways I tell people it can happen. One way is like the way I told you with the story of Laura knocking on my door and saying, "I've got a problem." But there's another way it can happen, and this is what happened in the work on data exchange that we won the two Test-of-Time awards for. The data exchange project called Clio, also led by Laura Haas, had been going on for over a year, and because of how well things went with Laura earlier on the Garlic project, I had been regularly attending Clio meetings. Then Phokion Kolaitis, Lucian Popa, Renee Miller and I (later joined by Wang-Chiew Tan) said, "This data exchange work at IBM has been going on for a long time. But let's see how we would do data exchange if we did it from scratch. Let's see what the right way to do it is. Let's have no preconceived ideas, and just say: if we were doing data exchange and no one told us anything about it, how would we do it, using principles from database theory?" That's what we did with data exchange, and it led to a very successful body of work. In fact, our ideas were implemented in Clio. This included the use of second-order TGDs as the internal mapping language of Clio. (As I mentioned earlier, second-order TGDs arose as the result of our theory work on composition of TGDs that received the PODS Test-of-Time Award in 2014.) And because of our work, every major database conference started having special sections on data exchange. We felt good that we brought data exchange out as a discipline with interesting technical results. There has been a lot of work done on data exchange ever since.

That list of people that you gave… Most of them I'd say are more from the theory side.

True. Lucian, however, played a very key role. Lucian lives on both sides of the aisle. Lucian was heavily involved in the actual implementation side of Clio, and he also does theory. One of the great things about working with Lucian was that he was our bridge to the other world. He understood what the issues were and he would keep us honest. For example, when we discussed technical issues, he might say "Okay, guys, now we're going off into never-never land. No practitioner cares about the issue we are now discussing, so let's consider this other direction instead."

What about the finite model theory?

Finite model theory is the topic of my Ph.D. thesis, and so is really near and dear to my heart. My thesis is where Fagin's Theorem, the 0-1 Law, and Fagin games appeared. I'm happy that people consider me the founder of finite model theory. Now, lots of work is being done in the area, and finite model theory has been applied in a number of different ways. That's something I'm proud of.

That idea, did that come from talking to practitioners?

No. I was at Berkeley writing my Ph.D. thesis, and the ideas all arose in different ways. For example, my 0-1 Law arose from a huge question in finite model theory, which is closure under complement of various classes. For example, if a property can be expressed in existential second-order logic, which I showed is the class NP, is the complement also expressible in this logic? This is really close to the P vs. NP problem: it's the NP vs. Co-NP problem. While playing with the notion of closure under complement, I realized to my surprise that in ordinary first-order logic, if a property is interesting, then its complement seemed to be very uninteresting. For example, consider the conjunction of the field axioms. That is an interesting first-order sentence. But its negation (which defines the complement) is very uninteresting, since there are many ways to fail to be a field. I wondered how I might prove some theorem that says that in first-order logic, if a property is interesting, then its complement is very uninteresting. I concluded, "I can use asymptotic probabilities." Specifically, I decided to interpret "very uninteresting" to mean "almost always true". This would imply that either a first-order

sentence or its negation is almost always true. And that's what I proved, via the 0-1 Law. As for your question, I got to this without talking to any practitioners.

Okay. You've been at IBM for over 40 years. The IT companies that were big back in the mid-70s are all dead now, except for IBM. Why did IBM survive when so many others did not?

I think its adaptability. IBM has been around for more than 100 years now, and you're not around for more than 100 years by just continuing to do what you have always done, because what you do eventually becomes obsolete and then you have to move on. There are Harvard Business School studies about this issue. If your company is extremely successful at something, and you see something new coming up that is going to replace your very profitable line of business, it's hard to switch to it, because, in the short term you're going to lose a lot of money since you're suddenly pushing customers from your expensive solution that was your bread and butter to something else. But if you don't do it, someone else will, so you better do it. IBM has learned that lesson, and IBM has adapted a number of times. IBM is doing that right now, by the way.

What have they been giving up right now?

The issue isn't a matter of IBM giving up on things, but rather a matter of IBM devoting more and more of its resources to areas that are crucial for the future.

What are the new big things at IBM?

The big new things are "CAMSS": cloud, analytics, mobile, social, and security – and, of course, artificial intelligence. So IBM is moving heavily into all these areas.

You knew Ted Codd, didn't you? Tell me a story from the early days.

