SIGMOD Officers, Committees, and Awardees

Chair	Vice-Chair	Secretary/Treasurer
Donald Kossmann Systems Group ETH Zürich Cab F 73 8092 Zuerich SWITZERLAND +41 44 632 29 40 <donaldk AT inf.ethz.ch>	Anastasia Ailamaki School of Computer and Communication Sciences, EPFL EPFL/IC/IIF/DIAS Station 14, CH-1015 Lausanne SWITZERLAND +41 21 693 75 64 <natassa AT epfl.ch>	Magdalena Balazinska Computer Science & Engineering University of Washington Box 352350 Seattle, WA USA +1 206-616-1069 <magda AT cs.washington.edu>

SIGMOD Executive Committee:
Donald Kossmann (Chair), Anastasia Ailamaki (Vice-Chair), Magdalena Balazinska, K. Selçuk Candan, Yanlei Diao, Curtis Dyreson, Yannis Ioannidis, Christian Jensen, and Tova Milo.

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

SIGMOD Information Director:
Curtis Dyreson, Utah State University <curtis.dyreson AT usu.edu>

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

SIGMOD Record Editor-in-Chief:
Yanlei Diao, University of Massachusetts Amherst <yanlei AT cs.umass.edu>

SIGMOD Record Associate Editors:
Vanessa Braganholo, Marco Brambilla, Chee Yong Chan, Rada Chirkova, Zachary Ives, Anastasios Kementsietsidis, Jeffrey Naughton, 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:
Tova Milo (Chair), Diego Calvanse, Wenfei Fan, Martin Grohe, Rick Hull, Maurizio Lenzerini

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

Awards Committee:
Elisa Bertino (Chair), Surajit Chaudhuri, Maurizio Lenzerini, Kartin Kersten, Umesh Dayal

Jim Gray Doctoral Dissertation Award Committee:
Tova Milo (Co-Chair), Juliana Freire (Co-Chair), Ashraf Aboulnaga, Minos Garofalakis, Chris Jermaine, Renee Miller, Aditya Parameswaran, Andy Pavlo, Kian-Lee Tan.

SIGMOD Officers, Committees, and Awardees (continued)

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. Formerly known as the "SIGMOD Innovations Award", it now honors Dr. E. F. (Ted) Codd (1923 - 2003) who invented the relational data model and was responsible for the significant development of the database field as a scientific discipline. Recipients of the award are the following:

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

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)	Gio Wiederhold (1995)	Yahiko Kambayashi (1995)
Jeffrey Ullman (1996)	Avi Silberschatz (1997)	Won Kim (1998)
Raghu Ramakrishnan (1999)	Michael Carey (2000)	Laura Haas (2000)
Daniel Rosenkrantz (2001)	Richard Snodgrass (2002)	Michael Ley (2003)
Surajit Chaudhuri (2004)	Hongjun Lu (2005)	Tamer Özsu (2006)
Hans-Jörg Schek (2007)	Klaus R. Dittrich (2008)	Beng Chin Ooi (2009)
David Lomet (2010)	Gerhard Weikum (2011)	Marianne Winslett (2012)
H.V. Jagadish (2013)	Kyu-Young Whang (2014)	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, University of Washington. *Honorable Mentions*: Marcelo Arenas and Yanlei Diao.
- **2007** *Winner*: Boon Thau Loo, University of California at Berkeley. *Honorable Mentions*: Xifeng Yan and Martin Theobald.
- **2008** *Winner*: Ariel Fuxman, University of Toronto. *Honorable Mentions*: Cong Yu and Nilesh Dalvi.
- **2009** *Winner*: Daniel Abadi, MIT. *Honorable Mentions*: Bee-Chung Chen and Ashwin Machanavajjhala.
- **2010** *Winner:* Christopher Ré, University of Washington. *Honorable Mentions*: Soumyadeb Mitra and Fabian Suchanek.
- **2011** *Winner*: Stratos Idreos, Centrum Wiskunde & Informatica. *Honorable Mentions*: Todd Green and Karl Schnaitterz.
- **2012** *Winner*: Ryan Johnson, Carnegie Mellon University. *Honorable Mention*: Bogdan Alexe.
- **2013** *Winner*: Sudipto Das, University of California, Santa Barbara. *Honorable Mention*: Herodotos Herodotou and Wenchao Zhou.
- **2014** *Winners*: Aditya Parameswaran, Stanford University, and Andy Pavlo, Brown University.
- **2015** *Winners*: Alexander Thomson, Yale University. *Honorable Mentions*: Marina Drosou, University of Ioannina and Karthik Ramachandra, IIT Bombay

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

[Last updated : March 31, 2016]

Editor's Notes

Welcome to the March 2016 issue of the ACM SIGMOD Record!

The new year of 2016 begins with five new members of the SIGMOD Record Editorial Broad, as well as new reforms that are currently taking place in the SIGMOD Record.

First, I would like to welcome five new members of the SIGMOD Record Editorial Broad:

- Zack Ives, Jeff Naughton, and Wang-Chiew Tan joined as Associate Editors of the Special Issue of Research Highlights;
- Frank Neven joined as Associate Editor of the Database Principles Column; and
- Huiping Cao joined as our new Information Director.

I am also pleased to announce that we have launched a new website of the SIGMOD Record:

http://sigmod.org/sigmodrecord/

Besides a modern look, this website provides new features such as leaving comments on published articles and searching for articles in a specific column in the previous issues. We will continue to improve the website and your feedback will be appreciated.

Next, I am thrilled to introduce the inaugural **ACM SIGMOD Research Highlight Award**, which is a new award for the database community to showcase a set of research projects that exemplify core database research. In particular, these projects address an important problem, represent a definitive milestone in solving the problem, and have the potential of significant impact. The initiative of the new award also aims to make the selected works widely known in the database community, to our industry partners, and potentially to the broader ACM community.

The March 2016 issue of the SIGMOD Record is a special issue dedicated to the 2015 ACM SIGMOD Research Highlights. The award committee and editorial board included Zack Ives, Jeff Naughton, Wang-Chiew Tan, and Yanlei Diao. We solicited articles from SIGMOD 2015, PODS 2015, and VLDB 2015, out of which 8 articles were selected as 2015 Research Highlights. The authors of each article worked closely with one associate editor to rewrite the article into a compact 8-page format, and improve it to appeal to the broad data management community.

The 2015 **Research Highlights** cover a broad set of topics, ranging from a new interface for human-database interaction, to multi-objective query optimization, to query answering in the face of inconsistent data, to new theory and practice for distributed query processing and optimization, to knowledge base construction and clustering of time series data. Each research highlight is also accompanied by a one-page **Technical Perspective**, which was written by our associate editor or an external expert on a given research topic. The technical perspective provides readers with an overview of the underlying motivation, the important ideas of the featured Research Highlight, and its scientific and practical significance.

Finally, this special issue closes with a message from the Editor-in-Chief of ACM TODS and Call for Papers for ACM SoCC 2016.

On behalf of the SIGMOD Record Editorial board, I hope that you all enjoy reading the March 2016 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:
http://sigmod.org/sigmodrecord/authors/

Yanlei Diao

March 2016

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)

Technical Perspective: Natural Language to SQL Translation by Iteratively Exploring a Middle Ground

Jeffrey F. Naughton
University of Wisconsin–Madison
Madison WI, U.S.A.

A fundamental question in data management is how relational database management systems (RDBMSs) should be queried. Ideally, the query interface should be powerful enough to express arbitrary queries, yet simple enough to learn that users require virtually no training. Natural language is an obvious and appealing approach – presumably most users already know at least one natural language and use it to "query" other humans constantly. Unfortunately, employing natural language to query RDBMSs is highly nontrivial, and for the most part, not used. However, with the growing power and ubiquity of Natural Language Processing (NLP) systems, it makes sense to redouble efforts in applying NLP to database querying.

At the most basic level, relational database systems are queried using SQL. (For that matter, most "NoSQL" systems are also queried using SQL.) SQL is very powerful and precise, and, for novices, very hard to write. So SQL cannot be used as a user interface for anyone but power users. Nonetheless, as the most widely used RDBMS query language, SQL is the most natural language into which to translate natural language questions over relational data. This translation is the focus of the following paper, "Understanding Natural Language Queries over Relational Databases", by Li and Jagadish.

The first important decision made by the authors of this paper is to reject a one-shot, one-way translation process from a natural language query to a corresponding SQL query. Instead, the authors advocate an iterative dialog between the person posing the query and the system building the relational query. This makes perfect sense – even in the much simpler world of keyword search systems, users iteratively refine their queries. Unfortunately, adopting this approach for RDBMS querying does not yield an easy problem – in fact, it uncovers a highly interesting and difficult challenge: how should the user and the system communicate in this iterative process?

Answering this question is difficult. Unlike the case for keyword search systems, the answer to the query may not help the user know if the executed query was what they really wanted. For example, consider the simple query "find the difference between sales this year and last year." In general the RDBMS will return a number – and it is very hard to tell just from that number if the query was correct or not. It would be far more precise for the system to respond to the user by presenting the generated SQL query itself. But this would require the person posing the natural language query to be able to read and understand SQL, which contradicts a major motivation for the system in the first place.

Now we come to what is perhaps the heart of this paper: the decision to adopt an intermediate language the authors call "Query Tree," a two-way domain-independent communication model allowing the user and system to understand one other. A query tree aids mapping a user query to its corresponding semantically correct SQL and translating a query plan to its corresponding natural language interpretation. The authors harness the schema knowledge, schema-driven similarity metrics, query tree reformulation and ranking to make the problem tractable for the system and the user.

The authors close with a user study evaluating the approach. The user study itself is interesting, including the aspect of using Chinese to convey the queries to the subjects instead of English to avoid bias through the phrasing in the query description (presumably the subjects already spoke Chinese!) The experiments show that the approach is best for simple to medium complexity queries.

This paper represent a significant improvement in the state of the art, and it is an ideal springboard for future advances. In an area as difficult and important as natural language querying of relational database systems, this is indeed a major contribution.

Understanding Natural Language Queries over Relational Databases

Fei Li
Univ. of Michigan, Ann Arbor
lifei@umich.edu

H. V. Jagadish
Univ. of Michigan, Ann Arbor
jag@umich.edu

ABSTRACT

Natural language has been the holy grail of query interface designers, but has generally been considered too hard to work with, except in limited specific circumstances. In this paper, we describe the architecture of an interactive natural language query interface for relational databases. Through a carefully limited interaction with the user, we are able to correctly interpret complex natural language queries, in a generic manner across a range of domains. By these means, a logically complex English language sentence is correctly translated into a SQL query, which may include aggregation, nesting, and various types of joins, among other things, and can be evaluated against an RDBMS. We have constructed a system, NaLIR (Natural Language Interface for Relational databases), embodying these ideas. Our experimental assessment, through user studies, demonstrates that NaLIR is good enough to be usable in practice: even naive users are able to specify quite complex ad-hoc queries.

1. INTRODUCTION

Querying data in relational databases is often challenging. SQL is the standard query language for relational databases. While expressive and powerful, SQL is too difficult for users without technical training. As the database user base is shifting towards non-experts, designing user-friendly query interfaces will be an important goal in the database community.

In the real world, people ask questions in natural language, such as English. Not surprisingly, a natural language interface is regarded by many as the ultimate goal for a database query interface, and many natural language interfaces to databases (NLIDBs) have been built towards this goal [1, 7, 9, 11, 13]. NLIDBs have many advantages over other widely accepted query interfaces (keyword-based search, form-based interface, and visual query builder). For example, a typical NLIDB would enable naive users to specify *complex*, *ad-hoc* query intent *without training*. In contrast, flat-structured keywords are often insufficient to convey complex query intent, form-based interfaces can be used only when queries are predictable and limited to the encoded logic, and visual query builders still requires extensive schema knowledge of the user.

Despite these advantages, NLIDBs have not been adopted widely. The fundamental problem is that understanding natural language is hard. Even in human-to-human interaction, there are miscommunications. Therefore, we cannot reasonably expect an NLIDB to be perfect. Therefore, users may be provided with a wrong answer due to the system incorrectly handling the query, and in many cases, it is impossible for users to verify the answer by themselves.

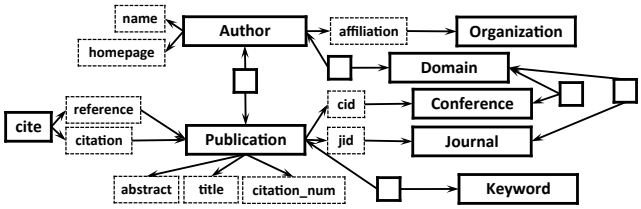
When humans communicate with one another in natural language, the query-response cycle is not as rigid as in a traditional database system. If a human is asked a query that she does not understand, she will seek clarification. She may do so by asking specific questions back, so that the question-asker understands the point of potential confusion. She may also do so by stating explicitly how she interpreted the query. Drawing inspiration from this natural human behavior, we design the query mechanism to facilitate collaboration between the system and the user in processing natural language queries. First, the system explains how it interprets a query, from each ambiguous word/phrase to the meaning of the whole sentence. These explanations enable the user to verify the answer and to be aware where the system misinterprets her query. Second, for each ambiguous part, we provide multiple likely interpretations for the user to choose from. Since it is often easier for users to recognize an expression rather than to compose it, we believe this query mechanism can achieve satisfactory reliability without burdening the user too much.

A question that then arises is how should a system represent and communicate its query interpretation to the user. SQL is too difficult for most non-technical humans. We need a representation that is both "human understandable" and "RDBMS understandable." In this paper, we present a data structure, called *Query Tree*, to meet this goal. As an intermediate between a linguistic parse tree and a SQL statement, a query tree is easier to explain to the user than a SQL statement. Also, given a query tree verified by the user, the system will almost always be able to translate it into a correct SQL statement.

Putting the above ideas together, we propose an NLIDB comprising three main components: a first component that transforms a natural language query to a query tree, a second component that verifies the transformation interactively with the user, and a third component that translates the query tree into a SQL statement. We have constructed such an NLIDB, and we call it a NaLIR (Natural Language Interface to Relational databases).

The remaining parts of the paper are organized as follows. We discuss the query mechanism in Section 2. The system architecture of our system is described in Section 3. Given a query, we show how to interpret each its words/phrases in

The original version of this article was published in PVLDB 2014.



Query 1: Return the average number of publications by Bob in each year.
Query 2: Return authors who have more papers than Bob in VLDB after 2000.
Query 3: Return the conference in each area whose papers have the most total citations.

Figure 1: A Simplified Schema for Microsoft Academic Search and Sample Queries.

Section 4 and infer the semantic meaning of the whole query (represented by a query tree) in Section 5. We discuss how our system translates a query tree to a SQL statement in Section 6. In Section 7, our system is evaluated experimentally. We discuss related work in Section 8 and draw conclusions in Section 9.

2. QUERY MECHANISM

Search engines are popular and effective in inferring the user intent from limited information. As we think about the architecture of an NLIDB, it is worthwhile to draw inspiration from search engines. First, given a query, a search engine returns a list of results, rather than a single result. This is central to providing acceptable recall. Second, users are able to verify whether a result is correct (useful) by reading the abstract/content. Third, these results are well ranked, to minimize user burden to verify potential answers. These strategies work very well in search engines. However, due to some fundamental differences between search engines and NLIDBs, as we will discuss next, this query mechanism cannot be directly applied to NLIDBs.

First, users are often able to verify the results from a search engine by just reading the results. However, the results returned by an NLIDB cannot usually explain themselves. For example, suppose a user submits Query 1 in Figure 1 to an NLIDB and gets an answer "5." How could she verify whether the system understands her query and returns to her the correct answer?

Second, unlike search engines, users tend to express *sophisticated query logics* to an NLIDB and expect *perfect results*. That requires the NLIDB to fix all the ambiguities correctly. However, when a natural language query has a complex structure, it may contain several ambiguities that cannot be fixed by the system with confidence. As a result, the candidate interpretations of the query are often permutations of the solutions for each single ambiguity. Such interpretations are hard to rank and their number often grows exponentially with the number of such ambiguities. Users will be frustrated in verifying them.

Given the above two observations, instead of explaining the query results, we explain the query interpretation process, especially how each ambiguity is fixed, to the user. In our system, we fix each "easy" ambiguity quietly. For each "hard" ambiguity, we provide multiple interpretations for the user to choose from. In such a way, even for a rather complex natural language query, verifications for 3-4 ambiguities is enough, in which each verification is just making choices from several options.

The ambiguities in processing a natural language query are not often independent of each other. The resolution of some ambiguities depends on the resolution of some other ambiguities. For example, the interpretation of the whole sentence depends on how each of its words/phrases is interpreted. So the disambiguation process and the verification process should be organized in a few steps. In our system, we organize them in three steps, as we will discuss in detail in the next section. In each step, for a "hard" ambiguity, we generate multiple interpretations for it and, at the same time, use the best interpretation as the default choice to process later steps. Each time a user changes a choice, our system immediately reprocesses all the ambiguities in later steps and updates the query results.

3. SYSTEM OVERVIEW

Figure 2 depicts the architecture of NaLIR. The entire system we have implemented consists of three main parts: the query interpretation part, interactive communicator and query tree translator. The query interpretation part, which includes *parse tree node mapper* (Section 4) and *structure adjustor* (Section 5), is responsible for interpreting the natural language query and representing the interpretation as a query tree. The *interactive communicator* is responsible for communicating with the user to ensure that the interpretation process is correct. The query tree, possibly verified by the user, will be translated into a SQL statement in the *query tree translator* (Section 6) and then evaluated against an RDBMS.

Dependency Parser. The first obstacle in translating a natural language query into a SQL query is to understand the natural language query linguistically. In our system, we use the Stanford Parser [2] to generate a linguistic parse tree from the natural language query. The linguistic parse trees in our system are dependency parse trees, in which each node is a word/phrase specified by the user while each edge is a linguistic dependency relationship between two words/phrases. The simplified linguistic parse tree of Query 2 in Figure 1 is shown in Figure 3 (a).

Parse Tree Node Mapper. The parse tree node mapper identifies the nodes in the linguistic parse tree that can be mapped to SQL components and tokenizes them into different tokens. In the mapping process, some nodes may fail in mapping to any SQL component. In this case, our system generates a warning to the user, telling her that these nodes do not directly contribute in interpreting her query. Also, some nodes may have multiple mappings, which causes ambiguities in interpreting these nodes. For each such node, the parse tree node mapper outputs the best mapping to the parse tree structure adjustor by default and reports all candidate mappings to the interactive communicator.

Parse Tree Structure Adjustor. After the node mapping (possibly with interactive communications with the user), we assume that each node is understood by our system. The next step is to correctly understand the tree structure from the database's perspective. However, this is not easy since the linguistic parse tree might be incorrect, out of the semantic coverage of our system or ambiguous from the database's perspective. In those cases, we adjust the structure of the linguistic parse tree and generate candidate interpretations (query trees) for it. In particular, we adjust the structure of the parse tree in two steps. In the first step, we reformulate the nodes in the parse tree to make it fall in the syntactic

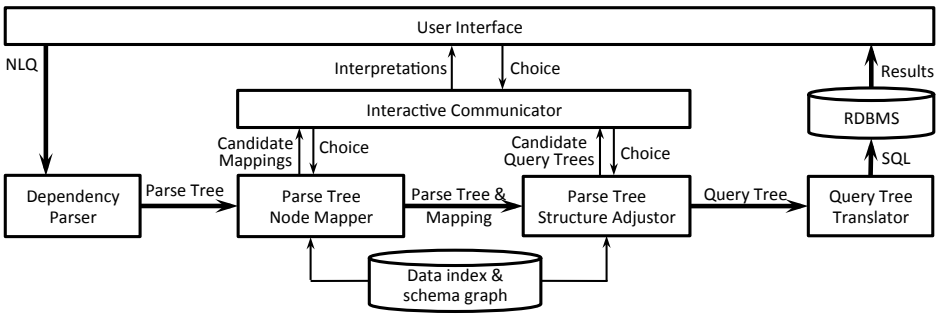


Figure 2: System Architecture.

coverage of our system (valid parse tree). If there are multiple candidate valid parse trees for the query, we choose the best one as default input for the second step and report top k of them to the interactive communicator. In the second step, the chosen (or default) valid parse tree is analyzed semantically and implicit nodes are inserted to make it more semantically reasonable. This process is also under the supervision of the user. After inserting implicit nodes, we obtain the exact interpretation, represented as a query tree, for the query.

Interactive Communicator. In case the system possibly misunderstands the user, the interactive communicator explains how her query is processed. In our system, interactive communications are organized in three steps, which verify the intermediate results in the parse tree node mapping, parse tree structure reformulation, and implicit node insertion, respectively. For each ambiguous part, we generate a multiple choice selection panel, in which each choice corresponds to a different interpretation. Each time a user changes a choice, our system immediately reprocesses all the ambiguities in later steps.

Example 1. *Consider the linguistic parse tree T in Figure 3(a). In the first step, the parse tree node mapper generates the best mapping for each node (represented as M and shown in Figure 3 (b)) and reports to the user that the node "VLDB" maps to "VLDB conference" and "VLDB Journal" in the database and that our system has chosen "VLDB conference" as the default mapping. According to M, in the second step, the parse tree adjustor reformulates the structure of T and generates the top k valid parse trees $\{T_i^M\}$, in which T_1^M (Figure 3 (c)) is the best. The interactive communicator explains each of the k valid parse trees in natural language for the user to choose from. For example, T_1^M is explained as "return the authors, where the papers of the author in VLDB after 2000 is more than Bob." In the third step, T_1^M is fully instantiated in the parse tree structure adjustor by inserting implicit nodes (shown in Figure 3 (d)). The result query tree T_{11}^M is explained to the user as "return the authors, where the number of papers of the author in VLDB after 2000 is more than the number of paper of Bob in VLDB after 2000.", in which the underline part can be canceled by the user. When the user changes the mapping strategy M to M', our system will immediately use M' to reprocess the second and third steps. Similarly, if the user choose T_i^M instead of T_1^M as the best valid parse tree, our system will fully instantiate T_i^M in the third step and update the interactions.*

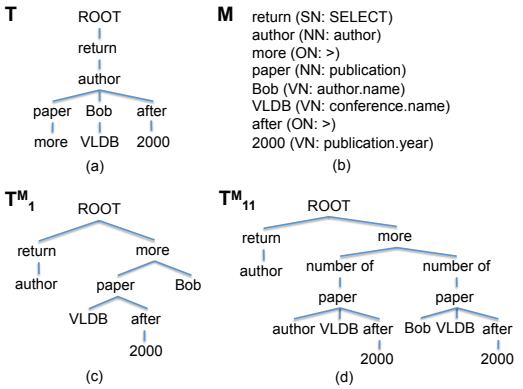


Figure 3: (a) Linguistic Parse Tree. (b) Node Mapping Strategy. (c) Valid Parse Tree. (d) Query Tree.

Query Tree Translator. Given the query tree verified by the user, the translator utilizes its structure to generate appropriate structure in the SQL expression and completes the foreign-key-primary-key (FK-PK) join paths. The result SQL statement may contain aggregate functions, multi-level subqueries, and various types of joins, among other things. Finally, our system evaluates the translated SQL statement against an RDBMS and returns the query results.

4. PARSE TREE NODE INTERPRETATION

To understand the linguistic parse tree from the database's perspective, we first need to identify the parse tree nodes that can be mapped to SQL components. Such nodes can be further divided into different types as shown in Figure 4, according to the type of SQL components they mapped to. The identification of select node, operator node, function node, quantifier node and logic node is independent of the database being queried. In NaLIR, enumerated sets of phrases are served as the real world "knowledge base" to identify these five types of nodes.

In contrast, name nodes and value nodes are entirely depend on the database being queried. For name nodes, we use the WUP similarity function [16] (based on Wordnet) to evaluate the similarity in meaning and adopt the square root of the Jaccard Coefficient to evaluate the similarity in spelling [17]. For value nodes, we only use the Jaccard Coefficienct to evaluate their similarity in spelling. When their similarity is above a predefined threshold, we say that the schema element/value is a candidate mapping for the node.

Node Type	Corresponding SQL Component
Select Node (SN)	SQL keyword: SELECT
Operator Node (ON)	an operator, e.g. =, <=, !=, contains
Function Node (FN)	an aggregation function, e.g., AVG
Name Node (NN)	a relation name or attribute name
Value Node (VN)	a value under an attribute
Quantifier Node (QN)	ALL, ANY, EACH
Logic Node (LN)	AND, OR, NOT

Figure 4: Different Types of Nodes.

1 Q -> (SClause)(ComplexCondition)*
2 SClause -> SELECT + GNP
3 ComplexCondition -> ON + (leftSubtree*rightSubtree)
4 leftSubtree -> GNP
5 rightSubtree -> GNP|VN|MIN|MAX
6 GNP -> (FN + GNP) | NP
7 NP -> NN + (NN)*(Condition)*
8 condition -> VN | (ON + VN)

\+ represents a parent-child relationship
* represents a sibling relationship

Figure 5: Grammar of Valid Parse Trees.

5. PARSE TREE STRUCTURE ADJUSTMENT

Given the correct mapping strategy, each node in the linguistic parse tree can be perfectly understood by our system. In this section, we infer the relationship between the nodes in the linguistic parse tree from the database's perspective and then understand the whole query. However, three obstacles lie in the way of reaching this goal.

First, the linguistic parse tree generated from an off-the-shelf parser may be incorrect. Second, the structure of the linguistic parse tree does not directly reflect the relationship between the nodes from the database's perspective. Consider the following three sentence fragments: (a) author who has more than 50 papers, (b) author who has more papers than Bob, and (c) author whose papers are more than those of Bob. The linguistic parse structures of these three sentence fragments are very different while their semantic meanings are similar. Third, natural language sentences often contain elliptical expressions. Take the parse tree in Figure 3 (c) as an example. Although the relationship between each pair of nodes is clear, it still has multiple possible interpretations.

In this section, we describe the construction of the Parse Tree Structure Adjustor in detail.

5.1 Query Tree

The semantic coverage of our system is essentially constrained by the expressiveness of SQL. So, given a database, we represent our semantic coverage as a subset of parse trees, in which each such parse tree explicitly corresponds to a SQL statement and all such parse trees could cover all possible SQL statements (with some constraints). We call such parse trees *Query Tree*s. As such, interpreting a natural language query (currently represented by a linguistic parse tree and the mapping for each its node) is indeed the process of mapping the query to its corresponding query tree in the semantic coverage.

We defined in Figure 5 the grammar of the parse trees that are syntactically valid in our system (all terminals are different types of nodes defined in Figure 4.). Query trees are the syntactically valid parse trees whose semantic meanings are reasonable, which will be discussed in Section 5.3, or approved by the user. Given the three obstacles in interpreting a linguistic parse tree, as we have discussed before, there is often a big gap between the linguistic parse tree and its corresponding query tree, which makes the mapping between them difficult. In our system, we take the following two strategies to make the mapping process accurate.

First, our system explains a query tree in natural language, which enables the user to verify it. Query trees are intermediates between natural language sentences and SQL statements. Thus the translation from a query tree to a natural language sentence is quite straightforward, compared to that from a SQL statement [5].

Second, given a natural language query, our system will generate multiple candidate query trees for it, which can significantly enhance the probability that one of them is correct. The problem is that, when the query is complex, there may be many candidate query trees, which are similar to each other. To show the user more candidate query trees without burdening them too much in verifying them, we do the mapping in two rounds and communicate with the user after each round. In the first round, we return the top k parse trees, which are syntactically valid according to the grammar defined and can be obtained by only reformulating the nodes in the parse tree. Each such parse tree represents a rough interpretation for the query and we call them valid parse trees. In the second round, if there are any implicit nodes, they will be inserted to the chosen (or default) valid parse tree to generate its exact interpretation. Our system inserts implicit nodes one by one under the supervision of the user. In such a way, suppose that there are k' possible implicit nodes in each of the k valid parse tree, the user only needs to verify k valid parse trees and k' query trees instead of all $k * 2^{k'}$ candidate query trees. Figure 3 (c) shows a valid parse tree generated in the first round, while this valid parse tree is full-fledged to the query tree in Figure 3 (d) after inserting implicit nodes.

5.2 Parse Tree Reformulation

In this section, given a linguistic parse tree, we reformulate it in multiple ways and generate its top k rough interpretations. The basic idea in the algorithm is to use subtree move operations to edit the parse tree until it is syntactically valid according to the grammar we defined. The resulting algorithm is shown in Figure 6. Each time, we use the function $adjust(tree)$ to generate all the possible parse trees in one subtree move operation (line 6). Since the number of possible parse trees grows exponentially with the number of edits, the whole process would be slow. To accelerate the process, our algorithm evaluates each new generated parse tree and filter out bad parse trees directly (line 11 - 12). Also, we hash each parse tree into a number and store all the hashed numbers in a hash table (line 10). By checking the hash table (line 8), we can make sure that each parse tree will be processed at most once. We also set a parameter t as the maximum number of edits approved (line 8). Our system records all the valid parse trees appeared in the reformulation process (line 13 - 14) and returns the top k of them for the user to choose from (line 15 - 16). Since our algorithm stops after t edits and retains a parse tree only if it is no

worse than its corresponding parse tree before the last edit (line 8), some valid parse trees may be omitted.

Algorithm 1: QueryTreeGen(*parseTree*)

```
1: results ← φ; HT ← φ
2: PriorityQueue.push(parseTree)
3: HT.add(h(tree))
4: while PriorityQueue ≠ φ do
5:   tree = PriorityQueue.pop()
6:   treeList = adjust(tree)
7:   for all tree' ∈ treeList do
8:     if tree' not exists in HT && tree'.edit < t then
9:       tree'.edit = tree.edit+1;
10:      HT.add(h(tree'));
11:      if evaluate(tree') >= evaluate(tree) then
12:        PriorityQueue.add(tree')
13:        if tree' is valid
14:          results.add(tree')
15: rank(results)
16: Return results
```

Figure 6: Parse Tree Reformulation Algorithm.

To filter out bad parse trees in the reformulating process and rank the result parse trees, we evaluate whether a parse tree is desirable from three aspects.

First, a good parse tree should be valid according to the grammar defined in Figure 5. Second, the nodes next to each other in the parse tree should be close to each other on the schema graph. Third, the parse tree should be similar to the original linguistic parse tree, which is measured by the number of the subtree move operations used in the transformation. When ranking the all the generated parse trees (line 15 in Figure 6), our system takes all the three factors into account. However, in the tree adjustment process (line 11), to reduce the cases when the adjustments stop in local optima, we only consider the first two factors, in which the first factor dominates the evaluation.

5.3 Implicit Nodes Insertion

Natural language sentences often contain elliptical expressions, which make some nodes in their parse trees implicit. In this section, for a rough interpretation, which is represented by a valid parse tree, we obtain its exact interpretation by detecting and inserting implicit nodes.

In our system, implicit nodes mainly exist in complex conditions, which correspond to the conditions involving aggregations, nestings, and non-FKPK join constraints. As can be derived from Figure 5, the semantic meaning of a complex condition is its comparison operator node operating on its left and right subtrees. When implicit nodes exist, such syntactically valid conditions are very likely to be semantically "unreasonable." To detect such unreasonable query logics, we define the concept of core node.

Definition 1 (Core Node). *Given a complex condition, its left (resp. right) core node is the name node that occurs in its left (right) subtree with no name node as ancestor.*

Inspired from [19], given a complex condition, we believe that its left core node and right core node are the concepts that are actually compared. So they should have the same type (map to the same schema element in the database). When they are in different types, we believe that the actual right core node, which is of the same type as the left core node, is implicit.

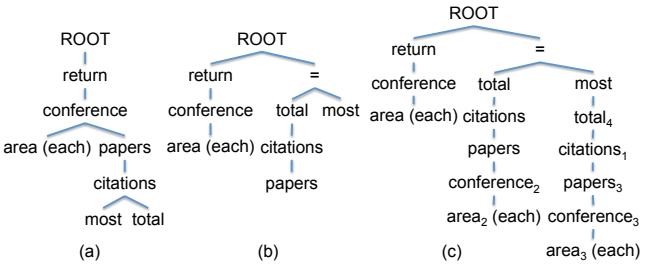


Figure 7: (a) Linguistic Parse Tree for Query 3 in Figure 1. (b) Valid Parse Tree. (c) Query Tree.

The name nodes in a left subtree are always related to the name nodes under the "SELECT" node. Take the parse tree in Figure 7 (b) as an example. The nodes "conference" and "area" are related to the nodes "citations" and "paper." In our system, we connect them by duplicating the name nodes under the "SELECT" node and inserting them to left subtree. Each of the nodes inserted in this step is considered as the same entity with its original node and marked "outside" for the translation in Section 6.

Furthermore, the constraints for the left core node and the right core node should be consistent. Consider the parse tree in Figure 3 (c). Its complex condition compares the number of papers by an author in VLDB after 2000 with the number of all the papers by Bob (in any year on any conference or journal), which is unfair. As such, the constraints of "in VLDB" and "after 2000" should be added to the right subtree. Note that the nodes duplicated from "outside" nodes are also marked "outside" and are considered corresponding to the same entity with the original nodes.

The last kind of implicit node is the function node. We consider two cases where function nodes may be implicit. First, the function node "count" is often implicit in the natural language sentences. Consider the parse tree in Figure 3 (c). The node "paper" is the left child of node "more" and it maps to the relation "Publication", which is not a number attribute. The comparison between papers is unreasonable without a "count" function. Second, the function nodes operating on the left core node should also operate on the right core node. Figure 7 (c) shows an example for this case. We see that the function node "total" operates on the left core node "citations" but does not operate on the right core node "$citations_1$." Our system detects such implicit function node and insert "$total_4$" to the right core node.

In our system, the detection and insertion of implicit nodes is just an inference of the semantic meaning for a query, which cannot guarantee the accuracy. As such, the whole process is done under the supervision of the user.

6. SQL GENERATION

In the cases when the query tree does not contain function nodes or quantifier nodes, which means the target SQL query will not have aggregate functions or subqueries, the translation is quite straightforward. The schema element mapped by the name node under the SELECT node is added to the SELECT clause. Each value node (together with its

```
1. Block 2: SELECT SUM(Publication.citation_num) as sum_citation,
2.                  Conference.cid, Domain.did
3.          FROM Publication, Conference, Domain, ConferenceDomain
4.          WHERE Publication.cid = Conference.cid
5.                  AND Conference.cid = ConferenceDomain.cid
6.                  AND ConferenceDomain.did = Domain.did
7.          GROUP BY Conference.cid, Domain.did

8. Block 3: SELECT MAX(block4.sum_citation) as max_citation,
9.                  block4.cid, block4.did
10.         FROM (CONTENT OF BLOCK4) as block4
11.         GROUP BY block4.did

12.Block 1: SELECT Conference.name, Domain.name
13.         FROM Conference, Domain, ConferenceDomain
14.                 (CONTENT OF BLOCK2) as block2
15.                 (CONTENT OF BLOCK3) as block3
16.         WHERE Conference.cid = ConferenceDomain.cid
17.                 AND ConferenceDomain.did = Domain.did
18.                 AND block2.citation_num = block3.max_citation
19.                 AND Conference.cid = block2.cid
20.                 AND Conference.cid = block3.cid
21.                 AND Domain.did = block2.did
22.                 AND Domain.did = block3.did
```

Figure 8: Translated SQL Statement

operation node if specified) is translated to a selection condition and added to the WHERE clause. Finally, a FK-PK join path is generated, according to the schema graph, to connect each name node and its neighbors. Such an FK-PK join path is translated into a series of FK-PK join conditions and all the schema elements in the FK-PK join path are added to the FROM clause.

When the query tree contains function nodes or quantifier nodes, the target SQL statements will contain subqueries. In our system, we use the concept of *block* to clarify the scope of each target subquery.

Definition 2 (Block). *A block is a subtree rooted at the select node, a name node that is marked "all" or "any", or a function node. The block rooted at the select node is the main block, which will be translated to the main query. Other blocks will be translated to subqueries. When the root of a block b_1 is the parent of the root of another block b_2, we say that b_1 is the direct outer block of b_2 and b_2 is a direct inner block of b_1. The main block is the direct outer block of all the blocks that do not have other outer blocks.*

Given a query tree comprising multiple blocks, we translate one block at a time, starting from the innermost block, so that any correlated variables and other context is already set when outer blocks are processed.

Example 2. *The query tree shown in Figure 7 (c) consists of four blocks: b_1 rooted at node "return", b_2 rooted at node "total", b_3 rooted at node "most", and b_4 rooted at node "total$_4$." b_1 is the main block, which is the direct outer block of b_2 and b_3. b_3 is the direct outer block of b_4. For this query tree, our system will first translate b_2 and b_4, then translate b_3 and finally translate the main block b_1.*

For each single block, the major part of its translation is the same as the basic translation as we have described. In addition, some SQL fragments must be added to specify the relationship between these blocks.

Example 3. *Consider the query tree in Figure 7 (c), whose target SQL statement is shown in Figure 8 (the block b_4 is omitted since it is almost the same as b_2). In the query tree, b_4 is included by b_2 while b_2 and b_3 are included by b_1 as direct inner blocks. Thus their corresponding subqueries are added to the FROM clause of their direct outer blocks (line 10 and line 14 - 15). The complex condition rooted at node "=" is translated to the condition in line 18. The nodes "conference$_2$", "area$_2$", "conference$_3$" and "area$_3$" are marked "outside" in the implicit node insertion, which means they correspond to the same entity as the nodes "conference" and "area" in the main block. These relationships are translated to the conditions in line 19 - 22.*

Relation	#tuples	Relation	#tuples
Publication	2.45 M	Author	1.25 M
cite	20.3 M	Domain	24
Conference	2.9 K	Journal	1.1 K
Organizations	11 K	Keywords	37 K

Figure 9: Summary Statistics for MAS Database.

7. EXPERIMENTS

The motivation of our system is to enable non-technical users to compose logically complex queries over relational databases and get perfect query results. So, there are two crucial aspects we must evaluate: the quality of the returned results (effectiveness) and whether our system is easy to use for non-technical users (usability).

Effectiveness. Evaluating the effectiveness of NaLIR is a challenging task. The objective in NaLIR is to allow users to represent SQL statements using natural language. In our experiments, the effectiveness of our system was evaluated as the percentage of the queries that were perfectly answered by our system. (Note that this is a stiff metric, in that we get zero credit if the output SQL query is not perfect, even if the answer set has a high overlap with the desired answer). Since the situations where users accept imperfect/wrong answers would cause severe reliability problems, for the cases when the answers were wrong, we recorded whether the users were able to recognize such failures, whether from the answers themselves or from the explanations generated by our system. Also, for the failure queries, we analyzed the specific reasons that caused such failures.

Usability. For the correctly processed queries, we recorded the actual time taken by the participants. In addition, we evaluated our system subjectively by asking each participant to fill out a post-experiment questionnaire.

7.1 Experiments Design

The experiment was a user study, in which participants were asked to finish the query tasks we designed for them.

Data Set and Comparisons. We used the data set of Microsoft Academic Search (MAS). Its simplified schema graph and summary statistics are shown in Figure 1 and Figure 9, respectively. We chose this data set because it comes with an interesting set of (supported) queries, as we will discuss next.

We compared our system with the faceted interface of the MAS website. The website has a carefully designed ranking system and interface. By clicking through the site, a user is able to get answers to many quite complex queries. We enumerated all query logics that are "directly supported" by the MAS website and can be accomplished by SQL state-

Easy: Return all the conferences in database area.
Medium: Return the number of papers in each database conference.
Hard: Return the author who has the most publications in database area.

Figure 10: Sample Queries in Different Complexity.

ments. "Directly supported" means that the answer of the query can be obtained in a single webpage (or a series of continuous webpages) without further processing. Through exhaustive enumeration, we obtained a set of 196 query logics and we marked their complexity according to the levels of aggregation/nesting in their corresponding SQL. Sample queries with different complexity are shown in Figure 10. In the query set, the number of easy/medium/hard queries are 63/68/65, respectively.

A central innovation in NaLIR is the user interaction as part of query interpretation. To understand the benefit of such interaction, we also experimented with a version of NaLIR in which the interactive communicator was disabled, and the system always chose the default (most likely) option.

Participants. 14 participants were recruited with flyers posted on a university campus. A questionnaire indicated that all participants were familiar with keyword search interfaces (e.g. Google) and faceted search interfaces (e.g. Amazon), but had little knowledge of formal query languages (e.g. SQL). Furthermore, they were fluent in both English and Chinese.

Procedures. We evenly divided the query set into 28 task groups, in which the easy/medium/hard tasks were evenly divided into each task group. This experiment was a within-subject design. Each participant randomly took three groups of tasks and completed three experimental blocks. In the first (resp. second) experimental block, each participant used our system without (with) the Interactive Communicator to accomplish the tasks in her first (second) task group. Then in the third experimental block, each participant used the MAS interface to do her third task group. For each task group, the participants started with sample query tasks, in order to get familiar with each interface.

For our system, it is hard to convey the query task to the participants since any English description would cause bias in the task. To overcome this, we described each query task in Chinese and asked users to compose English query sentences. Since English and Chinese are in entirely different language families, we believe this kind of design can minimize such bias.

7.2 Results and Analysis

Effectiveness. Figure 11 compares the effectiveness of our system (with or without the interactive communicator) with the MAS website. As we can see, when the interactive communicator was disabled, the effectiveness of our system decreased significantly when the query tasks became more complex. Out of the 32 failures, the participants only detected 7 of them. Actually, most of undetected wrong answers were aggregated results, which were impossible to verify without further explanation. In other undetected failures, the participants accepted wrong answers mainly because they were not familiar with what they were querying. In the 7 detected failures, although the participants were aware of the failure, they were not able to correctly reformulate the queries in the time constraint. (In 5 of the detected failures, the participants detected the failure only

	with Interaction	without Interaction	MAS
Simple:	34/34	26/32	20/33
Medium:	34/34	23/34	18/32
Hard:	20/30	15/32	18/33

Figure 11: Effectiveness.

	Mapper	Reformulation	Insertion	Translation
w/o Interaction	15	19	0	0
with Interaction	0	10	0	0

Figure 12: Failures in each Component.

because the query results were empty sets). The situation got much better when the interactive communicator was involved. The users were able to handle 88 out of the 98 query tasks. For the 10 failed tasks, they only accepted 4 wrong answers, which was caused by the ambiguous (natural language) explanations generated from our system. In contrast, the participants were only able to accomplish 56 out of the 98 tasks using the MAS website, although all the correct answers could be found. In the failure cases, the participants were simply not able to find the right webpages, which often required several clicks from the initial search results.

Figure 12 shows the statistics of the specific components that cause the failures. We can see that our system could always correctly detect and insert the implicit parse tree nodes, even without interactive communications with the user. Also, when the query tree was correctly generated, our system translated it to the correct SQL statement. When the interactive communicator was enabled, the accuracy in the parse tree node mapper improved significantly, which means for each the ambiguous parse tree node, the parse tree node mapper could at least generate one correct mapping in the top 5 candidate mappings, and most importantly, the participants were able to recognize the correct mapping from others. The accuracy in parse tree structure reformulation was also improved when the participants were free to choose from the top 5 candidate valid parse trees. However, when the queries were complex, the number of possible valid parse trees was huge. As a result, the top 5 guessed interpretations could not always include the correct one.

Usability. The average time needed for the successfully accomplished query tasks is shown in Figure 13. When the interactive communicator was disabled, the only thing a participant could do was to read the query task description, understand the query task, translate the query task from Chinese to English and submit the query. So most of the query tasks were done in 50 seconds. When the interactive communicator was enabled, the participants were able to read the explanations, choose interpretations, reformulate the query according to the warnings, and decide to whether to accept the query results.

It is worth noting that, using our system (with interactive communicator), there was no instance where the participant became frustrated with the natural language interface and abandoned his/her query task within the time constraint. However, in 9 of the query tasks, participants decided to stop the experiment due to frustration with the MAS website. According to the questionnaire results, the users felt that MAS website was good for browsing data but not well de-

	with Interaction	without Interaction	MAS
Simple:	48	34	49
Medium:	70	42	67
Hard:	103	51	74

Figure 13: Average Time Cost (s).

signed for conducting specific query tasks. They felt NaLIR can handle simple/medium query tasks very well but they encountered difficulties for some of the hard queries. In contrast, the MAS website was not sensitive to the complexity of query tasks. Generally, they welcomed the idea of an interactive natural language query interface, and found our system easy to use. The average level of satisfaction with our system was 5, 5 and 3.8 for easy, medium, and hard query tasks, respectively, on a scale of 1 to 5, where 5 denotes extremely easy to use.

8. RELATED WORK

The problem of constructing Natural Language Interfaces to DataBases (NLIDB) has been studied for several decades, starting from early systems based on manually crafted grammars [1]. While quite successful in their specific scenario, the manually crafted grammars are hard to scale up, both to other domains and to new natural language expressions.

Later, researchers begin to build NLIDBs that can learn semantic grammars from training examples, in which the grammars can be improved incrementally by adding new examples [3, 4, 14, 15, 18]. A major obstacle is that these methods depend crucially on having examples of natural language queries paired with meaning representations, which requires substantial human effort to obtain [10].

As pointed out in [13], the major usability problem in NLIDBs is its limited reliability. Natural language sentences do not have formally defined semantics. The goal of NLIDBs is to infer the user's query intent, which cannot guarantee accuracy due to ambiguities. To deal with the reliability issue, PRECISE [12, 13] defines a subset of natural language queries as semantically tractable queries and precisely translates these queries into corresponding SQL queries. However, natural language queries that are not semantically tractable will be rejected by PRECISE.

The idea of interactive NLIDB was discussed in previous literature [1, 6, 8]. Early interactive NLIDBs [1, 6] mainly focus on generating cooperative responses from query results (over-answering). NaLIX [8] takes a step further, generating suggestions for the user to reformulate her query when it is beyond the semantic coverage. This strategy greatly reduces the user's burden in query reformulation. This paper is a short version of NaLIR [7], to which given a natural language query, we explain to the user how we process her query and interactively resolve ambiguities with the user. As a result, under the supervision of the user, our system could confidently handle rather complex queries.

9. CONCLUSION AND FUTURE WORK

We have described an interactive natural language query interface for relational databases. Given a natural language query, our system first translates it to a SQL statement and then evaluates it against an RDBMS. To achieve high reliability, our system explains to the user how her query is actually processed. When ambiguities exist, for each ambiguity, our system generates multiple likely interpretations for the user to choose from, which resolves ambiguities interactively with the user. The query mechanism described in this paper has been implemented, and actual user experience gathered. Using our system, even naive users are able to accomplish logically complex query tasks, in which the target SQL statements include comparison predicates, conjunctions, quantifications, multi-level aggregations, nestings, and various types of joins, among other things.

10. REFERENCES

[1] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch. Natural language interfaces to databases - an introduction. *Natural Language Engineering*, 1(1):29–81, 1995.

[2] M.-C. de Marneffe, B. MacCartney, and C. D. Manning. Generating typed dependency parses from phrase structure parses. In *LREC*, pages 449–454, 2006.

[3] R. Ge and R. Mooney. A statistical semantic parser that integrates syntax and semantics. In *CoNLL*, pages 9–16, 2005.

[4] R. J. Kate and R. J. Mooney. Using string-kernels for learning semantic parsers. In *ACL*, 2006.

[5] A. Kokkalis, P. Vagenas, A. Zervakis, A. Simitsis, G. Koutrika, and Y. E. Ioannidis. Logos: a system for translating queries into narratives. In *SIGMOD Conference*, pages 673–676, 2012.

[6] D. Küpper, M. Strobel, and D. Rösner. Nauda - a cooperative, natural language interface to relational databases. In *SIGMOD Conference*, pages 529–533, 1993.

[7] F. Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *PVLDB*, 8(1):73–84, 2014.

[8] Y. Li, H. Yang, and H. V. Jagadish. Nalix: an interactive natural language interface for querying xml. In *SIGMOD Conference*, pages 900–902, 2005.

[9] Y. Li, H. Yang, and H. V. Jagadish. Nalix: A generic natural language search environment for XML data. *ACM Trans. Database Syst.*, 32(4), 2007.

[10] P. Liang, M. I. Jordan, and D. Klein. Learning dependency-based compositional semantics. In *ACL*, 2011.

[11] M. Minock. A STEP towards realizing codd's vision of rendezvous with the casual user. In *PVLDB*, pages 1358–1361, 2007.

[12] A.-M. Popescu, A. Armanasu, O. Etzioni, D. Ko, and A. Yates. Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability. In *COLING*, 2004.

[13] A.-M. Popescu, O. Etzioni, and H. A. Kautz. Towards a theory of natural language interfaces to databases. In *IUI*, pages 149–157, 2003.

[14] L. R. Tang and R. J. Mooney. Using multiple clause constructors in inductive logic programming for semantic parsing. In *EMCL*, pages 466–477, 2001.

[15] Y. W. Wong and R. J. Mooney. Learning synchronous grammars for semantic parsing with lambda calculus. In *ACL*, 2007.

[16] Z. Wu and M. S. Palmer. Verb semantics and lexical selection. In *ACL*, pages 133–138, 1994.

[17] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *ACM Trans. Database Syst.*, 36(3):15, 2011.

[18] J. M. Zelle and R. J. Mooney. Learning to parse database queries using inductive logic programming. In *AAAI*, 1996.

[19] Y. Zeng, Z. Bao, T. W. Ling, H. V. Jagadish, and G. Li. Breaking out of the mismatch trap. In *ICDE*, pages 940–951, 2014.

Technical Perspective: Attacking the Problem of Consistent Query Answering

Wang-Chiew Tan
University of California, Santa Cruz
tan@cs.ucsc.edu

Inconsistent data refers to data that do not adhere to one or more constraints. The term *constraints* refers to conditions that need to be imposed on the data. Constraints often arise from organizational requirements or business logic, such as the requirement that every employee in the database must be uniquely identified by the employee id, or every employee must work on some project, or the expenses cannot exceed the credit limit, or even a desired designated format for storing phone numbers. The need to manage *inconsistent data* arises in many settings. Quite typically, when one integrates data from different sources, the integrated data can be inconsistent data even when the data sources may be individually consistent. Another scenario where inconsistency in data can arise is when data and/or schema evolves, for example, through the addition or removal of data, changes in schema, or knowledge of new constraints.

There are generally two themes of research centered around the management of inconsistent data. One theme of research aims to make the data *clean* before the data is allowed to be used or queried. In other words, data is modified so that it becomes consistent prior to the execution of queries over the data. The data cleaning process is typically algorithmic, and sometimes heuristic, so that data can be manipulated, somehow, into a final consistent state. While this approach produces one final clean dataset that one can work with, it is generally difficult to understand how the consistent state of the data was arrived at. In contrast to data cleaning, the other theme of research adopts a "lazy" perspective towards the management of inconsistent data. In this line of research, the inconsistent data is left unmodified and work to determine what are the right answers is done only when queries are to be executed over the inconsistent data. In other words, the management of inconsistencies only occurs at query time. To compute the answer of the query, one considers all possible ways to *repair* the inconsistent database and the intersection of the results of the query on each repair forms the answer to the query. Since these are answers that appear in the result of the query applied to every repair, they are called the *consistent answer* to the query [1]. A notion of repair that has been widely considered in the past is that of a consistent database instance that differs from the original inconsistent database in a "minimal" way. If the only constraints were key constraints, this amounts to minimally removing tuples from the inconsistent database so that the key constraints will no longer be violated.

The problem of computing the consistent query answer of a query over an inconsistent database (CQA for short) has received significant attention in the past several years. This problem is known to be coNP-complete in general for the class of conjunctive queries under primary key constraints [2]. However, for a number of years, it was unclear whether one can provide exact conditions to determine, even for the special class of self-join-free conjunctive queries, the exact complexity of the query. There was a flurry of research activities on this problem in the past decade or so and finally, the problem is resolved in [3], where the authors showed a trichotomy result; for any self-join-free boolean conjunctive query, it can be decided with an effective procedure whether or not the CQA problem is in FO, P, or coNP-complete.

There are immediate practical implications of this result. Existing implementations of systems for CQA tend to adopt an (overly) expressive and hence computationally expensive engine for computing the answer, or identify special classes of queries for which CQA can be computed by means of SQL queries. The latter allows one to leverage highly optimized relational database management systems and tends to run fast. With this result at hand, it is now possible to determine and delegate the computation of CQA to the right engine; pushing computations to the RDBMS whenever possible, and relying on more algorithmic or expressive engines otherwise.

Now, if you are curious about how they "attacked" the CQA problem, read on.

References

[1] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the 18th ACM Symposium on Principles of Database Systems (PODS)*, pages 68–79, 1999.

[2] J. Chomicki and J. Marcinkowski. Minimal-change integrity maintenance using tuple deletions. In *Information and Computation*, 197(1-2):90–121, 2005.

[3] P. Koutris and J. Wijsen. The data complexity of consistent query answering for self-join-free conjunctive queries under primary key constraints. In *Proceedings of the 34th ACM Symposium on Principles of Database Systems (PODS)*, pages 17–29, 2015.

Consistent Query Answering for Primary Keys

Paraschos Koutris
University of Wisconsin-Madison
Madison, Wisconsin, USA
paris@cs.wisc.edu

Jef Wijsen
University of Mons
Mons, Belgium
jef.wijsen@umons.ac.be

ABSTRACT

We study the complexity of consistent query answering with respect to primary key violations, for self-join-free conjunctive queries. A repair of a possibly inconsistent database is obtained by selecting a maximal number of tuples without selecting two distinct tuples with the same primary key value. For any Boolean query q, CERTAINTY(q) is the problem that takes a database as input, and asks whether q is true in every repair of the database. The complexity of this problem has been extensively studied for q ranging over the class of self-join-free Boolean conjunctive queries. A research challenge is to determine, given q, whether CERTAINTY(q) belongs to complexity classes **FO**, **P**, or **coNP**-complete. We show that for any self-join-free Boolean conjunctive query q, it can be decided whether or not CERTAINTY(q) is in **FO**. Further, CERTAINTY(q) is either in **P** or **coNP**-complete, and the complexity dichotomy is effective. This settles a research question of practical relevance that has been open for ten years.

1. INTRODUCTION

A database is inconsistent if it violates one or more integrity constraints that the data is required to obey. Inconsistency in the data can arise in various settings, for example, when we integrate data from heterogeneous sources, or when the original data sources are imprecise or noisy.

The standard method of dealing with inconsistency is *data cleaning*, where the database is first *repaired* to satisfy the integrity constraints, and then we use the clean version to answer queries. However, since we can typically repair a database in many different ways, it is often the case that we have to make arbitrary choices about which data to keep, which means information may be lost. An alternative to data cleaning is *consistent query answering* (CQA), which was first introduced in [1]. In this framework, we answer queries by considering all possible repairs of the inconsistent database, and returning the intersection of the answers on all repairs, which is called the *consistent* (or *certain*) answer.

In this article, we focus on integrity constraints that are *primary key constraints*. Consider the database in Fig. 1, which includes the table E that stores employee information and the table D that stores department information. The primary keys of E and D are EID and DNAME, respectively. There are two foreign keys: every DNAME-value in E must occur in the DNAME-column of D, and every MGR-value in D must occur in the EID-column of E. The CITY column in table E stores the city of birth for each employee.

E

EID	ENAME	CITY	DNAME
E1	Smith	London	Training
E2	Jones	Paris	Training
E3	Blake	Paris	HR
E3	Blake	London	HR
E4	Clark	London	HR
E5	Adams	Athens	HR

D

DNAME	BUDGET	CITY	MGR
Training	120	London	E3
HR	300	Paris	E3
HR	310	Paris	E5

Figure 1: Example of inconsistent database that violates the primary key constraints.

Both tables in the database contain primary key violations: for example, the employee with EID E3 has two entries in table E. Two or more tuples that agree on their primary key represent mutually exclusive possibilities. Such tuples are said to form a *block*; in Fig. 1, blocks are separated by dashed lines for readability. The reader should notice that even though we do not know the exact city where Blake was born, we still know that it is either Paris or London; contrast this with the case where we would represent the city with a single uninformative NULL.

Blocks with two or more tuples model uncertainty: exactly one of the tuples is true, but we do not know which one is true. Therefore, we use the term *uncertain database* to refer to databases that can contain primary key violations. A *repair* (or *possible world*) of an uncertain database is obtained by selecting exactly one tuple from each block.

In this article, we study how to answer conjunctive queries on uncertain databases. We allow joins, but we disallow that a table be joined with itself (called a *self-join*). It is natural to distinguish between possible and certain answers: the *possible* answer to a query consists of the tuples that are in the answer to the query on some repair, while the *certain* (or consistent) answer consists of the tuples that are in the answer to the query on every repair. Consider, for example, the query *"Get the names of employees who were born in London,"* which is encoded in SQL as follows.

The original version of this article was published in PODS 2015.

```
SELECT E1.ENAME
FROM   E AS E1
WHERE E1.CITY='London';
```

The possible answer to this query consists of Smith, Clark, and Blake. The certain answer consists of Smith and Clark. Blake does not belong to the certain answer, since the database leaves open the possibility that Blake was born in Paris. For conjunctive queries without self-join, it is easy to see that the possible answer is obtained by executing the query on the uncertain database. Computing certain answers is a more difficult task. Interestingly, the certain answer to the above query is computed by the following SQL query.

```
SELECT E1.ENAME
FROM   E AS E1
WHERE E1.CITY='London'
AND    NOT EXISTS ( SELECT *
                    FROM   E AS E2
                    WHERE E2.EID=E1.EID
                    AND    E2.CITY≠'London' );
```

The NOT EXISTS subquery checks the non existence of a city of birth other than London, and thereby excludes Blake from the answer. Unfortunately, it is not always possible to obtain certain answers directly in SQL. We show in this paper that for all conjunctive queries without self-joins, CQA with respect to primary keys can be classified into one of three exclusive classes of increasing complexity:

1) For some queries, the certain answer can be expressed in relational calculus (or first-order logic), and hence can be written in SQL. One such query was shown before; as we will see in Section 3, another example is the query *"Get names for departments which are self-managed (i.e., are managed by one of their own employees)."*

```
SELECT D.DNAME FROM E, D
WHERE E.EID=D.MGR
AND    E.DNAME=D.DNAME;
```

The certain answer consists of HR. Notice that although there are two possibilities for the manager of HR, we know for certain that HR is self-managed.

2) For some queries, the certain answer can be computed in polynomial time (with respect to the database size), but cannot be expressed in relational calculus. We will see in Section 3 that an example of such a query is *"Get names for employees who manage the department for which they work."*

```
SELECT E.ENAME FROM E, D
WHERE E.EID=D.MGR
AND    E.DNAME=D.DNAME;
```

The certain answer is empty.

3) For some queries, the certain answer cannot be obtained in polynomial time (unless $\mathbf{P} = \mathbf{NP}$), since it is **coNP**-hard to compute the certain answer. We will see in Section 3 that an example of such query is *"Get names for employees who work in the city of their birth."*

```
SELECT E.ENAME FROM E, D
WHERE E.CITY=D.CITY
AND    E.DNAME=D.DNAME;
```

The certain answer consists only of Smith.

Given a conjunctive query q without self-join, it can be decided which of the three classes it belongs to. Moreover, if the query falls into the first or second class, then we know how to construct an SQL query or a polynomial-time algorithm for computing its certain answer.

Our result provides a complexity classification for CQA with respect to primary keys, when the query ranges over the set of self-join-free conjunctive queries. This complexity classification task has been an open problem since 2005 [8], and culminates a long line of research [8, 10, 12, 20, 22].

In Section 6, we explain why our theoretical results are of interest to practitioners and system builders. In short, CQA has often been implemented by means of expressive and computationally expensive languages, like Disjunctive Logic Programming. The efficiency of these algorithms is likely to be far from optimal in the case that consistent answers can be computed in polynomial time or, even faster, by executing a single SQL query. The practical relevance of our results is that they tell us when computationally expensive formalisms can be avoided. Moreover, it turns out that for many natural queries, consistent answers can be obtained with low complexity.

Organization. Section 2 introduces the data and query model. In Section 3, we define a syntactic tool, called *attack graph*, and use it to state Theorem 1, which is the main result of the article. Sections 4 and 5 zoom in on two important complexity boundaries: Section 4 elaborates on first-order expressibility, and Section 5 on polynomial-time tractability. Section 6 discusses both the theoretical and systems-oriented related work.

2. PRELIMINARIES

We assume disjoint sets of *variables* and *constants*. If $\vec{x}$ is a sequence containing variables and constants, then $\mathsf{vars}(\vec{x})$ denotes the set of variables that occur in $\vec{x}$. A *valuation* over a set U of variables is a total mapping θ from U to the set of constants. At several places, it is implicitly understood that such a valuation θ is extended to be the identity on constants and on variables not in U.

Atoms and key-equal facts. Each *relation name* R of arity n, $n \geq 1$, has a unique *primary key* which is a set $\{1, 2, \ldots, k\}$ where $1 \leq k \leq n$. We say that R has *signature* $[n, k]$ if R has arity n and primary key $\{1, 2, \ldots, k\}$. We say that R is *simple-key* if $k = 1$. Elements of the primary key are called *primary-key positions*. For all positive integers n, k such that $1 \leq k \leq n$, we assume denumerably many relation names with signature $[n, k]$. For example, the relation name E from Fig. 1 has signature $[4, 1]$ and is simple-key.

If R is a relation name with signature $[n, k]$, then the expression $R(s_1, \ldots, s_n)$ is called an *R-atom* (or simply atom), where each s_i is either a constant or a variable ($1 \leq i \leq n$). Such an atom is commonly written as $R(\underline{\vec{x}}, \vec{y})$ where the primary key value $\vec{x} = s_1, \ldots, s_k$ is underlined and $\vec{y} = s_{k+1}, \ldots, s_n$. An *R-fact* (or simply fact) is an R-atom in which no variable occurs. Two facts $R_1(\underline{\vec{a}_1}, \vec{b}_1), R_2(\underline{\vec{a}_2}, \vec{b}_2)$ are *key-equal*, denoted $R_1(\underline{\vec{a}_1}, \vec{b}_1) \sim R_2(\underline{\vec{a}_2}, \vec{b}_2)$, if $R_1 = R_2$ and $\vec{a}_1 = \vec{a}_2$. An R-atom or an R-fact is *simple-key* if R is simple-key.

We will use letters F, G, H for atoms. For an atom $F = R(\underline{\vec{x}}, \vec{y})$, we denote by $\mathsf{key}(F)$ the set of variables that occur in $\vec{x}$, and by $\mathsf{vars}(F)$ the set of variables that occur in F, that is, $\mathsf{key}(F) = \mathsf{vars}(\vec{x})$ and $\mathsf{vars}(F) = \mathsf{vars}(\vec{x}) \cup \mathsf{vars}(\vec{y})$.

Uncertain databases, blocks, and repairs. An *uncertain database* is a finite set **db** of facts using only the relation names of a fixed database schema. We refer to databases as "uncertain databases" to emphasize that such databases can violate primary key constraints.

A *block* of **db** is a maximal set of key-equal facts of **db**. The term R-block refers to a block of R-facts, i.e., facts with relation name R. An uncertain database **db** is *consistent* if no two distinct facts are key-equal (i.e., if every block of **db** is a singleton). A *repair* of **db** is a maximal (with respect to set inclusion) consistent subset of **db**.

Boolean conjunctive queries. A *Boolean query* is a mapping q that associates true or false to each uncertain database, such that q is closed under isomorphism [15]. We write $\mathbf{db} \models q$ to denote that q associates true to **db**, in which case **db** is said to *satisfy* q. A *Boolean first-order query* is a Boolean query that can be defined in first-order logic (with equality and constants, but without other built-in predicates). A *Boolean conjunctive query* is a finite set $q = \{R_1(\underline{\vec{x}_1}, \vec{y}_1), \ldots, R_n(\underline{\vec{x}_n}, \vec{y}_n)\}$ of atoms. We denote by $\mathsf{vars}(q)$ the set of variables that occur in q. The set q represents the first-order sentence

$$\exists u_1 \cdots \exists u_k \left(R_1(\underline{\vec{x}_1}, \vec{y}_1) \wedge \cdots \wedge R_n(\underline{\vec{x}_n}, \vec{y}_n) \right),$$

where $\{u_1, \ldots, u_k\} = \mathsf{vars}(q)$. This query q is satisfied by uncertain database **db** if there exists a valuation θ over $\mathsf{vars}(q)$ such that for each $i \in \{1, \ldots, n\}$, $R_i(\underline{\vec{a}}, \vec{b}) \in \mathbf{db}$ with $\vec{a} = \theta(\vec{x}_i)$ and $\vec{b} = \theta(\vec{y}_i)$.

We say that a Boolean conjunctive query q has a *self-join* if some relation name occurs more than once in q. If q has no self-join, then it is called *self-join-free*. If q is a Boolean conjunctive query, $\vec{x} = \langle x_1, \ldots, x_\ell \rangle$ is a sequence of distinct variables that occur in q, and $\vec{a} = \langle a_1, \ldots, a_\ell \rangle$ is a sequence of constants, then $q_{[\vec{x} \mapsto \vec{a}]}$ denotes the query obtained from q by replacing all occurrences of x_i with a_i, for all $1 \leq i \leq \ell$.

We write BCQ for the class of Boolean conjunctive queries, and sjfBCQ for the class of self-join-free Boolean conjunctive queries. If q is a sjfBCQ query with an R-atom, then, by an abuse of notation, we write R to mean the R-atom of q.

Consistent query answering. For every Boolean query q, the decision problem $\mathsf{CERTAINTY}(q)$ takes as input an uncertain database **db**, and asks whether q is satisfied by every repair of **db**.

PROBLEM:	$\mathsf{CERTAINTY}(q)$
INPUT:	uncertain database **db**
QUESTION:	Does every repair of **db** satisfy q?

Notice that the Boolean query q is not part of the input. Hence, every Boolean query q gives rise to a new problem. Since the input to $\mathsf{CERTAINTY}(q)$ is an uncertain database, we consider the *data complexity* of the problem. The extension to non-Boolean queries will be discussed in Section 3.

3. ATTACK GRAPHS

The construct of *attack graph* is the main tool for solving the complexity classification task of $\mathsf{CERTAINTY}(q)$. Attack graphs were first introduced in [20] for studying first-order expressibility of $\mathsf{CERTAINTY}(q)$ for acyclic (in the sense of [2]) self-join-free conjunctive queries q.

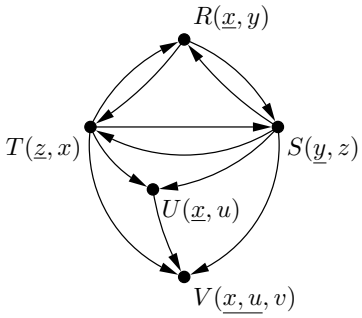


Figure 2: Attack graph of the query in Example 1.

Let $q \in \mathsf{sjfBCQ}$. We define $\mathcal{K}(q)$ as the following set of functional dependencies:

$$\mathcal{K}(q) := \{\mathsf{key}(F) \to \mathsf{vars}(F) \mid F \in q\}.$$

For every atom $F \in q$, we define $F^{+,q}$ as the following set of variables:

$$F^{+,q} := \{x \in \mathsf{vars}(q) \mid \mathcal{K}(q \setminus \{F\}) \models \mathsf{key}(F) \to x\}.$$

The *attack graph of* q is a directed graph whose vertices are the atoms of q. There is a directed edge from F to G ($F \neq G$) if there exists a sequence

$$F_0 \overset{z_1}{\frown} F_1 \overset{z_2}{\frown} F_2 \ldots \overset{z_n}{\frown} F_n \tag{1}$$

where

- $F_0, \ldots, F_n$ are atoms of q;
- $F_0 = F$ and $F_n = G$; and
- for all $i \in \{1, \ldots, n\}$, $z_i \in \mathsf{vars}(F_{i-1}) \cap \mathsf{vars}(F_i)$ and $z_i \notin F^{+,q}$.

A directed edge from F to G in the attack graph of q is also called an *attack from* F *to* G, denoted by $F \overset{q}{\rightsquigarrow} G$. The sequence (1) is called a *witness* for the attack $F \overset{q}{\rightsquigarrow} G$.

If $F \overset{q}{\rightsquigarrow} G$, then we also say that F *attacks* G (or that G is attacked by F). An attack from F to G is called *weak* if $\mathcal{K}(q) \models \mathsf{key}(F) \to \mathsf{key}(G)$; otherwise it is *strong*. A directed cycle in the attack graph of q is called *weak* if all attacks in the cycle are weak; otherwise the cycle is called *strong*.

Example 1. Let $q = \{R(\underline{x}, y),\ S(\underline{y}, z),\ T(\underline{z}, x),\ U(\underline{x}, u),\ V(\underline{x, u}, v)\}$. We have $R^{+,q} = \{x, u, v\}$. A witness for $R \overset{q}{\rightsquigarrow} T$ is $R \overset{y}{\frown} S \overset{z}{\frown} T$. The complete attack graph is shown in Fig. 2. All attacks are weak.

The above definition of an attack graph is purely syntactic. Semantically, an attack from an R-atom to an S-atom means that there exists an uncertain database **db** such that every repair of **db** satisfies q, and such that two key-equal R-facts join exclusively with two S-facts that are not key-equal. For the query of Example 1, such a database could be $\mathbf{db} = \{R(\underline{1}, a),\ R(\underline{1}, b),\ S(\underline{a}, \alpha),\ S(\underline{b}, \beta), \ldots\}$, in which the two R-facts are key-equal, $R(\underline{1}, a)$ joins exclusively with $S(\underline{a}, \alpha)$, and $R(\underline{1}, b)$ joins exclusively with $S(\underline{b}, \beta)$, and the two S-facts are not key-equal. Therefore, the attack graph of Fig. 2 contains a directed edge from the R-atom to the S-atom.

Equipped with the notion of attack graph, we can now present our result for the complexity classification task of

CERTAINTY(q). **FO** is the descriptive complexity class of decision problems expressible in first-order logic.

THEOREM 1. *For every $q \in$ sjfBCQ,*

1. *if the attack graph of q is acyclic, then* CERTAINTY(q) *is in* **FO**;
2. *if the attack graph of q is cyclic but contains no strong cycle, then* CERTAINTY(q) *is in* **P** *and is* **L**-*hard; and*
3. *if the attack graph of q contains a strong cycle, then* CERTAINTY(q) *is* **coNP**-*complete.*

Furthermore, it can be decided in polynomial time in the size of q which of the above three cases applies.

Before giving some examples, we explain how to deal with non-Boolean queries, which are common in practice. If $q \in$ sjfBCQ and $\vec{x}$ is a sequence of distinct variables of vars(q), then the *rule* $\vec{x} \leftarrow q$ denotes the conjunctive query that is obtained from q by letting the variables of $\vec{x}$ be free variables. Given an uncertain database **db**, the *certain answer* to this rule is the set of tuples $\vec{a}$ (of the same length as $\vec{x}$), such that $q_{[\vec{x} \mapsto \vec{a}]}$ is satisfied by every repair of **db**.

The attack graph of $\vec{x} \leftarrow q$ is the attack graph of $q_{[\vec{x} \mapsto \vec{c}]}$ for some sequence $\vec{c}$ of constants (of the same length as $\vec{x}$). The results that follow are independent of the choice of $\vec{c}$. This is tantamount to saying that free variables can be treated like constants.

It is easy to show that Theorem 1 extends to rules $\vec{x} \leftarrow q$, where q is always assumed to be self-join-free:

1. if the attack graph is acyclic, then there exists a first-order query that computes the certain answer;
2. if the attack graph is cyclic but contains no strong cycle, then there exists a polynomial-time algorithm (but no first-order query) that computes the certain answer; and
3. if the attack graph constains a strong cycle, then, unless **P** = **NP**, there exists no polynomial-time algorithm that computes the certain answer.

The query *"Get names for departments which are self-managed (i.e., are managed by one of their own employees),"* introduced in Section 1, is expressed by the following rule:

$$q_1 : d \leftarrow \mathsf{E}(\underline{m}, n, c_1, d), \mathsf{D}(\underline{d}, b, c_2, m).$$

Treating the free variable d as a constant, we obtain $\mathsf{E}(\underline{m}, n, c_1, d)^{+,q_1} = \{m, b, c_2\}$ and $\mathsf{D}(\underline{d}, b, c_2, m)^{+,q_1} = \{\}$. The only attack is from the D-atom to the E-atom. Since the attack graph is acyclic, the certain answer to this query can be computed in SQL.

Next, consider the query *"Get names for employees who manage the department for which they work."*

$$q_2 : n \leftarrow \mathsf{E}(\underline{m}, n, c_1, d), \mathsf{D}(\underline{d}, b, c_2, m).$$

We have $\mathsf{E}(\underline{m}, n, c_1, d)^{+,q_2} = \{m\}$ and $\mathsf{D}(\underline{d}, b, c_2, m)^{+,q_2} = \{d\}$. The E-atom attacks the D-atom because of the shared variable d, and the D-atom attacks the E-atom because of the shared variable m. The attack graph is cyclic, but contains no strong cycle. Therefore, the certain answer to this query can be computed in polynomial time, but not in first-order logic or SQL.

Finally, consider the query *"Get names for employees who work in the city of their birth."*

$$q_3 : n \leftarrow \mathsf{E}(\underline{e}, n, c, d), \mathsf{D}(\underline{d}, b, c, m).$$

We have $\mathsf{E}(\underline{e}, n, c, d)^{+,q_3} = \{e\}$ and $\mathsf{D}(\underline{d}, b, c, m)^{+,q_3} = \{d\}$. Both atoms attack each other because of the shared variable c. Moreover, the attack from the D-atom to the E-atom is strong. Since the attack graph contains a strong cycle, there exists no polynomial-time algorithm for computing the certain answer to this query (unless **P** = **NP**).

Before providing more insights into the proof of Theorem 1, one more definition is needed. So far we have defined an attack from an atom to another atom. The following definition introduces attacks from an atom to a variable.

Definition 1. Let $q \in$ sjfBCQ. Let R be a relation name with signature $[1, 1]$ such that R does not occur in q. For $F \in q$ and $z \in \mathsf{vars}(q)$, we say that F *attacks* z, denoted $F \overset{q}{\rightsquigarrow} z$, if $F \overset{q'}{\rightsquigarrow} R(\underline{z})$ where $q' = q \cup \{R(\underline{z})\}$.

Example 2. Clearly, if $F_0 \overset{z_1}{\frown} F_1 \ldots \overset{z_n}{\frown} F_n$ is a witness for $F_0 \overset{q}{\rightsquigarrow} F_n$, then $F_0 \overset{q}{\rightsquigarrow} z_i$ for every $i \in \{1, \ldots, n\}$. Notice also that if $q = \{R(\underline{x}, y)\}$, then the attack graph of q contains no edge, yet $R \overset{q}{\rightsquigarrow} y$.

4. FIRST-ORDER EXPRESSIBILITY

In this section, we elaborate on the first item in the statement of Theorem 1, as well as the **L**-hard lower complexity bound stated in the second item. Taken together, this leads to the following characterization of first-order expressibility of CERTAINTY(q).

THEOREM 2. *For every $q \in$ sjfBCQ,* CERTAINTY(q) *is in* **FO** *if and only if the attack graph of q is acyclic.*

To prove the only-if direction, the first step is to show that for $q_0 = \{R_0(\underline{x}, y), S_0(\underline{y}, x)\}$, CERTAINTY($q_0$) is **L**-hard. The query q_0 is in some sense the "simplest" query for which consistent query answering is **L**-hard (an earlier result in [21] showed the weaker result that CERTAINTY(q_0) is not in **FO**). Then, it is shown that for every $q \in$ sjfBCQ, if the attack graph of q is cyclic, there exists a first-order reduction from CERTAINTY(q_0) to CERTAINTY(q), which implies that CERTAINTY(q) is **L**-hard (and thus not in **FO**).

For the if-direction, assume that $q \in$ sjfBCQ such that the attack graph of q is acyclic. Assume $q = \{R_1(\underline{\vec{x}_1}, \vec{y}_1), \ldots, R_n(\underline{\vec{x}_n}, \vec{y}_n)\}$, where the atoms are listed in a topological ordering of the attack graph. Then, it can be shown that CERTAINTY(q) is solved by Algorithm 1, in which $\sim$ denotes "is key-equal to."

Input : Uncertain database **db**
Output: Does every repair of **db** satisfy q?

if $\exists s_1 \in R_1 \quad \forall r_1 \in R_1$ *s.t.* $r_1 \sim s_1$*:*
  $\exists s_2 \in R_2 \quad \forall r_2 \in R_2$ *s.t.* $r_2 \sim s_2$*:*
    $\ldots$
      $\exists s_n \in R_n \quad \forall r_n \in R_n$ *s.t.* $r_n \sim s_n$*:*
        the tuples $r_1, r_2, \ldots, r_n$ together satisfy q
then
  return "yes"
else
  return "no"

Algorithm 1: Algorithm for CERTAINTY(q) for an sjfBCQ query q whose (acyclic) attack graph has topological ordering $R_1, R_2, \ldots, R_n$.

In the **if**-condition of Algorithm 1, every existential quantifier $\exists s_i$ selects an R_i-block, and the subsequent universal quantifier $\forall r_i$ lets r_i range over all facts of that block. At the end, all facts r_i together must satisfy the query. It is easy to encode Algorithm 1 in first-order logic (or in SQL). This implies that if the attack graph of $q \in \mathsf{sjfBCQ}$ is acyclic, we can effectively construct a first-order definition of $\mathsf{CERTAINTY}(q)$. Such a first-order definition is commonly called a *consistent first-order rewriting* for q.

5. POLYNOMIAL-TIME TRACTABILITY

In this section, we outline a polynomial-time algorithm for $\mathsf{CERTAINTY}(q)$ in case that the attack graph of q contains no strong cycles (stated in the second item of Theorem 1). We refer the reader to [13, 14] for the proof of the **coNP**-hard lower bound in case that the attack graph contains a strong cycle.

THEOREM 3. *For every $q \in \mathsf{sjfBCQ}$, if the attack graph of q contains no strong cycle, then $\mathsf{CERTAINTY}(q)$ is in* **P**.

Theorem 3 is proved roughly as follows by syntactic induction. Let $q \in \mathsf{sjfBCQ}$ such that the attack graph of q contains no strong cycle. If the attack graph of q contains an R_1-atom without incoming attacks, then this atom can be processed like the R_1-atom in the first line of Algorithm 1. The more difficult case is if all atoms have incoming attacks in the attack graph of q. In this case, $\mathsf{CERTAINTY}(q)$ can be reduced in polynomial time to some problem $\mathsf{CERTAINTY}(q')$ which is in polynomial time by induction hypothesis. The query q' is obtained from q by a technique called *"dissolution of Markov cycles."*

The proof of Theorem 3 is technically involved, and thus we will only sketch the main ideas. The first important idea is an extension of the data model that allows some syntactic simplifications, which we explain in Section 5.1. The central idea is the notion of *Markov cycle*, which we present in Section 5.2. The dissolution of Markov cycles is illustrated in Section 5.3.

5.1 Relations Known to Be Consistent

We extend the data model by distinguishing between two kinds of relation names: those that can be inconsistent, and those that cannot.

Every relation name has a unique and fixed *mode*, which is an element in $\{i, c\}$. It will come in handy to think of i and c as inconsistent and consistent respectively. We often write R^c to denote that R is a relation name with mode c. If $q \in \mathsf{sjfBCQ}$, then $[\![q]\!]$ denotes the subset of q containing each atom whose relation name has mode c. The *inconsistency count* of q, denoted $\mathsf{incnt}(q)$, is the number of relation names with mode i in q. Modes carry over to atoms and facts: the mode of an atom $R(\underline{\vec{x}}, \vec{y})$ or a fact $R(\underline{\vec{a}}, \vec{b})$ is the mode of R. The intended semantics is that if a relation name R has mode c, then the set of R-facts of an uncertain database will always be consistent.

The problem $\mathsf{CERTAINTY}(q)$ now takes as input an uncertain database **db** such that for every relation name R in q, if R has mode c, then the set of R-facts of **db** is consistent. The problem is, as before, to determine whether every repair of **db** satisfies q.

All constructs and results shown in previous sections assumed that all relation names had mode i. However, having relation names with mode c is convenient but not fundamental, since we can simulate relation names with mode c by using relation names with mode i. Indeed, consider any $q \in \mathsf{sjfBCQ}$ and let $R^c(\underline{\vec{x}}, \vec{y})$ be an atom in q. Construct a new query $q' = (q \setminus \{R^c(\underline{\vec{x}}, \vec{y})\}) \cup \{R_1(\underline{\vec{x}}, \vec{y}), R_2(\underline{\vec{x}}, \vec{y})\}$, where R_1, R_2 are two new relation names with mode i and the same signature as R. Then $\mathsf{CERTAINTY}(q)$ and $\mathsf{CERTAINTY}(q')$ are equivalent under first-order reductions.

If relation names with mode c are allowed for syntactic convenience, the definition of $F^{+,q}$ needs slight change:

$$F^{+,q} := \{x \in \mathsf{vars}(q) \mid \mathcal{K}((q \setminus F) \cup [\![q]\!]) \models \mathsf{key}(F) \to x\}.$$

Modulo this redefinition, the notion of attack graph remains unchanged.

Our polynomial-time algorithm relies on the notion of saturated query, defined next, which we admit to be technical and not intuitive. Lemma 1 states that $\mathsf{CERTAINTY}(q)$ can be polynomially reduced to $\mathsf{CERTAINTY}(q')$, where q' is saturated and syntactically simplified.

Definition 2. A query $q \in \mathsf{sjfBCQ}$ is said to be *saturated* if whenever $x, z \in \mathsf{vars}(q)$ such that $\mathcal{K}(q) \models x \to z$ and $\mathcal{K}([\![q]\!]) \not\models x \to z$, then there exists an atom $F \in q$ with $\mathcal{K}(q) \models x \to \mathsf{key}(F)$ such that $F \overset{q}{\rightsquigarrow} x$ or $F \overset{q}{\rightsquigarrow} z$.

LEMMA 1. *For each $q \in \mathsf{sjfBCQ}$, there exists a polynomial-time many-one reduction from the problem $\mathsf{CERTAINTY}(q)$ to $\mathsf{CERTAINTY}(q')$ for some $q' \in \mathsf{sjfBCQ}$ with the following properties:*

- $\mathsf{incnt}(q') \leq \mathsf{incnt}(q)$*;*
- *no atom in q' has two occurrences of the same variable;*
- *constants occur in q' exclusively at the primary-key position of simple-key atoms;*
- *every atom with mode i in q' is simple-key;*
- *q' is saturated; and*
- *if the the attack graph of q contains no strong cycle, then the attack graph of q' contains no strong cycle either.*

5.2 Dissolving Markov Cycles

Our polynomial-time algorithm relies on a new tool called *Markov graph*, which is defined next.

Definition 3. Let $q \in \mathsf{sjfBCQ}$ such that every atom with mode i in q is simple-key. For every $x \in \mathsf{vars}(q)$, we define

$$\mathsf{C}_q(x) := \{F \in q \mid F \text{ has mode } i \text{ and } \mathsf{key}(F) = \{x\}\}.$$

The *Markov graph* of q is a directed graph whose vertex set is $\mathsf{vars}(q)$. There is a directed edge from x to y, denoted $x \xrightarrow{\mathsf{M}} y$, if $x \neq y$ and $\mathcal{K}(\mathsf{C}_q(x) \cup [\![q]\!]) \models x \to y$.

The use of the term Markov refers to the intuition that along a path in the Markov graph, each variable functionally determines the next variable on the path, independently of preceding variables. A *Markov cycle* refers to a (directed) cycle in the Markov graph.

Definition 4. A Markov cycle $\mathcal{C}$ is said to be *premier* if there exists a variable $x \in \mathsf{vars}(q)$ such that

- $\{x\} = \mathsf{key}(F_0)$ for some atom F_0 with mode i that belongs to an initial strong component of the attack graph of q; and
- for some y in $\mathcal{C}$, $\mathcal{K}(q) \models y \to x$ and the Markov graph of q contains a directed path from x to y.

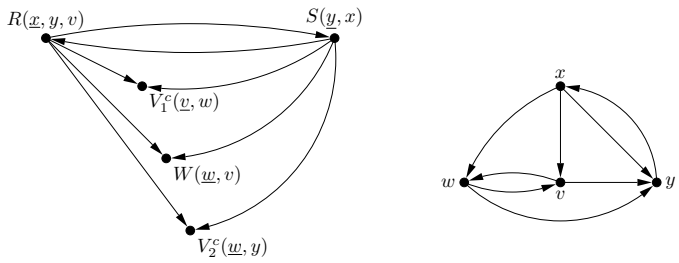


Figure 3: Attack graph (left) and Markov graph (right) of query $\{R(\underline{x}, y, v)$, $S(\underline{y}, x)$, $V_1^c(\underline{v}, w)$, $W(\underline{w}, v)$ $V_2^c(\underline{w}, y)\}$.

Example 3. Let $q = \{R(\underline{x}, y, v), S(\underline{y}, x), V_1^c(\underline{v}, w), W(\underline{w}, v)$ $V_2^c(\underline{w}, y)\}$. All atoms in q are simple-key. Then, $[\![q]\!] = \{V_1^c(\underline{v}, w), V_2^c(\underline{w}, y)\}$.

We have $\mathsf{C}_q(x) = \{R(\underline{x}, v, y)\}$. Since $\mathcal{K}(\mathsf{C}_q(x) \cup [\![q]\!]) \models x \to \{y, v, w\}$, the Markov graph of q contains directed edges from x to each of y, v, and w.

We have $\mathsf{C}_q(v) = \emptyset$. Since $\mathcal{K}(\mathsf{C}_q(v) \cup [\![q]\!]) \models v \to \{y, w\}$, the Markov graph of q contains directed edges from v to both y and w. The complete Markov graph of q is shown in Fig. 3 (right).

The attack graph of q is shown in Fig. 3 (left). The two atoms $R(\underline{x}, y, v)$ and $S(\underline{y}, x)$ together constitute an initial strong component of the attack graph. It is then straightforward that each Markov cycle containing x or y must be premier. Further, the Markov cycle $\langle v, w, v\rangle$ is also premier, because $\mathcal{K}(q) \models v \to x$ and the Markov graph contains a directed path from x to v.

Let q be like in Definition 3 and assume that the Markov graph of q contains an elementary directed cycle $\mathcal{C}$. Our key result (Lemma 2) states that $\mathsf{CERTAINTY}(q)$ can be reduced in polynomial time to $\mathsf{CERTAINTY}(q^*)$, where q^* is obtained from q by "dissolving" the Markov cycle $\mathcal{C}$, a notion that is defined next.

Definition 5. Let $q \in$ sjfBCQ such that every atom with mode i in q is simple-key. Let $\mathcal{C}$ be an elementary directed cycle of length $k \geq 2$ in the Markov graph of q. Then, $\mathsf{dissolve}(\mathcal{C}, q)$ denotes the sjfBCQ query defined next. Let $x_0, \ldots, x_{k-1}$ be the variables in $\mathcal{C}$, and let $q_0 = \bigcup_{i=0}^{k-1} \mathsf{C}_q(x_i)$. Let $\vec{y}$ be a sequence of variables containing exactly once each variable of $\mathsf{vars}(q_0) \setminus \{x_0, \ldots, x_{k-1}\}$. Let

$$q_1 = \{T(\underline{u}, x_0, \ldots, x_{k-1}, \vec{y})\} \cup \{U_i^c(\underline{x_i}, u)\}_{i=0}^{k-1}$$

where u is a fresh variable, T is a fresh relation name with mode i, and $U_1, \ldots, U_{k-1}$ are fresh relation names with mode c. Then, we define

$$\mathsf{dissolve}(\mathcal{C}, q) := (q \setminus q_0) \cup q_1.$$

Notice that $\mathsf{dissolve}(\mathcal{C}, q)$ is unique up to a renaming of the variable u and the relation names in q_1.

Example 4. Let q be the query of Fig. 3. Let $\mathcal{C}$ be the cycle $\langle x, w, y, x\rangle$ in the Markov graph of q. Using the notation of Definition 5, we have

$$\begin{aligned} q_0 &= \{R(\underline{x}, y, v), S(\underline{y}, x), W(\underline{w}, v)\}, \\ q_1 &= \{T(\underline{u}, x, w, y, v), U_1^c(\underline{x}, u), U_2^c(\underline{w}, u), U_3^c(\underline{y}, u)\}. \end{aligned}$$

Hence, $\mathsf{dissolve}(\mathcal{C}, q) = \{V_1^c(\underline{v}, w)$, $V_2^c(\underline{w}, y)$, $T(\underline{u}, x, w, y, v)$, $U_1^c(\underline{x}, u)$, $U_2^c(\underline{w}, u)$, $U_3^c(\underline{y}, u)\}$.