Oh, so let me tell you how I got involved with Ted Codd. I transferred from IBM Watson to IBM San Jose, and when I transferred, I looked around and said, "Okay, who's interesting here to work with?" There were a number of interesting people, but the guy who I thought was most interesting was Ted Codd, and I went to him and said, "I'd like to work with you." I was thrilled that he said yes. So Ted was my mentor, and he was my hero. He really helped my career.

One thing I remember (even though it's not a big deal, but to me it was huge at the time) is that he took me to a SIGMOD conference after I'd done some work with him and he put his arm around me either literally or figuratively (I'm not sure which) and he introduced everyone to me saying "This is Ron Fagin, he's a new employee at IBM and he's doing great work on relational databases." I just glowed, and I thought, "Wow the great Ted Codd, who is already the icon, is saying these nice things about me." That was even before Ted was an IBM Fellow, and before he won the Turing Award. I got into databases because of Ted Codd. He was my mentor, and he was doing relational databases, so by golly, I did relational databases.

> ***[…] what's important for me, Marianne, is completely understanding something. Putting my arm around it, totally, deeply, completely understanding it.***

So besides positive feedback, you said he had a big influence on your career.

Just talking to him – I would talk about relational databases, and he would understand it, of course, totally, deeply and that would help me understand it better. This was why I got into relational database theory. From talking with Ted, I had a good feeling about what databases were all about, what they could do, and why relational databases were different from previous ways of doing databases. I then began to understand what he was doing and how important it was, and I wanted to get involved, and I did.

Do you have any words of advice for fledgling or mid-career database researchers?

My advice I give all young people is to go to lots of talks, interact with lots of people, do different things, and open your mind. You'll never know when you will find something cool and exciting. And then follow your heart and work on what seems most important to you.

If you magically had enough extra time to do one additional thing at work that you are not doing now, what would it be?

This is kind of off the wall but believe it or not, cosmology. I am fascinated by the notion of multiple universes, and in fact I'm almost obsessed by it. Actually, I even wrote an unpublished paper about how

to calculate the probability of our own universe colliding with another universe. It sounds weird, but I based it on what I called the probability of a big bang per cubic meter per second.

How high is that probability?

It's pretty low, but I wrote this little paper on it. Then I put "colliding universes" into Google, and to my delight, it turned out that the world expert on colliding universes was a UC Santa Cruz professor whom I had met at a party! So I thought, "Okay, I'll send him my paper and see what he says." I thought he would just ignore it, but he was kind and said, "You have some nice new ideas. However, your paper violates both relativity and quantum mechanics." Oops. I'm not a physicist, so I wrote my paper from a Newtonian point of view. But I'm still fascinated by the notion of multiple universes, even though I'm a bit discouraged about my prospects for winning the Nobel Prize in physics through my cosmology work, given that it violates fundamental laws of physics.

By the way, I want to say something about laws of physics. I don't understand quantum mechanics. I admit it freely. I didn't go into physics but went into mathematics and later computer science because I just don't understand quantum mechanics. It isn't at all intuitive to me. And I felt much better, years later, when I found out that Richard Feynman said, "If you think you understand quantum mechanics, then you don't understand quantum mechanics." I thought, "Yes, it's not just me, it's everybody! If Richard Feynman, a Nobel Laureate in physics, says that, then none of us understand quantum mechanics, so it's okay that I don't understand it at all."

But does that mean that you should have gone into physics afterward?

No, because what's important for me, Marianne, is completely understanding something. Putting my arm around it, totally, deeply, completely understanding it. I feel like the reason I'm in mathematics is because I felt like I could do that. I felt like I could take that area and study it and think about it and read about it. It would be mine. I would own it. I would totally completely understand it in every way. In physics, I realized, I could never do that, because no one can. In some ways, physicists blindly follow some mysterious formalism that they don't completely understand. They may not view it that way, but I can't work like that. So I'm really glad I didn't go into physics, I wouldn't be happy just pushing equations around. I have to totally, completely understand things, and deep in my soul, I don't understand physics.

Well, it's good for computer science, I guess, that it turned out that way, but if you could change one thing about yourself as a computer science researcher, what would it be?

Actually, you know what? I'm not sure if I would change anything. I feel like I've been very lucky. Things have fallen my way, and I feel like I've made some good choices. Working with Ted Codd and getting into databases is an example. And it's gone so well, I don't think I'd change a thing. I never dreamed, by the way, of ever becoming an IBM Fellow, because the typical IBM Fellow brings like a billion dollars to IBM and I thought there's no chance that IBM would take a theoretician like me and make him an IBM Fellow. But, miraculously, they did. So things fell my way, and I'm delighted with how things have turned out. I couldn't ask for it to go any better so I wouldn't change a thing.