Lemma 2. *Let $q \in$ sjfBCQ such that every atom with mode i in q is simple-key. Let $\mathcal{C}$ be an elementary directed cycle in the Markov graph of q, and let $q^* = \mathsf{dissolve}(\mathcal{C}, q)$. Then, there exists a polynomial-time many-one reduction from $\mathsf{CERTAINTY}(q)$ to $\mathsf{CERTAINTY}(q^*)$.*

We will illustrate the reduction of Lemma 2 in Section 5.3. In order to show that dissolving Markov cycles leads to a polynomial-time algorithm, two more results are needed:

- We need to show that the "dissolution" of Markov cycles can be done while keeping the attack graph free of strong cycles (Lemma 3). Surprisingly, this turns out to be true only for Markov cycles that are premier.
- We need to show the existence of premier Markov cycles that can be "dissolved" (Lemma 4).

Lemma 3. *Let $q \in$ sjfBCQ such that every atom with mode i in q is simple-key. Let $\mathcal{C}$ be an elementary directed cycle in the Markov graph of q such that $\mathcal{C}$ is premier, and let $q^* = \mathsf{dissolve}(\mathcal{C}, q)$. If the attack graph of q contains no strong cycle, then the attack graph of q^* contains no strong cycle either.*

Lemma 4. *Let $q \in$ sjfBCQ such that*

- *for every atom $F \in q$, if F has mode i, then F is simple-key and $\mathsf{key}(F) \neq \emptyset$;*
- *q is saturated;*
- *the attack graph of q contains no strong cycle; and*
- *the attack graph of q contains an initial strong component with two or more atoms.*

Then, the Markov graph of q contains an elementary directed cycle that is premier and for every y in $\mathcal{C}$, $\mathsf{C}_q(y) \neq \emptyset$.

The condition $\mathsf{C}_q(y) \neq \emptyset$, for every y in $\mathcal{C}$, guarantees that $\mathsf{dissolve}(\mathcal{C}, q)$ will contain strictly less atoms of mode i than q. This condition is needed in the proof of Theorem 3 which runs by induction on the number of atoms with mode i.

5.3 The Reduction of Lemma 2

We will use an example to illustrate the main ideas behind the reduction of Lemma 2. Let $q \in$ sjfBCQ, and assume that q includes $q_0 = \{R(\underline{x}, y)$, $S(\underline{y}, z)$, $V(\underline{z}, x)\}$. Then, the Markov graph of q contains a cycle

$$x \xrightarrow{\mathsf{M}} y \xrightarrow{\mathsf{M}} z \xrightarrow{\mathsf{M}} x.$$

Let $\mathbf{db}$ be an uncertain database. Let $\mathbf{db}_0$ be the subset of $\mathbf{db}$ containing all R-facts, S-facts, and V-facts of $\mathbf{db}$.

Assume that the following three tables represent all facts of $\mathbf{db}_0$ (for convenience, we use variables as attribute names, and we blur the distinction between a relation name R and a table representing a set of R-facts).

R	$\underline{x}$	y
	1	a
	2	b
	2	c
	3	d
	3	e
	4	e
	4	f

S	$\underline{y}$	z
	a	α
	a	κ
	b	β
	c	γ
	d	δ
	e	ϵ
	e	δ
	f	ϕ

V	$\underline{z}$	x	
	α	1	$\}\ \mathbf{db}_{01}$
	κ	1	
	β	2	$\}\ \mathbf{db}_{02}$
	γ	2	
	δ	3	$\}\ \mathbf{db}_{03}$
	ϵ	3	
	δ	4	
	ϕ	4	

As indicated, we can partition $\mathbf{db}_0$ into three subsets $\mathbf{db}_{01}$, $\mathbf{db}_{02}$, and $\mathbf{db}_{03}$ whose active domains have, pairwise, no constants in common. Consider each of these three subsets in turn.

1. $\mathbf{db}_{01}$ has two repairs, both satisfying q_0. For every repair $\mathbf{r}$ of $\mathbf{db}$, we have either $\mathbf{r} \models q_{0[x,y,z\mapsto 1,a,\alpha]}$ or $\mathbf{r} \models q_{0[x,y,z\mapsto 1,a,\kappa]}$.
2. $\mathbf{db}_{02}$ has two repairs, both satisfying q_0. For every repair $\mathbf{r}$ of $\mathbf{db}$, we have either $\mathbf{r} \models q_{0[x,y,z\mapsto 2,b,\beta]}$ or $\mathbf{r} \models q_{0[x,y,z\mapsto 2,c,\gamma]}$.
3. $\mathbf{db}_{03}$ has 16 repairs, and for $\mathbf{s} := \{R(\underline{3}, d),\ S(\underline{d}, \delta),\ V(\underline{\delta}, 4),\ R(\underline{4}, e),\ S(\underline{e}, \epsilon),\ V(\underline{\epsilon}, 3),\ S(\underline{f}, \phi),\ V(\underline{\phi}, 4)\}$, we have that $\mathbf{s}$ is a repair of $\mathbf{db}_{03}$ that falsifies q_0. It can be easily seen that every repair of $\mathbf{db}$ satisfies q if and only if every repair of $\mathbf{db} \setminus \mathbf{db}_{03}$ satisfies q. That is, $\mathbf{db}_{03}$ can henceforth be ignored.

The following table T summarizes our findings. In the first column (named with a fresh variable u), the values 01 and 02 refer to $\mathbf{db}_{01}$ and $\mathbf{db}_{02}$ respectively. The table includes two blocks (separated by a dashed line for clarity). The first block indicates that for every repair $\mathbf{r}$ of $\mathbf{db}$, either $\mathbf{r} \models q_{0[x,y,z\mapsto 1,a,\alpha]}$ or $\mathbf{r} \models q_{0[x,y,z\mapsto 1,a,\kappa]}$. Likewise for the second block.

T	$\underline{u}$	x	y	z
	01	1	a	α
	01	1	a	κ
	02	2	b	β
	02	2	c	γ

The table U_x shown below is the projection of T on attributes x and u. This table must be consistent, because by construction, the active domains of $\mathbf{db}_{01}$ and $\mathbf{db}_{02}$ are disjoint. Likewise for U_y and U_z.

U_x	$\underline{x}$	u
	1	01
	2	02

U_y	$\underline{y}$	u
	a	01
	b	02
	c	02

U_z	$\underline{z}$	u
	α	01
	κ	01
	β	02
	γ	02

Let $\mathbf{db}'$ be the database that extends $\mathbf{db}$ with all the facts shown in the tables T, U_x, U_y, and U_z. Let $q^* = (q \setminus q_0) \cup \{T(\underline{u}, x, y, z),\ U_x^c(\underline{x}, u),\ U_y^c(\underline{y}, u),\ U_z^c(\underline{z}, u)\}$. From our construction, it follows that every repair of $\mathbf{db}$ satisfies q if and only if every repair of $\mathbf{db}'$ satisfies q^*.

Facts of $\mathbf{db}_0$ can be omitted from $\mathbf{db}'$, but that is not important.

6. RELATED WORK

Theoretical developments. Consistent query answering (CQA) goes back to the seminal work by Arenas, Bertossi, and Chomicki [1]. Fuxman and Miller [8] were the first to focus on CQA under the restrictions that consistency is only with respect to primary keys and that queries are self-join-free conjunctive queries. The term CERTAINTY(q) was coined in [20]. A recent and comprehensive survey on the problem CERTAINTY(q) is [23].

In the past decade, a variety of techniques have been used in the complexity classification task of CERTAINTY(q) for sjfBCQ queries q. In their pioneering work, Fuxman and Miller [8] introduced the notion of *join graph* (not to be confused with the classical notion of join tree). Later, Wijsen [20] introduced the notion of *attack graph*. Kolaitis and Pema [10] applied Minty's algorithm [18]. Koutris and Suciu [12] introduced the notion of *query graph* and the distinction between consistent and possibly inconsistent relations.

Little is known about CERTAINTY(q) beyond self-join-free conjunctive queries. An interesting recent result by Fontaine [6] goes as follows. Let UCQ be the class of Boolean first-order queries that can be expressed as disjunctions of Boolean conjunctive queries (possibly with constants and self-joins). A daring conjecture is that for every UCQ query q, CERTAINTY(q) is either in **P** or **coNP**-complete. Fontaine showed that this conjecture implies Bulatov's dichotomy theorem for conservative CSP [4], the proof of which is highly involved (the full paper contains 66 pages).

The counting variant of CERTAINTY(q), which has been denoted ♯CERTAINTY(q), asks to determine the exact number of repairs that satisfy some Boolean query q. In [16], it was shown that for every sjfBCQ query q, the counting problem ♯CERTAINTY(q) is either in **FP** or ♯**P**-complete. For conjunctive queries q with self-joins, the complexity of ♯CERTAINTY(q) has been established under the restriction that all atoms are simple-key [17].

Implemented systems. In the past, the paradigm of consistent query answering, and CERTAINTY(q) in particular, has been implemented in expressive formalisms, such as Disjunctive Logic Programming [9] and Binary Integer Programming (BIP) [11]. In these formalisms, it is relatively easy to express an exponential-time algorithm for CERTAINTY(q). The drawback is that the efficiency of these algorithms is likely to be far from optimal in case that the certain answer is computable in polynomial-time or expressible in first-order logic. In the latter case, the consistent answer can be computed by a single SQL query using standard database technology, including query optimization. In [3, page 38], the author mentions that logic programs for CQA cannot compete with first-order query rewriting mechanisms when they exist. Likewise, in an experimental comparison of EQUIP [11] and ConQuer [7], the authors of the former system found that BIP never outperformed first-order query rewriting.

The Hippo system [5] implements a polynomial-time algorithm for CQA with respect to denial constraints, for quantifier-free first-order queries. Since primary keys are denial constraints, this algorithm can be used in our setting for computing certain answers to self-join-free conjunctive queries in which all variables are free. Note, however, that such queries have an empty (and hence acyclic) attack graph, in which case our results imply that the consistent answer

can also be computed by a single SQL query.

In summary, the practical relevance of our results is that they tell us when computationally expensive formalisms can be avoided in the computation of consistent answers. Moreover, by looking at practical examples, we found that many natural self-join-free conjunctive queries have acyclic attack graphs, meaning that the certain answer can be computed by a single SQL query. That is, the "easiest" case is by no means exceptional. For example, as soon as a self-join-free conjunctive query, expressed in SQL, on the example database of Fig. 1 satisfies one of the following conditions, then its certain answer can be computed in SQL:

- the FROM clause contains only one table;
- the SELECT clause includes one or both primary keys (i.e., E.EID or D.DNAME); or
- the WHERE clause joins E and D on either E.EID = D.MGR or E.DNAME = D.DNAME (but not on both). In other words, the join is a simple primary-to-foreign key join.

7. CONCLUSION

This paper settles a long-standing open question in consistent query answering, by providing a solution to the complexity classification task of CERTAINTY(q) for sjfBCQ queries q. We show that it is decidable, given q, whether the problem CERTAINTY(q) is in **FO**, in $\mathbf{P} \setminus \mathbf{FO}$, or **coNP**-complete. Moreover, if the problem is in **FO** or in **P**, then we can effectively construct a first-order query or a polynomial-time algorithm for solving it.

An exciting question is whether our results can be extended beyond self-join-free conjunctive queries, to conjunctive queries with self-joins and unions of conjunctive queries.

Acknowledgments

This work is supported in part by the NSF through NSF grant NSF IIS-1115188.

8. REFERENCES

[1] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS*, pages 68–79. ACM Press, 1999.

[2] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30(3):479–513, 1983.

[3] L. E. Bertossi. *Database Repairing and Consistent Query Answering*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[4] A. A. Bulatov. Complexity of conservative constraint satisfaction problems. *ACM Trans. Comput. Log.*, 12(4):24, 2011.

[5] J. Chomicki, J. Marcinkowski, and S. Staworko. Hippo: A system for computing consistent answers to a class of SQL queries. In E. Bertino et al., editors, *EDBT*, volume 2992 of *Lecture Notes in Computer Science*, pages 841–844. Springer, 2004.

[6] G. Fontaine. Why is it hard to obtain a dichotomy for consistent query answering? In *LICS*, pages 550–559. IEEE Computer Society, 2013.

[7] A. Fuxman, E. Fazli, and R. J. Miller. ConQuer: Efficient management of inconsistent databases. In F. Özcan, editor, *SIGMOD Conference*, pages 155–166. ACM, 2005.

[8] A. Fuxman and R. J. Miller. First-order query rewriting for inconsistent databases. In T. Eiter and L. Libkin, editors, *ICDT*, volume 3363 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2005.

[9] G. Greco, S. Greco, and E. Zumpano. A logical framework for querying and repairing inconsistent databases. *IEEE Trans. Knowl. Data Eng.*, 15(6):1389–1408, 2003.

[10] P. G. Kolaitis and E. Pema. A dichotomy in the complexity of consistent query answering for queries with two atoms. *Inf. Process. Lett.*, 112(3):77–85, 2012.

[11] P. G. Kolaitis, E. Pema, and W. Tan. Efficient querying of inconsistent databases with binary integer programming. *PVLDB*, 6(6):397–408, 2013.

[12] P. Koutris and D. Suciu. A dichotomy on the complexity of consistent query answering for atoms with simple keys. In Schweikardt et al. [19], pages 165–176.

[13] P. Koutris and J. Wijsen. The data complexity of consistent query answering for self-join-free conjunctive queries under primary key constraints. In T. Milo and D. Calvanese, editors, *PODS*, pages 17–29. ACM, 2015.

[14] P. Koutris and J. Wijsen. A trichotomy in the data complexity of certain query answering for conjunctive queries. *CoRR*, abs/1501.07864, 2015.

[15] L. Libkin. *Elements of Finite Model Theory*. Springer, 2004.

[16] D. Maslowski and J. Wijsen. A dichotomy in the complexity of counting database repairs. *J. Comput. Syst. Sci.*, 79(6):958–983, 2013.

[17] D. Maslowski and J. Wijsen. Counting database repairs that satisfy conjunctive queries with self-joins. In Schweikardt et al. [19], pages 155–164.

[18] G. J. Minty. On maximal independent sets of vertices in claw-free graphs. *J. Comb. Theory, Ser. B*, 28(3):284–304, 1980.

[19] N. Schweikardt, V. Christophides, and V. Leroy, editors. *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*. OpenProceedings.org, 2014.

[20] J. Wijsen. On the first-order expressibility of computing certain answers to conjunctive queries over uncertain databases. In J. Paredaens and D. V. Gucht, editors, *PODS*, pages 179–190. ACM, 2010.

[21] J. Wijsen. A remark on the complexity of consistent conjunctive query answering under primary key violations. *Inf. Process. Lett.*, 110(21):950–955, 2010.

[22] J. Wijsen. Certain conjunctive query answering in first-order logic. *ACM Trans. Database Syst.*, 37(2):9, 2012.

[23] J. Wijsen. A survey of the data complexity of consistent query answering under key constraints. In C. Beierle and C. Meghini, editors, *FoIKS*, volume 8367 of *Lecture Notes in Computer Science*, pages 62–78. Springer, 2014.

Technical Perspective: Broadening and Deepening Query Optimization Yet Still Making Progress

Jeffrey F. Naughton
University of Wisconsin–Madison
Madison WI, U.S.A.

Query optimization is a fundamental problem in data management. Simply put, most database query languages are declarative rather than imperative – that is, they specify properties that the answer should satisfy, rather than give an algorithm to compute the answer. The best known and most widely used database query language, SQL, is a prime example of a language for which optimization is essential.

By "essential" I mean that database optimization is not a matter of shaving 10% or even a factor of 2X from a query's execution time. In database query evaluation, the difference between a good plan and a bad or even average plan can be multiple orders of magnitude – so successful query optimization makes the difference between a plan that runs quickly and one that never finishes at all. Accordingly, since the seminal papers in the 1970s, query optimization has received and continues to receive a great deal of attention from both the industrial and research database communities.

Early work on optimization focused on a scenario in which the query was fully specified, and the optimization goal was query evaluation time. That is, the problem was this: what is the fastest way to evaluate this query? While this problem was (and is!) challenging, it is not broad enough to capture the optimization problem faced by modern systems. As an important example, many times the query is not fully specified in advance (as a simple example, it may contain variables, or "parameters" that are only discovered at run time). This generalization gives rise to *parametric query optimization*, where the problem is as follows:

> Given a partially specified query, find a set of good evaluation plans, one of which will be chosen at run time when the parameter is instantiated.

Yet another necessary generalization has to do with the optimization goal. Sometimes execution time is not the only criterion by which plans should be selected. As a prominent and current example, if the query is being run in the cloud, the system may of course want to find fast evaluation plans, but may also desire inexpensive ones. That is, now we have two objectives: running time and cost. This gives rise to *multi-objective query optimization*, where the problem is as follows:

> Given a query and a set of objectives, find a set of plans that are Pareto-optimal for these objectives (a plan is "Pareto-optimal" if it is not dominated in all objectives by other plans.)

Both parametric and multi-objective query optimization have been studied in the past, but the following paper, by Trummer and Koch, is a remarkable tour de force exploration of the combination of the two. Here, the problem is roughly the following. Given a partially specified query, and multiple objectives for the resulting plan, find a set of Pareto-optimal plans that can be chosen at run time by filling in all parameters.

Since the original query optimization problem and its variants are already very difficult, one might despair that simultaneously treating two substantial extensions would yield a hopelessly intractable problem. Thus the current paper is surprising in its elegance and effectiveness. The paper embeds the problem in an insightful and expressive formal framework, and specifies a solution that combines aspects of piecewise linear functions, dynamic programming with pruning based upon Pareto polytope analyses, and linear programming. A thorough set of experiments with an implementation of their algorithm completes the paper, and indicates that all of this actually works.

Multi-Objective Parametric Query Optimization

Immanuel Trummer and Christoph Koch
Ecole Polytechnique Fédérale de Lausanne
{firstname}.{lastname}@epfl.ch

ABSTRACT

We propose a generalization of the classical database query optimization problem: multi-objective parametric query optimization (MPQ). MPQ compares alternative processing plans according to multiple execution cost metrics. It also models missing pieces of information on which plan costs depend upon as parameters. Both features are crucial to model query processing on modern data processing platforms.

MPQ generalizes previously proposed query optimization variants such as multi-objective query optimization, parametric query optimization, and traditional query optimization. We show however that the MPQ problem has different properties than prior variants and solving it requires novel methods. We present an algorithm that solves the MPQ problem and finds for a given query the set of all relevant query plans. This set contains all plans that realize optimal execution cost tradeoffs for any combination of parameter values. Our algorithm is based on dynamic programming and recursively constructs relevant query plans by combining relevant plans for query parts. We assume that all plan execution cost functions are piecewise-linear in the parameters. We use linear programming to compare alternative plans and to identify plans that are not relevant. We present a complexity analysis of our algorithm and experimentally evaluate its performance.

1. INTRODUCTION

Context. The goal of database query optimization is to map a query (describing the data to generate) to the optimal query plan (describing how to generate the data). Query optimization is a classical optimization problem with first work dating back to the seventies [14]. The original query optimization problem model has been motivated by the capabilities of data processing systems at that time. However, there have been fundamental advances in data processing techniques and systems in the meantime. Hence the original problem model is not sufficiently expressive to capture all relevant aspects of modern data processing systems. In this paper, we propose an extension of the classical query optimization problem model and a corresponding optimization algorithm.

Query optimization variants can be classified according to how they model the execution cost of a single query plan. Traditional query optimization [14] models the cost of a query plan as scalar cost value $c \in \mathbb{R}$. This implies that query plans are compared according to one single cost metric. It also implies that all information required to produce cost estimates is available to the query optimizer. The goal in classical query optimization is to find a query plan with minimal execution cost. Multi-objective query optimization [1, 7, 11, 16, 17] generalizes the classical model and associates each query plan with a cost vector $c \in \mathbb{R}^n$ instead of a scalar value. This allows to model scenarios where multiple execution cost metrics are of interest. If data processing takes place in the Cloud then we are for instance interested in execution time but also in monetary execution fees. Different components of the plan cost vector represent cost according to different cost metrics. The goal is to find the set of Pareto-optimal query plans which are plans for which no alternative plan offers better cost according to all metrics. Parametric query optimization [3, 4, 6, 8, 10, 13] generalizes the standard model in a different way. It associates each query plan with a cost function $c \in \mathbb{R}^m \rightarrow \mathbb{R}$, mapping from a multi-dimensional parameter space to a one-dimensional cost space. Parameters represent pieces of information that are not yet available at optimization time but required to estimate plan execution cost. Parametric query optimization allows for instance to optimize query classes that are defined via query templates in which some predicates are unspecified. One parameter could then represent the selectivity of one unspecified predicate. The goal in parametric query optimization is typically to find a set of plans containing for each possible parameter value combination the plan with minimal execution cost.

Problem. We propose multi-objective parametric query optimization (MPQ), a query optimization variant that generalizes multi-objective query optimization, parametric query optimization, and classical query optimization at the same time. MPQ models the cost of a single query plan as a cost function $c \in \mathbb{R}^m \rightarrow \mathbb{R}^n$ that maps a multi-dimensional parameter space to a multi-dimensional cost space. MPQ assumes that query plans are compared according to multiple cost metrics and that cost estimates depend on parameters whose values are unknown at optimization time. The goal in MPQ is to find the set of Pareto-optimal plans for each possible parameter value combination. This problem model is required wherever the application scenarios of multi-objective query optimization intersect with the ones of parametric query optimization. The following example describes a scenario in which MPQ is necessary.

EXAMPLE 1. *Assume that we need to process the same query in regular time intervals. Query processing takes place*

The original version of this article was published in the Proceedings of the VLDB Endowment, Volume 8. A video recording of the associated conference talk can be found at `http://www.itrummer.org/Talks.html`.

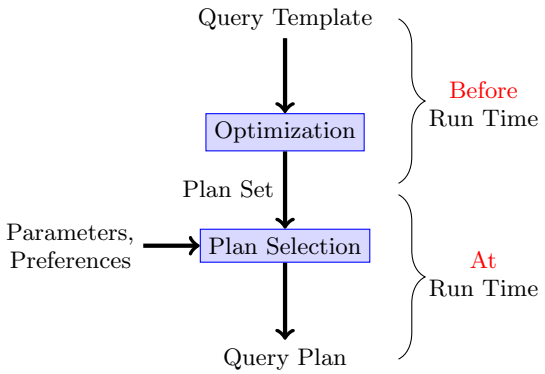


Figure 1: Multi-objective parametric query optimization pre-computes a set of relevant query plans. The optimal plan is selected from that set according to parameter values and user preferences.

in the Cloud and we would like to use Amazon EC2 Spot Instances. Here we care about two execution cost metrics which is execution time and monetary execution fees. We can trade between them by adapting type and number of the computational resources that we rent from Amazon. On the other side, the processing cost of the query depends on parameters that we cannot directly influence: the pricing of Amazon Spot Instances. As we process the same query repeatedly, we can determine the set of all potentially relevant query plans in a pre-processing step. At run time, given concrete Spot prices and execution cost bounds, we can efficiently select the best query plan out of the pre-computed set. This avoids expensive optimization at run time. The pre-processing step requires however MPQ since multiple plan cost metrics and parameters need to be considered.

There are many other scenarios in which multiple processing cost metrics are of interest. Techniques for approximate query processing allow for instance to trade between execution time and result precision [1]. Different query plans can realize different tradeoffs between energy consumption and execution time for the same query [18]. If data is processed by crowd workers then latency, execution fees, and result precision are all relevant cost metrics [12]. If the queries we want to process at run time correspond to query templates that are known before run time then we can make query optimization a pre-processing step. At pre-processing time, plan cost estimates depend on parameters with unknown values. Those parameters can represent query properties which are not fully specified in the template or properties of the query execution platform (e.g., the Spot Instance prices) that will become known only at run time. MPQ is applicable in such scenarios and allows to avoid query optimization at run time.

The result of MPQ is the set of all potentially relevant query plans for a given query or query template. It contains all Pareto-optimal plans for each possible parameter value combination. At run time, we can select the best query plan out of that set based on the concrete parameter values and based on user preferences. Users can specify their preferences in advance (e.g., by specifying cost bounds and priorities between different cost metrics [1, 16]) such that the optimal plan according to those preferences can be selected automatically. As an alternative, we can use the pre-computed plan set to visualize all Pareto-optimal cost tradeoffs for given parameter values. This allows users to select the preferred cost tradeoff directly [17]. Figure 1 illustrates the context of MPQ.

So far we have introduced a very generic problem model for MPQ. In order to make the problem tractable, we restrict ourselves to a specific class of cost functions in this paper: we consider piecewise-linear plan cost functions. Many approaches for parametric query optimization [6, 8] consider only piecewise-linear plan cost functions as well since such functions can approximate arbitrary functions [8]. The difference between our model and the one used in parametric query optimization is that we associate each plan with multiple piecewise-linear cost functions representing cost according to different metrics.

Algorithm. We present an algorithm that solves MPQ for piecewise-linear plan cost functions. Our algorithm is based on dynamic programming. It recursively decomposes the input query for which we need to determine the set of relevant query plans into sub-queries. In a bottom-up approach, it recursively calculates sets of relevant plans for a query out of optimal plan sets for its sub-queries: it combines plans that are relevant for the sub-queries to form new plans that are potentially relevant for the decomposed query. Dynamic programming is a classical approach for query optimization. The crucial difference between our algorithm and prior algorithms is the implementation of the pruning function, i.e. in how we compare alternative query plans and prune out sub-optimal plans.

We conceptually associate each plan for a query or subquery with a region in the parameter space for which the plan is Pareto-optimal. We call this region the Pareto region. The goal during pruning is to compare alternative plans generating the same result in order to discard sub-optimal plans. We compare plans pair-wise and determine for each plan the parameter space region in which it is dominated by another plan, i.e. in which the other plan has comparable or better cost according to each plan cost metric. Then we reduce the Pareto region of the first plan by the region in which it is dominated. If the Pareto region of a plan becomes empty then it is not Pareto-optimal for any parameter value combination. Then we can safely discard that plan.

All Pareto regions that could ever occur during the execution of that algorithm can be represented using the following formalism. We represent Pareto regions as a union of convex polytopes in the parameter space from which other convex polytopes have been subtracted. We prove that this representation is closed under all operations that the algorithm needs to perform on Pareto regions. Note that this region shape is a consequence of the class of cost functions (piecewise-linear functions) that we consider.

The algorithm needs to perform several elementary operations on Pareto regions and cost functions. It must for instance verify whether a Pareto region is empty or calculate a parameter space region in which one plan is preferable to a second one. We show how all those operations can be implemented based on the aforementioned representation of Pareto regions. We implement those operations using linear programming.

We will formally analyze the complexity of this algorithm and present experimental results in the following sections.

Outline. The remainder of this paper is organized as follows. We define the MPQ problem and related concepts more formally in Section 2. In Section 3, we describe our algorithm for MPQ with piecewise-linear plan cost functions. In Section 4, we analyze the MPQ problem and the asymptotic complexity of our algorithm. We present experimental results for an implementation of our algorithm in Section 5 and discuss related work in Section 6.

2. FORMAL PROBLEM STATEMENT

We define the MPQ problem and related concepts more formally than in the introduction. A query describes data to generate. The description of our algorithm for solving MPQ problems, given in the next section, focuses on simple SQL join queries. An SQL join query is defined by a set of tables to join. A sub-query joins a subset of tables. Standard methods exist by which a query optimization algorithm for this simple query language can be extended into an algorithm supporting full SQL queries [14].

A query plan describes how to generate data. We say that a query plan answers a query if it generates the data that is described by the query. We assume in the following that query plans consist of a sequence of scan operations and binary join operations. For a query q, we denote by $P(q)$ the set of alternative plans that answer the query.

We compare query plans according to their execution cost. The cost of a given plan depends on a set of real-valued parameters. The set of parameters is a property of the query. All alternative plans in $P(q)$ depend therefore on the same parameters. A parameter value vector contains for each parameter a corresponding value. We do not know the parameter values at optimization time. The parameter space is the set of all possible parameter value vectors. We assume in the following that there are n parameters and denote by $X \subseteq \mathbb{R}^n$ the n-dimensional parameter space. A parameter space region is a subset of the parameter space.

We compare query plans according to multiple execution cost metrics. A cost vector contains for each cost metric a non-negative cost value. We assume in the following that there are m execution cost metrics and denote by $C = \mathbb{R}^m$ the space of cost vectors. We associate each query plan p with a cost function $c_p : X \to C$ that maps the n-dimensional parameter space to the m-dimensional cost space. We can compare the cost of query plans for specific parameter values. Denote by $x \in X$ a parameter value vector and by p_1 and p_2 two plans answering the same query. We say that p_1 dominates p_2 for x, written $p_1 \preceq_x p_2$, if p_1 has lower or equivalent cost than p_2 according to each metric for parameter values x. In other words, p_1 dominates p_2 if $c_{p_1}(x)$ contains for no component a higher value than $c_{p_2}(x)$. Now we are ready to introduce the MPQ problem.

Definition 1. An **MPQ problem** is defined by a query q, a parameter space X, and a cost space C. A solution is a subset $S \subseteq P(q)$ of query plans such that for each possible plan $p \in P(q)$ and for each possible parameter value vector $x \in X$ there is a solution plan $s \in S$ such that s dominates p for x, i.e. $s \preceq_x p$.

We focus on a sub-class of MPQ problems that restricts the class of cost functions. In order to define the class of cost functions that we consider, we must first introduce convex polytopes. A convex polytope is defined by a set of linear inequalities. The convex polytope is the set of points in the parameter space that satisfy all its inequalities. We use the terms convex polytope and polytope as synonyms. A linear cost function is defined by a constant b and an n-dimensional weight vector $w \in \mathbb{R}^n$ such that $b + w^T \times x$ is the associated cost value for each parameter vector $x \in X$. A scalar piecewise-linear cost function is a cost function that allows to partition the parameter space into convex polytopes such that the function is linear in each polytope. A vector-valued piecewise-linear cost function consists of one piecewise-linear cost function for each cost metric. We use the terms vector-valued piecewise-linear cost function and piecewise-linear cost function as synonyms. We restrict our scope to MPQ with piecewise-linear cost functions.

3. ALGORITHM

Our algorithm produces a set of relevant plans for a given query. A plan is relevant if its execution cost is Pareto-optimal for some parameter value combination.

Overview. Our algorithm splits the input query recursively into smaller and smaller parts until we obtain atomic sub-queries. We start with atomic sub-queries and calculate the set of relevant plans for each of them. After that, larger sub-queries are treated. We treat sub-queries in an order which makes sure that before treating a query, we have calculated relevant plan sets for each of its sub-queries. The reason for restricting the order is that we want to calculate the set of relevant plans for a query out of the sets of relevant plans for its sub-queries. More precisely, we can guarantee that each relevant plan for a query can be obtained by splitting the query into two sub-queries and combining a relevant plan for the first sub-query with a relevant plan for the second sub-query, thereby generating a new query plan. Having calculated the set of relevant plans for each sub-query, we can therefore obtain a superset of relevant query plans by iterating over all possible splits into sub-queries and over all possible combinations of relevant sub-plans. In order to reduce the superset to the actual set of relevant query plans, we must prune plans answering the same query. Pruning them means to identify and to discard plans that are irrelevant. The input query is treated last. The set of relevant plans for the input query is the desired algorithm output. In summary, our algorithm can be written as follows:

- Iterate over all sub-queries s of the input query in ascending order of query size:
  - If sub-query s is an atomic sub-query then consider all possible plans for s
  - Otherwise, if s is not an atomic sub-query, then iterate over all possibilities to decompose s into two sub-queries s_1 and s_2:
    - For each split into two sub-queries s_1 and s_2, consider all plans that are combinations of a relevant plan for s_1 and a relevant plan for s_2
  - Prune all considered plans to obtain the set of relevant plans for s

As many query optimization algorithms [8, 14, 16], our algorithm is based on dynamic programming. We can use dynamic programming since the principle of optimality holds for query optimization [14]. Formulated in general terms, the principle of optimality designates the problem property

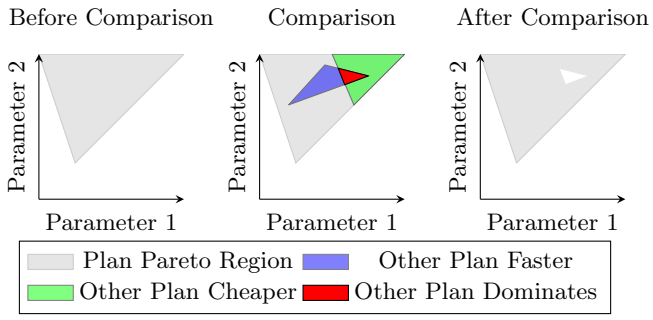


Figure 2: We subtract the area in which a plan is dominated from its Pareto region.

that optimal solutions can be obtained by combining optimal solutions to sub-problems. In the context of query optimization, the principle of optimality means more specifically that optimal query plans can be obtained by combining optimal plans for sub-queries. The principle of optimality has been shown to hold for all common execution cost metrics in multi-objective query optimization [16]. This means that a Pareto-optimal query plan can be combined from Pareto-optimal plans for sub-queries. A relevant plan is Pareto-optimal for some points in the parameter space. It is therefore intuitive that a relevant query plan can be combined from relevant plans for the sub-queries (we omit the formal proof). In other words, the principle of optimality holds for MPQ as well. It is the fundament of our MPQ algorithm.

Pruning. Many query optimization algorithms for classical query optimization [14], multi-objective query optimization [16], or parametric query optimization [8] are based on dynamic programming. The primary difference between all those algorithms is the realization of the pruning function. As we treat a novel problem variant, we must design a novel pruning function. In the following, we describe how our algorithm prunes query plans, i.e. how it compares plans for the same query and identifies irrelevant plans.

Our pruning function is based on the key concept of the Pareto region. Each query plan is associated with a Pareto region. This is a parameter space region in which it realizes Pareto-optimal cost tradeoffs. A plan is irrelevant if its Pareto region is empty. The goal of the pruning function is to compare a set of plans answering the same query in order to calculate their Pareto regions. The pruning function works as follows. At pruning start, we assume by default that each plan is Pareto-optimal in the entire parameter space. This means that we assign the entire parameter space as Pareto region to each query plan. During pruning, we compare all query plans answering the same query pair-wise in order to calculate their true Pareto regions. If we compare two plans p_1 and p_2 and we find that plan p_1 has better cost than or equivalent cost to p_2 according to all cost metrics for the parameter space region X then we reduce the Pareto region of p_2 by subtracting X. Pareto regions can only shrink during a pruning operation. Once the region of one plan becomes empty, it is irrelevant and can be safely discarded. We discard plans as soon as possible in order to avoid unnecessary comparisons.

More precisely, the pruning function iterates over all plan pairs and executes for each pair the following steps. First, it identifies the region in which one plan dominates the other plan. Second, it updates the Pareto region of the dominated plan by subtracting the region in which it is dominated. Third, it checks whether the Pareto region of the dominated plan becomes empty after the update. In that case, the plan is discarded and does not participate in further comparisons. Figure 2 illustrates how the Pareto region of a plan is reduced after comparing it to another plan. The example refers to a scenario where two parameters and two cost metrics are considered (execution time and fees).

Note that two plans can mutually dominate each other in different parameter space regions. Having determined that a first plan dominates a second plan for some parameter space region, we must therefore still verify if the second plan dominates the first plan as well.

Data Structures. We describe the data structures by which we represent plan cost functions and Pareto regions. Our plan cost model is based on piecewise-linear functions. A piecewise-linear function is linear in parameter space regions that form convex polytopes. A linear function can be represented by a constant and by weights capturing the slope of the function for each parameter. Hence a piecewise-linear function can be represented by a set of convex polytopes where each convex polytope is associated with a constant and weights. We consider multiple plan cost metrics. Each query plan is therefore associated with one piecewise-linear cost function per plan cost metric.

We consider the class of piecewise-linear cost functions to represent plan cost. We decided to use that class of functions since it allows to approximate arbitrary functions up to an arbitrary degree of precision (using more pieces increases precision). In contrast to that, we cannot freely decide which class of shapes we consider for representing Pareto regions. The algorithm must be able to represent each shape that could potentially occur during pruning. Our decision to use piecewise-linear cost functions implies the class of shapes that we need to consider as Pareto regions.

We describe our representation of Pareto regions. We motivate this representation in an informal way. It is however relatively easy to prove that the proposed representation covers all possible cases.

We start by considering the special case of linear cost functions. Parametric query optimization is a special case of MPQ. It has been shown in the domain of parametric query optimization that the parameter space region in which one plan is better than another plan according to one cost metric is a convex polytope if both plans have linear cost functions [6]. In a setting with multiple cost metrics, a plan is strictly better than another plan if it is better according to each cost metric. The region in which a plan is better than another one is therefore an intersection of multiple convex polytopes. An intersection of convex polytopes is a convex polytope again. The region in which all other plans are better than a given plan is hence a union of convex polytopes.

Now let us generalize that reasoning from linear cost functions to piecewise-linear cost functions. The generalization is straight-forward. Given two piecewise-linear cost functions, we can always partition the parameter space into convex polytopes such that both cost functions are linear in each polytope. Thereby we reduce the case of piecewise-linear cost functions to the case of linear cost functions. In summary, we can represent the Pareto region of a plan as a union of convex polytopes from which we subtract another union of convex polytopes.

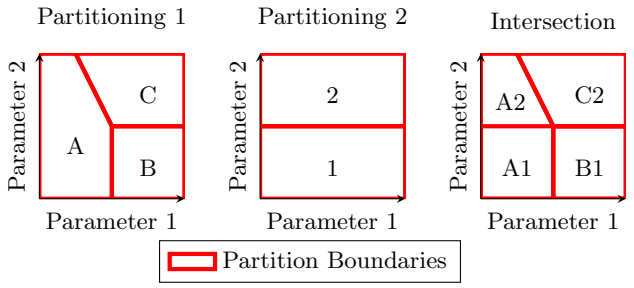


Figure 3: To compare two piecewise-linear cost functions, we intersect the parameter space partitions in which each function is linear. We compare the functions separately in each of the resulting partitions.

Elementary Operations. Having described the data structures used to represent cost functions and Pareto regions, we outline now how to implement elementary operations on those data structures. We require the following elementary operations to realize the pruning function as described before. First, given the cost functions of two plans, we must determine the parameter space region in which one plan dominates the other one. Second, given a Pareto region of a plan and a region in which it is dominated, we must reduce the Pareto region by that region. Third, given a Pareto region, we must determine whether it is empty.

Convex polytopes are described by a set of linear inequalities and we consider linear cost functions. All elementary operations that we describe in the following can hence be realized by solving systems of linear inequalities. Executing the elementary operations therefore requires a linear solver.

We describe how to determine the parameter space region in which one plan dominates another one. Assume first that we have only one cost metric and that cost functions are linear. Then we can directly use the linear solver to determine the parameter space region in which one function has lower values than the other one. Now we generalize from linear cost functions to piecewise-linear functions. Each piecewise-linear function partitions the parameter space into convex polytopes in which the function is linear. If we compare two piecewise-linear functions then we can partition the parameter space such that both functions are linear in each partition. More precisely, we obtain the aforementioned partitioning by intersecting the partitions associated with the first cost function with the partitions associated with the second function. Figure 3 illustrates how we intersect two parameter space partitionings in a two-dimensional parameter space. Having this partitioning, we apply the method for linear cost functions separately in each partition. If we have the sub-region in which a first plan dominates a second one for each parameter space region then the union of those sub-regions is the total area in which the first plan dominates. If we have multiple cost metrics instead of only one, then we can apply the method described before for each cost metric separately. If we have for each cost metric the parameter space region in which the first plan dominates the second one then the intersection of those areas (over all cost metrics) yields the area in which the first plan is better according to all cost metrics.

Given the area in which a plan is dominated, we must subtract it from that plan's Pareto region. The implementation of this operation is straight-forward: as discussed before, we represent Pareto regions as a union of convex polytopes from which other convex polytopes have been subtracted. The region in which one plan dominates another one must consist of convex polytopes. In order to subtract such a region from the Pareto region, we simply add the corresponding polytopes to the list of subtracted polytopes.

We must determine whether a given Pareto region is empty. A Pareto region is a set of polytopes from which other polytopes have been subtracted. We consider first the special case of one polytope P^+ from which another set of polytopes $\{P_i^-\}$ have been subtracted. We want to verify whether the given polytope becomes empty after the subtractions. We can verify that as follows. Assume that all subtracted polytopes P_i^- are contained within P^+. Then the region remaining after subtraction becomes empty if and only if $\cup_i P_i^- = P^+$. We can use the algorithm by Bemporad [2] to check the latter condition. The algorithm by Bemporad verifies whether the union of a given set of convex polytopes forms a convex polytope again. If this is the case then the algorithm constructs that polytope. The condition $\cup_i P_i^- = P^+$ can only be verified if $\cup_i P_i^-$ forms a convex polytope. In that case, the algorithm by Bemporad constructs the polytope $P^- = \cup_i P_i^-$ and a linear solver can verify whether P^- and P^+ are equivalent.

4. ANALYSIS

We analyze the formal properties of the freshly introduced MPQ problem in this section. We also analyze the complexity of the algorithm described in the last section.

Problem Analysis. MPQ generalizes parametric query optimization since it allows to consider multiple plan cost metrics instead of only one. We compare the formal properties of MPQ to the properties of parametric query optimization in the following.

The parametric query optimization problem with linear cost functions has the following property: if the same query plan is optimal at all vertices of a convex polytope in the parameter space then that plan must be optimal inside the polytope as well [6]. This property is commonly known as one of the "guiding principle of parametric query optimization" [5]. Many algorithms for parametric query optimization exploit this property as follows [6, 9]: they recursively decompose the parameter space into convex polytopes and calculate optimal query plans at the vertices. Due to the guiding principle, the decomposition of the parameter space can be stopped once the same plan is optimal at all vertices of a polytope. Such algorithms transform the parametric query optimization problem into a series of traditional query optimization problems (calculating the optimal plan at a polytope vertex is a traditional query optimization problem). This has the advantage that traditional query optimizers can be used for parametric query optimization with minimal changes. It is therefore interesting to verify whether an analogue property holds for MPQ.

Unfortunately this is not the case as we show next. The following property for MPQ would be analogue to the guiding principle of parametric query optimization: if the same set of plans is Pareto-optimal at all vertices of a polytope in the parameter space then that set of plans must be Pareto-optimal inside the polytope as well. Figure 4 illustrates a counter example showing that this property does not hold. The figure refers to a scenario in which two cost metrics,

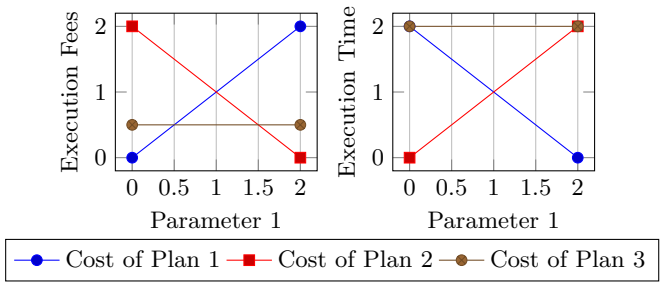


Figure 4: The guiding principle of parametric query optimization does not hold for multi-objective parametric query optimization.

namely execution time and execution fees, are of interest. Cost functions depend on a single parameter, called "Parameter 1" in the figure, that could refer to unspecified predicates in the input query template. We see the cost functions of three plans. For parameter value 0, plan 1 is Pareto-optimal since it has lowest execution fees. Plan 3 is Pareto-optimal since it has lower execution time than all other plans. Plan 2 is however dominated by plan 1 since plan 1 has equivalent execution time and lower execution cost. This means that plan 2 is not Pareto-optimal for parameter value 0. For parameter value 2, the situation is similar and plans 1 and 3 are Pareto-optimal while plan 2 is not. For parameter values between 0.5 and 1.5, plan 2 is however Pareto-optimal. Even though the same set of plans is Pareto-optimal at the borders of the parameter value interval $[0, 2]$, additional plans can be Pareto-optimal for values at the interior of that interval. All plan cost functions are linear in the example and an interval is a special case of a convex polytope. The example is minimal for MPQ: having less than two cost metrics would lead to parametric query optimization. Having less than one parameter would lead to multi-objective query optimization. Hence we can conclude from this example that the guiding principles do not apply for MPQ in general.