Okay, well thank you very much for talking with me today.

Thank you, Marianne. It was fun.

The Dresden Database Systems Group

Wolfgang Lehner
Technische Universitat Dresden – Faculty of Computer Science
01062 Dresden, Germany
wolfgang.lehner@tu-dresden.de

ABSTRACT

The *Dresden Database Systems Group* focuses on the advancement of data management techniques from a system level as well as information management perspective. With more than 15 PhD students the research group is involved in a variety of larger research projects ranging from activities to exploit modern hardware for scalable storage engines to advancing statistical methods for large-scale time series management. The group is visible at an international level as well as actively involved in cooperations with national and regional research partners.

1. INTRODUCTION

The efficient processing of large volumes of data without compromising many of the traditional database system properties like consistency, descriptive query specification, durability, etc. is one of the core pillars of many user-level applications or domains like Machine Learning. Data management solutions have therefore gained significant relevance and are also constantly faced with a wide variety of requirements ranging from application access (analytical vs. transactional) and different operator types (relational model, linear algebra, graph processing etc.) to different data characteristics (from relational rows to documents to tensors etc.). These application requirements are met by potentials and capacities on the hardware side. Especially in the recent past, hardware platforms have changed dramatically, providing substantial new opportunities for data management solutions in many areas. However, these golden prospects come along with severe constraints and increased overall system complexity.

Processing: The early multi-core era with double-digit numbers of cores per system has passed. Nowadays, multi-socket systems with up to 1,000 cores have become economically feasible. In addition, GPUs and FPGAs have made significant progress in providing general purpose processing units, but still require specific support from the software layer.

Memory: While disks are still highly usable for cold data, the increase of main memory capacities often allows to keep all working data close to the processing units. With this, the focus shifts from buffer pool management to cache optimization. In addition, non-volatile RAM will allow to directly work on primary data without copying content from the persistent to the transient memory world.

Network: Recent improvements in network technologies (e.g. Infiniband, Nx10GB Ethernet) in combination with RDMA etc. blur the boundaries between "local" (or in-node memory) and "remote" memory within a cluster, providing the opportunity to re-consider scale-up and scale-out.

Overview of research activities

Reflecting on the requirements from the application side and the opportunities on the hardware side, database systems are currently sandwiched between these two layers and have to mediate in order to provide the best service using the most efficient hardware environment. In order to holistically embrace these technological challenges and provide excellent research contributions, the *Dresden Database Systems Group* is structured into two topic areas:

System architecture: Research activities generally investigate novel system architectures as well as specific technologies to exploit modern hardware opportunities within modern storage engines. Individual topics, as detailed in Section 2, range from energy optimization via data encoding and compression to hybrid data structures for heterogeneous memory.

Data Processing: Within this field, research is conducted to push the envelope in the context of data extraction and data imputation for semi-structured data sets as well as forecasting and managing large-scale time series data. Section 3 will provide more detailed information.

Scientific environment

The *Dresden Database Systems Group* is located in Dresden (Germany), the capital of the state of Sax-

ony. Located at the heart of the Elbe valley, Dresden is famous for its baroque buildings, Mediterranean flair, and worldwide renowned cultural activities. In addition, Dresden is also one of the main research centers in Europe with important institutions like the Max Planck Society (3 institutes), Fraunhofer Society (11 institutes), Leibniz Society (4 institutes), and of course the Technische Universität Dresden (TUD), one of eleven German universities that were awarded the "University of Excellence". Moreover, Dresden is one of the largest semiconductor centers worldwide with more than 1,500 IT companies forming the region known as "Silicon Saxony".

The "Technische Universität Dresden[1]" was founded in 1828 as the "Saxon Technical School[2]" to educate workers in technological subjects such as mechanical engineering, and ship construction. Today it is among the Top-3 universities for Engineering in Germany and with approximately 37,000 students, it is one of the largest universities in Germany. The Faculty of Computer Science consists of six institutes with more than 1,700 bachelor and master students, and 180 doctoral students. Dresden has been the main research hub for computer science in Eastern Europe, making cutting-edge database research a big part of its long tradition. The database systems group is headed by Wolfgang Lehner since October 2002 and currently consists of 5 postdoctoral researchers and 15 PhD students. The group is involved in many national and international research projects and activities (Section 5). In addition to the summary below, the website of the group at `https://wwwdb.inf.tu-dresden.de/` is providing further information.