Algorithm Analysis. The space and time complexity of dynamic programming based query optimization algorithms depends on the number of plans stored per sub-query. In traditional query optimization, plans are compared according to one cost metric and cost functions do not depend on parameters. If we assume that alternative query plans are compared based on their cost values alone then exactly one plan, a plan with minimal cost, remains after pruning an arbitrary set of plans. In parametric query optimization, plans are compared according to one cost metric but cost functions depend on parameters. This means that different plans can be optimal for different parameter values. In multi-objective query optimization, we compare plans according to different cost metrics. Hence multiple plans can be Pareto-optimal for each sub-query. As a result, we generally need to store multiple plans per sub-query in parametric and in multi-objective query optimization. The number of plans to store depends on many factors. Research in parametric query optimization has focused on analyzing how the number of plans per sub-query depends on the number of parameters. Research in multi-objective query optimization has focused on the dependency between the number of plans and the number of cost metrics. Such analysis is necessarily based on simplifying assumptions. Traditionally, the weights that define the cost functions of different query plans are assumed to follow independent random distributions [7, 6]. Based on that assumption, the number of remaining plans after pruning can be considered a random variable as well and we can calculate its expected value. This reasoning led for instance to an asymptotic upper bound of 2^m, where m designates the number of cost metrics, on the expected number of plans per sub-query in multi-objective query optimization [7].

We perform a similar analysis to determine the expected number of plans per sub-query in MPQ. We consider linear cost functions. We denote the number of parameters by n. A linear function is therefore defined by a vector consisting of $n + 1$ components, specifying the function slope for each parameter and a constant. We still denote the number of considered plan cost metrics by m. Each query plan is therefore associated with m linear functions. The multi-dimensional cost function of each query plan can therefore be described by a matrix containing $m \cdot (n+1)$ components, specifying for each cost metric the cost slopes and a constant. Assume that we have two cost functions and that all constants and slopes describing the first function are lower than the corresponding entries for the second cost function. Then the first cost functions has for each cost metric a lower constant cost component and a lower slope in each parameter. In other words, the first cost function has lower values than the second one for arbitrary parameter values and cost metrics. If both cost functions are associated with query plans then the plan associated with the second function is clearly irrelevant.

We can exploit this fact as follows. Assume that we choose an arbitrary number of D-dimensional vectors randomly with independent identical distribution. Then the expected number of vectors such that no other vector has a lower or equivalent value in each component is bounded by 2^D [7]. We assume that vectors describing the cost functions of different query plans are chosen randomly with independent and identical distribution. Setting $D = m \cdot (n + 1)$, we infer that the expected number of vectors such that no other vector has lower or equal values in all components is bounded by $2^{m \cdot (n+1)}$. As outlined before, this is at the same time an upper bound on the expected number of relevant query plans per sub-query.

In order to obtain an upper bound on the asymptotic space complexity, we multiply the aforementioned bound by the number of sub-queries. We generate new plans by combining two relevant plans. The number of generated plans grows therefore as the square of the number of relevant plans. All generated plans for the same sub-query are compared pair-wise during pruning. The number of plan comparisons grows therefore as the fourth power of the number of relevant plans. Multiplying by the number of sub-query splits yields the time complexity measured by the number of plan comparisons.

5. EXPERIMENTS

Experimental Setup. We evaluate our MPQ algorithm experimentally. More precisely, we study how optimization time depends on the input query size and on the number of considered parameters. Our experiments are based on an example scenario in which SQL queries are processed in the Cloud. Hence we compare alternative query plans according to two cost metrics: execution time and monetary execution

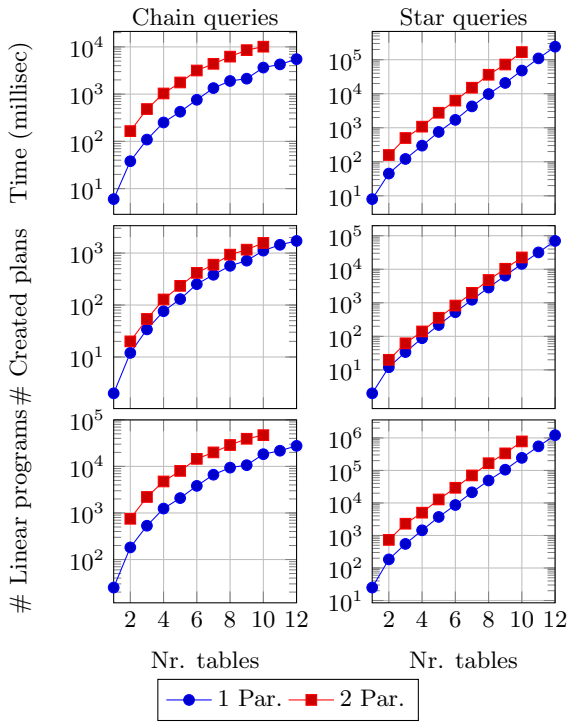


Figure 5: Optimization time, number of generated plans, and number of solved linear programs.

fees. We consider a restricted class of SQL queries: each query is described by a set of tables to join, by predicates defined on single tables, and by binary join predicates defined on table pairs. We assume that our MPQ algorithm is applied to query templates which are not fully specified: the predicates defined on single tables are placeholders. The selectivity of such a predicate, meaning the average fraction of tuples satisfying the predicate, is unknown to our MPQ algorithm. Hence the selectivity of each predicate placeholder must be represented by a parameter. Our algorithm finds all plans realizing optimal cost tradeoffs for each possible parameter value combination.

We generate the queries for our benchmark randomly. We use the method described by Steinbrunn et al. [15] to produce random queries that join a given number of tables. The number of rows in each table and the selectivity of each predicate is chosen randomly according to that method. We distinguish two classes of queries: chain queries and star queries. For chain queries, the binary join predicates connect query tables in a chain. For star queries, the binary join predicates connect one table (the middle of the "star") to all other query tables. The number of predicates is for both query classes one less than the number of tables.

We describe the plan search space that our algorithm considers. Our algorithm considers all possible orders in which tables can be joined with only one restriction: whenever we have the choice between joining two relations that are connected via a binary join predicate and joining two relations where this is not the case then only joins of the first category are considered. This restriction on the join order is often used in query optimization [14, 15]. In addition to the join orders, our algorithm considers different scan and join operators. For scanning single tables on which a predicate is defined, we consider a full scan and an index-based scan. Which of the two operators is preferable depends on the selectivity of the predicate. If the selectivity is low (few tuples will satisfy the predicate) then the index scan is often preferable. If the predicate is satisfied for most tuples then the full scan is more efficient. We model the selectivity of a predicate defined on a single table by a parameter. The optimal choice for the scan operator therefore depends on the value of that parameter. We consider two join operators: a distributed join and a single-node hash join. For sufficiently large amounts of input data, the distributed join saves execution time. On the other side, the distributed join requires to rent more computational resources from the Cloud provider and is therefore more expensive. Hence we can realize different tradeoffs between execution time and execution fees by selecting between alternative join operators. We implemented our MPQ algorithm in Java 1.7. We used Gurobi 5.6 as linear solver. All experiments were executed on an iMac equipped with an i5-3470S processor with 2.9 GhZ and 16 GB of RAM.

Experimental Results. Figure 5 shows our experimental results. Each data point in that figure corresponds to the median value of 25 randomly generated test cases. We report optimization time, the number of generated query plans (counting plans for the input query and plans for subqueries), and the number of solved linear programs. We generated query templates joining between two and 12 tables and having between one and two parameters.

Optimization time increases in the number of tables. As predicted by our formal analysis in the previous section, optimization time also increases in the number of parameters. Optimization time grows faster in the number of query tables for star queries than for chain queries. The reason is that the number of admissible join orders grows faster in the number of query tables for star queries. Speaking of admissible join orders, we mean join orders that comply with the restriction mentioned before. Optimization time, the number of generated plans, and the number of solved linear programs are all correlated. This is intuitive as the number of generated plans relates to the number of plan comparisons that are required during pruning. The number of linear programs is related to the number of plan comparisons since plan comparisons are realized by solving linear programs. The time required for generating plans and for solving linear programs adds to optimization time.

The query sizes that we consider in our benchmark are typical for query sizes as they appear in standard benchmarks: the queries in the popular TPC-H benchmark join for instance at most eight tables. MPQ takes longer than traditional query optimization. In contrast to traditional query optimization, MPQ takes however place before run time. This makes higher optimization times acceptable.

6. RELATED WORK

Figure 6 shows how multi-objective parametric query optimization relates to prior query optimization variants. The figure shows for each variant the type of cost function c that is associated with each query plan. Arrows point from a more restricted to a more general query optimization vari-

`http://www.gurobi.com/`
`http://www.tpc.org/tpch/`

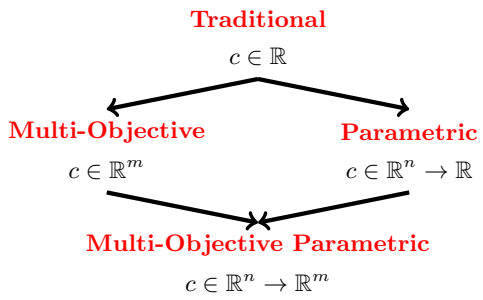


Figure 6: Multi-objective parametric query optimization generalizes the cost models of multi-objective and of parametric query optimization.

ant. Multi-objective query optimization [1, 7, 11, 16, 17] and parametric query optimization [3, 4, 6, 8, 10, 13] both generalize the traditional query optimization model [14]. Multi-objective parametric query optimization generalizes both of the aforementioned variants.

The algorithm that we propose in this paper allows to solve query optimization problems that prior algorithms cannot solve. Algorithms for parametric query optimization are not applicable to MPQ since they cannot handle multiple cost metrics. Algorithms for mult-objective query optimization are not applicable to MPQ since they cannot handle parameters. Note that parameters and cost metrics have a different semantic such that it is not possible to model parameters as cost metrics or vice versa. Intuitively, we want to "cover" the entire parameter space (by finding plans for each possible parameter value combination) while we do not want to cover the entire cost space (plans with higher cost values than necessary are not part of the result plan set).

The algorithm that we describe in this article is based on dynamic programming. It calculates optimal plans for a query by combining optimal plans for its sub-queries. Many query optimization algorithms for traditional query optimization [14], multi-objective query optimization [16, 17], and parametric query optimization [8] use the same dynamic programming scheme. The difference between our algorithm and all prior algorithms lies in the implementation of the pruning function. We use linear programming in the pruning function. Our algorithm shares this property with prior algorithms for parametric query optimization [8]. We support however multiple cost metrics and hence the definition of the pruning function, the type of the used data structures, and the implementation of elementary operations on those data structures differ.

Many algorithms for parametric query optimization are based on the guiding principles of parametric query optimization [5]. They partition the parameter space in a more and more fine-grained manner until a single query plan is optimal in each partition [6, 9]. The condition that allows to verify whether a single query plan is optimal in a given partition is based on the guiding principles. We have shown in Section 4 that the multi-objective analogue of the guiding principles does not hold for MPQ. Hence we cannot use generalizations of the aforementioned decomposition methods for MPQ.

7. CONCLUSION

The traditional query optimization model is outdated. We proposed a generalized problem model that allows to represent multiple plan cost metrics and multiple parameters. We described and analyzed a first algorithm that solves the novel optimization problem.

8. REFERENCES

[1] S. Agarwal, A. Iyer, and A. Panda. Blink and it's done: interactive queries on very large data. In *VLDB*, volume 5, pages 1902–1905, 2012.

[2] A. Bemporad, K. Fukuda, and F. Torrisi. Convexity recognition of the union of polyhedra. *Computational Geometry*, 18(3):141–154, 2001.

[3] P. Bizarro, N. Bruno, and D. DeWitt. Progressive parametric query optimization. *KDE*, 21(4):582–594, 2009.

[4] P. Darera and J. Haritsa. On the production of anorexic plan diagrams. *PVLDB*, 2007.

[5] A. Dey, S. Bhaumik, and J. Haritsa. Efficiently approximating query optimizer plan diagrams. In *VLDB*, pages 1325–1336, 2008.

[6] S. Ganguly. Design and analysis of parametric query optimization algorithms. In *VLDB*, pages 228–238, 1998.

[7] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *SIGMOD*, pages 9–18, 1992.

[8] A. Hulgeri and S. Sudarshan. Parametric query optimization for linear and piecewise linear cost functions. In *VLDB*, pages 167–178, 2002.

[9] A. Hulgeri and S. Sudarshan. AniPQO: almost non-intrusive parametric query optimization for nonlinear cost functions. In *PVLDB*, pages 766–777, 2003.

[10] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric Query Optimization. *VLDBJ*, 6(2):132–151, may 1997.

[11] C. Papadimitriou and M. Yannakakis. Multiobjective query optimization. In *PODS*, pages 52–59, 2001.

[12] H. Park and J. Widom. Query optimization over crowdsourced data. *VLDB*, pages 781–792, 2013.

[13] N. Reddy and J. Haritsa. Analyzing plan diagrams of database query optimizers. *VLDB*, pages 1228–1239, 2005.

[14] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[15] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDBJ*, 6(3):191–208, aug 1997.

[16] I. Trummer and C. Koch. Approximation schemes for many-objective query optimization. In *SIGMOD*, pages 1299–1310, 2014.

[17] I. Trummer and C. Koch. An incremental anytime algorithm for multi-objective query optimization. In *SIGMOD*, pages 1941–1953, 2015.

[18] Z. Xu, Y. C. Tu, and X. Wang. PET: Reducing Database Energy Cost via Query Optimization. *VLDB*, 5(12):1954–1957, 2012.

Technical Perspective: Data Distribution for Fast Joins

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

A model database theory paper is usually assumed to have several key ingredients:

- it should consider a real data management problem that is of interest in practice;
- it should provide a clean and simple formalism that can be followed by theoreticians and practitioners alike;
- it should provide theoretical results that give us insights about the original practical problem.

In your favorite database theory papers you will surely find all these three ingredients. A recent paper that has them as well – and that serves as the basis for the highlights paper that follows – is the PODS 2015 paper by Ameloot, Geck, Ketsman, Neven, and Schwentick that considers single-round multi-way join algorithms in parallel systems. This brief overview explains why this is so, and hopefully convinces you to read the full highlights paper.

The problem.

Large-scale data analytics and massive parallelism are two concepts that go hand-in-hand; hence the problem of efficient evaluation of join queries is one that is actively studied. The challenges are quite different from the usual join processing, as the dominant factor is no longer the I/O, but rather communication cost. The most drastic way to reduce it is to have just a single round of communication: that is, distribute data to servers, let them do their work, and then collect the results to produce the answer to the join query.

Afrati and Ullman's EDBT 2010 paper initiated the study of such multi-join algorithms. A refinement, *Hypercube*, algorithm was proposed in a PODS 2013 paper by Beame, Koutris, and Suciu and then experimentally evaluated. In those algorithms, the network topology is a hypercube. To evaluate a query, one replicates each tuple in several of its nodes and then lets each node perform its computation.

While the hypercube is a rather natural distribution policy, it is certainly not the only one. But can we reason about single-round join evaluation under *arbitrary* distribution policies?

Also, distribution policies are query-dependent. While finding one policy for all scenarios is of course unrealistic, what about a more down-to-earth requirement: if we already know that a policy works for a query Q, perhaps we can use *the same* policy for another query Q', without redistributing data? These are the questions addressed in the paper.

The formalism.

It is very simple and elegant. A network is a set of node names; a distribution policy simply assigns each fact (a tuple in a relation) to a set of nodes. This is the communication round. The query Q is then executed locally at each node. It is *parallel-correct* if such a distributed evaluation gives the result of Q; that is, tuples in the answer to Q are exactly those that are produced locally at network nodes.

Next, if we have two queries Q and Q', and we know that each distribution policy that makes Q parallel-correct does the same for Q', we say that parallel-correctness transfers from Q to Q'. In this case, the work done for Q in terms of looking for the right distribution policy need not be re-done for Q'.

The results, and what they tell us.

This is a theory paper, and the main results are about the complexity of checking parallel-correctness and parallel-transferability. The paper concentrates on the class of *conjunctive queries*, i.e., slightly more general queries that multi-way joins.

Parallel-correctness, under mild assumptions, is Π_2^p-complete. That is, it is a bit harder than NP or coNP, but still well within polynomial space. In practice, this means that checking whether a distribution policy guarantees correctness for all databases can be done in exponential time. Note that this is a static analysis problem (the database is *not* an input), and exponential time is tolerable and in fact the expected best case for conjunctive queries (as their containment is NP-complete).

The paper then shows that the same problems for conjunctive queries with negations requires (modulo some complexity theory assumptions) *double*-exponential time, i.e., is realistically unsolvable, which means that one needs to restrict attention to simple joins.

Finally, parallel-transferability for conjunctive queries is solvable in exponential time (remember, this is a problem about queries, not about data), and importantly it is in NP for many classes of conjunctive queries, likely multi-joins (which hints at the possibility of using efficient NP solvers to address this problem in practice).

In summary, the paper provides an elegant theoretical investigation of a practically important problem, and gives a good set of results that delineate the feasibility boundary.

Data partitioning for single-round multi-join evaluation in massively parallel systems*

Tom J. Ameloot*
Hasselt University & transnational University of Limburg
tom.ameloot@uhasselt.be

Gaetano Geck
TU Dortmund University
gaetano.geck@udo.edu

Bas Ketsman†
Hasselt University & transnational University of Limburg
bas.ketsman@uhasselt.be

Frank Neven
Hasselt University & transnational University of Limburg
frank.neven@uhasselt.be

Thomas Schwentick
TU Dortmund University
thomas.schwentick@udo.edu

ABSTRACT

A dominant cost for query evaluation in modern massively distributed systems is the number of communication rounds. For this reason, there is a growing interest in single-round multiway join algorithms where data is first reshuffled over many servers and then evaluated in a parallel but communication-free way. The reshuffling itself is specified as a distribution policy. We introduce a correctness condition, called *parallel-correctness*, for the evaluation of queries w.r.t. a distribution policy. We provide a semantical characterization for when conjunctive queries (and extensions thereof) are parallel-correct and give matching complexity bounds for the associated decision problem.

Motivated by scenarios for workload optimization, we further consider the problem of parallel-correctness transfer from a query $\mathcal{Q}$ to a query $\mathcal{Q}'$, that is, whether $\mathcal{Q}'$ is parallel-correct for all distribution policies for which $\mathcal{Q}$ is parallel-correct. In this case, $\mathcal{Q}'$ can always be evaluated after $\mathcal{Q}$ *without* repartitioning the data. We provide a semantical characterization for parallel-correctness transfer and provide matching complexity bounds for the associated decision problem for conjunctive queries (and extensions). Finally, we investigate restrictions of queries and families of distribution policies with better complexities, including, for instance, the Hypercube distributions.

1. INTRODUCTION

The background scenario for our work is that of large-scale data analytics where massive parallelism is utilized to answer complex join queries. As, for instance, described by Chu et al. [6], data analytics engines face new kinds of workloads, where multiple large tables are joined, or where the query graph has cycles. Furthermore, recent in-memory systems (like, e.g., [9, 10, 16, 19]) can fit data in main memory by utilizing multitudes of servers. For such systems, performance is no longer dominated by the number of I/O requests to external memory as in traditional systems but by the communication cost for reshuffling data during query execution. When queries need to be evaluated in several rounds, such reshuffling can repartition the whole database and can thus be very expensive. For this reason, it is paramount to reduce the number of evaluation rounds.

While in traditional distributed query evaluation, multi-join queries are computed in several stages over a join tree possibly transferring data over the network at each step, we focus in this paper on query evaluation algorithms that only require *one* round of communication.[1] Such algorithms consist of two phases: a *distribution phase* (where data is repartitioned or reshuffled over the servers) followed by a naive *evaluation phase* where each server contributes to the query answer in isolation by evaluating the query at hand over the local data without any further communication. We refer to such algorithms as *one-round algorithms*. Afrati and Ullman [1] describe an algorithm that computes a multi-join query in a *single* communication round. The algorithm uses a technique that can be traced back to Ganguli, Silberschatz, and Tsur [7] and received quite some attention in the literature. For instance, Beame et al. [4, 5] refined the algorithm, named it *Hypercube*, and showed that it is a communication-optimal algorithm for distributed evaluation, while in a subsequent study, Chu et al. performed an empirical evaluation of Hypercube [6].

Specifically, for a given conjunctive query $\mathcal{Q}$, Hypercube defines a reshuffling of the data over the available servers: the algorithm organizes all the servers in a hypercube and assigns each fact to a subset of points within this cube. The Hypercube reshuffling is thus defined on the granularity of facts and assigns each fact in isolation (that is, independent of the presence or absence of any other facts) to a subset of the servers. This means that the Hypercube reshuffling is independent of the current distribution of the data and can therefore be applied locally at every server. Furthermore, once the data is repartitioned, the target query $\mathcal{Q}$ can be

*The original version of this article was published in PODS 2015 as [2].

*Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

†PhD Fellow of the Research Foundation - Flanders (FWO).

[1]The novel algorithms, introduced, for instance, in [13] or [18] do process multiple joins at once but are targeted towards a sequential setting.

naively evaluated independently on all servers. Hence, Hypercube is a one-round algorithm. We note that the common term Hypercube (algorithm) refers to the described combination of Hypercube distribution/reshuffling followed by naive evaluation of the query at each server. For this reason, the "hypercube" aspect relates exclusively to the way the data is initially reshuffled. Beame et al. [5] obtained that choosing the optimal shape of the hypercube is related to the fractional edge packing of the query graph.

We present a framework for reasoning about the correctness of one-round algorithms for the evaluation of queries under *arbitrary* distribution policies. To target the widest possible range of repartitioning strategies, the initial distribution phase is modeled by a distribution policy that can be *any* mapping from facts to subsets of servers. In particular, this includes any primary horizontal fragmentation of the database as for instance hash partitioning or range partitioning [14].

In this setting, we study two fundamental problems:

Parallel-Correctness: Given a distribution policy and a query, can we be sure that the corresponding one-round algorithm will always compute the query result correctly — no matter the actual data?

Parallel-Correctness Transfer: Given two queries $\mathcal{Q}$ and $\mathcal{Q}'$, can we infer from the fact that $\mathcal{Q}$ is computed correctly under the current distribution policy, that $\mathcal{Q}'$ is computed correctly as well?

Apart from their fundamental nature, the just mentioned problems have practical relevance in settings where a workload of queries has to be evaluated. Recall that the naive one-round Hypercube algorithm requires a reshuffling of the base data for *every* separate query, it therefore makes sense to consider scenarios for which this reshuffling can be avoided. It is in this context that we consider the problem of parallel-correctness transfer. Specifically, parallel-correctness allows to decide whether a query can be evaluated on the current distribution without reshuffling the data, that is, with zero communication cost. Furthermore, we say that parallel-correctness *transfers* from $\mathcal{Q}$ to $\mathcal{Q}'$, denoted $\mathcal{Q} \xrightarrow{\text{pc}} \mathcal{Q}'$, when $\mathcal{Q}'$ is parallel-correct under *every* distribution policy for which $\mathcal{Q}$ is parallel-correct. This implies that $\mathcal{Q}'$ can *always* be evaluated after $\mathcal{Q}$ *without* redistributing the data. Therefore parallel-correctness transfer is particularly relevant in a setting of automatic data partitioning where an optimizer tries to automatically partition the data across multiple nodes to achieve overall optimal performance for a specific workload of queries (see, e.g., [12, 15]). Indeed, when parallel-correctness transfers from a query $\mathcal{Q}$ to a set of queries $\mathcal{S}$, then any distribution policy under which $\mathcal{Q}$ is parallel-correct can be picked to evaluate all queries in $\mathcal{S}$ without reshuffling the data. Of course, parallel-correctness transfer is a very strong notion. In our view, parallel-correctness transfer relates to optimization of data partitioning like query containment relates to traditional query optimization: a relevant fundamental property but perhaps too strong for immediate use.

We focus in this paper on conjunctive queries (possibly extended with union, inequalities and negation) and first consider the complexity of deciding parallel-correctness. The latter problem is equivalent to testing whether the distribution policy *saturates* the query, that is, whether for every *minimal* valuation of the conjunctive query there is a node in the network containing all facts required by that valuation. For various representations of distribution policies, testing parallel-correctness is Π_2^P-complete. These results continue to hold in the presence of union and inequalities. When negation is added, deciding parallel-correctness can no longer be reduced to testing properties of minimal valuations but might involve counterexample databases of exponential size. More specifically, in the presence of negation deciding parallel-correctness is coNEXPTIME-complete. Interestingly, the latter result is related to the new result that query containment for conjunctive queries with negation is coNEXPTIME-complete, as well.

Parallel-correctness transfer can be semantically characterized. In particular, $\mathcal{Q} \xrightarrow{\text{pc}} \mathcal{Q}'$ if and only if $\mathcal{Q}$ *covers* $\mathcal{Q}'$. The latter is a (value-based) containment condition for minimal valuations of $\mathcal{Q}'$ and $\mathcal{Q}$. Deciding transferability of parallel-correctness for conjunctive queries is Π_3^P-complete, again even in the presence of unions and inequalities. We note that the implied exponential time algorithm for parallel-correctness transfer does not rule out practical applicability since the running time is exponential in the size of the queries not in the size of a database. Still, it would be interesting to lower the complexity. Therefore, we consider a further condition that is merely necessary for parallel-correctness transfer, but can be tested more easily, in NP. We refer to this condition as *weak parallel-correctness transfer* and consider two settings where this necessary condition is also sufficient. The first setting applies when $\mathcal{Q}$ is strongly minimal. The second setting considers particular families of distribution policies which generalize the Hypercube distribution families. In particular this means that it is NP-complete to decide whether a query $\mathcal{Q}'$ is parallel-correct under all Hypercube distributions for a query $\mathcal{Q}$.

Outline. In Section 2, we introduce the necessary preliminaries regarding databases, queries, and distribution policies, including Hypercube distributions. In Section 3 and Section 4, we discuss parallel-correctness and parallel-correctness transfer. In Section 5, we consider the setting with lower complexity. We present concluding remarks together with direction for further research in Section 6. Although most of the results hold in the presence of union and inequality, the presentation will focus on CQs most of the time.

2. PRELIMINARIES

2.1 Databases and queries

Throughout the rest of the paper, we assume an infinite domain **dom** and a database scheme consisting of relation names with associated arities. A *(database) instance* I is simply a finite set of facts. A *conjunctive query (CQ)* $\mathcal{Q}$ is an expression of the form

$$H(\mathbf{x}) \leftarrow R_1(\mathbf{y_1}), \ldots, R_m(\mathbf{y_m})$$

where every R_i is a relation name and every $\mathbf{y_i}$ matches the arity of R_i. We require that every variable in $\mathbf{x}$ occurs in some $\mathbf{y}_i$. We refer to the *head atom* $H(\mathbf{x})$ by $head_{\mathcal{Q}}$ and to the set $\{R_1(\mathbf{y_1}), \ldots, R_m(\mathbf{y_m})\}$ by $body_{\mathcal{Q}}$.

We denote by $vars(\mathcal{Q})$ the set of all variables occurring in $\mathcal{Q}$. A *valuation* for a CQ $\mathcal{Q}$ is a total function $V : vars(\mathcal{Q}) \to \mathbf{dom}$. We refer to $V(body_{\mathcal{Q}})$ as the facts *required* by V. A valuation V *satisfies* $\mathcal{Q}$ on instance I if all facts required by

V are in I. In that case, V *derives* the fact $V(head_{\mathcal{Q}})$. The *result* of $\mathcal{Q}$ on instance I, denoted $\mathcal{Q}(I)$, is defined as the set of facts that can be derived by satisfying valuations for $\mathcal{Q}$ on I.

Example 2.1. Let I_e be the example database instance

$$\big\{R(a,b), R(b,a), R(b,c), S(a,a), S(c,a)\big\},$$

and $\mathcal{Q}_e$ be the example CQ

$$H(x_1,x_3) \leftarrow R(x_1,x_2), R(x_2,x_3), S(x_3,x_1).$$

Then $V_1 = \{x_1 \mapsto a, x_2 \mapsto b, x_3 \mapsto a\}$ and $V_2 = \{x_1 \mapsto a, x_2 \mapsto b, x_3 \mapsto c\}$ are the only satisfying valuations. Consequently, $\mathcal{Q}_e(I_e) = \big\{H(a,a), H(a,c)\big\}$. □

We denote the class of all CQs by **CQ**.

2.2 Distribution policies

A *network* $\mathcal{N}$ is a nonempty finite set of node names. A *distribution policy* $\boldsymbol{P} = (U, rfacts_{\boldsymbol{P}})$ for a network $\mathcal{N}$ consists of a universe U and a total function $rfacts_{\boldsymbol{P}}$ that maps each node of $\mathcal{N}$ to a set of facts in $facts\big(U\big)$.[2] Here, $facts\big(U\big)$ denotes the set of all possible facts over U. A node κ is *responsible for a fact* $\boldsymbol{f}$ (under policy $\boldsymbol{P}$) if $\boldsymbol{f} \in rfacts_{\boldsymbol{P}}(\kappa)$. For a distribution policy $\boldsymbol{P}$, an instance I over $\mathcal{D}$ and a $\kappa \in \mathcal{N}$, let $loc\text{-}inst_{\boldsymbol{P},I}(\kappa)$ denote $I \cap rfacts_{\boldsymbol{P}}(\kappa)$, that is, the set of facts in I for which node κ is responsible. We refer to a given instance I as the *global instance* and to $loc\text{-}inst_{\boldsymbol{P},I}(\kappa)$ as the *local instance on node* κ.

The result $[\mathcal{Q}, \boldsymbol{P}](I)$ of the distributed evaluation in one round of a query $\mathcal{Q}$ on an instance I under a distribution policy $\boldsymbol{P}$ is defined as the union of the results of $\mathcal{Q}$ evaluated over every local instance. Formally,

$$[\mathcal{Q}, \boldsymbol{P}](I) \stackrel{\text{def}}{=} \bigcup_{\kappa \in \mathcal{N}} \mathcal{Q}(loc\text{-}inst_{\boldsymbol{P},I}(\kappa)).$$

Example 2.2. Continuing Example 2.1, consider a network $\mathcal{N}_e$ consisting of two nodes $\{\kappa_1, \kappa_2\}$. Let $\boldsymbol{P}_1 = (\{a,b,c\}, rfacts_{\boldsymbol{P}})$ be the distribution policy that assigns all R-facts to both nodes κ_1 and κ_2, and every fact $S(d_1,d_2)$ to node κ_1 when $d_1 = d_2$ and to node κ_2 otherwise. Then,

$$loc\text{-}inst_{\boldsymbol{P}_1,I_e}(\kappa_1) = \big\{R(a,b), R(b,a), R(b,c), S(a,a)\big\},$$

and

$$loc\text{-}inst_{\boldsymbol{P}_1,I_e}(\kappa_2) = \big\{R(a,b), R(b,a), R(b,c), S(c,a)\big\}.$$

Furthermore,

$$[\mathcal{Q}_e, \boldsymbol{P}_1](I_e) = \mathcal{Q}_e\big(loc\text{-}inst_{\boldsymbol{P}_1,I_e}(\kappa_1)\big) \cup \mathcal{Q}_e\big(loc\text{-}inst_{\boldsymbol{P}_1,I_e}(\kappa_2)\big),$$

which is just $\{H(a,b)\} \cup \{H(a,c)\}$.

Next, consider the alternative distribution policy $\boldsymbol{P}_2$ that assigns all R-facts to node κ_1 and all S-facts to node κ_2, then $[\mathcal{Q}_e, \boldsymbol{P}_2](I_e) = \emptyset$. □

Obviously, every primary horizontal fragmentation (see, e.g., [14]) can be modeled as a distribution policy. Consider, for instance, a range partitioning on a relation Customer that assigns tuples to network nodes determined by a threshold on the area code.

[2]We mention that for Hypercube distributions, the view is reversed: facts are assigned to nodes. However, both views are essentially equivalent and we will freely adopt the view that fits best for the purpose at hand.

2.3 Hypercube distributions

A Hypercube distribution distributes the space of all valuations of $\mathcal{Q}$ over the computing servers in an instance independent way through hashing of domain values. Let $\mathcal{Q}$ be a CQ of the following form

$$H(\mathbf{x}) \leftarrow R_1(\mathbf{y_1}), \ldots, R_m(\mathbf{y_m}),$$

and let $|\mathcal{N}| = p$. A Hypercube distribution is parameterized by numbers $p_1, \ldots, p_k$ for which $p = p_1 \times \cdots \times p_k$. Here, k is equal to the number of distinct variables in Q, that is, $|vars(\mathcal{Q})| = k$. For simplicity, assume $vars(\mathcal{Q}) = \{x_1, \ldots, x_k\}$. Still, there can be variables in $vars(\mathcal{Q})$ that do not occur in the head $H(\mathbf{x})$. The distribution organizes the p servers in a hypercube of k dimensions where the size of the i-th dimension is p_i. So, every server corresponds to one point in $\{1, \ldots, p_1\} \times \cdots \times \{1, \ldots, p_k\}$. Furthermore, for $i \in \{1, \ldots, k\}$, let h_i be a hash function mapping each value in **dom** to $\{1, \ldots, p_i\}$.

Then the Hypercube distribution policy $\boldsymbol{P}_H$ assigns each fact to a set of points in the hypercube and is defined as follows: each fact $R_i(a_1, \ldots, a_\ell)$ in the local instance that can be mapped to an atom $R_i(x_{i_1}, \ldots, x_{i_\ell})$ in $\mathcal{Q}$ is sent to every coordinate $(\alpha_1, \ldots, \alpha_k)$ for which $\alpha_{i_j} = h_{i_j}(a_j)$ for all j in $\{1, \ldots, \ell\}$. In particular, this means that when a variable x_r occurs in $R_i(x_{i_1}, \ldots, x_{i_\ell})$ on position j then the r-th dimension of the coordinate has to be $h_r(a_j)$; otherwise, it can be any value in the codomain of h_r.

Example 2.3. Consider query $\mathcal{Q}_e$

$$H(x_1,x_3) \leftarrow R(x_1,x_2), R(x_2,x_3), S(x_3,x_1).$$

from Example 2.1, and a network $\mathcal{N}$ and numbers p_1, p_2, p_3 with $|\mathcal{N}| = p = p_1 \times p_2 \times p_3$. So, every computing node is addressed by a triple (i_1, i_2, i_3) with $i_j \in \{1, \ldots, p_j\}$ for $j \leq 3$. We abuse notation and denote, for instance, by $(i_1, i_2, \star)$ the set of coordinates that match on (i_1, i_2), that is, $\{(i_1, i_2, i_3) \mid i_3 \leq p_3\}$. Then, for any choice of hash functions, $\boldsymbol{P}_H$ assigns each

- $R(d_1, d_2)$ to
  - $\big(h_1(d_1), h_2(d_2), \star\big)$ because of $R(x_1, x_2)$; and
  - $\big(\star, h_2(d_1), h_3(d_2)\big)$ because of $R(x_2, x_3)$;
- $S(d_1, d_2)$ to $\big(h_1(d_2), \star, h_3(d_1)\big)$ because of $S(x_3, x_1)$.

It is not hard to see that, for each instance I,

$$\mathcal{Q}_e(I) = \bigcup_{\kappa \in \mathcal{N}} \mathcal{Q}_e\big(loc\text{-}inst_{\boldsymbol{P},I}(\kappa)\big) = [\mathcal{Q}_e, \boldsymbol{P}_H](I),$$

and thus the one-round algorithm evaluates $\mathcal{Q}_e$ correctly. We will refer to this property as parallel-correctness in the next section. □

3. PARALLEL-CORRECTNESS

In this section, we introduce the notion of parallel-correctness. Informally, it states for a query and a distribution policy that the naive one-round evaluation algorithm yields the correct result, for every possible instance. Specifically, this algorithm first distributes (reshuffles) the data over the computing nodes according to $\boldsymbol{P}$ and then evaluates $\mathcal{Q}$ in a subsequent parallel step at every computing node. Notice that, since $\boldsymbol{P}$ is defined on the granularity of a fact, the

reshuffling does not depend on the current distribution of the data and can be done in parallel as well.

First, we define parallel-correctness w.r.t. a given instance:

Definition 3.1. A query $\mathcal{Q}$ is *parallel-correct on instance I under distribution policy $\boldsymbol{P}$* if $\mathcal{Q}(I) = [\mathcal{Q}, \boldsymbol{P}](I)$.

We note that parallel-correctness is the combination of

- *parallel-soundness*: $[\mathcal{Q}, \boldsymbol{P}](I) \subseteq \mathcal{Q}(I)$, and
- *parallel-completeness*: $\mathcal{Q}(I) \subseteq [\mathcal{Q}, \boldsymbol{P}](I)$.

For monotone queries, like conjunctive queries, parallel-soundness is granted, and therefore parallel-correctness and parallel-soundness coincide. Next, we lift parallel-correctness to all instances:

Definition 3.2. A query $\mathcal{Q}$ is *parallel-correct under distribution policy $\boldsymbol{P} = (U, rfacts_{\boldsymbol{P}})$*, if $\mathcal{Q}$ is parallel-correct on all instances $I \subseteq facts\big(U\big)$.

While Definitions 3.1 and 3.2 are in terms of general queries, in the rest of this section, we only consider (extensions of) conjunctive queries.

3.1 Conjunctive queries

We first focus on a characterization of parallel-correctness. It is easy to see that a CQ $\mathcal{Q}$ is parallel-correct under distribution policy $\boldsymbol{P} = (U, rfacts_{\boldsymbol{P}})$ if, for each valuation for $\mathcal{Q}$, the required facts meet at some node. That is, if the following condition holds:

$$\text{For every valuation } V \text{ for } \mathcal{Q} \text{ over } U, \text{ there is a node } \kappa \in \mathcal{N} \text{ such that } V(body_{\mathcal{Q}}) \subseteq rfacts_{\boldsymbol{P}}(\kappa). \qquad (PC_0)$$

Even though Condition (PC_0) is sufficient for parallel-correctness, it is not necessary as the following example shows.

Example 3.3. We consider the CQ $\mathcal{Q}$,

$$H(x, z) \leftarrow R(x, y), R(y, z), R(x, x),$$

and the valuation $V = \{x \mapsto a, y \mapsto b, z \mapsto a\}$. Let further $\mathcal{N} = \{\kappa_1, \kappa_2\}$ and let $\boldsymbol{P}$ distribute every fact except $R(a, b)$ onto node κ_1 and every fact except $R(b, a)$ onto node κ_2. Since $R(a, b)$ and $R(b, a)$ do not meet under $\boldsymbol{P}$, valuation V witnesses the failure of Condition (PC_0) for $\boldsymbol{P}$ and $\mathcal{Q}$.

However, $\mathcal{Q}$ is parallel-correct under $\boldsymbol{P}$. Indeed, every valuation that derives a fact $\boldsymbol{f}$ with the help of the facts $R(a, b)$ and $R(b, a)$, also requires the fact $R(a, a)$ (or $R(b, b)$). But then, $R(a, a)$ (or $R(b, b)$) alone is sufficient to derive $\boldsymbol{f}$ by mapping all variables to a (or b). Therefore, if $\boldsymbol{f} \in \mathcal{Q}(I)$, for some instance I, then $\boldsymbol{f} \in [\mathcal{Q}, \boldsymbol{P}](I)$ and thus $\mathcal{Q}$ is parallel-correct under $\boldsymbol{P}$. □

It turns out that for a semantical characterization only such valuations have to be considered that are minimal in the following sense:

Definition 3.4. Let $\mathcal{Q}$ be a CQ. A valuation V for $\mathcal{Q}$ is *minimal* for $\mathcal{Q}$ if there does *not* exist a valuation V' for $\mathcal{Q}$ that derives the same head fact with a strict subset of body facts, that is, such that $V(body_{\mathcal{Q}}) \subsetneq V'(body_{\mathcal{Q}})$ and $V(head_{\mathcal{Q}}) = V'(head_{\mathcal{Q}})$.

Example 3.5. For a simple example of a minimal valuation and a non-minimal valuation, consider the CQ $\mathcal{Q}$,

$$H(x, z) \leftarrow R(x, y), R(y, z), R(x, x).$$

Both $V_1 = \{x \mapsto a, y \mapsto b, z \mapsto a\}$ and $V_2 = \{x \mapsto a, y \mapsto a, z \mapsto a\}$ are valuations for $\mathcal{Q}$. Notice that both valuations agree on the head variables of $\mathcal{Q}$, but they require different sets of facts. In particular, for V_1 to be satisfying on I, instance I must contain the facts $R(a, b), R(b, a)$, and $R(a, a)$, while V_2 only requires I to contain $R(a, a)$. This observation implies that V_1 is not minimal for $\mathcal{Q}$. Further, since V_2 requires only one fact for $\mathcal{Q}$, valuation V_2 is minimal for $\mathcal{Q}$. □

It turns out that it suffices to restrict valuations to minimal valuations in Condition (PC_0) to get a sufficient *and* necessary condition for parallel-correctness.

Proposition 3.6. *Let $\mathcal{Q}$ be a CQ. Then $\mathcal{Q}$ is parallel-correct under distribution policy $\boldsymbol{P}$ if and only if the following holds:*

$$\text{For every minimal valuation } V \text{ for } \mathcal{Q} \text{ over } U, \text{ there is a node } \kappa \in \mathcal{N} \text{ such that } V(body_{\mathcal{Q}}) \subseteq rfacts_{\boldsymbol{P}}(\kappa). \qquad (PC_1)$$

We emphasize that the word *minimal* is the only difference between Conditions (PC_0) and (PC_1).[3] The latter conditions are so fundamental when reasoning over parallel-correctness that they deserve their own terminology:

Definition 3.7. For a CQ $\mathcal{Q}$ and a distribution policy $\boldsymbol{P}$:

- $\boldsymbol{P}$ *saturates* $\mathcal{Q}$ if they fulfill Condition (PC_1); and,
- $\boldsymbol{P}$ *strongly saturates* $\mathcal{Q}$ if they fulfill Condition (PC_0).

Every Hypercube distribution $\boldsymbol{P}_H$ for a conjunctive query $\mathcal{Q}$ strongly saturates $\mathcal{Q}$. Indeed, consider Example 2.3. Then, for every valuation V, all facts

$$R\big(V(x_1), V(x_2)\big), R\big(V(x_2), V(x_3)\big), S\big(V(x_3), V(x_1)\big)$$

meet at node $\big(h_1(V(x_1)), h_2(V(x_2)), h_3(V(x_3))\big)$. Therefore, $\mathcal{Q}$ is parallel-correct under $\boldsymbol{P}_H$.

The quantifier structure in Condition (PC_1) hints at a Π_2^p upper bound for the complexity of testing parallel-correctness.[4] Of course, the exact complexity can not be judged without having a bound on the number of nodes κ and the complexity of the test $V(body_{\mathcal{Q}}) \subseteq rfacts_{\boldsymbol{P}}(\kappa)$. The largest classes of distribution policies for which we established the Π_2^p upper bound, are gathered in the set $\mathfrak{P}_{\text{npoly}}$ that contains classes $\mathcal{P}$ of distribution policies, for which each policy comes with an algorithm $\mathcal{A}$ and a bound n on the representation size of nodes in the network, respectively, such that whether a node κ is responsible for a fact $\boldsymbol{f}$ is decided by $\mathcal{A}$ *nondeterministically* in time $\mathcal{O}(n^k)$, for some k that depends only on $\mathcal{P}$.

It turns out that the problem of testing parallel-correctness is also Π_2^p-hard, even for the simple class $\mathcal{P}_{\text{fin}}$ of distribution policies, for which all pairs $(\kappa, \boldsymbol{f})$ of a node and a fact are

[3] We mention that Conditions (PC_0) were named (C_0) and (C_1), respectively, in [2].

[4] This holds, even if one takes into account that testing minimality of V requires an additional existential quantification of a valuation V' that might serve as a witness, in case V is not minimal.

explicitly enumerated. Thus, in a sense, Condition (PC_1) can essentially not be simplified.

To state the results more formally, we define the following two algorithmic problems.

PCI($\mathbf{CQ}, \mathcal{P}$)	
Input:	$\mathcal{Q} \in \mathbf{CQ}$, $\boldsymbol{P} \in \mathcal{P}$, instance I
Question:	Is $\mathcal{Q}$ parallel-correct on I under $\boldsymbol{P}$?