2. SYSTEM ARCHITECTURE

The group's research field in the context of efficient and scalable data processing systems embraces different research directions investigating the benefits of modern hardware and developing novel algorithms and data structures. The core question that drives the research activities is: "How should database systems be designed to optimally match new application requirements with new hardware opportunities?". To answer this question, the group develops a scalable data management platform (ERIS[3]), which is agnostic with respect to logical data models as well as physical implementations. The basic idea of this platform is to factor out as many general data management services like visibility, data and query distribution, connection management, etc. as possible and provide a plug-in mechanism for operator implementations as well as individual physical designs. In addition, the platform also systematically deploys the concept of control loops for different aspects at different levels and provides a rich set of telemetry data. For example, access statistics at the physical container level serve as input for self-optimizing access path selection. Performance counters at the CPU-level serve as input for energy optimization as well as data placement strategies. In general, this project acts as an envelope and implementation sandbox, to which individual and specific PhD projects contribute.

Energy Management

While energy consumption is a well-known issue for large-scale computing, it has also become a serious challenge in the context of individual computing systems. In this domain, our research work investigates the potentials and opportunities for fine-tuning energy consumption without compromising the overall system performance. To our own surprise, there are significant opportunities for saving energy – both from the energy efficiency and the energy proportionality perspective. Figure 1 outlines energy profiles for different workloads defined by operating individual cores and the socket infrastructure (caches, controller etc.) at different frequencies. The diagrams show individual configurations (dot size = number of active cores, dot color = average core frequency with uncore frequency in the middle) organized in a performance versus energy efficiency manner [24]. As we can see, different performance for the same work can be achieved by different configurations exhibiting different energy behaviors. With background knowledge of the type of work (column scan, hash-based aggregation, etc.) the system may pick the most energy-efficient configuration for a particular task. As we demonstrated in the context of ERIS, we can achieve up to 30% energy savings compared to the standard Linux power governor without compromising performance.

Heterogenous Systems

While using GPUs for data management activities has a long research history, most of the work focused on special implementations for highly specialized hardware configurations. In the context of heterogeneous systems research, we investigated different approaches to integrate different compute units into a single query processing environment. In [14], we propose an iteratively refined cost model to determine the optimal work distribution with respect to

[1] `https://tu-dresden.de/`
[2] `https://en.wikipedia.org/wiki/TU_Dresden`
[3] `https://wwwdb.inf.tu-dresden.de/research-projects/eris/`

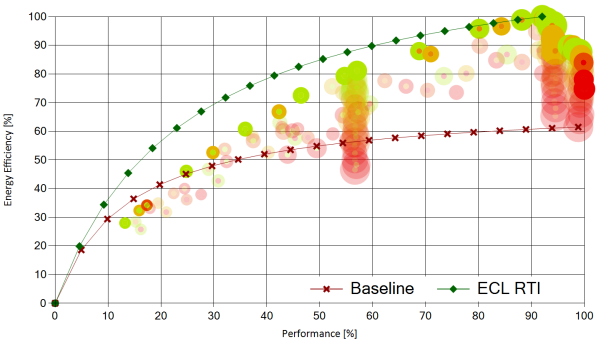


Figure 1: Energy profiles using different workloads; 12 core, 4 different core frequencies, 3 different uncore frequencies, resulting in 144 configurations

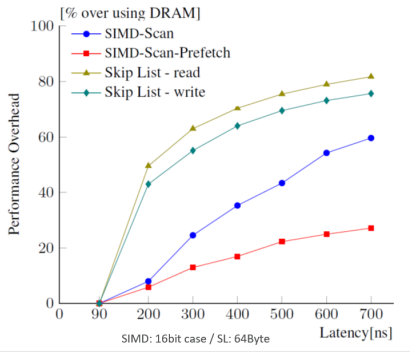


Figure 2: Performance overhead for varying memory latencies

traditional CPUs or an alternative implementation using a GPU. We also showed that the distribution of a single operations over heterogenous devices is often not practical [13] and devised an allocation scheme on a per-operator basis. The approach nicknamed HERO ("HEterogeneous Resource Optimization") provides a pseudo OpenCL device which on the one side can be registered at any OpenCL-enabled database engine and on the other side may decide on the optimal operator as well as data placement.