PC($\mathbf{CQ}, \mathcal{P}$)	
Input:	$\mathcal{Q} \in \mathbf{CQ}$, $\boldsymbol{P} \in \mathcal{P}$
Question:	Is $\mathcal{Q}$ parallel-correct under $\boldsymbol{P}$?

Theorem 3.8. *Problems* PC($\boldsymbol{CQ}, \mathcal{P}$) *and* PCI($\boldsymbol{CQ}, \mathcal{P}$) *are* Π_2^p*-complete, for every policy class* $\mathcal{P} \in \{\mathcal{P}_{fin}\} \cup \mathfrak{P}_{npoly}$.

The upper bounds follow from the characterization in Proposition 3.6 and the fact that pairs $(\kappa, \boldsymbol{f})$ can be tested in NP.

We note that Proposition 3.6 continues to hold true in the presence of union and inequalities (under a suitable definition of minimal valuation for unions of CQs) leading to the same complexity bounds as stated in Theorem 3.8 [8].

3.2 Conjunctive queries with negation

In this section, we consider conjunctive queries with negation. Specifically, queries can be of the form

$$H(\mathbf{x}) \leftarrow R_1(\mathbf{y_1}), \ldots, R_m(\mathbf{y_m}), \neg S_1(\mathbf{z_1}), \ldots, \neg S_n(\mathbf{z_n}).$$

To ensure safety, we require that every variable in $\mathbf{x}$ occurs in some $\mathbf{y}_i$ or $\mathbf{z}_j$, and that every variable occurring in a negated atom has to occur in a positive atom as well. A valuation V now derives a fact $H\big(V(\mathbf{x})\big)$ on an instance I if every positive atom $R_i\big(V(\mathbf{y_i})\big)$ occurs in I while none of the negative atoms $S_j\big(V(\mathbf{z_j})\big)$ do. We refer to the class of conjunctive queries with negation as $\mathbf{CQ}^\neg$.

We note that, as queries in $\mathbf{CQ}^\neg$ are no longer monotone, parallel-soundness is no longer guaranteed and thus parallel-correctness need not coincide with parallel-soundness.

We illustrate through an example that in the case of conjunctive queries *with negation*, the parallel-correctness problem becomes much more involved, since it might involve counterexample databases of exponential size. We emphasize that this exponential explosion can only occur if, as in our framework, the arity of the relations in the database schema are not a-priori bounded by some constant.

Example 3.9. Let $\mathcal{Q}$ be the following conjunctive query with negation:

$$\begin{aligned} H() \quad \leftarrow \quad & \texttt{Val}(w_0, w_0), \texttt{Val}(w_1, w_1), \neg\texttt{Val}(w_0, w_1), \\ & \texttt{Val}(x_1, x_1), \ldots, \texttt{Val}(x_n, x_n), \neg\texttt{Rel}(x_1, \ldots, x_n). \end{aligned}$$

Let $\boldsymbol{P}$ be the policy defined over universe $U = \{0, 1\}$ and two-node network $\{\kappa_1, \kappa_2\}$, which distributes all facts except $\texttt{Rel}(0, \ldots, 0)$ to node κ_1 and only fact $\texttt{Rel}(0, \ldots, 0)$ to node κ_2.

Query $\mathcal{Q}$ is not parallel-sound under policy $\boldsymbol{P}$, as witnessed by the counter-example $I \stackrel{\text{def}}{=} \{\texttt{Val}(0, 0), \texttt{Val}(1, 1)\} \cup \{\texttt{Rel}(a_1, \ldots, a_n) \mid (a_1, \ldots, a_n) \in \{0, 1\}^n\}$. Indeed, $\mathcal{Q}(I) = \emptyset$ but $\mathcal{Q}(\textit{loc-inst}_{\boldsymbol{P},I}(\kappa_1)) \neq \emptyset$, as witnessed by the valuation that maps all variables to 0.

However, I has $2^n + 2$ facts and is a counter-example of minimal size as can easily be shown as follows. First, it is impossible that $\mathcal{Q}(I^*) \neq \emptyset$ and $\mathcal{Q}(\textit{loc-inst}_{\boldsymbol{P},I^*}(\kappa_1)) = \emptyset$, for any I^*, since $\texttt{Rel}(0, \ldots, 0)$ is the only fact that can be missing at node κ_1, and $\mathcal{Q}$ is antimonotonic with respect to $\texttt{Rel}$. On the other hand, if $\mathcal{Q}(\textit{loc-inst}_{\boldsymbol{P},I^*}(\kappa_1)) \neq \emptyset$, then the literals $\texttt{Val}(w_0, w_0), \texttt{Val}(w_1, w_1)$, and $\neg\texttt{Val}(w_0, w_1)$ ensure that there are at least two different data values (and thus 0 and 1) in I^*. But then $\mathcal{Q}(I^*) = \emptyset$ can only hold if all 2^n n-tuples over $\{0, 1\}$ are in I^*. □

Although this example requires an exponential size counter-example, in this particular case, the existence of the counter-example is easy to conclude. However, the following result shows that, in general, there is essentially no better algorithm than guessing an exponential size counter-example.

Theorem 3.10. *[8] For every class* $\mathcal{P} \in \mathfrak{P}_{npoly}$ *of distribution policies, the following problems are coNEXPTIME-complete.*

- Parallel-Sound($\boldsymbol{UCQ}^\neg, \mathcal{P}$)
- Parallel-Complete($\boldsymbol{UCQ}^\neg, \mathcal{P}$)
- Parallel-Correct($\boldsymbol{UCQ}^\neg, \mathcal{P}$)

The result and, in particular, the lower bound even holds if $\mathfrak{P}_{\text{npoly}}$ is replaced by the corresponding $\mathfrak{P}_{\text{poly}}$, where the decision algorithm for pairs $(\kappa, \boldsymbol{f})$ is deterministic and in polynomial time.

The proof of the lower bounds comes along an unexpected route and exhibits a reduction from query containment for $\mathbf{CQ}^\neg$ to parallel-correctness for $\mathbf{CQ}^\neg$. Specifically, query containment asks the question whether, given two queries $\mathcal{Q}$ and $\mathcal{Q}'$, it holds that $\mathcal{Q}(I) \subseteq \mathcal{Q}'(I)$ for all instances I. The latter is denoted by $\mathcal{Q} \subseteq \mathcal{Q}'$. It is shown in [8] that query containment for $\mathbf{CQ}^\neg$ is coNEXPTIME-complete, implying coNEXPTIME-hardness for parallel-correctness as well. The result regarding containment of $\mathbf{CQ}^\neg$ answers the observation in [11] that the Π_2^p-completeness result for query containment for $\mathbf{CQ}^\neg$ mentioned in [17] only holds for fixed database schemas (or a fixed arity bound, for that matter).

4. PARALLEL-CORRECTNESS TRANSFER

As mentioned in the introduction, the one-round Hypercube algorithm requires a reshuffling of the data before the evaluation of a new query. In the context of multiple query evaluation, where an optimizer tries to automatically partition the base data across multiple nodes to achieve overall optimal performance for a specific workload (see, e.g., [12, 15]), it makes sense to consider scenarios in which such reshuffling can be avoided. To this end, *parallel-correctness transfer* was introduced in [2] which states that a subsequent query $\mathcal{Q}'$ can always be evaluated over a distribution for which a query $\mathcal{Q}$ is parallel-correct.

Definition 4.1. For two queries $\mathcal{Q}$ and $\mathcal{Q}'$ over the same input schema, *parallel-correctness transfers from* $\mathcal{Q}$ *to* $\mathcal{Q}'$ if $\mathcal{Q}'$ is parallel-correct under every distribution policy for which $\mathcal{Q}$ is parallel-correct. In this case, we write $\mathcal{Q} \xrightarrow{\text{pc}} \mathcal{Q}'$.

Example 4.2. We illustrate parallel-correctness transfer with the help of the following example queries:

$$\begin{aligned} &\mathcal{Q}_1 : H() \leftarrow S(x), R(x, x), T(x). \\ &\mathcal{Q}_2 : H() \leftarrow R(x, x), T(x). \\ &\mathcal{Q}_3 : H() \leftarrow S(x), R(x, y), T(y). \\ &\mathcal{Q}_4 : H() \leftarrow R(x, y), T(y). \end{aligned}$$

Figure 1 (a) shows how these queries relate with respect to parallel-correctness transfer. As an example, $\mathcal{Q}_3 \xrightarrow{\text{pc}} \mathcal{Q}_1$. As Figure 1 (b) illustrates, this relationships is entirely orthogonal to query containment. Indeed, there are examples where parallel-correctness transfer and query containment coincide ($\mathcal{Q}_3$ vs. $\mathcal{Q}_4$), where they hold in opposite directions ($\mathcal{Q}_4$ vs. $\mathcal{Q}_2$) and where one but not the other holds ($\mathcal{Q}_3$ vs. $\mathcal{Q}_2$ and $\mathcal{Q}_1$ vs. $\mathcal{Q}_4$, respectively). □

It turns out that, just like parallel-correctness, parallel-correctness transfer can be characterized in terms of minimal valuations. For this, we need the following notion:

Definition 4.3. For two CQs $\mathcal{Q}$ and $\mathcal{Q}'$, we say that $\mathcal{Q}$ *covers* $\mathcal{Q}'$ if the following holds:

> for every minimal valuation V' for $\mathcal{Q}'$, there is a minimal valuation V for $\mathcal{Q}$, such that $V'(body_{\mathcal{Q}'}) \subseteq V(body_{\mathcal{Q}})$.

Proposition 4.4. *For two CQs $\mathcal{Q}$ and $\mathcal{Q}'$, parallel-correctness transfers from $\mathcal{Q}$ to $\mathcal{Q}'$ if and only if $\mathcal{Q}$ covers $\mathcal{Q}'$.*

Proposition 4.4, allows us to pinpoint the complexity of parallel-correctness transferability. For a formal statement we define the following algorithmic problem:

PC-TRANS (**CQ**)	
Input:	Queries $\mathcal{Q}$ and $\mathcal{Q}'$ from **CQ**
Question:	Does parallel-correctness transfer from $\mathcal{Q}$ to $\mathcal{Q}'$?

When the defining condition of "covers" is spelled out by rewriting "minimal valuations" one gets a characterization with a Π_3-structure. Again, it can be shown that this is essentially optimal.

Theorem 4.5. *Problem* PC-TRANS($\boldsymbol{CQ}$) *is Π_3^p-complete.*

The upper bounds follow directly from the characterization in Proposition 4.4, implying that these characterizations are essentially optimal. We note that the same complexity bounds continue to hold in the presence of inequalities and for unions of conjunctive queries [3].

5. LOWERING COMPLEXITY

In static analysis of conjunctive queries, one is used to face NP-complete algorithmic problems, most prominently, containment testing. Since queries are often small, especially compared with the data, NP-algorithms might still be helpful in query optimization. We saw in the previous two sections that the complexity of parallel-correctness and parallel-correctness transfer are higher than that, namely Π_2^p-complete and Π_3^p-complete, respectively, even without union, inequalities and negation.

In this section, we consider two settings, in which, for both problems, the complexity drops to the "usual" NP-complete level. In both cases, the complexity reduction is based on a simpler condition for parallel-correctness transfer which can be tested in NP. We state this condition next.

5.1 A necessary condition for pc-transfer

We use the following two additional notions, the first of which is the simpler condition for parallel-correctness transfer (in restricted settings), and the second characterizes the first, as we will see soon.

Definition 5.1. For CQs $\mathcal{Q}, \mathcal{Q}'$ and a distribution policy $\boldsymbol{P}$:

- parallel correctness *weakly transfers* from a CQ $\mathcal{Q}$ to $\mathcal{Q}'$, if $\mathcal{Q}'$ is parallel-correct under every policy $\boldsymbol{P}$ that strongly saturates $\mathcal{Q}$;[5] and
- $\mathcal{Q}$ *weakly covers* $\mathcal{Q}'$, if there are mappings ρ and θ such that
  - ρ maps the variables of $\mathcal{Q}$ to some variables (ρ is a *substitution*),
  - θ maps the variables of $\mathcal{Q}'$ to variables of $\mathcal{Q}'$ such that $head_{\theta(\mathcal{Q}')} = head_{\mathcal{Q}'}$ and $body_{\theta(\mathcal{Q}')} \subseteq body_{\mathcal{Q}'}$ (θ is a *simplification*),
  - and $body_{\theta(\mathcal{Q}')} \subseteq body_{\rho(\mathcal{Q})}$.

Example 5.2. We consider the queries $\mathcal{Q}$

$$H(w) \leftarrow R(u', u), R(u, v), R(v, w), R(u, w)$$

and $\mathcal{Q}'$

$$H'(y) \leftarrow R(x', x), R(x, x), R(x, y), R(y, z).$$

Then $\mathcal{Q}$ weakly covers $\mathcal{Q}'$. Indeed, the substitution ρ can map u, u' to x, v to y and w to z, and θ can map x' to x and leave the other variables alone. Since $head_{\theta(\mathcal{Q}')} = head_{\mathcal{Q}'}$, and $body_{\theta(\mathcal{Q}')} = \{R(x,x), R(x,y), R(y,z)\} \subseteq body_{\mathcal{Q}'}$, θ is indeed a simplification and, furthermore,
$body_{\theta(\mathcal{Q}')} \subseteq \{R(x,x), R(x,y), R(y,z), R(x,z)\} = body_{\rho(\mathcal{Q})}$.
However, we show that parallel-correctness does *not* transfer from $\mathcal{Q}$ to $\mathcal{Q}'$. Towards a contradiction, we assume that parallel-correctness *does* transfer. We consider the valuation V' mapping the variables x, x', z to a and y to b for some $a \neq b$. Then, $V'(body_{\mathcal{Q}'}) = \{R(a,a), R(a,b), R(b,a)\}$, V' derives $H'(b)$, and V' is minimal. Thanks to Proposition 4.4, there must be a minimal valuation V for $\mathcal{Q}$ such that

$$\{R(a,a), R(a,b), R(b,a)\} \subseteq V(body_{\mathcal{Q}}) \qquad (\dagger)$$

holds. We distinguish two cases.

Case 1 $V(w) = a$. Thus V derives $H(a)$. However, the valuation that maps all variables to a also derives $H(a)$ and shows that V is not minimal.

Case 2 $V(w) = b$. Thus V derives $H(b)$. However, the valuation that maps all other variables to a also derives $H(b)$ and only requires the facts $R(a,a)$ and $R(a,b)$. Therefore, V is not minimal.

Hence, no minimal valuation V for $\mathcal{Q}$ exists satisfying (†), the desired contradiction. Therefore, $\mathcal{Q} \xrightarrow{\text{pc}} \mathcal{Q}'$.

On the other hand, it follows from Proposition 5.4 that parallel-correctness *weakly* transfers from $\mathcal{Q}$ to $\mathcal{Q}'$. □

We note that, as the naming already suggests, if parallel-correctness transfers from $\mathcal{Q}$ to $\mathcal{Q}'$ it also weakly transfers, and if $\mathcal{Q}$ covers $\mathcal{Q}'$ it also weakly covers it.

Although the definition of "weakly covers" seems quite involved, it can be tested in NP.

Proposition 5.3. *The decision problem, whether a given CQ $\mathcal{Q}$ weakly covers a given CQ $\mathcal{Q}'$ is* NP*-complete.*

[5] Recall that the notion of strong saturation is introduced in Definition 3.7.

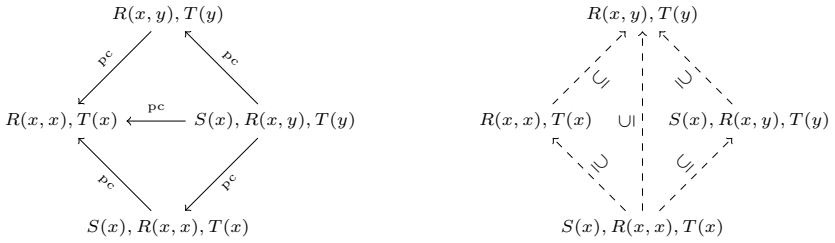


Figure 1: Relationship between the queries of Example 4.2 with respect to (a) parallel-correctness transfer and (b) query containment.

It turns out that "weak cover" characterizes "weak transfer" and thus the following counterpart of Proposition 4.4 holds:

Proposition 5.4. *For CQs $\mathcal{Q}$ and $\mathcal{Q}'$, parallel correctness weakly transfers from $\mathcal{Q}$ to $\mathcal{Q}'$ if and only if $\mathcal{Q}$ weakly covers $\mathcal{Q}'$.*

Thus, if in some setting, weak transferability and transferability coincide, Propositions 5.3 and 5.4 yield an NP upper bound for parallel-correctness transfer. We present two such settings in the following two subsections.

5.2 Strongly minimal queries

Definition 5.5. A CQ query is *strongly minimal* if all its valuations are minimal.

We write $\mathbf{CQ}[sm]$ for the class of strongly minimal CQs.

Although strong minimality is a non-trivial notion (with a coNP-complete decision problem), there are some very common examples like full queries (where all variables occur in the head) and queries without self-joins (where every relation name occurs at most once), and further kinds of queries like

$$H(x_1, x_2) \leftarrow R(x_1, x_3), R(x_2, x_3), S(x_3, x_2).$$

For strongly minimal conjunctive queries, transfer and weak transfer coincide and therefore the following holds.

Theorem 5.6. pc-trans($\boldsymbol{CQ}[sm], \boldsymbol{CQ}$) *is* NP*-complete.*

Also the complexity of parallel-correctness drops for strongly minimal queries, if the representation of the distribution policy allows to figure out in polynomial time whether there is a node that is responsible for a given set of facts. In particular, the following holds:

Theorem 5.7. *For policy class $\mathcal{P} \in \mathfrak{P}_{poly}$,* PCI($\boldsymbol{CQ}[sm], \mathcal{P}$) *and* PC($\boldsymbol{CQ}[sm], \mathcal{P}$) *are in* coNP.

5.3 Feasible Families of Distribution Policies

Parallel-correctness transfer can be seen as a generalization of parallel-correctness. In both cases, the goal is to decide whether a query can be correctly evaluated by evaluating it locally at each node. However, for parallel-correctness *transfer*, the question whether $\mathcal{Q}'$ is parallel-correct is not asked for a particular distribution policy but for the *family*[6] of those distribution policies for which $\mathcal{Q}$ is parallel-correct, which by Proposition 3.6 is just the family of all distribution policies that saturate $\mathcal{Q}$.

[6]A family of distribution policies is just a set of distribution policies.

Definition 5.8. A query is *parallel-correct for a family $\mathcal{F}$* of distribution policies if it is parallel-correct under every distribution policy from $\mathcal{F}$.

We next define a criterion for families of distribution policies which guarantees that parallel-correctness can be tested in NP.

For an instance I, a distribution policy $\boldsymbol{P}$ is called $(\mathcal{Q}, I)$-*scattered* if, for each node κ, there is a valuation V for $\mathcal{Q}$ such that $\mathit{loc\text{-}inst}_{\boldsymbol{P},I}(\kappa) \subseteq V(\mathit{body}_{\mathcal{Q}})$. Intuitively, a $(\mathcal{Q}, I)$-scattered policy ensures that facts are sufficiently spread out such that every network node only contains a subset of the facts related to one valuation. We call a family $\mathcal{F}$ of distribution policies $\mathcal{Q}$-*scattered* if $\mathcal{F}$ contains a $(\mathcal{Q}, I)$-scattered policy, for every instance I. A $\mathcal{Q}$-scattered family of distribution policies that strongly saturate $\mathcal{Q}$ is called a $\mathcal{Q}$-*family*.

It turns out that parallel-correctness with respect to $\mathcal{Q}$-families can be characterized in terms of weak coverability,

Proposition 5.9. *Let $\mathcal{Q}$ be a CQ and let $\mathcal{F}$ be a $\mathcal{Q}$-family. Then, for every CQ $\mathcal{Q}'$, query $\mathcal{Q}'$ is parallel correct for $\mathcal{F}$ if and only if $\mathcal{Q}$ weakly covers $\mathcal{Q}'$.*

And thus, Proposition 5.3 yields another NP-result. Indeed, the following can be shown:

Theorem 5.10. *It is* NP*-complete to decide, for given CQs $\mathcal{Q}$ and $\mathcal{Q}'$, whether $\mathcal{Q}'$ is parallel-correct for $\mathcal{Q}$-families of distribution policies.*

For a CQ $\mathcal{Q}$, let $\mathcal{H}_{\mathcal{Q}}$ denote the family

$$\{\boldsymbol{P}_H \mid H \text{ is a hypercube for } \mathcal{Q}\}$$

of distribution policies. Then the next lemma specifies that Hypercube distributions for a CQ $\mathcal{Q}$ form a $\mathcal{Q}$-family.

Lemma 5.11. *Let $\mathcal{Q}$ be a CQ. Then $\mathcal{H}_{\mathcal{Q}}$ is a $\mathcal{Q}$-family.*

As a corollary, we now have:

Corollary 5.12. *It is* NP*-complete to decide, for given conjunctive queries $\mathcal{Q}, \mathcal{Q}'$, whether $\mathcal{Q}'$ is parallel-correct for $\mathcal{H}_{\mathcal{Q}}$.*

6. DISCUSSION

Parallel-correctness serves as a framework for studying correctness and implications of data partitioning in the context of one-round query evaluation algorithms. A main insight of the work up to now is that testing for parallel-correctness as well as the related problem of parallel-correctness transfer reduces to reasoning about minimal valuations in the context of conjunctive queries (even in the presence of union and inequalities) but becomes considerably more involved when negation is allowed.

There are many questions left unexplored and we therefore highlight possible directions for further research.

From a foundational perspective, it would be interesting to explore the decidability boundary for parallel-correctness and transfer when considering more expressive query languages or even other data models. Obviously, the problems become undecidable when considering first-order logic, but one could consider monotone languages or for instance guarded fragment queries. At the same time, it would be interesting to find settings that render the problems tractable, for instance, by restricting the class of queries or by limiting to certain classes of distribution policies.

Parallel-correctness transfer is a rather strong notion as it requires that a query $\mathcal{Q}'$ is parallel-correct for *every* distribution policy for which another query $\mathcal{Q}$ is parallel-correct. As a consequence, query $\mathcal{Q}'$ can always be executed after $\mathcal{Q}$ without reshuffling of the data. From a practical perspective, however, it could be interesting to determine, given $\mathcal{Q}$ and $\mathcal{Q}'$, whether there is at least one distribution policy under which both queries are correct. Other questions concern the least costly way to migrate from one distribution to another. As an example, assume a distribution $\boldsymbol{P}$ on which $\mathcal{Q}$ is parallel-correct but $\mathcal{Q}'$ is not. Find a distribution $\boldsymbol{P}'$ under which $\mathcal{Q}'$ is parallel-correct and that minimizes the cost to migrate from $\boldsymbol{P}$ to $\boldsymbol{P}'$. Similar questions can be considered for a workload of queries.

Even though the naive one-round evaluation model considered in this paper suffices for Hypercube, it is rather restrictive. Other possibilities are to consider more complex aggregator functions than union and to allow for a different query than the original one to be executed at computing nodes. Furthermore, it could be interesting to generalize the framework beyond one-round algorithms, that is, towards evaluation algorithms that comprise of several rounds.

Acknowledgments

We thank Serge Abiteboul, Luc Segoufin, Cristina Sirangelo, Dan Suciu, and Thomas Zeume for helpful remarks and Jan-Eric Lenssen for careful proof reading.

7. REFERENCES

[1] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT 2010, 13th International Conference on Extending Database Technology*, pages 99–110, 2010.

[2] T. J. Ameloot, G. Geck, B. Ketsman, F. Neven, and T. Schwentick. Parallel-correctness and transferability for conjunctive queries. In *Proceedings of the 34th ACM Symposium on Principles of Database Systems, PODS 2015*, pages 47–58, 2015.

[3] T. J. Ameloot, G. Geck, B. Ketsman, F. Neven, and T. Schwentick. Parallel-correctness and transferability for conjunctive queries. Invited Journal Submission, 2015.

[4] P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. In *Proceedings of the 32nd Symposium on Principles of Database Systems*, PODS '13, pages 273–284, 2013.

[5] P. Beame, P. Koutris, and D. Suciu. Skew in parallel query processing. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '14, pages 212–223, 2014.

[6] S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 63–78, 2015.

[7] S. Ganguly, A. Silberschatz, and S. Tsur. Parallel bottom-up processing of datalog queries. *J. Log. Program.*, 14(1&2):101–126, 1992.

[8] G. Geck, B. Ketsman, F. Neven, and T. Schwentick. Parallel-correctness and containment for conjunctive queries with union and negation. In *International Conference on Database Theory*, pages 9:1–9:17, 2016.

[9] D. Halperin, V. Teixeira de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, S. Xu, M. Balazinska, B. Howe, and D. Suciu. Demonstration of the myria big data management service. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 881–884, 2014.

[10] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3(1-2):330–339, Sept. 2010.

[11] M. Mugnier, G. Simonet, and M. Thomazo. On the complexity of entailment in existential conjunctive first-order logic with atomic negation. *Inf. Comput.*, 215:8–31, 2012.

[12] R. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1137–1148, 2011.

[13] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012*, pages 37–48, 2012.

[14] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[15] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 558–569, 2002.

[16] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed sql database that scales. *Proc. VLDB Endow.*, 6(11):1068–1079, Aug. 2013.

[17] J. D. Ullman. Information integration using logical views. *Theor. Comput. Sci.*, 239(2):189–210, 2000.

[18] T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *Proc. 17th International Conference on Database Theory (ICDT)*, pages 96–106, 2014.

[19] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 13–24, 2013.

Technical Perspective: Taming Hardware Skew as Parallel DBMSs Scale Out

David J. DeWitt
Microsoft Corporation
Madison WI, U.S.A.

For almost 40 years now, relational database management systems have successfully used data parallelism to speed up the evaluation of large queries. Here, by "data parallelism" we mean taking one operation (for example, a "join" or an "aggregation") and spreading it over multiple machines, each operating on a part of the data. In general this approach works spectacularly well, yielding almost linear speedups over a wide variety of workloads. However, like any form of parallelism, data-parallel relational query processing is vulnerable to "skew." The database literature is full of work dealing with the skew that arises when one node in a parallel system is allocated more work than the average.

The following paper, by Li, Naughton, and Nehme, is interesting in that it deals with another kind of skew, one that has received much less attention: "hardware skew," that is, skew that arises because the processing units in a parallel system are not all of equal power. Such skew can arise in several ways – for example, a parallel system could be constructed "on the fly" by allocating available nodes in a cloud, or a company could upgrade an on-premises system with the addition of new nodes that are of a different generation and class of hardware than the existing ones. If the DBMS is oblivious to the fact that the underlying system is not uniform, the result will be the same as that achieved if the system were constructed entirely of the slowest nodes in the system.

If all the nodes in the system are equally "balanced" the solution is simple – if one node is 1/2 as fast as the average, give that node 1/2 the average work, and you are set. Unfortunately, in practice, things are not that simple. One node may have a faster CPU but the same I/O performance, or vice-versa; or nodes may have differing amounts of memory or network bandwidth. In such cases simple proportional allocation of work will be suboptimal. The situation is further complicated by the fact that different queries make different demands on the system with respect to CPU, memory, network, and disk; in fact, different stages of a single query can make very different demands.

This, finally, is the situation addressed by the paper, "Resource Bricolage for Parallel DBMSs on Heterogeneous Clusters." The authors make use of techniques for cost estimation growing out of the query optimization and query running time prediction literature; they combine these techniques with a linear programming model that chooses an optimal allocation for a given query on a given system. They demonstrate through an analytic model as well as experiments with an implementation that their proposed solution dominates simpler alternatives.

An interesting question this work raises is the duality between "on-demand" load balancing of the type employed by MapReduce-like systems and the predictive, up-front allocation of work advocated by this paper. My suspicion is that both approaches have their place, and the choice of which to use depends on issues such as the predictability of the workload and the importance of "locality" in the performance of the system. Perhaps hybrid solutions will be the answer in some cases.

Resource Bricolage for Parallel DBMSs on Heterogeneous Clusters

Jiexing Li †, Jeffrey Naughton #, Rimma V. Nehme *

†Google Inc. jiexing@google.com

#University of Wisconsin, Madison naughton@cs.wisc.edu

*Microsoft Jim Gray Systems Lab rimman@microsoft.com

ABSTRACT

Running parallel database systems in an environment with heterogeneous resources has become increasingly common, due to cluster evolution and increasing interest in moving applications into public clouds or shared infrastructures. For database systems running in a heterogeneous cluster, the default uniform data partitioning strategy may overload some of the slow machines while at the same time it may underutilize the more powerful machines. Since the processing time of a parallel query is determined by the slowest machine, such an allocation strategy may result in a significant query performance degradation.

We take a first step to address this problem by introducing a technique we call *resource bricolage* that improves database performance in heterogeneous environments. Our approach quantifies the performance differences among machines with various resources as they process workloads with diverse resource requirements. We formalize the problem of minimizing workload execution time and view it as an optimization problem, and then we employ linear programming to obtain a recommended data partitioning scheme. We verify the effectiveness of our technique with an extensive experimental study on a commercial database system.

1. INTRODUCTION

With the growth of the Internet, our ability to generate extremely large amounts of data has dramatically increased. This sheer volume of data that needs to be managed and analyzed has led to the wide adoption of parallel database systems (DBMSs). Gamma [14] and Teradata [13] were some of the early parallel DBMSs to address the need to process this massive amount of data. Later on, many commercial parallel DBMSs were developed to provide this support. Examples include Aster nCluster [1], IBM DB2 Parallel Edition [7], Oracle nCUBE [8], Pivotal Greenplum [2], and Microsoft SQL Server Parallel Data Warehouse [3]. In order to get strategic insights to make business decisions, running online analytical processing (OLAP) workloads on parallel DBMSs has become increasingly important. To achieve high performance and scalability, these parallel DBMSs typically partition data across machines in a shared-nothing cluster to exploit data parallelism. A query running on such a system is typically broken up into subqueries, which are executed in parallel on the separate data chunks.

Nowadays, running parallel DBMSs in an environment with heterogeneous resources has become increasingly common, due to cluster evolution and increasing interest in moving applications into public clouds or shared infrastructures.

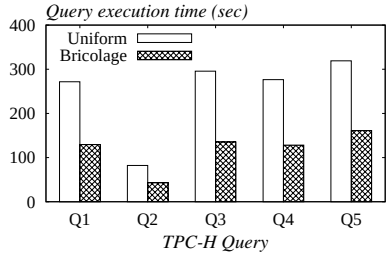


Figure 1: Query execution times with different data partitioning strategies.

Cluster evolution. When a cluster is first built, it typically begins with a set of identical machines. Over time, old machines may be reconfigured, upgraded, or replaced, and new machines may be added, thus resulting in a heterogeneous cluster.

Public clouds. With the proliferation of cloud computing, more and more parallel DBMSs are moving into public clouds. Previous research has revealed that the supposedly identical instances provided by public clouds often exhibit measurably different performance. Performance variations exist extensively in disk, CPU, memory, and network [16, 24, 32, 34].

Shared infrastructures. For systems that remain in private clusters, it is desirable to consolidate different workloads into a shared infrastructure to exploit data locality and multiplex physical resources. However, in a shared environment, it is hard to guarantee that resources available on different machines will always be the same for a given application. Since applications might request different amounts of resources on different machines, this may leave behind machines with vastly different available resources and capacities for new applications.

1.1 Motivation

Performance differences among machines (either physical or virtual) in the same cluster pose new challenges for parallel database systems. By default, parallel systems ignore differences among machines and try to assign the same amount of data to each. If these machines have different disk, CPU, memory, and network resources, they will take varying amounts of time to process the same amount of data. Unfortunately, the execution time of a query in a parallel DBMS is determined by its slowest machine. At worst, a

The original version of this article was published in VLDB 2015.

slow machine can substantially degrade the performance of the query.

On the other hand, a fast machine in such a system will be under-utilized, finishing its work early, sitting idle and waiting for the slower machines to finish. This suggests that we can reduce execution time by allocating more data to more powerful machines and less data to the overloaded slow machines, in order to reduce the execution time of the entire query. In Figure 1, we compare the execution times of the first 5 TPC-H queries running on a heterogeneous cluster with two different data partitioning strategies. One strategy partitions the data uniformly across all the machines, while the other partitions the data using our proposed technique, which we present in Section 4. The detailed cluster setup is described in Section 5. As can be seen from the graph, we can significantly reduce total query execution time by carefully partitioning the data.

Our task is complicated by the fact that whether a machine should be considered powerful or not depends on the workload. For example, a machine with powerful CPUs is considered "fast" if we have a CPU-intensive workload. For an I/O-intensive workload, it is considered "slow" if it has limited disks. Furthermore, to partition the data in a better way, we also need to know how much data we should allocate per machine. Obviously, enough data should be assigned to machines to fully exploit their potential for the best performance, but at the same time, we do not want to push too far to turn things around by overloading the powerful machines. The problem gets more complicated when queries in a workload have different (mixed) resource requirements, as usually happens in practice. Thus, for a workload with a mix of I/O, CPU, and network-intensive queries, the partitioning of data with the goal of reducing overall execution time is a non-trivial task.

1.2 Our Contributions

To improve performance of parallel DBMSs running in heterogeneous environments, we propose a technique we call ***resource bricolage***. The term bricolage refers to construction or creation of a work from a diverse range of things that happen to be available, or a work created by such a process. The keys to the success of bricolage are knowing the characteristics of the available items, and knowing a way to utilize and get the most out of them during construction.

In the context of our problem, a set of heterogeneous machines are the available resources, and we want to use them to process a database workload as fast as possible. Thus, to implement resource bricolage, we must know the performance characteristics of the machines that execute database queries, and we must also know which machines to use and how to partition data across them to minimize the workload execution time. To do this, we quantify differences among machines by using query optimizer and a set of profiling queries that estimate the machines' performance parameters. We then formalize the problem of minimizing workload execution time and view it as an optimization problem that takes the performance parameters as input. We solve the problem using a standard linear program solver to obtain a recommended data partitioning scheme. In Section 4.4, we also discuss alternatives for handling nonlinear situations. We implemented our techniques and tested them in Microsoft SQL Server Parallel Data Warehouse (PDW) [3], and our experimental results show the effectiveness of our proposed solution.

The rest of the paper is organized as follows. Section 2 formalizes the resource bricolage problem. Section 3 describes our way of characterizing the performance of a machine. Section 4 presents our approach for finding an effective data partitioning scheme. Section 5 experimentally confirms the effectiveness of our proposed solution. Section 6 briefly reviews the related work. Finally, Section 7 concludes the paper with directions for future work.

2. THE PROBLEM

2.1 Formalization

To enable parallelism in a parallel database system, tables are typically horizontally partitioned across machines. The tuples of a table are assigned to a machine either by applying a partitioning function, such as a hash or a range partitioning function, or in a round-robin fashion. A partitioning function maps the tuples of a table to machines based on the values of specified column(s), which is (are) called the partitioning key of the table. As a result, a partitioning function determines the number of tuples that will be mapped to each machine.

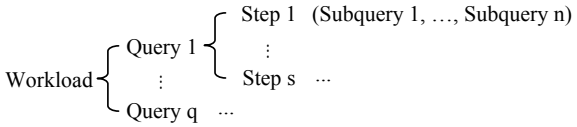


Figure 2: A query workload.

In our work, we target the problem of improving the performance of OLAP workloads in heterogeneous environments. In general, we may have a set of heterogeneous machines with different disk, CPU, network performance, and different amounts of memory. At the same time, we have a workload with a set of SQL queries as shown in Figure 2. A query can be further decomposed into a number of *steps* with different resource requirements. For each step, there will be a set of identical subqueries executing concurrently on different machines to exploit data parallelism. A step will not start until all steps upon which it depends on, if any, have finished. Thus, the running time of a step is determined by the longest-running subquery. The query result of a step will be repartitioned to be utilized by later steps, if needed.

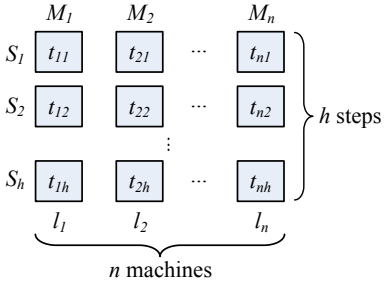


Figure 3: Problem setting.

We visually depict our problem setting in Figure 3. Let M_1, M_2, ..., M_n be a set of machines in the cluster, and let W be a workload consisting of multiple queries. Each query consists of a certain number of steps, and we concatenate all the steps in all of the queries to get a total of h steps: S_1, S_2, ..., S_h. Assume that t_{ij} would be the execution time for step S_j running on machine M_i if all the data were assigned to

M_i. Each column in the picture corresponds to a machine, and each row represents the set of subqueries running on the machines for a particular step. In addition, we assume that a machine M_i also has a storage limit l_i, which represents the maximum percentage of the entire data set that it can hold. The goal of resource bricolage is to find the best way to partition data across machines in order to minimize the total execution time of the entire workload.

2.2 Challenges

Whether it is worth allocating data to machines in a non-uniform fashion is dependent on the characteristics of the available computing resources. In [19], we compared the performance of three data partitioning strategies: *Uniform*, *Delete*, and *Optimal*. *Uniform* assigns the same amount of data to each machine. *Delete* is a simple heuristic that attempts to handle resource heterogeneity. It excludes some slow machines before it partitions the data uniformly to the remaining ones. Starting with the slowest set of machines, the machine exclusion attempt is repeated until no further improvement can be made. *Optimal* is the ideal data partitioning strategy that distributes data to machines in a way that minimized the overall execution time. We proved that in the worst case, the workload execution time of the *Delete* approach can be $\frac{n}{4}$ times as long as that of the *Optimal* approach [19].

Thus, it is important for us to come up with the optimal partitioning strategy to better utilize computing resources. To do this, there are a number of challenges that need to be tackled. First of all, we need to quantify performance differences among machines in order to assign the proper amounts of data to them. Second, we need to know which machines to use and how much data to assign to each of them for best performance. Intuitively, we should choose "fast" machines, and we should add more machines to a cluster to reduce query execution times. However, we found that this is not true for the worst-case example we discussed in [19]. In the worst-case example, the set of "fast" machines do not collaborate well with others in the same system, resulting in longer execution times.

3. QUANTIFYING PERFORMANCE DIFFERENCES

For each machine in the cluster, we use the runtimes of the queries that will be executed to quantify its performance. Since we do not know the actual query execution times before they finish, we need to estimate these values.

There has been a lot of work in the area of query execution time estimation [9, 10, 20, 22, 27]. Unlike previous work, we do not need to get perfect time estimates to make a good data partitioning recommendation. Instead, the ratios in time among machines are the key information that we need in order to assign the proper amounts of data to machines. Consider a very simple example with just two machines. Even if we can not estimate the query runtimes perfectly for these two machines, we can still assign the right amount of data to them, as long as we can correctly estimate how much faster or slower one machine is compared to the other. Thus, we adopt a less accurate but much simpler approach to estimate query execution times. Our approach can be summarized as follows. For a given database query, we retrieve its execution plan from optimizer and divide the plan into a set of pipelines. We then use the optimizer's cost model to estimate the CPU, I/O, and network "work" that needs to be done by each pipeline. To estimate the times to execute the pipelines on different machines, we run profiling queries to measure the speeds to process the estimated work for each machine.

3.1 Estimating the Cost of a Pipeline

Like previous work on execution time estimation [10, 22], we use the execution plan for a query to estimate its runtime. An execution plan is a tree of physical operators chosen by a query optimizer. In addition to the most commonly used operators in a single-node DBMS, such as Table Scan, Filter, Hash Join, etc., a parallel database system also employs data movement operators, which are used for transferring data between DBMS instances running on different machines.

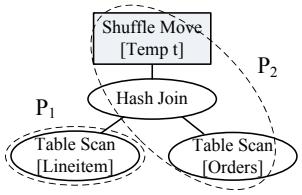


Figure 4: An execution plan with two pipelines.

An execution plan is divided into a set of pipelines delimited by blocking operators (e.g., Hash Join, Group-by, and data movement operators). The example plan in Figure 4 is divided into two different pipelines P_1 and P_2. Pipelines are executed one after another, and the total runtime of a query is the sum of the execution time(s) of its pipeline(s). To estimate a pipeline's execution time, we first predict what is the work of the pipeline and what is the speed to process the work. We then estimate the runtime of a pipeline as the estimated work divided by the processing speed.

For each pipeline, we use the optimizer's cost model to estimate the work (called *cost*) that needs to be done by CPUs, disks, and network, respectively. These costs are estimated based on the available memory size. We utilize the optimizer estimated cost units to define the work for an operator in a pipeline. We follow the idea presented in [20] to calculate the cost for a pipeline, and the interested reader is referred to that paper for details.

The default optimizer estimated cost is calculated using parameters with predefined values (e.g., the time to fetch a page sequentially), which are set by optimizer designers without taking into account the resources that will be available on the machine for running a query. Thus, it is not a good indicator of the actual query execution time for a specific machine. To obtain more accurate predictions, we keep the original estimates and treat them as estimated work if a query were to run on a "standard" machine with default parameters. Then, we test on a given machine to see how fast it can go through this estimated work with its resources (the speeds).

3.2 Measuring Speeds to Process the Cost

Measuring I/O speed. To get the speed to process the estimated I/O cost for a machine, we execute the following query with a cold buffer cache: *select count(*) from T*. This query simply scans a table T and returns the number of tuples in the table. It is an I/O-intensive query with negligible CPU cost. For this query, we use the query optimizer to get its estimated I/O cost, and then we run it to obtain

its execution time for the given machine. Then we calculate the I/O speed for this machine as the estimated I/O cost divided by the query execution time.

Measuring CPU speed. To measure the CPU speed, we test a CPU-intensive query: *select T.a from T group by T.a* from a warm buffer cache. We get its estimated CPU cost and runtime, and we calculate the CPU speed for this machine by dividing cost by runtime. Since small queries tend to have higher variation in the cost estimates and execution times, one practical suggestion is to use a sufficiently big table for the test. Meanwhile, since the time spent on transferring query results from a database engine to an external test program is not used to process the estimated CPU cost, we need to limit the number of tuples that will be returned. In our experiment, T contains $18M$ unsorted tuples, and only 4 distinct $T.a$ values are returned.

Measuring network speed. We use a separate program to test the network speed instead of a query running on an actual database system. The reason is that it is hard to find a query to test the network speed while isolating all other factors that can contribute to query execution times. For a query with data movement operators in a fully functional system, the query may need to read data from a local disk and store data in a destination table. If network is not the bottleneck resource, we can not observe the true network speed. Thus, we wrote a small program to simulate the actual system for transmitting data between machines. We run this program at its full speed to send (receive) data to (from) another machine that is known to have a fast network connection. At the end, we calculate the average bytes of data that can be transferred per second as the network speed for the tested machine.

Finally, for a pipeline P, we estimate its execution time as the maximum of $C_{Res}(P)/Speed_{Res}$, for any Res in {CPU, I/O, network}. The execution time of a plan is the sum of the execution times of all pipelines in the plan.

4. RESOURCE BRICOLAGE

After we estimate the performance differences among machines for running our workload, we now need to find a better way to utilize the machines to process a given workload as fast as possible. We model and solve this problem using linear programming, and we deploy special strategies to handle nonlinear scenarios.

4.1 Base and Intermediate Data Partitioning

Data partitioning can happen in two different places. One is base table partitioning when loading data into a system, and the other one is intermediate result reshuffling at the end of an intermediate step. For example, consider a subquery of a step that uses the execution plan shown in Figure 4. This plan scans two base tables: *Lineitem* and *Orders*, which may be partitioned across all machines. The result of this subquery, which can be viewed as a temporary table, is served as input to the next steps, if there are any. Thus, the output table may also be redistributed among the machines.

The execution time of a plan running on a given machine is usually determined by the input table sizes. For example, the runtime of the plan in Figure 4 depends on the number of *Lineitem* and *Orders* (L and O for short) tuples. The runtime of a plan that takes a temporary table as input is again determined by the size of the temporary table.

In some cases, the partitioning of an immediate table can be independent of the partitioning of any other tables. For example, if the output of $L \bowtie O$ is used to perform a local aggregate in the next step, we can use a partitioning function different from the one used to partition L and O to redistribute the join results. However, if the output of $L \bowtie O$ is used to join with other tables in a later step, we must partition all tables participating in the join in a distribution-compatible way. In other words, we have to use the same partitioning function to allocate the data for these tables.