Non-Volatile RAM

While heterogeneity is a quite well-understood fact at the level of processing units, we see a similar trend at the level of memory. The "black-and-white"-model of RAM with buffer pool against a disk is long gone. Large main memories with different characteristics have taken over, ranging from extremely fast MCDRAM-like memories to non-volatile but still byte addressable memory systems. Within different research activities, we investigate basic characteristics and the impact on data structure design of future NVRAM. For example, Figure 2 shows that the HW-prefetcher of a CPU is able to hide most of the penalty derived from higher memory latencies for scan-based memory access patterns. However, for data structures required to follow pointers (e.g. SkipLists), the latency is directly visible as additional overhead [22].

Based on these characteristics, we developed the FP-Tree [21], a hybrid data structure spanning volatile DRAM (for the inner nodes) as well as NVRAM holding the leaf nodes for the raw data. This allows the data structure to be completely self-constrained, i.e. it does not rely on a global log but provides a micro-logging approach to bring the data structure into a consistent state after failure recovery [23]. Moreover, since HTM is a scalable method for DRAM-based data manipulation, it is inherently incompatible with NVRAM-based data structure modifications. Therefore, the FP-tree intertwines different concurrency schemes, the volatile as well as the non-volatile part. Research on efficient data structure design has a tradition within the group. A team of PhD students won the SIGMOD 2011 Programming Contest with a solution based on in-memory optimized prefix trees [17], which again was followed by the KISS-tree, a highly optimized prefix tree for supporting 32bit key lookups with exactly 3 memory accesses, independent of data cardinality and skew [18].

Data encoding schemes

The traditional disk-based layout uses a row-based or columnar data layout to represent the raw data (in combination with secondary index structures). Due to extremely high access latencies, the physical data layout of logical entities was not in the focus of optimization. Main-memory systems however demand and allow more sophisticated encoding schemes, especially compression schemes to limit the data transfer between memory and CPU as well as to increase cache utilization. Moreover, since raw as well as intermediate data are both located in main memory and therefore exhibit the same access characteristics, it seems beneficial to apply lightweight compression schemes also for intermediates.

Unfortunately, compression algorithms are highly dependent on the individual data characteristics and implementation details. Within [4], we reported on 39 different implementations of different compression algorithms ranging from logical schemes like RLE, Differential Coding, Dictionary Encoding to physical schemes like null suppression to eliminate leading zeroes in the binary representation. As expected, there is no single best algorithm, the decision is not trivial and depends on system environment as well as data characteristics. The experimental study however provides a solid base for an automated selection mechanism.

As counterpart to compression schemes, we also investigate the impact of encoding schemes for failure detection and failure discovery, which becomes more and more relevant with larger and denser main-memories. We look at applying AN coding schemes for column stores, which turns out to be a great solution for detecting multi-bit flips, as it results in a significantly lower probability for silent data corruption in combination with a simple arithmetic model. While previous work only used expensive division operations for decoding, AN coding allows transforming divisions into relatively cheap multiplications by using inverses.

3. DATA PROCESSING

As already mentioned, the research field of data processing addresses applications for managing and analyzing data. Research activities range from extracting structured data out of unstructured data to large-scale time series management.

Database Augmentation

In the era of Big Data, the number and variety of data sources is increasing every day. However, not all of this new data is available in well-structured databases or warehouses. Instead, heterogeneous collections of individual datasets such as data lakes are becoming more prevalent. This new wealth of data, though not integrated, has enormous potential for generating value in ad-hoc analysis processes, which are becoming more and more common with increasingly agile data management practices. However, in today's database management systems there is a lack of support for ad-hoc data integration of such heterogeneous data sources.

We therefore developed the entity augmentation system REA [7] that, given a set of entities and a large corpus of possible data sources, automatically retrieves the missing attributes. Due to the inherent uncertainty of the data sources and the matching process in general, REA produces not one but k different augmentations from which the user can choose. To this end, we developed an extended version of the Set Cover problem, called Top-k Consistent Set Covering, onto which we map our requirements.

On top of that, we built DrillBeyond [6] by integrating REA with PostgreSQL, that allows to combine structured and unstructured query processing and enables seamless SQL queries over both RDBMS and the Web of Data. Therefore, we designed a novel plan operator that encapsulates the retrieval part and allows direct integration of such systems into relational query processing. The operator is placed in a cost-based manner to create query plans, that are optimized for large invariant intermediate results which can be reused between multiple query evaluations.

Dresden Web Table Corpus (DTWC)

The Web has become a comprehensive resource not only for unstructured or semi-structured data, but also for relational data. Millions of relational tables embedded in HTML pages or published in the course of Open Data/Open Government initiatives provide extensive information on entities and their relationships from almost every domain. Researchers have recognized these Web tables as an important source of information for applications such as factual search, entity augmentation and ontology enrichment. Therefore, we extracted the Dresden Web Table Corpus[4] [5] a large corpus consisting of 125 million unique tables extracted from the July 2014 incarnation of the Common Crawl. The DWTC is used as a source of semi-structured data for our augmentation project but also triggered other research projects, e.g. in [3] we proposed a semantic normalization approach for Web tables containing multiple concepts, whereas in [1, 2] we proposed techniques to recover the meaning of columns by inferring knowledge base class labels and considering the Web table context.

DeExcelerator

Spreadsheets are one of the most successful content generation tools, used in almost every enterprise to perform data transformation, visualization, and analysis. The high degree of freedom provided by these tools results in very complex sheets, intermingling the actual data with formatting, formulas, layout artifacts, and textual metadata. To unlock the wealth of data contained in spreadsheets, a human analyst will often have to understand and transform the data manually. To overcome this cumbersome process, we proposed the DeExcelerator [8] that is able to automatically infer the structure and extract the data from these documents in a canonical form [19, 20].

Large-scale time series forecasting

Many analytical applications are based on empirically collected data sets derived from sensors that form large time series. Our research activities started by treating timeseries as first class citizens within a database system and introducing forecast operators to support predictive modeling. In a first step, we developed a solution that natively integrates time series forecasting into an existing DBMS, the Flash-

[4] `https://wwwdb.inf.tu-dresden.de/misc/dwtc/`

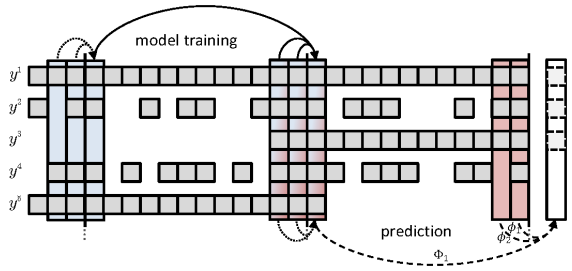


Figure 3: CSAR model with one seasonal and two non-seasonal AR components

Forward Database System (F^2DB) [9]. It supports a new query type–the forecast query–that enables forecasting for any database user and is transparently processed by the core engine of an existing DBMS. A key component of our system is a specialized model index structure that stores pre-built forecast models, transparently finds existing models for a given query, and maintains materialized models.

Based on this work, we reached out into multiple directions. On the one hand, we devised a novel forecasting method "The Cross-sectional Autoregression Model" (CSAR) for large-scale data sets that are highly dynamic and often noisy. While traditional forecasting approaches are focused on individual time series, resulting in a high model creation effort for a large data sets, CSAR trades depth for width by incorporating only the relevant sections of multiple time series into a model. In doing so, it provides a balance between low latency and high accuracy at individual aggregate levels. Figure 3 outlines the basic idea of CSAR with details provided–for example–in [10].

On the other hand, we investigate ways for the systematic description of time series characteristics. We developed a feature-based approach that allows us to create synthetic time series based on a given set of reference series data. As shown in [16], the approach allows users to formulate specific what-if scenarios by "tweaking" individual characteristics of the underlying time series and instantaneously see the impact in the time series data. This mechanism can be used to systematically generate time series data for simulations, model evaluation, or scalability experiments [15].

4. PARTICIPATION IN MAJOR RESEARCH ACTIVITIES

All individual research activities of the database systems group are integrated into different larger research projects funded by industrial partners, the German Research Foundation (DFG), and the European Union. The following list provides a comprehensive overview.

SAP HANA Database Campus: The group maintains a research relationship with the product development group of SAP HANA mostly located in Walldorf, Seoul, and Waterloo for more than 10 years. A variety of research activities have jointly resulted in high-profile publications as well as direct product impact. Directly involved PhD students of the Dresden group are physically located in Walldorf together with fellow PhD students from other universities forming the SAP HANA Database Campus[5].

Center for advancing electronics Dresden[6] (cfAED): cfAED was established within the German excellence initiative that represents the flagship of research funding instruments in Germany. The center aims at exploring new technologies for electronic information processing which overcomes the limits of today's predominant CMOS technology. The database systems group is actively involved in two research paths: the investigation of resilience mechanisms for data structures (using different encoding schemes), and the creation of mechanisms to bridge the gap between traditional silicon-based systems and systems based on novel materials potentially providing completely different computing characteristics.