In our work, we consider data partitioning for both base and intermediate tables. Note that our technique can also be applied to systems that do not partition base tables a priori or do not store data in local disks. For these systems, our approach can be used to decide the initial assignment of data to the set of parallel tasks running on machines with heterogeneous resources, and similarly, our approach can be used for intermediate result reshuffling. Instead of reading pre-partitioned data from local disks, these systems read data from distributed file systems or remote servers. In order to apply our technique, we need to replace the time estimates for reading data locally with the time estimates for accessing remote data. We omit the details here since it is not the focus of our paper.

4.2 The Linear Programming Model

Next, we will first give our solution to the situation where all tables must be partitioned using the same partitioning function, and then we extend it to cases where multiple partitioning functions are allowed at the same time.

Recall that in our problem setting, we have n machines, and the maximum percentage of the entire data set that machine M_i can hold is l_i. Our workload consists of h steps, and it would take time t_{ij} for machine M_i to process step S_j if all data were assigned to M_i. The actual t_{ij} values are unknown, and we use the technique proposed in Section 3 to estimate them. We want to find a data partitioning scheme that can minimize the overall workload execution time.

When all tables are partitioned in the same way, we can use just one variable to represent the percentage of data that goes to a particular machine for different tables. Let p_i be the percentage of the data that is allocated to M_i for each table. We assume that the time it takes for M_i to process step S_j is proportional to the percentage of data assigned to it. Based on this assumption, $p_i t_{ij}$ represents the total time to process p_i of the data for step S_j running on machine M_i. The execution time of S_j, which is determined by the slowest machine, is $\max_{i=1}^{n} p_i t_{ij}$. Then the total execution time of the workload can be calculated as $\sum_{j=1}^{h} \max_{i=1}^{n} p_i t_{ij}$. In order to use a linear program to model this problem, we introduce an additional variable x_j to represent the execution time of step S_j. Thus, the total execution time of the workload can also be represented as $\sum_{j=1}^{h} x_j$. The linear program that minimizes the total execution time of the workload can be formulated as below.

For step S_j, since the execution time x_j is the longest execution time of all machines, we must have $p_i t_{ij} \leq x_j$ for machine M_i. We also know that the percentage of data that can be allocated to M_i must be at least 0 and at most l_i. The sum of all p_is is 1, since all data must be processed. We can solve this linear programming model using standard linear optimization techniques to obtain the values for p_is ($0 \leq i \leq n$) and x_js ($0 \leq j \leq h$), where the set of p_i values represents

$$\text{minimize} \sum_{j=1}^{h} x_j$$
$$\text{subject to } p_i t_{ij} \le x_j \qquad 1 \le i \le n, 1 \le j \le h$$
$$\sum_{i=1}^{n} p_i = 1$$
$$0 \le p_i \le l_i \qquad 1 \le i \le n$$

a data partitioning scheme that minimizes $\sum_{j=1}^{h} x_j$. Note that we may use only a subset of the machines, since we do not need to run queries on a machine with 0% of the data. Thus, the data partitioning scheme suggests a way to select the most suitable set of machines and a way to utilize them to process the database workload efficiently.

4.3 Allowing Multiple Partitioning Functions

When different partitioning functions are allowed to be used by different tables, we are given more flexibility for making improvements. Thus, we want to apply different partitioning functions whenever possible. In order to do this, we need to identify sets of tables that must be partitioned in the same way to produce join-compatible distributions, and we apply different partition functions to tables in different sets.

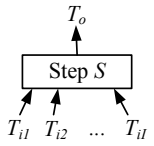


Figure 5: The input and output tables for a step.

For step S in workload W, let $\{T_{i1}, T_{i2}, ..., T_{iI}\}$ be the set of its input tables and T_o be its output table as we show in Figure 5. An input table to S could be a base table or an output table of another step, and all input tables will be joined together in step S. In order to perform joins, tuples in these tables must be mapped to machines using the same partitioning function, otherwise tuples that can be joined together may end up on different machines.

We define a ***distribution-compatible group*** as the set of input and output tables for W that must be partitioned using the same function, together with the set of steps in W that take these tables as input. Placing a step to a group implies that how the tables can be partitioned in a group has a significant impact on the execution time of the step. If we can find all distribution-compatible groups for W, we can apply different functions to tables in different groups for data allocation.

Given a database, we assume that the partitioning keys for base tables and whether two base tables should be partitioned in a distribution-compatible way or not are designed by a database administrator or an automated algorithm [4, 29, 31]. As a result, we know which base tables should belong to a distribution-compatible group. For intermediate tables, we need to figure this out. We generate the distribution-compatible groups for a workload W in the following way:

1. Create initial groups with corresponding distribution-compatible base tables according to the database design.
2. For each step S in W, perform the following three instructions.
   (a) For the input tables in S, find the groups that they belong to. If more than one group is found, ***merge*** them into a single group.
   (b) ***Assign*** S to the group.
   (c) ***Create*** a new group with the output table of S.

After the distribution-compatible groups are generated, we can employ the linear programming model proposed above to obtain a partitioning scheme for the tables to minimize total runtime of the steps in the group. For more details, we refer the reader to [19].

4.4 Handling Nonlinear Growth in Time

In our proposed linear programming model, we assume that query execution time changes linearly with the data size. Unfortunately, this assumption does not always hold true for database queries. This assumption is valid for the network cost of a query, where the transmission time increases in proportion to data size. It is also true for the CPU and I/O costs of many database operators, such as Table/Index Scan, Filter, and Compute Scalar. These operators take a large proportion of query execution times for analytical workloads.

The linear assumption may, however, be invalid for multi-phase operators such as Hash Join and Sort. We may introduce errors by choosing fixed linear functions for these operators in the following way. To estimate the t_{ij} value for step S_j running on machine M_i, we first assume that M_i gets $1/n$ of the data. We then use the query optimizer to generate the execution plan for S_j, and we estimate the runtime for the plan. Finally, the estimated value is magnified n times and returned as the t_{ij} value for S_j running on M_i. Based on all t_{ij}s we predict, a recommended partitioning is computed using the linear programming model, and the data we eventually allocate to M_i may be less or more than $1/n$.

If the plan is the same as the estimated plan and the operator costs increase linearly with the data size, everything will work as is. However, since the input table sizes could be different from our assumption, the plan may change, and some multi-phase operators may need more or fewer passes to perform their tasks. Thus, the estimated times we used to quantify the performance differences among machines may be wrong.

The impact of the changes in plans and operator executions is twofold. When a plan with lower cost is selected or fewer passes are needed for an operator, the actual query runtime should be shorter than our estimate, leaving more room for improvement. When things change in the opposite direction, query execution times may be longer than expected, and we may place too much data on a machine. The latter case is an unfavorable situation that we should watch out for. We use the following strategies to avoid making a bad recommendation.

- **Detection**: before we actually adopt a partitioning recommendation, we involve the query optimizer again

We omit replicated tables in our problem. Since a full copy of a replicated table will be kept on a machine, there is no need to worry about partitioning.

to generate execution plans. We re-estimate query execution times when assuming that each machine gets the fraction of data as suggested by our model. We return a warning to the user, if we find that the new estimated workload runtime is longer than the old estimate. This approach works for both plan and phase changes.

- **Safeguard**: to avoid overloading a machine M_i, we can add a new constraint $p_i \leq p_{isafe}$ to our model. By selecting a suitable value for p_{isafe} as a guarding point, we can force the problem to stay in the region, where query execution times grow linearly with data size.

Even if additional passes are required for some operators, the data processing time of a powerful machine may still be shorter than that of a slow machine. One possible direction would be to use a mixed-integer program to fully exploit the potential of a powerful machine. Due to lack of space, we leave this as an interesting direction for future work.

5. EXPERIMENTAL EVALUATION

This section experimentally evaluates the effectiveness and efficiency of our proposed techniques.

5.1 Experimental Setup

We implemented and tested our techniques in SQL Server PDW. Our cluster consisted of 9 physical machines, and each machine had two quad-core processors, 16GB of main memory, and eight disks. On top of each physical machine, we created a virtual machine (VM) to run our database system. One VM served as a control node for our system, while the remaining eight were compute nodes. We artificially introduced heterogeneity by allowing VMs to use varying numbers of processors and disks, limiting the amount of main memory, and by "throttling" the network connection.

Table	Partition Key	Table	Partition Key
Customer	c_custkey	Part	p_partkey
Lineitem	l_orderkey	Partsupp	ps_partkey
Nation	(replicated)	Region	(replicated)
Orders	o_orderkey	Supplier	s_suppkey

Table 1: Partition keys for the TPC-H tables.

The parallel database system we ran consists of single-node DBMS instances connected by a distribution layer. We have eight instances of this single-node DBMS, each running in one of the VMs. The single-node DBMS is responsible for exploiting the resources within the node. We used a TPC-H 200GB database for our experiments. Each table was either hash partitioned or replicated across all compute nodes. Table 1 summarizes the partition keys used for the TPC-H tables. Replicated tables were stored at every compute node on a single disk.

5.2 Overall Performance

To test the performance of different data partitioning approaches, we used a workload of 22 TPC-H queries. By default, each VM used 4 disks, 8 CPUs, 1Gb/s network bandwidth, and 8GB memory. In our experiments, we created 6 different heterogeneous environments as summarized below to run the queries.

1. ***CPU-intensive configuration***: to make more queries CPU bound, we use as few CPUs as possible for the VMs. In this setting, we use just one CPU for half of the VMs, and two CPUs for the other half. As a result, CPU capacity of the fast machines is twice that of the slow machines.

2. ***Network-intensive configuration***: similarly, to make more queries network bound, we reduce network bandwidth for the VMs. For half of the VMs we set the bandwidth to 10 Mb/s and for the other half to 20 Mb/s.

3. ***I/O-intensive configuration (2)***: we reduce the number of disks that are used by the VMs. For half of the VMs we limit the number of disks to one and for the remainder to two.

4. ***I/O-intensive configuration (4)***: in this setting, we have 4 types of machines. We set the number of disks used by the VMs to 1, 1, 2, 2, 4, 4, 8, and 8, respectively. Note that the I/O speeds of the machines with 8 disks (the fastest machines) are roughly 4 times as fast as the I/O speeds of the machines with just 1 disk (the slowest machines), and the I/O speeds of the machines with 4 disks are roughly 3.2 times as fast as the I/O speeds of the slowest machines.

5. ***CPU and I/O-intensive configuration***: the number of disks used by the VMs is the same as in the above configuration, but we reduce their CPU capability. We set the number of CPUs that they use to 2, 4, 2, 4, 2, 4, 2, and 4, respectively. In this setting, all VMs are different. If we calculate a ratio to represent the number of CPUs to the number of disks for a VM, we can conclude that subqueries running on a VM with a small ratio tend to be CPU bound, while subqueries running on a VM with a large ratio tend to be I/O bound. We refer to this configuration as *Mix-2*.

6. ***CPU, I/O, and network-intensive configuration***: The CPU and I/O settings are the same as above. We also reduce network bandwidth to make some of the subqueries network bound. We set the bandwidth for the VMs in Mb/s to 30, 30, 30, 10, 10, 30, 30, and 30, respectively. We refer to this configuration as *Mix-3*.

For each heterogeneous cluster configuration, we evaluate the performance of the strategy proposed in this paper (we refer to it as Bricolage). We use Uniform and Delete as the competitors. In Table 2(a), we illustrate the predicted workload execution time for different approaches running with different cluster configurations. We also calculate the percentage of time that can be reduced compared to the Uniform approach. We load the data into our cluster using different data partitioning strategies to run the queries, and we measure the actual workload processing times and the improvements. In Table 2(b), we list the numbers we observe after running the workload. As we can see from the tables, although in some cases, our absolute time estimates are not very precise, the percentage improvement we achieve is close to our predictions. As a result, we can conclude that our model is reliable for making recommendations.

In the above cases, we varied only the number of disks, CPUs, and network bandwidth for the VMs. The impact of heterogeneous memory sizes was investigated in [19], and our results indicated that the two strategies proposed in Section

Strategy	CPU-intensive	Network-intensive	I/O-intensive (2)	I/O-intensive (4)	Mix-2	Mix-3
Uniform (sec)	5346	5628	5302	5583	6451	8709
Delete (sec)	5346 (0.0%)	5628 (0.0%)	5103 (3.7%)	3522 (36.9%)	4760 (26.2%)	8052 (7.5%)
Bricolage (sec)	4115 (23.0%)	4583 (18.6%)	3317 (37.4%)	2431 (56.5%)	3420 (47.0%)	5202 (40.3%)

(a) Estimated execution time and percentage of time reduction for different data partitioning strategies

Strategy	CPU-intensive	Network-intensive	I/O-intensive (2)	I/O-intensive (4)	Mix-2	Mix-3
Uniform (sec)	7371	8720	6037	6275	7680	11564
Delete (sec)	7371 (0.0%)	8720 (0.0%)	6581 (-9.0%)	4026 (35.8%)	6107 (20.5%)	9202 (20.4%)
Bricolage (sec)	6024 (18.3%)	7205 (17.4%)	4195 (30.5%)	3236 (48.4%)	5131 (33.2%)	5767 (50.1%)

(b) Actual execution time and percentage of time reduction for different data partitioning strategies

Table 2: Overall performance (22 TPC-H queries).

4.4 can effectively avoid overloading machines with scarce memory. In [19], we also presented (*i*) TPC-H query execution time comparison with different strategies, (*ii*) evaluations of the accuracy of query performance prediction, and (*iii*) studies on whether or not further improvements might be possible, if we had better system performance predictions. For (*i*), our results showed that Bricolage can provide the best performance compared to the other alternatives we have considered. For (*ii*) and (*iii*), we found that accurate relative performance prediction is critical for Resource Bricolage, and it may not be worth trying too hard to improve the absolute performance prediciton accuracy. Due to space constraints, the interested reader is referred to the paper for details.

6. RELATED WORK

Our work is related to query execution time estimation, which can be loosely classified into two categories. The first category includes the work on progress estimation for running queries [9, 18, 20, 22, 23, 26]. The key idea for this work is to collect runtime statistics from the actual execution of a query to dynamically predict the remaining work/time for the query. In general, no prediction can be made before the query starts. The debug run-based progress estimator for MapReduce jobs proposed in [28] is an exception. However, it cannot provide accurate estimates for queries running in parallel database systems [21]. On the other hand, the second category of work focuses on query running time prediction before a query starts [35, 36]. In [36], the authors proposed a technique to calibrate the cost units in the optimizer cost model to match the true performance of the hardware and software on which the query will be run, in order to estimate query execution time. This paper gave details on how to calibrate the five parameters used by PostgreSQL. However, different database optimizers may use different cost formulas and parameters. Additional work is required before we can apply the technique to other database systems.

Another related research direction is automated partitioning design for parallel databases. The work in [17] investigates different multi-attribute partitioning strategies, and it tries to place tuples that satisfy the same selection predicates on fewer machines. The work in [11, 25] studies three data placement issues: choosing the number of machines over which to partition base data, selecting the set of machines on which to place each relation, and deciding whether to place the data on disk or cache it permanently in memory. In [29, 31], the most suitable partitioning key for each table is automatically selected in order to minimize estimated costs, such as data movement costs. While these approaches can substantially improve system performance, they focus on base table partitioning and treat all machines in the cluster as identical. In our work, we aim at improving query performance in heterogeneous environments. Instead of always applying a uniform partitioning function to these keys, we vary the amount of data that will be assigned to each machine for the purpose of better resource utilization and faster query execution. The work in [12, 30] attempts to improve scalability of distributed databases by minimizing the number of distributed transactions for OLTP workloads. Our work targets resource-intensive analytical workloads where queries are typically distributed. An adaptive and query-workload-aware mechanism for partitioning large-scale spatial data is proposed in [6], while other systems for processing spatial data typically employ static data-partitioning structures that cannot adapt to data changes. An elastic partitioning framework is developed in [33] for distributed OLTP DBMSs, which automatically scales resources in response to demand spikes and gradual changes in an application's workload. Our work currently employs a static partitioning strategy, and dynamic data partitioning would be an interesting direction for future research.

Our work is also related to skew handling in parallel database systems [15, 37, 38]. Skew handling is in a sense the dual problem of the one that we deal with in the paper. It assumes that the hardware is homogeneous, but data skew can lead to load imbalances in the cluster. It then tries to level the imbalances that arise.

Finally, our paper is related to various approaches proposed for improving system performance in heterogeneous environments. The work in [40] proposed solutions to improve performance for latency-sensitive applications running on clusters with heterogeneous network connectivity. A suite of optimizations are proposed in [5] to improve MapReduce performance on heterogeneous clusters. Zaharia et al. [39] developed a scheduling algorithm to dispatch straggling tasks to reduce execution times of MapReduce jobs. Since a MapReduce system does not use knowledge of data distribution and location, our technique cannot be used to pre-partition the data in HDFS. However, we can apply our technique to partition intermediate data in MapReduce systems with streaming pipelines.

7. CONCLUSION AND FUTURE IDEAS

We studied the problem of improving database performance in heterogeneous environments. We developed a technique to quantify performance differences among machines with heterogeneous resources and to assign proper amounts of data to them. Extensive experiments confirm that our technique can provide good and reliable partition recommendations for given workloads with minimal overhead.

This paper lays down a foundation for several directions towards future studies to improve database performance running in the cloud. Our work aims at improving the performance of OLAP workload on heterogeneous clusters, where similar problems on performance prediction and performance optimization for OLTP workloads on heterogeneous environments remain open. While the focus of this work has been on static data partitioning strategies, a natural follow-up will be to study how to dynamically repartition the data at runtime, when our initial predictions were not accurate or system conditions have changed. The model proposed in the paper assumes that queries are running sequentially in the workload. Since queries might run concurrently in a system, the relatively short running ones might have negligible impact on the total execution time. One promising direction would be to take into account concurrent query execution and explicitly model how queries interact with each other to better utilize resources. As we mentioned in the introduction, previous research has revealed that the supposedly identical instances provided by a public cloud often exhibit measurable performance differences. In the cases where we are given a budget constraint or a time constraint, we will encounter a problem of selecting the most suitable computing resources for building a cluster. In addition to the performance prediction and data allocation challenges we are tackling in the paper, we also need to either (*i*) select a set of most suitable computing resources that minimize the total execution time for a given budget, or (*ii*) select a set of most suitable computing resources that minimize the cost for a given performance goal. We think both are very interesting directions for future research.

Acknowledgment

This research was supported by a grant from Microsoft Jim Gray Systems Lab, Madison, WI. We would like to thank everyone in the lab for valuable suggestions on this project.

8. REFERENCES

[1] Aster Data nCluster. http://download.101com.com/tdwi/ww29/Aster_Data_A_New_Architecture.pdf.

[2] Pivotal Greenplum. http://pivotal.io/big-data/pivotal-greenplum.

[3] SQL Server 2012 Parallel Data Warehouse. http://www.microsoft.com/en-ca/server-cloud/products/analytics-platform-system/.

[4] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. SIGMOD, 2004.

[5] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Tarazu: Optimizing mapreduce on heterogeneous clusters. ASPLOS, 2012.

[6] A. M. Aly, A. R. Mahmood, M. S. Hassan, W. G. Aref, M. Ouzzani, H. Elmeleegy, and T. Qadah. AQWA: Adaptive query workload aware partitioning of big spatial data. *PVLDB*, 2015.

[7] C. Baru, G. Fecteau, A. Goyal, H.-I. Hsiao, A. Jhingran, S. Padmanabhan, W. Wilson, and A. G. H. i Hsiao. DB2 Parallel Edition. *IBM Systems Journal*, 1995.

[8] R. Buck. The Oracle Media Server for nCUBE Massively Parallel Systems. *Parallel Processing Symposium*, 1994.

[9] S. Chaudhuri, R. Kaushik, and R. Ramamurthy. When can we trust progress estimators for SQL queries? In *SIGMOD*, 2005.

[10] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. Estimating progress of execution for SQL queries. In *SIGMOD*, 2004.

[11] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in Bubba. *SIGMOD Record*, 1988.

[12] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 2010.

[13] M. Dempsey. Monitoring active queries with Teradata manager 5.0. http://www.teradataforum.com/attachments/a030318c.doc, 2001.

[14] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *TKDE*, 1990.

[15] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. VLDB, 1992.

[16] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift. More for your money: exploiting performance heterogeneity in public clouds. SoCC, 2012.

[17] S. Ghandeharizadeh, D. J. DeWitt, and W. Qureshi. A performance analysis of alternative multi-attribute declustering strategies. SIGMOD, 1992.

[18] A. C. König, B. Ding, S. Chaudhuri, and V. Narasayya. A statistical approach towards robust progress estimation. *PVLDB*, 2012.

[19] J. Li, J. Naughton, and R. V. Nehme. Resource bricolage for parallel database systems. *PVLDB*, 2014.

[20] J. Li, R. V. Nehme, and J. F. Naughton. GSLPI: A cost-based query progress indicator. In *ICDE*, 2012.

[21] J. Li, R. V. Nehme, and J. F. Naughton. Toward progress indicators on steroids for big data systems. In *CIDR*, 2013.

[22] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke. Toward a progress indicator for database queries. SIGMOD, 2004.

[23] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke. Increasing the accuracy and coverage of SQL progress indicators. In *ICDE*, 2005.

[24] D. Mangot. EC2 variability: The numbers revealed. http://tech.mangot.com/roller/dave/entry/ec2_variability_the_numbers_revealed, 2009.

[25] M. Mehta and D. J. DeWitt. Data placement in shared-nothing parallel database systems. *The VLDB Journal*, 1997.

[26] C. Mishra and N. Koudas. A lightweight online framework for query progress indicators. *ICDE*, 2007.

[27] K. Morton, M. Balazinska, and D. Grossman. Paratimer: a progress indicator for MapReduce DAGs. In *SIGMOD*, 2010.

[28] K. Morton, A. Friesen, M. Balazinska, and D. Grossman. Estimating the progress of MapReduce pipelines. ICDE, 2010.

[29] R. Nehme and N. Bruno. Automated partitioning design in parallel database systems. SIGMOD, 2011.

[30] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. SIGMOD, 2012.

[31] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. SIGMOD, 2002.

[32] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. In *PVLDB*, 2010.

[33] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *PVLDB*, 2014.

[34] G. Wang and T. S. E. Ng. The impact of virtualization on network performance of amazon EC2 data center. INFOCOM, 2010.

[35] W. Wu, Y. Chi, H. Hacígümüş, and J. F. Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. *PVLDB*, 2013.

[36] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs, and J. F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *ICDE*, 2013.

[37] Y. Xu and P. Kostamaa. Efficient outer join data skew handling in parallel DBMS. *PVLDB*, 2009.

[38] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen. Handling data skew in parallel joins in shared-nothing systems. SIGMOD, 2008.

[39] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. OSDI, 2008.

[40] T. Zou, R. L. Bras, M. V. Salles, A. Demers, and J. Gehrke. ClouDiA: A deployment advisor for public clouds. PVLDB, 2013.

Technical Perspective - Implicit Parallelism through Deep Language Embedding

Zachary G. Ives
University of Pennsylvania
zives@cis.upenn.edu

Modern "big data" analysis was motivated by the needs of the large Internet players, but it was enabled by two main technical developments: parallel data processing technologies that support reliable and scalable computation over *unreliable* shared-nothing clusters of computers, and continued advances in machine learning algorithms and techniques. Initial work on these two areas happened largely independently: MapReduce was developed for aggregate computations over large multitudes of records, with minimal control flow and no evident goal of supporting machine learning. Conversely, many of the advances in machine learning research targeted a single machine.

However, many subsequent developments in parallel data processing have indeed been motivated by machine learning tasks. Additionally, many machine learning algorithms have been ported to parallel data processing libraries, and certain machine learning techniques, such as deep learning, almost entirely owe their success to parallel processing techniques. Tremendous activity, in both the distributed systems and database communities, and in both industry and academia, continues in improving the runtime architectures and algorithms. Key points of focus have included more effective data partitioning and load balancing strategies, as well as mechanisms for making reliable computation more efficient, such as checkpointing and recomputation schemes at different levels of granularity. Such work has led to many parallel data processing platforms [1, 2, 3].

Systems-level issues are not the only concern. The database community continues to search for better programming abstractions and interfaces. Efficient parallel dataflow programs adopt control and dataflow patterns where processing is done independently on different compute nodes, then messages are exchanged, and the task iterates until some condition is reached. Parallel data processing researchers have extensively investigated how to express distributed data and control flow. We now understand much more about the trade-offs between iterative computation with synchronization (where all nodes are in the same interation step) versus fully asynchronous execution (which can be more efficient but makes it harder to port algorithms developed in a centralized setting); how to abstract working state within the program; and whether the actual dataflow across nodes should be modeled as values, deltas, or generic messages.

A much less-studied, but at least equally important, issue revolves around the fact that programmers want to write code in familiar languages and take advantage of existing libraries, so there needs to be a strong coupling between the parallel data processing runtime system and an existing *host language* like Java, Scala, or Python. Ideally, the runtime platform could still predict and optimize for the workload. Most existing approaches to host language integration use variations of Microsoft's Language-Integrated Querying (LINQ), which incorporates some SQL constructs into the host language; or user-defined `map` and `reduce` functions, applied to collections of objects, along with second-order functions like `join` and `cogroup`. However, increasingly there are applications where the data or the computation has some innate structure requiring navigation and assembly of results, where such abstractions do little to help.

The "Implicit Parallelism through Deep Language Embedding" paper makes an astute observation that parallel data processing is the first setting that couples structured navigation operations with host programming languages: in the 1990s, the fields of object-oriented databases and persistent programming languages focused on these concerns. While object-oriented databases perhaps never found their "killer application," many innovative ideas were developed.

Alexandrov and collaborators propose Emma, an extension to Scala for parallel data processing, which operates on bags of structured objects. Emma leverages *monad comprehensions*, first developed as primitive query operations for object-oriented databases, to traverse and assemble these structured objects. These monad comprehensions form a very natural and powerful mechanism for declaratively specifying structuring and navigation operations, as illustrated in this paper. The authors also highlight some of the sophisticated query optimization techniques that can be applied, and further techniques and an evaluation appear in the full conference version of the paper. This paper does a great job of taking ideas that may have been "ahead of their time," and using them to cleanly tackle some of the open problems in the parallel data processing space.

1. REFERENCES

[1] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc VLDB*, 1(2):1265–1276, 2008.

[2] V. Markl. Breaking the chains: On declarative data analysis and data independence in the big data era. *Proc VLDB*, 7(13):1730–1733, 2014.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 2010.

Implicit Parallelism through Deep Language Embedding

Alexander Alexandrov Asterios Katsifodimos Georgi Krastev
Volker Markl

TU Berlin
first.last@tu-berlin.de

ABSTRACT

Parallel collection processing based on second-order functions such as `map` and `reduce` has been widely adopted for scalable data analysis. Initially popularized by Google, over the past decade this programming paradigm has found its way in the core APIs of parallel dataflow engines such as Hadoop's MapReduce, Spark's RDDs, and Flink's DataSets. We review programming patterns typical of these APIs and discuss how they relate to the underlying parallel execution model. We argue that fixing the abstraction leaks exposed by these patterns will reduce the cost of data analysis due to improved programmer productivity. To achieve that, we first revisit the algebraic foundations of parallel collection processing. Based on that, we propose a simplified API that (*i*) provides proper support for nested collection processing and (*ii*) alleviates the need of certain second-order primitives through comprehensions – a declarative syntax akin to SQL. Finally, we present a metaprogramming pipeline that performs algebraic rewrites and physical optimizations which allow us to target parallel dataflow engines like Spark and Flink with competitive performance.

1. INTRODUCTION

One can argue that the success of Google's MapReduce programming model [5] is largely due to its expressiveness and simplicity. Exposing an API built around second-order functions such as `map` f and `reduce` h enables general-purpose programming with collections via user-defined functions (UDFs) f and h. At the same time, the semantics of `map` and `reduce` alone (i.e., regardless of their UDF parameters) enable data-parallelism and facilitate scalability.

Vanilla MapReduce is a perfect fit for generalized processing and aggregation of a single collection of complex objects, but programmers stretch its limits when trying to express algorithms characterized by multiple inputs and non-trivial data and control flow dependencies. To overcome these limitations without sacrificing the benefits of seamless integration of UDFs and collection processing primitives in a general-purpose host language like Java or Scala, projects like Cascading [1], SCOPE [9], Spark [23], and Stratosphere/Flink [3] emerged, proposing programming model extensions that focus on two basic aspects:

- **Expressive dataflows APIs**. This includes more second-order primitives (e.g., `cogroup`, `cross`, `join`) as well as means to construct advanced dataflows by composing them freely.
- **Non-trivial data and control flow**. This includes primitives for data exchange between the driver and the UDFs, caching, as well as primitives that enable native control flow.

To motivate the goal of this work, we revisit some programming patterns associated with the features listed above. The code examples are given in Spark's Resilient Distributed Datasets (RDD) API, although similar observations can be made in the other APIs as well. The domain used in the examples consists of the following Scala types:

```
case class Person(id: Long, email: String, name: String)

case class Email(id: Long, from: String,
                 to: String, msg: String)
```

Join Cascades. In our first example, we want to write code that associates all emails with their sender and receiver.

```
// 1) join 'people' with 'emails' on 'sender'
val xs = people.map(p => (p.email, p)) join
         emails.map(e => (e.from, e))
// 2) join 'people' with 'xs' on 'receiver'
val ys = people.map(p => (p.email, p)) join
         xs.map(x => (x._2._2.to, x._2))
// 3) transform 'ys' as an RDD of flat tuples
val rs = ys.map(y => {
  val to    = y._2._1     // project 'to'
  val from  = y._2._2._1  // project 'from'
  val email = y._2._2._2  // project 'email'
  (from, to, email)
})
```

Two problems become evident from the above code snippet. First, since Spark models keys as data, inputs on key-based operators such as `join` need to be explicitly transformed into RDDs of `(key, value)` pairs, which enforces an extra `map` before each `join`. Second, n-way joins must be specified as a cascade of binary joins. The element type in the end-result (`ys`) is therefore a tuple of nested pairs whose shape reflects the tree shape of the join cascade. Accessing base data requires projection chains that traverse the tuple tree to its leafs. As the example above illustrates, such idiomatic programming patterns lead to cluttered and hard to read code. Flink and Scalding manage to resolve the first issue by modeling keys as functions rather than data. The second is usually alleviated through weaker schema models, like the field literal lists used in Cascading and Scalding.

The original version of this article was published in SIGMOD 2015.

Aggregates. For our second example, consider a situation where we want to compute a tf-idf statistic over the collection of emails. We start by calculating the term frequencies – we tokenize each email message into terms `t` and compute their frequencies `tfrq` using a library function, extend the resulting `(t, tfrq)` sequence with the enclosing email identifier, and finally flatten the result.

```
val tf = emails.flatMap(email => {
  tokenizeAndCount(email.msg).map {
    case (t, tfrq) => Tf(email.id, t, tfrq)
  }
})
```

Next, we have to calculate the inverse document frequencies, which for a term t and a document corpus D with $|D| = N$ are given by the following formula.

$$\text{idf}(t, D) = \log \frac{N}{|\{\, d \in D \mid t \in d \,\}|} \quad .$$

One way to do this in Spark is to group by term and map over the groups to calculate the idf values.

```
val N = emails.count().toDouble
val idf = tf
  .groupBy { case (_, t, _) => t }
  .map { case(t, docs) => (t, math.log(N / docs.size)) }
```

However, the proper way to express this computation is with a `reduceByKey` followed by a `map`.

```
val idf = tf
  .map { case (_, t, _) => (t, 1) }
  .reduceByKey(_ + _)
  .map { case (t, dfrq) => (t, math.log(N / dfrq)) }
```

Although they denote the same result, the two variants define different computations. The first shuffles the entire input and materializes the groups at the receiver side, whereas the second computes partial aggregates locally before shuffling and merging the final document frequencies `dfrq`. The difference in performance gets even more dramatic for naturally occurring skewed distributions, as in this case above where the terms follow a Zipf distribution. Programmers are therefore advised to use the `reduceByKey` pattern whenever possible. As in the previous case, this leads to situations where the code is organized according to execution model specifics rather than readability. Again, the problem can be witnessed across all similar APIs (Flink, Cascading, etc.).

Caching. To motivate the need for caching, consider the following piece of code which computes the `tfidf` values out of `tf` and `idf` and uses them to iteratively update a model:

```
val tfidf: RDD[TfIdf] = /* compute 'tf*idf' per term */
var model: RDD[Model] = /* initialize an ML model */
while (...) { // update 'model' until convergence
  model = /* derive from 'tfidf' and old 'model' */
}
```

The problem with that code is that `RDD` expressions are lazy. Under the hood, the `RDD` type implements a builder pattern that accumulates calls of *transformation* primitives like `map`, `flatMap`, `groupBy` into a parallel execution plan. Evaluation is implicitly forced by *action* primitives like `count`, `collect`, `first`. The implication for the code above is that the `tfidf` term is evaluated once for every loop. To fix this type of problems, Spark offers a `cache` primitive that forces an `RDD` result to be persisted. In our example, we can append `cache` to the definition of `tfidf`. As before, figuring out when to use it is left to the programmer and requires understanding and consideration of Spark's execution model. Cascading does not offer explicit support for caching, while Flink can infer and enforce it, but only in limited situations (as we will discuss shortly).

Broadcast Variables. Another problem can arise in a variation of the last example:

```
val tfidf: Seq[TfIdf] = /* compute as above */.collect()
var model: RDD[Model] = /* initialize an ML model */
while (...) { // update 'model' until convergence
  model = /* transform old 'model' */.map(x => {
    /* anonymous function that reads 'tfidf' */
  })
}
```

Evaluation of `tfidf` is now triggered outside of the loop by the `collect` action. The result of the computation is collected as a local sequence and subsequently read in the `map` UDF. As part of its closure, the `tfidf` value is therefore serialized and shipped to the cluster in each iteration. In this situation, performance can be improved by wrapping the `tfidf` definition in a `sc.broadcast(...)` call. The value is then broadcast to the cluster only once outside of the loop and can be subsequently accessed from the `map` UDF through a `tfidf.value` call. As in the previous case, the decision whether to ship a read-only value as part of the UDF closure or as a broadcast variable is left to the programmer. A similar construct exists in Flink's API.

Control Flow. The final issue we highlight is concerned with the form of certain control flow primitives. In order to optimally express the while loop from the caching example in Flink, for example, one has to phrase the program as follows:

```
val tfidf: DataSet[TfIdf] = /* compute per term */
var model: DataSet[Model] = /* initialize an ML model */
model = model.iterate(model => {
  /* transform 'tfidf' and old 'model' */
})
```

Note how Flink relies on a dedicated `iterate` construct for the iterative part of the program. The reason for this is once more tied to the underlying execution mode. Spark supports only acyclic dataflows and realizes iterative computation by lazily unrolling and evaluating dataflows from a Scala-driven loop. Flink's runtime, in contrast, offers restricted support for native iterations. This approach has performance benefits (e.g. less scheduling overhead, loop-invariant data caching), but requires special feedback edges in the dataflow graph. To introduce those edges, a dedicated construct like `iterate` is required at the API level. Ideally, however, this should be hidden from the programmer, who should be able to use a native Scala `while` loop in both cases.

Problem Statement. The above examples highlight the existence of specific programming patterns and primitives across various parallel dataflow APIs and execution engines. The common theme that shines through is the tight interplay between the programming interface and the underlying execution model. We end up in a situation with several well-known problems: (*i*) high barrier of entry due to the required level of understanding of the underlying execution model, (*ii*) hard to read and maintain code due to low-level

abstractions, and (*iii*) missed opportunities for optimization due to hard-coded execution strategies.

One way to tackle these problems is to provide high-level programming abstractions on top of the low-level dataflow APIs. The merits of this strategy are validated by the popularity of external languages such as Pig, Hive, and SystemML, on the one side, and specialized internal libraries for relational processing (DataFrame APIs), graph analysis (GraphX, GraphLab), and machine learning (MLLib, Mahout), on the other. This development, however, does not resolve the need for seamless, high-level integration of parallel collection processing in a general-purpose language. External languages introduce a language barrier, while internal libraries introduce a domain barrier.

In this paper, we argue in favor of an alternative approach based on *deep language embedding* through quotation and metaprogramming. The ability to manipulate data analysis programs at compile time has twofold impact. First, it facilitates declarative, SQL-like dataflow definitions through host language constructs such as comprehensions. Second, it allows to decompose the program code as combination of (parallel) dataflow and (sequential) driver fragments, and make holistic decisions for optimal dataflows execution based on the surrounding driver context. The overall effect is a high-level collection processing API where notions of parallelism associated with an underlying dataflow engine are hidden from the programmer.

The remainder of this paper is structured as follows. Section 2 reviews a theoretical model for parallel collection processing and its relation to comprehensions – a declarative syntax generalizing SQL. Section 3 presents Emma [2] – a domain-specific language (DSL) embedded in Scala which enables parallel collection processing through comprehensions. Section 4 sketches Emma's compiler pipeline and discusses how traditional database optimizations such as partial aggregates and join order can be revised in light of the model from Section 2. Section 5 reviews related work, and Section 6 discusses ideas for future research. For a more detailed technical description, we refer the reader to the original version of this paper [4].

2. FORMAL FOUNDATIONS

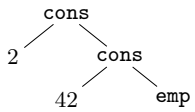
Spark's `RDD`, Cascading's `Collection`, and Flink's `DataSet` all represent homogeneous distributed collections with so called "bag semantics". That is, the elements in a bag share the same type, their order is not fixed, and duplicates are allowed. Our point of departure therefore is a suitable formal model for distributed collections (Section 2.1) and parallel computations on those (Section 2.2), which also facilitates declarative expression syntax (Section 2.3).

2.1 Bags as Algebraic Data Types

Bag Structure. We define the polymorphic type $\mathsf{Bag}\,A$ structurally, using a *recursive constructor algebra*.

$$\mathtt{type}\ \mathsf{Bag}\,A\ =\ \mathtt{emp}\ \mid\ \mathtt{cons}\,x{:}\,A\ xs{:}\,\mathsf{Bag}\,A \quad \text{(AlgBag-Ins)}$$

The above definition states that all possible $\mathsf{Bag}\,A$ values can be constructed inductively by two primitive constructor functions: `emp` (which denotes the empty bag), and `cons` (which denotes the bag where the element x is added to the bag xs). In other words, for each $xs \in \mathsf{Bag}\,A$ there is a corresponding functional expression t_{xs} (which we call the *constructor application tree*) that constructs xs. For example, the tree associated with $\{\!\{\,2, 42\,\}\!\}$ looks as follows.



Bag Semantics. By definition, constructor algebras like AlgBag-Ins are *initial* and thereby, following Lambek's lemma [17], *bijective*. This means that the association between constructor application trees and values is bidirectional – each constructor application tree t_{xs} represents precisely one bag xs and vice versa. This poses a problem, as it contradicts our intended semantics, which state that element order should does not matter. Using only the algebra definition, we have $\{\!\{\,2, 42\,\}\!\} \neq \{\!\{\,42, 2\,\}\!\}$ because the corresponding trees are different. To overcome this problem, we must add an appropriate *semantic equation*.

$$\mathtt{cons}\,x_1\,\mathtt{cons}\,x_2\,xs = \mathtt{cons}\,x_2\,\mathtt{cons}\,x_1\,xs \quad \text{(EQ-Comm-Ins)}$$

The equation states that the order of element insertion is irrelevant for the constructed value. Based on this equation, we can create an equivalence relation on trees and use the induced equivalence classes $[t_{xs}]$ instead of the original trees to ensure $xs \leftrightarrow [t_{xs}]$ bijectivity. In our running example, substituting the trees for $\{\!\{\,2, 42\,\}\!\}$ and $\{\!\{\,42, 2\,\}\!\}$ in the left- and right-hand sides of (EQ-Comm-Ins), correspondingly, renders them equivalent and puts them in the same equivalence class $[\,\{\!\{\,2, 42\,\}\!\}\,]$.

Relevance for Data Management. Conceptually, the $xs \mapsto t_{xs}$ direction can be interpreted as a recursive parser that decomposes a bag xs into its constituting elements. Database runtimes encapsulate this behavior in a reusable operator called `Scan`, and use it to sequentially read the records in a base table. Indeed, we can define a simple iterator-based version of `Scan` with the help of the AlgBag-Ins constructors (again using Scala syntax).

```
class Scan(var xs: Bag[A]) {
  def next(): Option[A] = xs match {
    case emp          => Option.empty[A]
    case cons(x, ys) => xs = ys; Some(x)
  }
}
```

Union Representation. The constructors in AlgBag-Ins impose a *left-deep* structure on the constructor application trees. There is, however, another algebra and a corresponding set of semantic equations that encodes the same initial semantics by means of *general binary trees*.

$$\begin{aligned}\mathtt{type}\ \mathsf{Bag}\,A\ =\ &\mathtt{emp}\\ \mid\ &\mathtt{sng}\,x{:}\,A\\ \mid\ &\mathtt{uni}\,xs{:}\,\mathsf{Bag}\,A\ ys{:}\,\mathsf{Bag}\,A\end{aligned} \quad \text{(AlgBag-Union)}$$

$$\begin{aligned}\mathtt{uni}\,xs\,\mathtt{emp}\ &=\ \mathtt{uni}\,\mathtt{emp}\,xs\ =\ xs && \text{(EQ-Unit)}\\ \mathtt{uni}\,xs\,(\mathtt{uni}\,ys\,zs)\ &=\ \mathtt{uni}\,(\mathtt{uni}\,xs\,ys)\,zs && \text{(EQ-Assoc)}\\ \mathtt{uni}\,xs\,ys\ &=\ \mathtt{uni}\,ys\,xs && \text{(EQ-Comm)}\end{aligned}$$

Here `emp` denotes the empty bag, `sng` x denotes the singleton bag, and `uni` xs ys denotes the union of xs and ys.

Although AlgBag-Ins and AlgBag-Union model the same type semantics, the latter reflects better the essence of parallel collection processing, as we will show in Section 2.2.

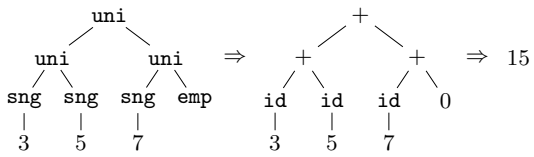
2.2 Structural Recursion on Bags

The previous section described a conceptual model for the structure of bags that identifies bag values with equivalence classes of constructor application trees. We now describe the principle of *structural recursion* – a method for defining functions on bags xs by means of systematic substitution of the constructor applications in the associated t_{xs} tree.

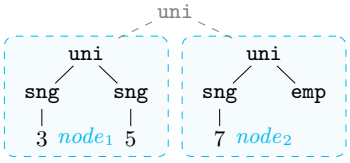
Basic Principle. Consider a case where we want to compute the sum of the elements in $xs = \{\{3, 5, 7\}\}$. We can define this operation with a higher-order function called `fold`.

```
// structural recursion on union-style bags
def fold[A,B](e: B, s: A => B, u: (B, B) => B)
             (xs: Bag[A]) = xs match {
  case emp        => e
  case sng(x)     => s(x)
  case uni(ys,zs) => u(fold(e,s,u)(ys), fold(e,s,u)(zs))
}
```

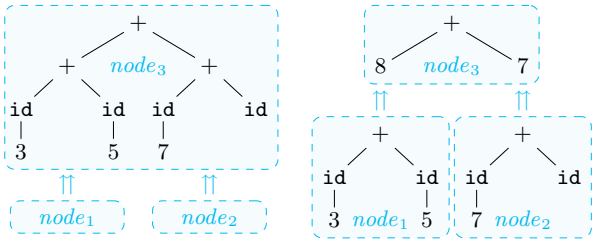
The `fold` function takes three function arguments: `e`, `s`, and `u`, substitutes them in place of the constructor applications in t_{xs}, and evaluates the resulting expression tree to get a final value $z \in \mathsf{B}$. To compute the sum of all elements, for example, we can substitute `e` $= 0$, `s` $=$ `id`, and `u` $= +$.



Relevance for Parallel Data Management. Again, we want to highlight the importance of this view on bag computations from a data management perspective. Imagine a scenario where xs is partitioned and distributed over two nodes: $xs_1 = \{\{3, 5\}\}$ and $xs_2 = \{\{7\}\}$. Conceptually, the value is still $xs =$ `uni` $xs_1\ xs_2$, but the `uni` is evaluated only if we have to materialize xs in a single node.



If we need the xs only to apply a `fold`, we can push the `fold` argument functions to the nodes containing xs_i, apply the `fold` locally, and ship the computed z_i values instead. In general, `e`, `s`, and `u` do not form an initial algebra. This implies loss of information when the substituted t_{xs} tree is evaluated to z, and thereby that z is "smaller" than t_{xs}. This is evident in the sum example – shipping the partial sums z_i is more efficient than shipping the partial bags xs_i.

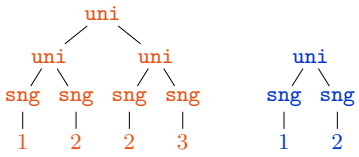


Fold Examples. The `fold` function provides a generic mold for specifying operations on collections. Aggregation functions like `min`, `max`, `sum`, and `count`, existential qualifiers like `exists` and `forall`, as well as collection processing operators like `map` and `filter` can be defined as folds.

Moreover, starting from `fold` we can define an algebraic structure known as *monad* on top of Bag A and enable declarative specification of computations.

2.3 Bag Comprehensions

Consider two bags $xs = \{\{1, 2, 2, 3\}\}$ and $ys = \{\{1, 2\}\}$ and their corresponding constructor application trees



in an example where you want to compute the bag of all pairs (x, y) where $x \in xs$, $y \in ys$ and $x = y$.

If xs and ys were sets, we could describe this computation mathematically as a set comprehension.

$$\{\, (x, y) \mid x \in xs,\ y \in ys,\ x = y \,\}$$

If xs and ys were lists, we could write a Python list comprehension.

```
[ (x, y) for x in xs for y in ys if x == y ]
```

If xs and ys were database relations, we could write a select-from-where query (effectively a SQL comprehension).

```
SELECT x, y FROM xs AS x, ys AS y WHERE x = y
```

Finally, let xs and ys be values from the user-defined type Bag $\mathbb{N}$ presented in Section 2.1. Modern functional languages like Scala allow us to use native comprehension syntax for arbitrary types, as long as those implement the so-called monad operators. We can therefore write the intended computation like this.

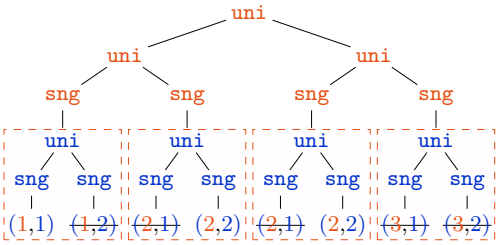
```
for (x <- xs; y <- ys; if x == y) yield (x, y)
```

At parse time, the above comprehension is transformed into a chain of nested `flatMap` applications ending with a `map` and interleaved with `withFilter` applications as follows.

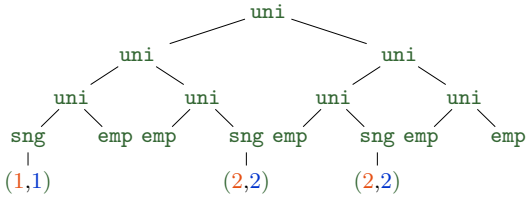
```
xs.flatMap(x =>
  ys.withFilter(y => x == y).map(y => (x, y)))
```

This desugaring scheme can be interpreted in terms of the structural recursion scheme discussed above. The `map` part of the `flatMap` application operates on the level of the (orange) xs tree – each value x is substituted with a copy

of the entire ys tree. The inner `map` operates on the level of the ys trees and maps their y values to a (x, y) pair using the x from the outer `map`. We end up with an outer (orange) bag of inner (blue) bags.



The `withFilter` application substitutes singleton bags that do not satisfy the $x = y$ predicate with `emp`, and the `flat` part of `flatMap` "forgets" the nested bag structure by inlining the inner trees into the outer one.



Theory to Practice. Bag comprehensions provide the key ingredient for solving two long-standing problems. First, as first-class citizen in a general-purpose source language, comprehension syntax offers direct means for declarative parallel collection processing (see Section 3). Second, as a first-class citizen in an object language subject to metaprogramming, comprehensions can serve as an entry point for the integration of dataflow optimization techniques into general-purpose languages (see Section 4).

3. LANGUAGE DESIGN

Based on the formal foundations outlined in Section 2, we present Emma [2] – a DSL for parallel collection processing embedded in Scala. We first discuss the core API features by example, revisiting the issues outlined in Section 1, and then list the requirements of our approach to the host language.

3.1 Programming Abstractions

The core abstraction of our API is a generic type called `DataBag` which models bags in UNION-representation. The complete set of methods can be found in [4]. In the following, we discuss characteristic features by example.

For Comprehensions. Binary operators like `join` and `cross` are missing from the API. Instead, the `DataBag` type implements the monad operations discussed in Section 2.3. This allows us to write `Select-From-Where` expressions like the join from Section 1 in a declarative way.

```
for {
  email <- emails
  from  <- people
  to    <- people
  if from.email == email.from
  if to.email == email.to
} yield (from, to, email)
```

Folds. Computation on `DataBag` values is allowed only by means of structural recursion. To that end, we expose the `fold` operator from Section 2.2 as well as aliases for commonly used folds (e.g. `count`, `exists`, `minBy`). Counting the number of emails, for example, can be written as follows.

```
val N = emails.fold(0, x => 1, plus) // or
val N = emails.count() // alias for the above
```

Nesting. The grouping operator introduces proper nesting:

```
val ys: DataBag[Group[K, DataBag[A]]] = xs.groupBy(k)
```

The resulting bag contains groups of `values` that share the same `key`. Note that the `values` type is again `DataBag[A]`. This is fundamentally different from Spark, Flink, and Hadoop MapReduce, where the group values have the type `Iterable[A]` or `Iterator[A]`. An ubiquitous support for `DataBag` nesting allows us to hide the complexity of primitives like `groupByKey`, `reduceByKey`, and `aggregateByKey` behind a simple "`groupBy` and `fold`" programming model. For instance, calculating the `idf` term shown in Section 1 can be expressed as follows.

```
val idf = for {
  (t, docs) <- tf.groupBy{ case (_, t, _) => t }
} yield (t, math.log(N / docs.count()))
```

Due to the deep embedding approach we commit to, we can recognize nested `DataBag` patterns like the one above at compile time and rewrite them into more efficient equivalent expressions using primitives like `aggregateByKey`.

Coarse-Grained Parallelism Contracts. Current parallel dataflow APIs provide data-parallelism contracts at the operator level (e.g. `map` for element-at-a-time, `join` for pair-at-a-time, etc.). Emma takes a different approach as its `DataBag` abstraction itself serves as a *coarse-grained* contract for data-parallel computation. The promise Emma gives is to (i) discover all maximal `DataBag` expressions in a quoted code fragment, (ii) rewrite them logically in order to maximize the degree of data-parallelism, and (iii) take a holistic approach while translating them as parallel dataflows. This also allows to transparently insert primitives influencing execution like `broadcast` and `cache` as part of the compilation process.

3.2 Host Language Desiderata

The decision to base our implementation on Scala is motivated by purely pragmatic reasons: (i) Scala supports `for`-comprehensions for user-defined types, (ii) the runtimes we target have Scala APIs, and (iii) lightweight embedding and metaprogramming are enabled through Scala's macro and reflection facilities. In theory, however, any language which satisfies the above requirements can be used as a host-language for a similar compiler pipeline.

4. COMPILER PIPELINE

The basic compilation steps are depicted in Figure 1. At compile time, a Scala macro is used to (i) lift the Scala AST into a suitable intermediate representation (IR), (ii) apply logical rewrites that maximize data-parallelism to the Emma IR (see Section 4.1), and (iii) compile the result as a binary driver program with staged comprehensions. At runtime, Scala's reflection API is used to (iv) translate the

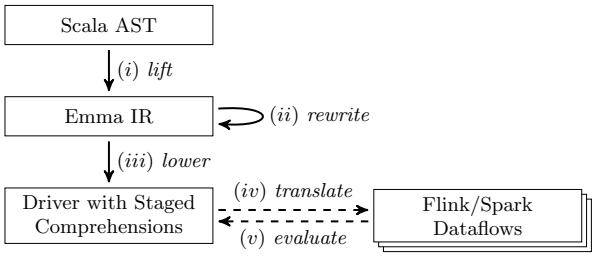


Figure 1: Basic Compiler Pipeline. Solid arrows represent static, and dashed dynamic (JIT) compilation.

staged comprehensions into the API of the targeted parallel dataflow engine, and to (*v*) evaluate those and feed the results back in the driver. In the rest of this section, we focus on some aspects of steps (*ii*) and (*iv*). The goal is to illustrate optimization techniques widely adopted by the database community on top of bag comprehensions as a model for program manipulation through metaprogramming. As in Section 3, more information can be found in [4].

4.1 Logical Optimizations

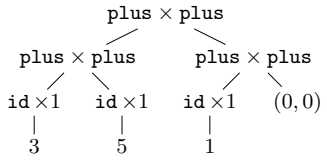
In this section, we show how two algebraic laws known as *banana-split* [6] and *fold-build fusion* [10] facilitate a logical rewrite upon generated groups. The rewrite transparently inserts partial aggregates whenever possible and thereby removes expensive group materializations.

Rewrite Candidates. Candidates for this rewrite are `groupBy` terms where (*i*) all occurrences of the group `values` are consumed by a `fold`, and (*ii*) these folds do not have data dependencies. When the optimization is triggered, the `groupBy` is replaced by an `aggBy` operator which fuses the group construction performed by the `groupBy` together with the subsequent `fold` applications on the group values. In terms of the APIs discussed in Section 1, this corresponds to replacing `groupBy` with `reduceByKey` whenever possible.

Banana Split. The banana split law generalizes the machinery behind loop fusion for arbitrary structural recursion. Informally, it states that a pair of folds can be rewritten as a fold over pairs. For example, the two folds below calculate a sum and a count over the same input collection.

```
val sum   = xs.fold(0, id, plus)
val count = xs.fold(0, x => 1, plus)
```

Those can be substituted by a single `fold` which operates by pairwise application of the original substitution functions. As a result, a pair containing the results of the two original folds can be computed in a single pass. To illustrate the idea, take a look at the application tree of the resulting pairwise `fold` over $\{\{3, 5, 1\}\}$.



Fold-Build Fusion. The second law enables a rewrite which in functional programming languages is commonly known as *cheap deforestation*. Intuitively, the law states that an operation that *constructs* a bag can be *fused together* (or in database terms – it can be *pipelined*) with a subsequent `fold` over the constructed value. To illustrate why `groupBy` can be seen as a build operator, consider the following naive definition of `tf.groupBy(...)`.

```
val tfGrp = for {
  t <- tf.map{ case (_, t, _) => t }.distinct()
} yield {
  val docs = tf.withFilter(_._2 == t)
  (t, docs)
}
```

We bind `t` over the set of distinct terms contained in `tf`, and pair each binding with its corresponding `docs` derived with a filter over `tf`. Substituting `tfGrp` in the code from Section 3.1, we consume the grouped result as follows.

```
val idf = for {
  (t, docs) <- tfGrp
} yield (t, math.log(N / docs.count()))
```

Knowing that the group values (`docs`) are used only in the context of a `count` application, we can build upon referential transparency and move the application up to `tfGrp`.

```
val tfAgg = for {
  t <- tf.map{ case (_, t, _) => t }.distinct()
} yield {
  val dfrq = tf.withFilter(_._2 == t).count()
  (key, dfrq)
}
```

The fold-build fusion law tells us that `withFilter` and the `count` can be fused to a single `fold` as follows.

```
val dfrq = tf.fold(0, if _._2 == t 1 else 0, plus)
```

The rewritten definition pairs terms directly with the aggregated document frequencies `dfrq` instead of materializing `docs`. To compensate for the rewrite at the consumer site, we remove the original `count` and directly refer to `dfrq`.

```
val idf = for {
  (term, dfrq) <- trAgg
} yield (term, math.log(N / dfrq))
```

In practice, we directly substitute `groupBy` with `aggBy`, but the comprehension model enables better understanding and reasoning about the soundness of such rewrites.

Other logical rewrites such as flattening [22] can also be performed at this step in order to maximize data-parallelism.

4.2 Dataflow Generation

As a result of the logical optimizations from Section 4.1, we obtain a modified AST for the original program. The JIT compiler then offloads the identified maximal `DataBag` terms to a parallel execution engine like Spark or Flink as promised. To illustrate the challenge here, consider again the (already normalized) join comprehension from Section 3.

```
for {
  email <- emails
  from  <- people
  to    <- people
  if from.email == email.from
  if to.email == email.to
} yield (from, to, email)
```

According to the semantics from Section 2.3 the above comprehension is equivalent to the following functional term.

```
emails.flatMap(email =>
  people.flatMap(from =>
    people
      .withFilter(to => from.email == email.from)
      .withFilter(to => to.email == email.to)
      .map(to => (from, to, email))))
```

A blunt translation approach would be to take the functional form and substitute the collection type from `DataBag` to either `RDD` or `DataSet`. This is not a good strategy for two reasons. First, it only allows for data-parallelism over the top-most collection (in this case `emails`), while everything else must be fully replicated across the cluster. Second, local evaluation amounts to brute-force nested loops.

Fortunately, the database community has solutions for this kind of problems. In relational databases, SQL queries are represented as expression trees called *logical plans*. For `Select-From-Where` queries, the leafs of the tree denote input relations, while inner nodes denote binary `join` applications. Due to the associativity and commutativity of `join`, however, multiple logical plans exist for the same query. Query optimizers use dynamic programming to pick an optimal plan based on a data-driven cost model.

To bridge the gap between the functional and the database world, we revise some old ideas from Grust [12] in the context of parallel dataflows. Observe that the abstract semantics of an (equi-)`join` operator can be defined in terms of a comprehension.

```
def join(k1, k2)(xs, ys) = for {
  x <- xs
  y <- ys
  if k1(x) == k2(y)
} yield (x, y)
```

Armed with that insight, we can devise recursive procedures that transform comprehension expressions into functionally closed cascades of joins. A bottom-up procedure may begin by joining `email` and `from`, and continue recursively until all generators are eliminated. This technique is known as "Selinger-style query optimization" [20].

```
// intermediate result (closed functional term)
val ir = join(_.from, _.email)(emails, people)
// final result (residual comprehension term)
for {
  (email, from)  <- ir
  to             <- people
  if to.email == email.to
} yield (from, to, email)
```

A top-down procedure, on the other side, may first split the original comprehensions in two, join their results, and continue recursively until all comprehensions have one generator. This technique is known as "Volcano-style query optimization" [11].

```
// intermediate result (residual comprehension term)
val ir = for {
  email <- emails
  from  <- people
  if from.email == email.from
} yield (from, email)
// final result (closed functional term)
join(_._2.to, _.email)(ir, people)
```

Either way, after the rewrite procedure terminates, we end up with an algebraic variant of the original comprehension based on second order combinators (like `join` and `flatMap`) in direct correspondence with the primitives offered by the targeted dataflow APIs. Translation from this form to the concrete target API can be done in a single traversal pass.

5. RELATED WORK

Object-Oriented Databases. A multitude of research has been performed during the OODBs era to bridge the gap between programming languages and database querying. Their primary goal was to make database access from the programming language transparent, focusing mainly on object persistence. OODBs attempted to support *all* the power of the host programming language (e.g., [16]) but did not succeed, mainly due to the complexity of such an undertaking. Another approach towards solving the impedance mismatch (i.e., using a string-quoted language like SQL within Java, C, Ruby, etc.) was PASCAL/R [14] – a first step towards integrated querying facility later resurrected by LINQ [18]. XQuery [7] (with scripting extensions for executing loops) provides a completely integrated programming and querying language that supports a flavor of comprehension syntax. However, XQuery is tied to the XML data model whereas our approach is oblivious to the data model.

More Recent Attempts. Similar to LINQ, Ferry [13] is a comprehensions-based programming language that facilitates database-supported execution of entire programs. To be evaluated, LINQ and Ferry programs both map into an intermediate form suitable for execution on SQL:1999-capable relational database systems. Similar to XQuery, Ferry, and LINQ, Emma is a comprehensions-based language, but targets JVM-based parallel dataflow engines. Moreover, the Emma compiler pipeline relies on a holistic view of the quoted code which simultaneously considers all comprehended terms.

Algebraic Approach. In spirit, the ideas presented here follow a line of work exploring the intersection between type theory, functional programming, and data management. The starting point is the work by Buneman et al. [8], who showed that monads can be used to generalize nested relational algebra to different types of collections and complex objects. The implications of using insert or union representation for parallel processing have been explored by Suciu and Wong [21] and more recently by Steele [15]. The approach of "comprehending dataflows" presented in Section 4.2 draws on ideas originally proposed by Grust [12]. Our aim is to highlight the importance of this line of work in facilitating seamless integration of declarative, data-parallel collection processing into a general-purpose host language.

To the best of our knowledge, Emma is the first DSL for parallel data analysis that promotes comprehensions as first-class citizen. The metaprogramming approach allows for hiding low-level API primitives behind a declarative, ubiquitous abstraction (`DataBag`) and maintaining competitive performance through a series of holistic optimizations.

6. SUMMARY & OUTLOOK

We illustrated common abstraction leaks shared between distributed collection processing APIs offered by state-of-the-art parallel dataflow engines. We argued that such leaks

hinder the adoption of these APIs as a basic tool for advanced data analysis due to the burden imposed on the programmer. To alleviate this burden, we promoted the use of monad comprehensions over bags in union representation as first-class citizens in embedded DSLs. As a proof-of-concept, we presented Emma – a Scala DSL that offers (*i*) declarative dataflow syntax, and (*ii*) advanced rewrite-based optimizations. Emma programs can thereby employ dataflow engines such as Flink and Spark as transparent co-processors.

Embedding Approach. The ideas behind Emma require embedding in a general-purpose host language. While we currently rely on quotation-based embedding (as advocated by Lisp), type-based embedding (as advocated by LMS [19]) is an appealing alternative due to a more flexible metaprogramming infrastructure. Investigating and comparing the practical benefits of the two approaches poses an interesting research question.

Future Work. At the moment, the comprehensions discovered by the the Emma compiler are translated into target-engine dataflows in a heuristic manner. This approach is sub-optimal with respect to the optimization potential that can be harvested at runtime. To this end, we are currently working on an optimizer that will statically pre-compute interesting physical properties *across* dataflows and use this information in the JIT compilation phase. Finally, we are developing a linear algebra API that will allow for mixed use of bags, matrices, and vectors. Interested readers can learn more about Emma at our project webpage [2].

Acknowledgements. The authors would like to thank Sebastian Breß, Lukas Egger, Zachary Ives, and Tamara Mendt for reviewing a draft version of this article and providing insightful improvement suggestions.

This work has been supported through grants by the German Science Foundation (DFG) as FOR 1306 Stratosphere, by the German Ministry for Education and Research as Berlin Big Data Center BBDC (ref. 01IS14013A), by the European Commission as Proteus (ref. 687691) and Streamline (ref. 688191), and by Oracle Labs.

7. REFERENCES

[1] Cascading Project. `http://www.cascading.org/`.

[2] Emma Language. `http://www.emma-language.org/`.

[3] A. Alexandrov, R. Bergmann, S. Ewen, J. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The Stratosphere platform for big data analytics. *VLDB J.*, 23(6):939–964, 2014.

[4] A. Alexandrov, A. Kunft, A. Katsifodimos, F. Schüler, L. Thamsen, O. Kao, T. Herb, and V. Markl. Implicit parallelism through deep language embedding. In *SIGMOD Conference*, pages 47–61. ACM, 2015.

[5] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C. Kanne, F. Özcan, and E. J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 4(12):1272–1283, 2011.

[6] R. Bird and O. de Moor. *The Algebra of Programming*. Prentice Hall, 1997.

[7] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. Xquery 1.0: An XML query language, 2002.

[8] P. Buneman, S. A. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1):3–48, 1995.

[9] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 2008.

[10] A. J. Gill and S. L. P. Jones. Cheap deforestation in practice: An optimizer for haskell. In *IFIP Congress (1)*, pages 581–586, 1994.

[11] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218. IEEE Computer Society, 1993.

[12] T. Grust. *Comprehending Queries (PhD Thesis)*. PhD thesis, Universität Konstanz, 1999.

[13] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. FERRY: database-supported program execution. In *SIGMOD Conference*, pages 1063–1066. ACM, 2009.

[14] M. Jarke and J. W. Schmidt. Query processing strategies in the PASCAL/R relational database management system. In *SIGMOD Conference*, pages 256–264. ACM Press, 1982.

[15] G. L. S. Jr. Organizing functional code for parallel execution or, foldl and foldr considered slightly harmful. In *ICFP*, pages 1–2. ACM, 2009.

[16] C. Lamb, G. Landis, J. A. Orenstein, and D. Weinreb. The objectstore database system. *Commun. ACM*, 34(10):50–63, 1991.

[17] J. Lambek. Least fixpoints of endofunctors of cartesian closed categories. *Mathematical Structures in Computer Science*, 3(2):229–257, 1993.

[18] E. Meijer, B. Beckman, and G. M. Bierman. LINQ: reconciling object, relations and XML in the .net framework. In *SIGMOD Conference*, page 706. ACM, 2006.

[19] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *GPCE*, pages 127–136. ACM, 2010.

[20] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD Conference*, pages 23–34. ACM, 1979.

[21] D. Suciu and L. Wong. On two forms of structural recursion. In *ICDT*, volume 893 of *Lecture Notes in Computer Science*, pages 111–124. Springer, 1995.

[22] A. Ulrich and T. Grust. The flatter, the better: Query compilation based on the flattening transformation. In *SIGMOD Conference*, pages 1421–1426. ACM, 2015.

[23] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In E. M. Nahum and D. Xu, editors, *HotCloud*. USENIX, 2010.

Technical Perspective: Incremental Knowledge Base Construction Using DeepDive

Alon Halevy
Recruit Institute of Technology

Imagine the task of creating a database of all the high-quality specialty cafes around the world so you never have to settle for an imperfect brew. There are plenty of online sources with content relevant to your envisioned database. Cafes may be featured in well-respected coffee publications such as sprudge.com or baristamagazine.com. Data of more fleeting nature may pop up when your coffee-savvy friends note their location by checking in on Facebook or tweeting. Naturally, there is a plethora of books that studied cafes around the world in even more detail.

The task of creating such a database is surprisingly hard. You would begin by deciding which attributes of cafes the database should model. Attributes such as address and opening hours would be obvious even to a novice, but you will need to consult a coffee expert who will suggest more refined attributes such as roast profile and brewing methods. The next step is to write programs that will extract structured data from these heterogeneous sources, distinguish the good extractions from the bad ones, and combine extractions from different sources to create tuples in your database. As part of the data cleaning process, you might want to employ crowd workers to confirm details such as opening hours that were extracted from text or whether two mentions of cafes in text refer to the same cafe in the real world. In the extreme case, you might even want to send someone out to a cafe to check on some of the details in person. The process of creating the database is iterative because your extraction techniques will be refined and because the cafe scene changes frequently.

This Knowledge Base Construction task (KBC) has been an ongoing challenge and an inspiration for deep collaborations between researchers and practitioners in multiple fields, including data management and integration, information extaction, machine learning, natural language understanding, and probabibilistic reasoning. Aside from the compelling application detailed above, the problem arises in many other settings. For example, imagine the task of creating a database (or ontology) of all job categories for a job-search site, or compiling a database of dishes served in Tokyo cafes for the purpose of restaurant search or trend analysis.

The paper you are about to read is a prime example of groundbreaking work in the area of KBC. DeepDive, a project led by Chris Ré at Stanford, is an end-to-end system for creating knowledge bases. The input to DeepDive is a set of data sources such as text documents, PDF files, and structured databases. DeepDive extracts, cleans and integrates data from the multiple sources and produces a database in which a probability is attached to every tuple. A user interacts with DeepDive in a high-level declarative language (DDLog) that uses predicates that are defined with functions in Python. The rules in DDLog specify how to extract entities, mentions of entities, and relationships from the data sources and the details of the extractions are implemented in Python. DeepDive then uses an efficient statistical inference engine to compute probabilities of the facts in the database. Using a set of tools that facilitate examining erroneous extractions, the user can iteratively adjust the DDLog rules to obtain the desired precision and recall. DeepDive has already been used in several substantial applications, such as detecting human trafficking and creating a knowledge base for Paleobiologists with quality higher than human volunteers.

This particular paper focuses on the incremental aspects of DeepDive. As noted in several applications of the system, knowledge base construction is an iterative process. As the user goes through the process of building the knowledge base, the rules used to extract the data change are modified and of course, the underlying data may change as well. The paper describes the algorithms used in DeepDive to efficiently recompute the facts in the knowledge base and to efficiently recompute the probabilities of facts coming from the inference engine. The results show that efficient incremental computation can make a substantial difference in the usability of a KBC system.

Like with any deep scientific endeavour, there is much more research to be done (and for now, too many coffee lovers need to settle for over-roasted coffee because the database of cafes does not exist yet). We hope that reading this paper will inspire you to work on the KBC problem and hopefully to contribute ideas from far-flung fields.

DeepDive: Declarative Knowledge Base Construction

Christopher De Sa Alex Ratner Christopher Ré Jaeho Shin
Feiran Wang Sen Wu Ce Zhang
Stanford University
{cdesa, ajratner, chrismre, jaeho, feiran, senwu, czhang}@cs.stanford.edu

ABSTRACT

The dark data extraction or knowledge base construction (KBC) problem is to populate a SQL database with information from unstructured data sources including emails, webpages, and pdf reports. KBC is a long-standing problem in industry and research that encompasses problems of data extraction, cleaning, and integration. We describe DeepDive, a system that combines database and machine learning ideas to help develop KBC systems. The key idea in DeepDive is that statistical inference and machine learning are key tools to attack classical data problems in extraction, cleaning, and integration in a unified and more effective manner. DeepDive programs are declarative in that one cannot write probabilistic inference algorithms; instead, one interacts by defining features or rules about the domain. A key reason for this design choice is to enable domain experts to build their own KBC systems. We present the applications, abstractions, and techniques of DeepDive employed to accelerate construction of KBC systems.

1. INTRODUCTION

The process of populating a structured relational database from unstructured sources has received renewed interest in the database community through high-profile start-up companies (e.g., Tamr and Trifacta), established companies like IBM's Watson [5, 13], and a variety of research efforts [9,24,29,38,43]. At the same time, communities such as those of natural language processing and machine learning are attacking similar problems under the name *knowledge base construction* (KBC) [3, 11, 20]. While different communities place differing emphasis on the extraction, cleaning, and integration phases, all seem to be converging toward a common set of techniques that include a mix of data processing, machine learning, and engineers-in-the-loop.

The ultimate goal of KBC is to obtain high-quality structured data from unstructured information. The output databases produced are richly structured with tens of different entity types in complex relationships. Typically, quality is assessed using two complementary measures: precision (how often a claimed tuple is correct) and recall (of the possible tuples to extract, how many are actually extracted). These systems can ingest massive numbers of documents–far outstripping the document counts of even well-funded human curation efforts. Industrially, KBC systems are constructed by skilled engineers in a months-long (or longer) process–not a one-shot algorithmic task. Arguably, the most important question in such systems is how to best use skilled engineers' time to rapidly improve data quality. In its full generality, this question spans a number of areas in computer science, including programming languages, systems, and HCI. We focus on a narrower question, with the axiom that *the more rapidly the programmer moves through the KBC construction loop, the more quickly she obtains high-quality data.*

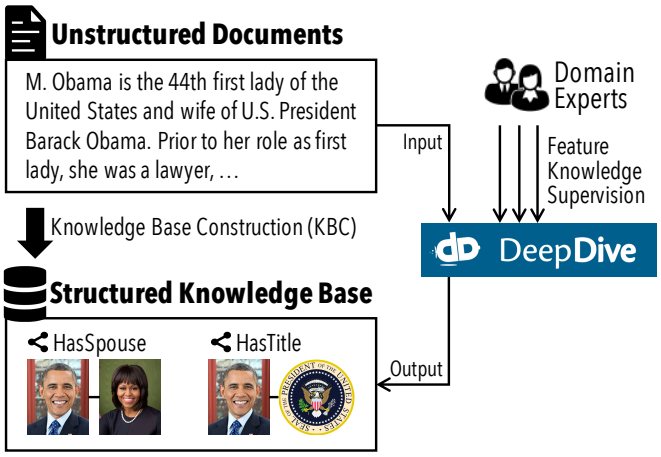


Figure 1: Knowledge Base Construction (KBC) is the process of populating a structured relational knowledge base from unstructured sources. DeepDive is a system aimed at facilitating the KBC process by allowing domain experts to integrate their domain knowledge without worrying about algorithms.

This paper presents DeepDive, our open-source engine for knowledge base construction. DeepDive's language and execution model are similar to other KBC systems: DeepDive uses a high-level declarative language [9, 29, 31]. From a database perspective, DeepDive's language is based on SQL. From a machine learning perspective, DeepDive's language is based on Markov Logic [10, 31]: DeepDive's language inherits Markov Logic Networks' (MLN's) formal semantics. Moreover, it uses a standard execution model for such systems [9, 29, 31] in which programs go through two main phases: *grounding*, in which one evaluates a sequence of SQL queries to produce a data structure called a *factor graph* that describes a set of random variables and how they are correlated. Essentially, every tuple in the database or result of a query is a random variable (node) in this factor graph. The *inference* phase takes the factor graph from the grounding phase and performs statistical inference using standard techniques, e.g., Gibbs sampling [44, 47]. The output of inference is the marginal probability of every tuple in the database. As with Google's Knowledge Vault [11] and others [32], DeepDive also produces marginal probabilities that are *calibrated*: if one examined all facts with proba-

The original version of this article was published in PVLDB 2015.

`http://deepdive.stanford.edu`

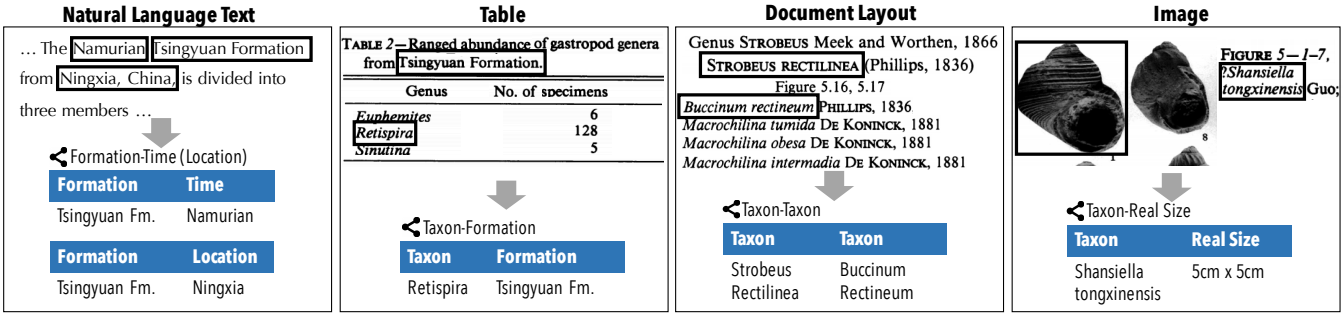


Figure 2: Example KBC Application Built with DeepDive.

bility 0.9, we would expect approximately 90% of these facts to be correct. To calibrate these probabilities, DeepDive estimates (i.e., learns) parameters of the statistical model from data. Inference is a subroutine of the learning procedure and is the critical loop. Inference and learning are computationally intense (hours on 1TB RAM/48-core machines).

In our experience with DeepDive, we found that KBC is an iterative process. In the past few years, DeepDive has been used to build dozens of high-quality KBC systems by a handful of technology companies, a number law enforcement agencies via DARPA's MEMEX, and scientists in fields such as paleobiology, drug repurposing, and genomics. Recently, we compared the quality of a DeepDive system's extractions to those provided by human volunteers over the last ten years for a paleobiology database, and we found that the DeepDive system had higher quality (both precision and recall) on many entities and relationships. Moreover, on all of the extracted entities and relationships, DeepDive had no worse quality [34]. Additionally, the winning entry of the 2014 TAC-KBC competition was built on DeepDive [1]. In all cases, we have seen the process of developing KBC systems is iterative: quality requirements change, new data sources arrive, and new concepts are needed in the application. This led us to develop a set of techniques to make not only the execution of statistical inference and learning efficient, but also the entire pipeline incremental in the face of changes both to the data and to the DeepDive program.

This paper aims at giving a broad overview of DeepDive. The rest of the paper is organized as follows. Section 2 describes the KBC process, its scientific applications, and technical challenges. Section 3 presents our language for modeling KBC systems inside DeepDive. We discuss the different techniques in Section 4 and give pointers for readers who are interested in each technique.

2. APPLICATIONS AND CHALLENGES

Knowledge base construction (KBC) is the process of populating a knowledge base with facts extracted from unstructured data sources such as text, tabular data expressed in text and in structured forms, and even maps and figures. In *sample-based science* [34], one typically assembles a large number of facts (typically from the literature) to understand macroscopic questions, e.g., about the amount of carbon in the Earth's atmosphere throughout time, the rate of extinction of species, or all the drugs that interact with a particular gene. To answer such questions, a key step is to construct a high-quality knowledge base, and some sciences have undertaken decade-long sample collection efforts, e.g., PaleoDB.org and PharmaGKB.org.

In parallel, KBC has attracted interest from industry [13, 49] and academia [2, 3, 6, 12, 21, 23, 29, 32, 35, 38, 40, 45]. To understand the common patterns in KBC systems, we are actively collaborating with scientists from a diverse set of domains, including geology [46], paleontology [34], pharmacology for drug repurposing, and others. We first describe one KBC application we built, called PaleoDeepDive, then present a brief description of other applications built with similar purposes, and then finally discuss the challenges.

2.1 PaleoDB and PaleoDeepDive

Paleontology is based on the description and biological classification of fossils, an enterprise that has been recorded in and an untold number of scientific publications over the past four centuries. One central task for paleontology is to construct a knowledge base about fossils from scientific publications, and an existing knowledge base compiled by human volunteers has greatly expanded the intellectual reach of paleontology and led to many fundamental new insights into macroevolutionary processes and the nature of biotic responses to global environmental change. However, the current process of using human volunteers is usually expensive and time-consuming. For example, PaleoDB, one of the largest such knowledge bases, took more than 300 professional paleontologists and 11 human years to build over the last two decades, resulting in `PaleoDB.org`. To get a sense of the impact of this database on this field, at the time of writing, this dataset has contributed to 205 publications, of which 17 have appeared in *Nature* or *Science*.

This provided an ideal test bed for our KBC research. In particular, we constructed a prototype called PaleoDeepDive [34] that takes in PDF documents. This prototype attacks challenges in optical character recognition, natural language processing, information extraction, and integration. Some statistics about the process are shown in Figure 3. As part of the validation of this system, we performed a double-blind experiment to assess the quality of the system versus the PaleoDB. We found that the KBC system built on DeepDive has achieved comparable—and sometimes better—quality than a knowledge base built by human volunteers over the last decade [34]. Figure 3 illustrates the accuracy of the results in PaleoDeepDive.

2.2 Beyond Paleontology

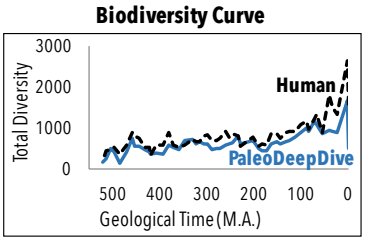
The success of PaleoDeepDive motivates a series of other KBC applications in a diverse set of domains including both natural and social sciences. Although these applications focus on very different types of KBs, they are usually built in a way similar to PaleoDeepDive. This similarity across applications motivate our study of building DeepDive as a unified framework to support these diverse applications.

Quality of PaleoDeepDive

Relation	PDD # Ext.	PDD Accuracy	Human Accuracy
Taxon-Taxon	27 M	97%	92%
Taxon-Fm.	4 M	96%	84%
Fm.-Time	3 M	92%	89%
Fm.-Location	5 M	94%	90%

Quality of Other Applications

Applications	F1 Score
Human-Trafficking	72%
TAC-KBP	34%
Medical Genetics	53%
Pharmacogenomics	57%



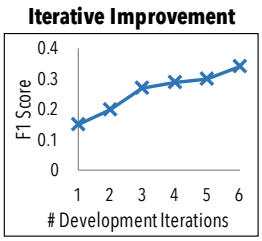


Figure 3: Quality of KBC systems built with DeepDive. On many applications, KBC systems built with DeepDive achieves comparable (and sometimes better) quality than professional human volunteers, and leads to similar scientific insights on topics such as biodiversity. This quality is achieved by iteratively integrating diverse sources of data- often quality scales with the amount of information we enter into the system.

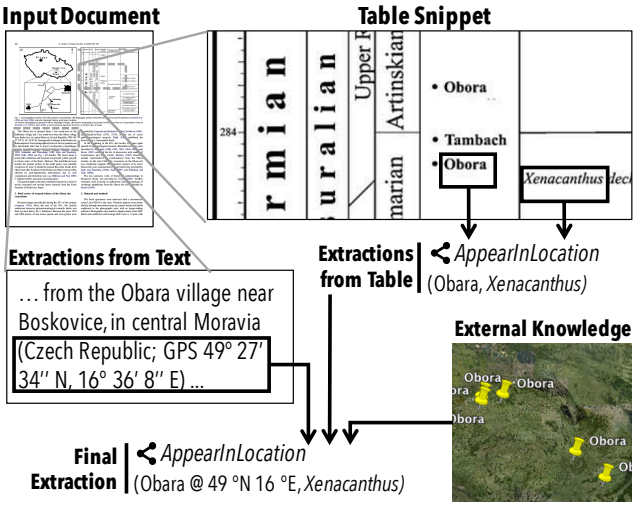


Figure 4: One challenge of building high-quality KBC systems is dealing with diverse sources jointly to make predictions. In this example page of a Paleontology journal article, information extracted from tables, text, and external structured knowledge bases are all required to reach the final extraction. This problem becomes even more challenging when many extractors are not 100% accurate, thus motivating the joint probabilistic inference engine inside DeepDive.

Human-Trafficking. MEMEX is a DARPA program that explores how next generation search and extraction systems can help with real-world use cases. The initial application is the fight against human trafficking. In this application, the input is a portion of the publicly-indexed and "dark" web in which human traffickers are likely to (surreptitiously) post supply and demand information about illegal labor, sex workers, and more. DeepDive processes such documents to extract evidential data such as names, addresses, phone numbers, job types, job requirements, information about rates of service, etc. Some of these data items are difficult for trained human annotators to accurately extract and have never been previously available, but DeepDive-based systems have high accuracy (Precision and Recall in the 90s, which may surpass that of non-experts). Together with provenance information, such structured, evidential data are then passed on to both other collaborators on the MEMEX program as well as law enforcement for analysis and consumption in operational applications. MEMEX has been featured extensively in the media and is supporting actual investigations. For example, every human trafficking investigation pursued by the Human Trafficking Response Unit in New York City now involves MEMEX, for which DeepDive is the main extracted data provider. In addition, future use cases such as applications in the war on terror are under active consideration.

Medical Genetics. The body of literature in life sciences has been growing at an accelerating speed, to the extent that it has been unrealistic for scientists to perform research solely based on reading and/or keyword search. Numerous manually-curated structured knowledge bases are likewise unable to keep pace with exponential increases in the number of publications available online. For example, OMIM is an authoritative database of human genes and mendelian genetic disorders which dates back to the 1960s, and so far contains about 6,000 hereditary diseases or phenotypes, growing at a rate of roughly 50 records / month for many years. Conversely, almost 10,000 publications were deposited into PubMed Central per month last year. In collaboration with Prof. Gill Bejerano at Stanford, we are developing DeepDive applications to create knowledge bases in the field of medical genetics. Specifically, we use DeepDive to extract mentions of genes, gene variants, and phenotypes from the literature, and statistically infer their relationships, presently being applied to clinical genetic diagnostics & reproductive counseling.

Pharmacogenomics. Understanding the interactions of chemicals in the body is key for drug discovery. However, the majority of this data resides in the biomedical literature and cannot be easily accessed. The Pharmacogenomics Knowledgebase is a high quality database that aims to annotate the relationships between drugs, genes, diseases, genetic variation, and pathways in the literature. With the exponential growth of the literature, manual curation requires prioritization of specific drugs or genes in order to stay up to date with current research. In collaboration with Emily Mallory and Prof. Russ Altman [27] at Stanford, we are developing DeepDive applications in the field of pharmacogenomics. Specifically, we use DeepDive to extract relations between genes, diseases, and drugs in order to predict novel pharmacological relationships.

TAC-KBP. TAC-KBP is a NIST-sponsored research competition where the task is to extract common properties of people and organizations (e.g., age, birthplace, spouses, and shareholders) from a 1.3 million newswire and web documents – this task is also termed Slot Filling. In the 2014 evaluation, 31 US and international teams participated in the competition, including a solution based on DeepDive from Stanford [1]. The DeepDive based solution achieved the highest precision, recall, and F1 among all submissions.

2.3 Challenges

On all the applications mentioned above, KBC systems built with DeepDive achieved high quality as illustrated in Figure 3. Achiev-

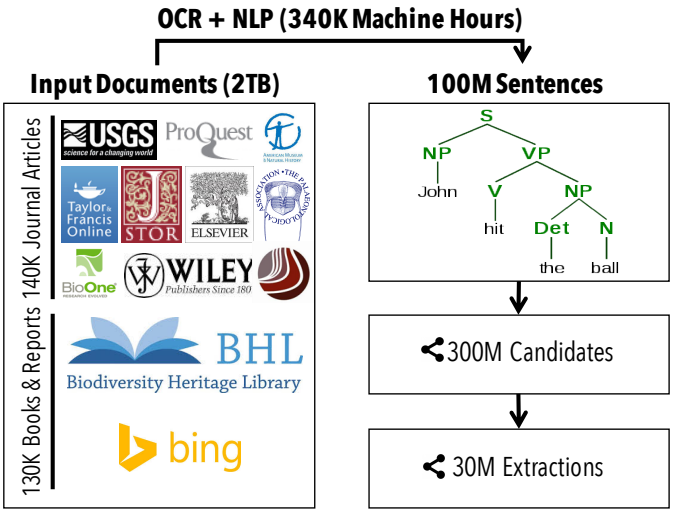


Figure 5: Another challenge of building high-quality KBC systems is that one usually needs to deal with data at the scale of tera-bytes. These data are not only processed with traditional relational operations, but also operations involving machine learning and statistical inference. Thus, DeepDive consists of a set of techniques to speed up and scale up inference tasks involving billions of correlated random variables.

ing this high quality level requires that we solve a set of challenges.

Joint Statistical Inference. We have found that text is often not enough: often, the data that are interesting to scientists are located in the tables, figures, and images of articles. For example, in geology, more than 50% of the facts that we are interested in are buried in tables [14]. For paleontology, the relationship between taxa, as known as taxonomy, is almost exclusively expressed in section headers [34]. For pharmacology, it is not uncommon for a simple diagram to contain a large number of metabolic pathways. To build a KBC system with the quality that scientists will be satisfied with, we need to deal with these diverse sources of input. Additionally, external sources of information (other knowledge bases) typically contain high-quality signals (e.g., Freebase and Macrostrat). Leveraging these sources in information extraction is typically not studied in the classical information extraction context. To perform high-quality and high-coverage knowledge extraction, one needs a model that is able to ingest whatever sources present themselves, *opportunistically*—that is, a model which is not tied solely to text but can handle more general extraction and integration from multiple source types *simultaneously*.

This challenge becomes more serious when the information from different sources are all *noisy*. Take Figure 4 for example, to reach the extraction that the genus *Xenacanthus* appears in the location of the name *Obara*, the extraction system needs to consult extractions from text, tables, and external structured sources. These extractions are often associated with a confidence score. To join these extractions with difference confidence level together, one needs a principled framework. The DeepDive approach to this challenge is based on a Bayesian probabilistic approach. DeepDive treats all these information sources as one joint probabilistic inference problem, with all predictions modeled as random variables within a factor graph model. This probabilistic framework ensures all facts that are produced by DeepDive are associated with a marginal probability. These marginal probabilities are meaningful in DeepDive, i.e., the empirical accuracy that one should expect for the extracted mentions, and provide a guideline to the developer to improve the KBC system built using DeepDive. In Section 3, we present a declarative language inside DeepDive to help developers specify a joint statistical inference problem easily.