Research Center on Highly Adaptive Energy-Efficient Computing[7] (HAEC): The HAEC project systematically and holistically investigates energy efficiency in computer systems. Starting from the hardware perspective it goes all the way up to implications for application development, compiler design, and runtime support. Wolfgang Lehner is acting a co-chairman and is responsible for all software-related activities.

Research Training Group on Role-based Software Infrastructures for continuous-context-sensitive Systems[8] (RoSI): Software with long life cycles is faced with continuously changing contexts, e.g. new functionality has to be added, new platforms have to be addressed, and existing business rules have to be adjusted. The concept of role modeling has been introduced in different fields and at different times in order to model context-related information. The central research goal of this project is to deliver proof of the capability of consistent role modeling and its practical applicability. Research activities within the database systems group try to integrate the notion of role-modeling into the database system and develop novel agile

[5] https://wiki.scn.sap.com/wiki/display/SAPHANA/Research+at+the+SAP+HANA+Database+Department
[6] https://cfaed.tu-dresden.de
[7] https://tu-dresden.de/ing/forschung/sfb912
[8] https://wwwdb.inf.tu-dresden.de/grk/

schema evolution methods to efficiently control the real-world constraints based on playing individual roles. Additionally, novel agile schema evolution methods [12, 11] are subsumed under this project [12, 11]. Wolfgang Lehner is the spokesman of this initiative, which is funded by the DFG.

Information Technologies for Business Intelligence - Doctoral College[9] (IT4BI-DC): IT4BI-DC is a doctoral program addressing six fundamental challenges in the area of Business Intelligence: Modeling and Semantics, Information Discovery, Information Integration, Business Analytics, Large-Scale Processing, and Collaboration and Privacy. The curriculum is jointly delivered by Université Libre de Bruxelles (Belgium), Aalborg Universitet (Denmark), Technische Universität Dresden (Germany), Universitat Politècnica de Catalunya (Spain), and Poznan University of Technology (Poland). Associated partners from around the world include top-ranked universities, leading industries in BI, public and private research organizations, consulting companies, and public authorities. The consortium jointly designs a set of research topics, which are jointly co-supervised by two partners of the consortium. Graduates perform their research at two of these universities and upon completion of the program are awarded with a joint degree.

5. CONTRIBUTION TO THE COMMUNITY

The Dresden Database Systems Group is supporting the database community at different levels and in different roles. At the regional and national level, Wolfgang Lehner was acting as the spokesman of the database special interest group within the "Gesellschaft für Informatik" (= German equivalent of ACM). Since April 2012, Wolfgang Lehner is elected member of the Computer science review panel of the German Research Foundation (DFG) and acts as the chairman since April 2016. At the international level, Wolfgang Lehner was member of the editorial board of the VLDB Journal from 2005 to 2011. He was Co-PC-Chair of VLDB 2011, ICDE 2015, and currently serves on the VLDB Endowment. Wolfgang Lehner is also PC member of all high-profile database conferences and was awarded with the "Distinguished PC Member" award at SIGMOD 2017. All of these activities have only been possible with a great team that is supporting and contributing. Thea team is the source of all of these fascinating research results and therefore deserves recognition for all of these remarkable achievements.

[9] https://it4bi-dc.ulb.ac.be/

6. REFERENCES

[1] K. Braunschweig, M. Thiele, J. Eberius, and W. Lehner. Column-specific context extraction for web tables. In *SAC*, pages 1072–1077, 2015.

[2] K. Braunschweig, M. Thiele, E. Koci, and W. Lehner. Putting web tables into context. In *KDIR*, 2016.

[3] K. Braunschweig, M. Thiele, and W. Lehner. From web tables to concepts: A semantic normalization approach. In *ER*, pages 247–260, 2015.

[4] P. Damme et al. Lightweight data compression algorithms: An experimental survey (experiments and analyses). In *EDBT*, pages 72–83, 2017.

[5] J. Eberius, K. Braunschweig, M. Hentsch, M. Thiele, A. Ahmadov, and W. Lehner. Building the dresden web table corpus: A classification approach. In *BDC*, 2015.

[6] J. Eberius, M. Thiele, K. Braunschweig, and W. Lehner. Drillbeyond: processing multi-result open world SQL queries. In *SSDBM*, pages 16:1–16:12, 2015.

[7] J. Eberius, M. Thiele, K. Braunschweig, and W. Lehner. Top-k entity augmentation using consistent set covering. In *SSDBM*, pages 8:1–8:12, 2015.

[8] J. Eberius et al. Deexcelerator: a framework for extracting relational data from partially structured documents. In *CIKM*, pages 2477–2480, 2013.

[9] U. Fischer, C. Schildt, C. Hartmann, and W. Lehner. Forecasting the data cube: A model configuration advisor for multi-dimensional data sets. In *ICDE*, 2013.

[10] C. Hartmann et al. CSAR: The cross-sectional autoregression model. In *DSAA*, 2017.

[11] K. Herrmann, H. Voigt, A. Behrend, J. Rausch, and W. Lehner. Living in parallel realities: Co-existing schema versions with a bidirectional database evolution language. In *SIGMOD*, pages 1101–1116, 2017.

[12] T. Jäkel, T. Kühn, H. Voigt, and W. Lehner. Towards a role-based contextual database. In *ADBIS*, 2016.

[13] T. Karnagel, D. Habich, and W. Lehner. Limitations of intra-operator parallelism using heterogeneous computing resources. In *ADBIS*, pages 291–305, 2016.

[14] T. Karnagel, D. Habich, and W. Lehner. Adaptive work placement for query processing on heterogeneous computing resources. *PVLDB*, 10(7):733–744, 2017.

[15] L. Kegel, M. Hahmann, and W. Lehner. Template-based time series generation with loom. In *EDBT*, 2016.

[16] L. Kegel, M. Hahmann, and W. Lehner. Generating what-if scenarios for time series data. In *SSDBM*, 2017.

[17] T. Kissinger et al. A high-throughput in-memory index, durable on flash-based SSD: insights into the winning solution of the SIGMOD programming contest 2011. *SIGMOD Record*, 41(3):44–50, 2012.

[18] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. *KISS-Tree*: smart latch-free in-memory indexing on modern architectures. In *DaMoN*, pages 16–23, 2012.

[19] E. Koci, M. Thiele, O. Romero, and W. Lehner. A machine learning approach for layout inference in spreadsheets. In *KDIR*, pages 77–88, 2016.

[20] E. Koci, M. Thiele, O. Romero, and W. Lehner. Table identification and reconstruction in spreadsheets. In *CAiSE*, pages 527–541, 2017.

[21] I. Oukid et al. Fptree: A hybrid SCM-DRAM persistent and concurrent b-tree for storage class memory. In *SIGMOD*, pages 371–386, 2016.

[22] I. Oukid and W. Lehner. Data structure engineering for byte-addressable non-volatile memory. In *SIGMOD*, pages 1759–1764, 2017.

[23] I. Oukid, W. Lehner, T. Kissinger, T. Willhalm, and P. Bumbulis. Instant recovery for main memory databases. In *CIDR*, 2015.

[24] A. Ungethüm, T. Kissinger, D. Habich, and W. Lehner. Work-energy profiles: General approach and in-memory database application. In *TPCTC*, pages 142–158, 2016.

CALL FOR NOMINATIONS
ICDT 2018 TEST-OF-TIME AWARD

Nominations are solicited for the ICDT 2018 Test of Time Award. In order to provide comprehensive coverage of past ICDT conferences, the ICDT Council has determined that the ICDT 2018 ToT award will cover papers published in ICDT 1999 and 2001.

The ICDT 2018 ToT Award Committee consists of Pablo Barcelo, Richard Hull, and Victor Vianu. The committee will select the paper or a small number of papers from the ICDT 1999 and 2001 proceedings that has had the most impact in terms of research, methodology, conceptual contribution, or transfer to practice. All papers are nominated by default, but the committee welcomes input from our community. Please feel free to nominate a paper if you think it has had great impact, even if you have not thoroughly compared it to the other eligible papers. The usual conflict of interest rules apply.

Please email your nominations to Pablo Barcelo (pbarcelo@gmail.com) with subject line "ICDT 2018 ToT Award nomination" together with a brief justification. Please send your nominations no later than November 1, 2017. Nominations are confidential and will only be shared among the committee members.

The ICDT ToT award for 2018 will be presented during the EDBT/ICDT 2018 Joint Conference, March 26-29, 2018 in Vienna, Austria.

The ICDT 1999 papers can be found at http://dblp.uni-trier.de/db/conf/icdt/icdt99.html and the ICDT 2001 papers at http://dblp.uni-trier.de/db/conf/icdt/icdt2001.html