Scalability and Efficiency. Performance is also a major challenge. In our KBC systems using DeepDive, we may need to perform inference and learning on billions of highly correlated random variables. For example, Figure 5 illustrates the data flow of PaleoDeepDive. The input to PaleoDeepDive contains nearly 300K journal articles and books, whose total size exceeds 2TB. These raw inputs are then processed with tools such as OCR and linguistic parsing, which are computationally expensive and may take hundreds of thousands of machine hours. The outputs of these tools are then used by DeepDive to construct factor graphs which contain more than 300 million variables as candidates for predictions (where over 30 million of these variables have probability $\geqslant 0.9$ and are thus output as final predictions). Therefore, one of our technical focus areas has been to speed up probabilistic inference [30, 31, 33, 47, 48]. In Section 4, we briefly describe these techniques and provide pointers to readers who are interested in further details.

3. KBC USING DEEPDIVE

We describe DeepDive, an end-to-end framework for building KBC systems with a declarative language.

3.1 Definitions for KBC Systems

The *input* to a KBC system is a heterogeneous collection of unstructured, semi-structured, and structured data, ranging from text documents to existing but incomplete KBs. The *output* of the system is a relational database containing relations extracted from the input and put into an appropriate schema. Creating the knowledge base may involve extraction, cleaning, and integration.

EXAMPLE 3.1. *Figure 6 illustrates a new running example: a knowledge base with pairs of individuals that are married to each other. The input to the system is a collection of news articles and an incomplete set of married persons; the output is a KB containing pairs of person that are asserted to be married by the input sources. A KBC system extracts linguistic patterns, e.g., "... and his wife ..." between a pair of mentions of individuals (e.g., "Barack Obama" and "M. Obama"). Roughly, these patterns are then used as features in a classifier deciding whether this pair of mentions indicates that they are married (in the* HasSpouse*) relation.*

We adopt standard terminology from KBC, e.g., ACE. There are four types of objects that a KBC system seeks to extract from input documents, namely *entities*, *relations*, *mentions*, and *relation mentions*. An **entity** is a real-world person, place, or thing.

http://www.freebase.com/
http://macrostrat.org/

There is a justification for probabilistic reasoning as Cox's theorem asserts (roughly) that if one uses numbers as degrees of belief, then one must either use probabilistic reasoning or risk contradictions in one's reasoning system, i.e., a probabilistic framework is the only sound system for reasoning in this manner. We refer the reader to Jaynes [19].
http://www.itl.nist.gov/iad/mig/tests/ace/2000/

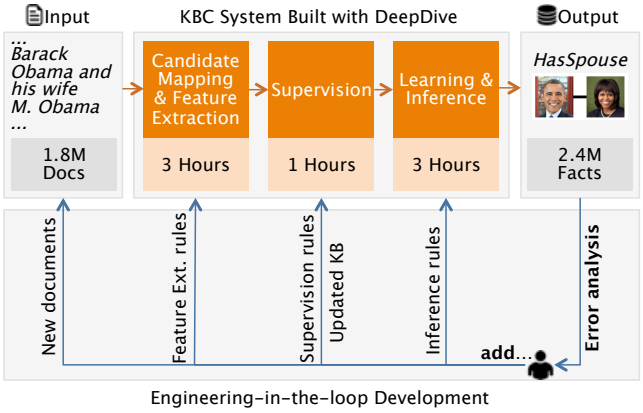


Figure 6: A KBC system takes as input unstructured documents and outputs a structured knowledge base. The runtimes are for the TAC-KBP competition system. To improve quality, the developer adds new rules and new data with error analysis conducted on the result of the current snapshot of the system. DeepDive provides a declarative language to specify each type of different rules and data, and techniques to incrementally execute this iterative process.

For example, "Michelle_Obama_1" represents the actual entity for a person whose name is "Michelle Obama"; another individual with the same name would have another number. A **relation** associates two (or more) entities, and represents the fact that there exists a relationship between the participating entities. For example, "Barack_Obama_1" and "Michelle_Obama_1" participate in the HasSpouse relation, which indicates that they are married. These real-world entities and relationships are described in text; a **mention** is a span of text in an input document that refers to an entity or relationship: "Michelle" may be a mention of the entity "Michelle_Obama_1." A **relation mention** is a phrase that connects two mentions that participate in a relation such as "(Barack Obama, M. Obama)". The process of mapping mentions to entities is called *entity linking*.

3.2 The DeepDive Framework

DeepDive is an end-to-end framework for building KBC systems, as shown in Figure 6. We walk through each phase. DeepDive supports both SQL and datalog, but we use datalog syntax for this exposition. The rules we describe in this section are manually created by the user of DeepDive and the process of creating these rules is application specific.

Candidate Mapping and Feature Extraction. All data in DeepDive is stored in a relational database. The first phase populates the database using a set of SQL queries and user-defined functions (UDFs) that we call *feature extractors*. By default, DeepDive stores all documents in the database in one sentence per row with markup produced by standard NLP pre-processing tools, including HTML stripping, part-of-speech tagging, and linguistic parsing. After this loading step, DeepDive executes two types of queries: (1) *candidate mappings*, which are SQL queries that produce possible mentions, entities, and relations, and (2) *feature extractors* that associate features to candidates, e.g., "... and his wife ..." in Example 3.1.

EXAMPLE 3.2. *Candidate mappings are usually simple. Here, we create a relation mention for every pair of candidate persons in the same sentence (*s*):*

```
(R1) MarriedCandidate(m1, m2) :-
       PersonCandidate(s, m1), PersonCandidate(s, m2).
```

Candidate mappings are simply SQL queries with UDFs that look like low-precision but high-recall ETL scripts. Such rules must be high recall: if the union of candidate mappings misses a fact, DeepDive has no chance to extract it.

We also need to extract features, and we extend classical Markov Logic in two ways: (1) *user-defined functions (UDFs)* and (2) *weight tying*, which we illustrate by example.

EXAMPLE 3.3. *Suppose that* phrase$(m1, m2, sent)$ *returns the phrase between two mentions in the sentence, e.g., "and his wife" in the above example. The phrase between two mentions may indicate whether two people are married. We would write this as:*

```
(FE1) MarriedMentions(m1, m2) :-
        MarriedCandidate(m1, m2), Mention(s, m1),
        Mention(s, m2), Sentence(s, sent)
        weight = phrase(m1, m2, sent).
```

One can think about this like a classifier: This rule says that whether the text indicates that the mentions m1 *and* m2 *are married is* influenced *by the phrase between those mention pairs. The system will infer based on training data its confidence (by estimating the weight) that two mentions are indeed indicated to be married.*

Technically, phrase returns an identifier that determines which weights should be used for a given relation mention in a sentence. If phrase returns the same result for two relation mentions, they receive the *same* weight. We explain weight tying in more detail in Section 3.3. In general, phrase could be an arbitrary UDF that operates in a per-tuple fashion. This allows DeepDive to support common examples of features such as "bag-of-words" to context-aware NLP features to highly domain-specific dictionaries and ontologies. In addition to specifying sets of classifiers, DeepDive inherits Markov Logic's ability to specify rich correlations between entities via weighted rules. Such rules are particularly helpful for data cleaning and data integration.

Supervision. Just as in Markov Logic, DeepDive can use training data or evidence about any relation; in particular, each user relation is associated with an evidence relation with the same schema and an additional field that indicates whether the entry is true or false. Continuing our example, the evidence relation MarriedMentions_Ev could contain mention pairs with positive and negative labels. Operationally, two standard techniques generate training data: (1) hand-labeling, and (2) *distant supervision*, which we illustrate below.

EXAMPLE 3.4. Distant supervision *[17, 28] is a popular technique to create evidence in KBC systems. The idea is to use an incomplete KB of married entity pairs to* heuristically *label (as* `True` *evidence) all relation mentions that link to a pair of married enti-*

For more information, including examples, please see `http://deepdive.stanford.edu`. Note that our engine is built on Postgres and Greenplum for all SQL processing and UDFs. There is also a port to MySQL.

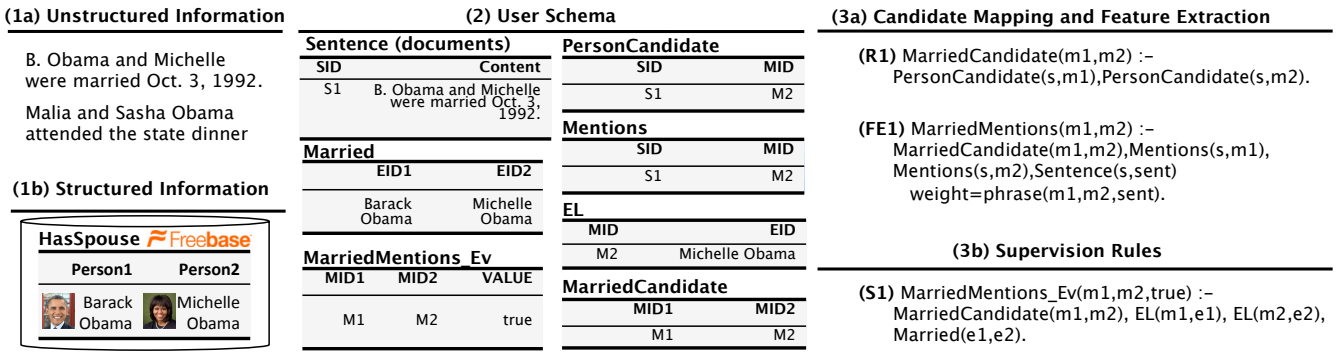


Figure 7: An example KBC system. See Section 3.2 for details.

ties:

$$(S1)\ \texttt{MarriedMentions\_Ev}(\mathrm{m1},\mathrm{m2},\mathrm{true}) :- \texttt{MarriedCandidates}(\mathrm{m1},\mathrm{m2}), \texttt{EL}(\mathrm{m1},\mathrm{e1}), \texttt{EL}(\mathrm{m2},\mathrm{e2}), \texttt{Married}(\mathrm{e1},\mathrm{e2}).$$

Here, `Married` is an (incomplete) list of married real-world persons that we wish to extend. The relation EL is for "entity linking" that maps mentions to their candidate entities. At first blush, this rule seems incorrect. However, it generates noisy, imperfect examples of sentences that indicate two people are married. Machine learning techniques are able to exploit redundancy to cope with the noise and learn the relevant phrases (e.g., "and his wife"). Negative examples are generated by relations that are largely disjoint (e.g., siblings). Similar to DIPRE [4] and Hearst patterns [16], distant supervision exploits the "duality" [4] between patterns and relation instances; furthermore, it allows us to integrate this idea into DeepDive's unified probabilistic framework.

Learning and Inference. In the learning and inference phase, DeepDive generates a factor graph, similar to Markov Logic, and uses techniques from Tuffy [31]. The inference and learning are done using standard techniques (Gibbs Sampling) that we describe below after introducing the formal semantics.

Error Analysis. DeepDive runs the above three phases in sequence, and at the end of the learning and inference, it obtains a marginal probability p for each candidate fact. To produce the final KB, the user often selects facts in which DeepDive is highly confident, e.g., $p > 0.95$. Typically, the user needs to inspect errors and repeat the previous steps, a process that we call *error analysis*. Error analysis is the process of understanding the most common mistakes (incorrect extractions, too-specific features, candidate mistakes, etc.) and deciding how to correct them [36]. To facilitate error analysis, users write standard SQL queries.

3.3 Discussion of Design Choices

We have found the following key aspects of the DeepDive approach that we believe enable non-computer scientists to build sophisticated KBC systems: (1) there is no reference in a DeepDive program to the underlying machine learning algorithms. Thus, DeepDive programs are declarative in a strong sense. Probabilistic semantics provide a way to debug the system independent of the algorithm it uses. (2) DeepDive allows users to write feature extraction code (UDFs) in familiar languages (Python, SQL, and Scala). (3) DeepDive fits into the familiar SQL stack, which allows standard tools to inspect and visualize the data. (4) The user constructs an end-to-end system and then refines the quality of the system in a pay-as-you-go way [26]. In contrast, traditional pipeline-based ETL scripts may lead to user's time and effort over-spent on a specific extraction or integration step–without the ability to evaluate how important each step is for the quality of the end result. Anecdotally, pay-as-you-go leads to more informed decisions about how to improve quality.

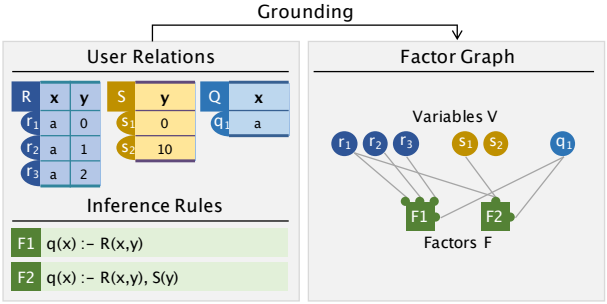


Figure 8: Schematic illustration of grounding. Each tuple corresponds to a Boolean random variable and node in the factor graph. We create one factor for every set of groundings.

4. TECHNIQUES

A DeepDive program is a set of rules with weights specified using the language we described above. During inference, the values of all weights are assumed to be known, while, in learning, one finds the set of weights that maximizes the probability of the evidence. The execution of a DeepDive program consists of two phases, namely grounding and statistical inference and learning. In this section, we briefly describe the techniques we developed in each phase to make DeepDive performant and scalable.

4.1 Grounding

As in Figure 8, DeepDive explicitly constructs a factor graph for inference and learning using a set of SQL queries. A factor graph is a triple $(V, F, \hat{w})$ in which V is a set of nodes that correspond to Boolean random variables, F is a set of hyperedges (for $f \in F$, $f \subseteq V$), and $\hat{w} : F \times \{0,1\}^V \to \mathbb{R}$ is a weight function. In DeepDive, each hyperedge f corresponds to the set of groundings for a rule. In DeepDive, V and F are explicitly created using a set of SQL queries, and this process is called *grounding*.

EXAMPLE 4.1. *Take the database instances and rules in Fig-*

ure 8 as an example: each tuple in relation R*,* S*, and* Q *is a random variable, and* V *contains all random variables. The inference rules* F1 *and* F2 *ground factors with the same name in the factor graph as illustrated in Figure 8. Both* F1 *and* F2 *are implemented as SQL statements in DeepDive.*

Incremental Grounding. Because DeepDive is based on SQL, we are able to take advantage of decades of work on incremental view maintenance. The input to this phase is the same as the input to the grounding phase, a set of SQL queries and the user schema. The output of this phase is how the output of grounding changes, i.e., a set of modified variables ΔV and their factors ΔF. Since V and F are simply views over the database, any view maintenance techniques can be applied to incremental grounding. DeepDive uses the DRED algorithm [15] which handles both additions and deletions. Recall that in DRED, for each relation R_i in the user's schema, we create a *delta relation*, R_i^δ, with the same schema as R_i and an additional column `count`. For each tuple t, $t.\texttt{count}$ represents the number of derivations of t in R_i. On an update, DeepDive updates delta relations in two steps. First, for tuples in R_i^δ, DeepDive directly updates the corresponding counts. Second, a SQL query called a *"delta rule"* is executed which processes these counts to generate modified variables ΔV and factors ΔF. We found that the overhead of DRED is modest and the gains may be substantial, so DeepDive always runs DRED–except on initial load.

4.2 Statistical Inference and Learning

The main task that DeepDive conducts on factor graphs is statistical inference, i.e. determining for a given node what the marginal probability is that this node takes the value 1. Since a node takes value 1 when a tuple is in the output, this process computes the marginal probability values returned to users. In general, computing these marginal probabilities is $\sharp$P-hard [42]. Like many other systems, DeepDive uses Gibbs sampling [37] to estimate the marginal probability of every tuple in the database.

Efficiency and Scalability. There are two components to scaling statistical algorithms: *statistical efficiency*, roughly how many steps an algorithm takes to converge, and *hardware efficiency*, how efficient each of those step is. We introduced this terminology and studied this extensively in a recent paper [48].

DimmWitted, the statistical inference and learning engine in DeepDive [48] is built upon our research of how to design a high-performance statistical inference and learning engine on a single machine [25, 30, 47, 48]. DimmWitted models Gibbs sampling as a "column-to-row access" operation: each row corresponds to one factor, each column to one variable, and the non-zero elements in the matrix correspond to edges in the factor graph. To process one variable, DimmWitted fetches one column of the matrix to get the set of factors, and other columns to get the set of variables that connect to the same factor. In standard benchmarks, DimmWitted was 3.7× faster than GraphLab's implementation without any application-specific optimization. Compared with traditional work, the main novelty of DimmWitted is that it considers *both* hardware efficiency and statistical efficiency for executing an inference and learning task.

- **Hardware Efficiency** DeepDive takes into consideration the architecture of modern Non-uniform memory access (NUMA) machines. A NUMA machine usually contains multiple nodes (sockets), where each sockets contains multiple CPU cores. To achieve high hardware efficiency, one wants to decrease the communication across different NUMA nodes.
- **Statistical Efficiency** Pushing hardware efficiency to the extreme might decrease statistical efficiency because the lack of communication between nodes might decrease the rate of convergence of a statistical inference and learning algorithm. DeepDive takes advantage of the theoretical results of model averaging [50] and our own results about lock-free execution [25, 30].

On the whole corpus of Paleobiology, the factor graph contains more than 0.2 billion random variables and 0.3 billion factors. On this factor graph, DeepDive is able to run Gibbs sampling on a machine with 4 sockets (10 cores per socket), and we find that we can generate 1,000 samples for all 0.2 billion random variables in 28 minutes. This is more than 4× faster than a non-NUMA-aware implementation.

Incremental Inference. Due to our choice of incremental grounding, the input to DeepDive's inference phase is a factor graph along with a set of changed variables and factors. The goal is to compute the output probabilities computed by the system. Our approach is to frame the incremental maintenance problem as approximate inference. Previous work in the database community has looked at how machine learning data products change in response to both to new labels [22] and to new data [7, 8]. In KBC, both the program and data change on each iteration. Our proposed approach can cope with both types of change simultaneously.

The technical question is which approximate inference algorithms to use in KBC applications. We choose to study two popular classes of approximate inference techniques: *sampling-based materialization* (inspired by sampling-based probabilistic databases such as MCDB [18]) and *variational-based materialization* (inspired by techniques for approximating graphical models [41]). Applying these techniques to incremental maintenance for KBC is novel, and it is not theoretically clear how the techniques compare. Thus, we conducted an experimental evaluation of these two approaches on a diverse set of DeepDive programs. We found these two approaches are sensitive to changes along three largely orthogonal axes: the size of the factor graph, the sparsity of correlations, and the anticipated number of future changes. The performance varies by up to two orders of magnitude in different points of the space. Our study of the tradeoff space highlights that neither materialization strategy dominates the other. To automatically choose the materialization strategy, we developed a simple rule-based optimizer [39].

5. RELATED WORK

Knowledge Base Construction (KBC) has been an area of intense study over the last decade [2, 3, 6, 12, 21, 23, 29, 35, 38, 40, 45, 49]. Within this space, there are a number of approaches.

Rule-Based Systems. The earliest KBC systems used pattern matching to extract relationships from text. The most well-known example is the "Hearst Pattern" proposed by Hearst [16] in 1992. In her seminal work, Hearst observed that a large number of hyponyms can be discovered by simple patterns, e.g., "X such as Y." Hearst's technique has formed the basis of many further techniques that attempt to extract high-quality patterns from text. Rule-

For example, for the grounding procedure illustrated in Figure 8, the delta rule for F1 is $q^\delta(x) : -R^\delta(x, y)$.

based (pattern matching-based) KBC systems, such as IBM's SystemT [23, 24], have been built to aid developers in constructing high-quality patterns. These systems provide the user with a (declarative) interface to specify a set of rules and patterns to derive relationships. These systems have achieved state-of-the-art quality on tasks such as parsing [24].

Statistical Approaches. One limitation of rule-based systems is that the developer needs to ensure that all rules provided to the system are high-precision rules. For the last decade, probabilistic (or machine learning) approaches have been proposed to allow the system to select from a range of a priori features automatically. In these approaches, the extracted tuple is associated with a marginal probability that it is true. DeepDive, Google's knowledge graph, and IBM's Watson are built on this approach. Within this space, there are three styles of systems that based on classification-based frameworks [2, 3, 6, 12, 45], maximum *a posteriori* (MAP) [21, 29, 40], and probabilistic graphical models [10, 35, 49]. Our work on DeepDive is based on graphical models.

6. ACKNOWLEDGMENTS

We gratefully acknowledge the support of the Defense Advanced Research Projects Agency (DARPA) XDATA program under No. FA8750-12-2-0335 and DEFT program under No. FA8750-13-2-0039, DARPA's MEMEX program and SIMPLEX program, the National Science Foundation (NSF) CAREER Award under No. IIS-1353606, the Office of Naval Research (ONR) under awards No. N000141210041 and No. N000141310129, the National Institutes of Health Grant U54EB020405 awarded by the National Institute of Biomedical Imaging and Bioengineering (NIBIB) through funds provided by the trans-NIH Big Data to Knowledge (BD2K) initiative, the Sloan Research Fellowship, the Moore Foundation, American Family Insurance, Google, and Toshiba. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA, AFRL, NSF, ONR, NIH, or the U.S. government.

7. REFERENCES

[1] G. Angeli et al. Stanford's 2014 slot filling systems. *TAC KBP*, 2014.
[2] M. Banko et al. Open information extraction from the Web. In *IJCAI*, 2007.
[3] J. Betteridge, A. Carlson, S. A. Hong, E. R. Hruschka Jr, E. L. Law, T. M. Mitchell, and S. H. Wang. Toward never ending language learning. In *AAAI Spring Symposium*, 2009.
[4] S. Brin. Extracting patterns and relations from the world wide web. In *WebDB*, 1999.
[5] E. Brown et al. Tools and methods for building watson. *IBM Research Report*, 2013.
[6] A. Carlson et al. Toward an architecture for never-ending language learning. In *AAAI*, 2010.
[7] F. Chen, A. Doan, J. Yang, and R. Ramakrishnan. Efficient information extraction over evolving text data. In *ICDE*, 2008.
[8] F. Chen et al. Optimizing statistical information extraction programs over evolving text. In *ICDE*, 2012.
[9] Y. Chen and D. Z. Wang. Knowledge expansion over probabilistic knowledge bases. In *SIGMOD*, 2014.
[10] P. Domingos and D. Lowd. *Markov Logic: An Interface Layer for Artificial Intelligence*. Morgan & Claypool, 2009.
[11] X. L. Dong et al. From data fusion to knowledge fusion. In *VLDB*, 2014.
[12] O. Etzioni et al. Web-scale information extraction in KnowItAll: (preliminary results). In *WWW*, 2004.
[13] D. Ferrucci et al. Building Watson: An overview of the DeepQA project. *AI Magazine*, 2010.
[14] V. Govindaraju et al. Understanding tables in context using standard NLP toolkits. In *ACL*, 2013.
[15] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. *SIGMOD Rec.*, 1993.
[16] M. A. Hearst. Automatic acquisition of hyponyms from large text corpora. In *COLING*, 1992.
[17] R. Hoffmann et al. Knowledge-based weak supervision for information extraction of overlapping relations. In *ACL*, 2011.
[18] R. Jampani et al. MCDB: A Monte Carlo approach to managing uncertain data. In *SIGMOD*, 2008.
[19] E. T. Jaynes. *Probability Theory: The Logic of Science*. Cambridge University Press, 2003.
[20] S. Jiang et al. Learning to refine an automatically extracted knowledge base using Markov logic. In *ICDM*, 2012.
[21] G. Kasneci et al. The YAGO-NAGA approach to knowledge discovery. *SIGMOD Rec.*, 2009.
[22] M. L. Koc and C. Ré. Incrementally maintaining classification using an RDBMS. *PVLDB*, 2011.
[23] R. Krishnamurthy et al. SystemT: A system for declarative information extraction. *SIGMOD Rec.*, 2009.
[24] Y. Li, F. R. Reiss, and L. Chiticariu. SystemT: A declarative information extraction system. In *HLT*, 2011.
[25] J. Liu and et al. An asynchronous parallel stochastic coordinate descent algorithm. *ICML*, 2014.
[26] J. Madhavan et al. Web-scale data integration: You can only afford to pay as you go. In *CIDR*, 2007.
[27] E. K. Mallory et al. Large-scale extraction of gene interactions from full text literature using deepdive. *Bioinformatics*, 2015.
[28] M. Mintz et al. Distant supervision for relation extraction without labeled data. In *ACL*, 2009.
[29] N. Nakashole et al. Scalable knowledge harvesting with high precision and high recall. In *WSDM*, 2011.
[30] F. Niu et al. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, 2011.
[31] F. Niu et al. Tuffy: Scaling up statistical inference in Markov logic networks using an RDBMS. *PVLDB*, 2011.
[32] F. Niu et al. Elementary: Large-scale knowledge-base construction via machine learning and statistical inference. *Int. J. Semantic Web Inf. Syst.*, 2012.
[33] F. Niu et al. Scaling inference for Markov logic via dual decomposition. In *ICDM*, 2012.
[34] S. E. Peters et al. A machine reading system for assembling synthetic Paleontological databases. *PloS ONE*, 2014.
[35] H. Poon and P. Domingos. Joint inference in information extraction. In *AAAI*, 2007.
[36] C. Ré et al. Feature engineering for knowledge base construction. *IEEE Data Eng. Bull.*, 2014.
[37] C. P. Robert and G. Casella. *Monte Carlo Statistical Methods*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
[38] W. Shen et al. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB*, 2007.
[39] J. Shin et al. Incremental knowledge base construction using deepdive. *PVLDB*, 2015.
[40] F. M. Suchanek et al. SOFIE: A self-organizing framework for information extraction. In *WWW*, 2009.
[41] M. Wainwright and M. Jordan. Log-determinant relaxation for approximate inference in discrete Markov random fields. *Trans. Sig. Proc.*, 2006.
[42] M. J. Wainwright and M. I. Jordan. Graphical models, exponential families, and variational inference. *FTML*, 2008.
[43] G. Weikum and M. Theobald. From information to knowledge: Harvesting entities and relationships from web sources. In *PODS*, 2010.
[44] M. Wick et al. Scalable probabilistic databases with factor graphs and MCMC. *PVLDB*, 2010.
[45] A. Yates et al. TextRunner: Open information extraction on the Web. In *NAACL*, 2007.
[46] C. Zhang et al. GeoDeepDive: statistical inference using familiar data-processing languages. In *SIGMOD*, 2013.
[47] C. Zhang and C. Ré. Towards high-throughput Gibbs sampling at scale: A study across storage managers. In *SIGMOD*, 2013.
[48] C. Zhang and C. Ré. DimmWitted: A study of main-memory statistical analytics. *PVLDB*, 2014.
[49] J. Zhu et al. StatSnowball: A statistical approach to extracting entity relationships. In *WWW*, 2009.
[50] M. Zinkevich and et al. Parallelized stochastic gradient descent. In *NIPS*, pages 2595–2603, 2010.

Technical Perspective - k-Shape: Efficient and Accurate Clustering of Time Series

Zachary G. Ives
University of Pennsylvania
zives@cis.upenn.edu

Database research frequently cuts across many layers of abstraction (from formal foundations to algorithms to languages to systems) and the software stack (from data storage and distribution to runtime systems and query optimizers). It does this in a way that is specialized to a particular class of data and workloads. Over the decades, we have seen this pattern applied to enterprise data, persistent objects, Web data, sensor data, data streams, and so on. Each time, the community has developed extensions to algebraic query primitives, specialized implementation techniques (index structures, pattern detection algorithms, update and consistency mechanisms, etc.), benchmarks, and new optimization techniques.

Today, we are on the cusp of another class of data and applications becoming of broad interest. *Time series data* were once viewed as being the purview of specialized scientific applications, forecasting settings, etc., with solutions that did not necessarily generalize to other domains. Today time series data are omnipresent: they are continuously emitted by smartphones (tracking location, acceleration, etc.), smartwatches and fitness bands (tracking activity and health data), as well as environmental sensors, medical devices, and flow monitors, and even server logs, network events, financial transactions, etc. Personalization, prediction, and event detection are increasingly reliant on these data.

At first glance, time series data management seems closely related to that of sensor networks, data streams, temporal data, complex event processing, and the like. The difference is largely in the goals: rather than computing properties of samples within time windows, or looking for particular sequences of events, time series processing often looks for general patterns that can manifest themselves with differences in both amplitude and phase: i.e., in some distinguishing *shape* in the readings, which may occur at different scales. For instance, the goal may be to find spikes in voltage levels measured in regions of the brain (EEG, ECoG, etc.) indicative of seizures or tremors, or to find motion in a wrist-worn fitness band that indicates the wearer is taking a step during walking, or to spot a signature in network traffic behavior indicative of a particular kind of DoS attack.

Database and data mining researchers have been developing techniques to extract motifs for time series, which are useful for compression, indexing, and search; query-by-example capabilities with waveforms of particular shapes; techniques to learn the structure of "notable" time series segments (e.g., seizures, tremors); and finally, unsupervised pattern detection methods like clustering, which can be used to find the underlying structure in the shape of the data.

Most work on clustering time series is focused on defining an effective distance metric between shapes, then using standard clustering methods (hierarchical, k-means, k-mediods, spectral) to group time series segments using these distance measures. However, the question of how to describe a shape is not completely evident, and there is also a need to tolerate certain kinds of variations (e.g., noise, stutters, faster or slower rates). As a result, many different measures have been defined, including Euclidean distance functions between waveforms, "time warping" where portions of a time series signal can be accelerated or decelerated, longest common subsequence measures with scaling, and edit distance models where samples can be added or removed to make two shapes look more similar. Excellent survey materials exist describing the different distance measures as well as broader time series techniques [1, 2, 3].

The k-Shape paper is differentiated from previous work by tackling the questions of clustering method and distance measure simultaneously: Paparizos and Gravano use a statistical measure, cross-correlation, as their measure for comparing time series sequences, and they alter the k-means algorithm to use a different centroid computation mechanism when computing the clusters. These two modifications are insightful and highly effective, both in terms of clustering quality and scale-up. In fact, the conference version of the paper includes an extensive performance evaluation with 48 different datasets and shows the superiority of their method against many of the previously proposed schems. This paper represents a very nice example of the progress being made in the time series arena, and we anticipate that many more developments lie ahead.

1. REFERENCES

[1] D. Gunopulos and G. Das. Time series similarity measures and time series indexing. In *SIGMOD*, page 624, 2001.

[2] E. Keogh. Machine learning in time series databases (and everything is a time series!). Tutorial, AAAI 2011. Available from `http://www.cs.ucr.edu/~eamonn/tutorials.html`.

[3] Y. Sakurai, Y. Matsubara, and C. Faloutsos. Mining and forecasting of big time-series data. In *SIGMOD*, pages 919–922, 2015. Tutorial available from `http://www.cs.kumamoto-u.ac.jp/~yasuko/TALKS/15-SIGMOD-tut/`.

k-Shape: Efficient and Accurate Clustering of Time Series

John Paparrizos
Columbia University
jopa@cs.columbia.edu

Luis Gravano
Columbia University
gravano@cs.columbia.edu

ABSTRACT

The proliferation and ubiquity of temporal data across many disciplines has generated substantial interest in the analysis and mining of time series. Clustering is one of the most popular data mining methods, not only due to its exploratory power, but also as a preprocessing step or subroutine for other techniques. In this paper, we describe k-Shape, a novel algorithm for time-series clustering. k-Shape relies on a scalable iterative refinement procedure, which creates homogeneous and well-separated clusters. As its distance measure, k-Shape uses a normalized version of the cross-correlation measure in order to consider the shapes of time series while comparing them. Based on the properties of that distance measure, we develop a method to compute cluster centroids, which are used in every iteration to update the assignment of time series to clusters. An extensive experimental evaluation against partitional, hierarchical, and spectral clustering methods, with the most competitive distance measures, showed the robustness of k-Shape. Overall, k-Shape emerges as a domain-independent, highly accurate, and efficient clustering approach for time series with broad applications.

1. INTRODUCTION

Temporal, or sequential, data mining deals with problems where data are naturally organized in sequences [28]. We refer to such data sequences as time-series sequences if they contain explicit information about timing (e.g., stock, audio, speech, and video) or if an ordering on values can be inferred (e.g., streams and handwriting). Large volumes of time-series sequences appear in almost every discipline, including astronomy, biology, meteorology, medicine, finance, robotics, engineering, and others [1, 5, 21, 23, 29, 43, 59, 62]. The ubiquity of time series has generated a substantial interest in querying [2, 38, 39, 41, 52, 61, 65], indexing [8, 11, 34, 35, 37, 63], classification [30, 47, 58, 70], clustering [36, 45, 54, 69, 71], and modeling [3, 31, 68] of such data.

Among all techniques applied to time-series data, clustering is the most widely used as it does not rely on costly human supervision or time-consuming annotation of data. With clustering, we can identify and summarize interesting patterns and correlations in the underlying data [27]. In the last few decades, clustering of time-series sequences has received significant attention [4, 14, 21, 40, 51, 54, 56, 69, 71], not only as a powerful stand-alone exploratory method, but also as a preprocessing step or subroutine for other tasks.

The original version of this paper was published in ACM SIGMOD 2015 [53].

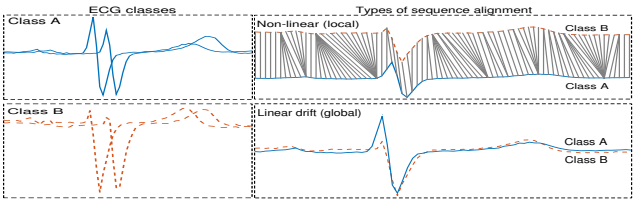


Figure 1: ECG sequence examples and types of alignments for the two classes of the ECGFiveDays dataset [1].

Most time-series analysis methods, including clustering, critically depend on the choice of distance measure. A key issue when comparing two sequences is how to handle the variety of distortions, as we will discuss, that are characteristic of the sequences. To illustrate this point, consider the ECGFiveDays dataset [1], with ECG sequences recorded for the same patient on two different days. While the sequences seem similar overall, they exhibit patterns that belong in one of the two distinct classes (see Figure 1): Class A is characterized by a sharp rise, a drop, and another gradual increase while Class B is characterized by a gradual increase, a drop, and another gradual increase. Ideally, a *shape-based* clustering method should generate a partition similar to the classes shown in Figure 1, where sequences exhibiting similar patterns are placed into the same cluster based on their *shape* similarity, regardless of differences in amplitude and phase. As the notion of shape cannot be precisely defined, dozens of distance measures have been proposed [9, 10, 12, 16, 18, 46, 64] to offer invariances to multiple inherent distortions in the data. However, it has been shown that distance measures offering invariances to amplitude and phase perform exceptionally well [15, 66] and, hence, such measures are used for shape-based clustering [44, 50, 54, 69].

Due to these difficulties and the different needs for invariances from one domain to another, more attention has been given to the creation of new distance measures rather than to the creation of new clustering algorithms. It is generally believed that the choice of distance measure is more important than the clustering algorithm itself [6]. As a consequence, time-series clustering relies mostly on classic clustering methods, either by replacing the default distance measure with one that is more appropriate for time series, or by transforming time series into "flat" data so that existing clustering algorithms can be directly used [67]. However, the choice of clustering method can affect: (i) accuracy, as every method expresses homogeneity and separation of clus-

ters differently; and (ii) efficiency, as the computational cost differs from one method to another. For example, spectral clustering [17] or certain variants of hierarchical clustering [33] are more appropriate to identify density-based clusters (i.e., areas of higher density than the remainder of the data) than partitional methods such as k-means [42] or k-medoids [33]. On the other hand, k-means is more efficient than hierarchical, spectral, or k-medoids methods.

Unfortunately, state-of-the-art approaches for shape-based clustering, which use partitional methods with distance measures that are scale- and shift-invariant, suffer from two main drawbacks: (i) these approaches cannot scale to large-volumes of data as they depend on computationally expensive methods or distance measures [44, 50, 54, 69]; and (ii) these approaches have been developed for particular domains [69] or their effectiveness has only been shown for a limited number of datasets [44, 50]. Moreover, the most successful shape-based clustering methods handle phase invariance through a local, non-linear alignment of the sequence coordinates, even though a global alignment is often adequate. For example, for the ECG dataset in Figure 1, an efficient linear drift can reveal the underlying differences in patterns of sequences of two classes, whereas an expensive non-linear alignment might match every corresponding increase or drop of each sequence, making it difficult to distinguish the two classes (see Figure 1). Importantly, to the best of our knowledge, these approaches have never been extensively evaluated against each other, against other partitional methods, or against different approaches such as hierarchical or spectral methods. We summarize such an experimental evaluation below. Our original paper [53] has further details.

In this article, we discuss k-Shape, a novel algorithm for shape-based time-series clustering that is efficient and domain independent. k-Shape is based on a scalable iterative refinement procedure similar to the one used by the k-means algorithm, but with significant differences. Specifically, k-Shape uses both a different distance measure and a different method for centroid computation from those of k-means. As argued above, k-Shape attempts to preserve the shapes of time-series sequences while comparing them. To do so, k-Shape requires a distance measure that is invariant to scaling and shifting. Unlike other clustering approaches [44, 54, 69], for k-Shape we adapt the cross-correlation statistical measure and we show: (i) how we can derive in a principled manner a time-series distance measure that is scale- and shift-invariant; and (ii) how this distance measure can be computed efficiently. Based on the properties of the normalized version of cross-correlation, we develop a novel method to compute cluster centroids, which are used in every iteration to update the assignment of time series to clusters.

To demonstrate the effectiveness of the distance measure and k-Shape, we have conducted an extensive experimental evaluation on 48 datasets and compared the state-of-the-art distance measures and clustering approaches for time series using rigorous statistical analysis. We took steps to ensure the reproducibility of our results, including making available our source code as well as using public datasets. Our experimental evaluation suggests that: (1) cross-correlation measures, which are not widely adopted as time-series distance measures, outperform Euclidean distance (ED) [16] and are as competitive as state-of-the-art measures, such as constrained Dynamic Time Warping (cDTW) [60], but significantly faster; (2) the k-means algorithm with ED, in contrast to what has been reported in the literature, is a robust approach for time-series clustering, but inadequate modifications of its distance measure and centroid computation can reduce its performance; (3) the choice of clustering method, which was believed to be less important than that of distance measure, is as important as the choice of distance measure; and (4) k-Shape outperforms all scalable approaches in terms of accuracy. Furthermore, k-Shape also outperforms all non-scalable (and hence impractical) approaches, with one exception that achieves similar accuracy results. However, unlike k-Shape, this approach requires tuning of its distance measure and is two orders of magnitude slower than k-Shape. Overall, k-Shape is a highly accurate and scalable choice for time-series clustering that performs exceptionally well across domains [53].

We start by reviewing the state of the art for clustering time series, as well as with our problem definition (Section 2). We then describe our approach, as follows:

- We show how a scale-, translate-, and shift-invariant distance measure can be derived in a principled manner from the cross-correlation measure and how this measure can be efficiently computed (Section 3.1).
- We present a novel method to compute a cluster centroid when that distance measure is used (Section 3.2).
- We describe k-Shape, a centroid-based algorithm for time-series clustering (Section 3.3).
- We summarize our extensive experimental evaluation (Sections 4 and 5).

We conclude with the implications of our work (Section 6). Please refer to [53] for further details on our approach and the experimental evaluation.

2. PRELIMINARIES

In this section, we review distortions that are common in time series (Section 2.1) and the most popular distance measures for such data (Section 2.2). Then, we summarize existing approaches for clustering time-series data (Section 2.3) and for centroid computation (Section 2.4). Finally, we formally present our problem of focus (Section 2.5).

2.1 Time-Series Invariances

Based on the domain, sequences are often distorted in some way, and distance measures need to satisfy a number of invariances in order to compare sequences meaningfully. In this section, we review common time-series distortions and their invariances. For a more detailed review, see [6].

Scaling and translation invariances: In many cases, it is useful to recognize the similarity of sequences despite differences in amplitude (scaling) and offset (translation). In other words, transforming a sequence $\vec{x}$ as $\vec{x}' = a\vec{x}+b$, where a and b are constants, should not change $\vec{x}$'s similarity to other sequences. For example, these invariances might be useful to analyze seasonal variations in currency values on foreign exchange markets without being biased by inflation.

Shift invariance: When two sequences are similar but differ in phase (global alignment) or when there are regions of the sequences that are aligned and others are not (local alignment), we might still need to consider them similar. For example, heartbeats can be out of phase depending on when we start taking the measurements (global alignment) and handwritings of a phrase from different people will need alignment depending on the size of the letters and on the spaces between words (local alignment).

Uniform scaling invariance: Sequences that differ in length require either stretching of the shorter sequence or shrinking of the longer sequence so that we can compare them effectively. For example, this invariance is required for heartbeats with measurement periods of different duration.
Occlusion invariance: When subsequences are missing, we can still compare the sequences by ignoring the subsequences that do not match well. This invariance is useful in handwritings if there is a typo or a letter is missing.
Complexity invariance: When sequences have similar shape but different complexities, we might want to make them have low or high similarity based on the application. For example, audio signals that were recorded indoors and outdoors might be considered similar, despite the fact that outdoor signals will be more noisy than indoor signals.

For many tasks, some or all of the above invariances are required when we compare time-series sequences. To satisfy the appropriate invariances, we could preprocess the data to eliminate the corresponding distortions before clustering. For example, by z-normalizing [24] the data we can achieve the scaling and translation invariances. However, for invariances that cannot be trivially achieved with a preprocessing step, we can define sophisticated distance measures that offer distortion invariances. In the next section, we review the most common such distance measures.

2.2 Time-Series Distance Measures

The two state-of-the-art approaches for time-series comparison first z-normalize the sequences and then use a distance measure to determine their similarity, and possibly capture more invariances. The most widely used distance metric is the simple ED [16]. ED compares two time series $\vec{x} = (x_1, \ldots, x_m)$ and $\vec{y} = (y_1, \ldots, y_m)$ of length m as follows:

$$ED(\vec{x}, \vec{y}) = \sqrt{\sum_{i=1}^{m} (x_i - y_i)^2} \quad (1)$$

Another popular distance measure is DTW [60]. DTW can be seen as an extension of ED that offers a local (non-linear) alignment. To achieve that, an m-by-m matrix M is constructed, with the ED between any two points of $\vec{x}$ and $\vec{y}$. A *warping path* $W = \{w_1, w_2, \ldots, w_k\}$, with $k \geq m$, is a contiguous set of matrix elements that defines a mapping between $\vec{x}$ and $\vec{y}$ under several constraints [37]:

$$DTW(\vec{x}, \vec{y}) = \min \sqrt{\sum_{i=1}^{k} w_i} \quad (2)$$

This path can be computed on matrix M with dynamic programming for the evaluation of the following recurrence:

$$\gamma(i,j) = ED(i,j) + \min\{\gamma(i-1,j-1), \gamma(i-1,j), \gamma(i,j-1)\}.$$

It is common practice to constrain the warping path to visit only a subset of cells on matrix M. The shape of the subset matrix is called *band* and the width of the band is called *warping window*. The most frequently used band for constrained Dynamic Time Warping (cDTW) is the Sakoe-Chiba band [60]. Figure 2a shows the difference in alignments of two sequences offered by ED and DTW distance measures, whereas Figure 2b presents the computation of the warping path (dark cells) for cDTW constrained by the Sakoe-Chiba band with width 5 cells (light cells).

Recently, Wang et al. [66] extensively evaluated 9 distance measures and several variants thereof. They found that ED is the most efficient measure with a reasonably high accuracy, and that DTW and cDTW perform exceptionally well in comparison to other measures. cDTW is slightly better

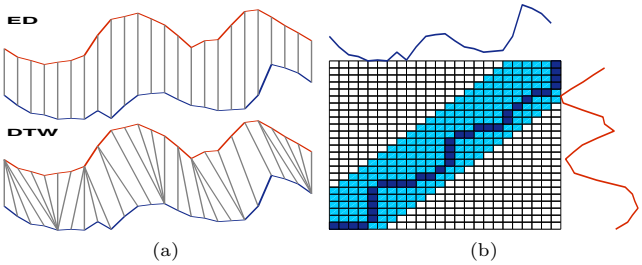


Figure 2: Similarity computation: (a) alignment under ED (top) and DTW (bottom), (b) Sakoe-Chiba band with a warping window of 5 cells (light cells in band) and the warping path computed under cDTW (dark cells in band).

than DTW and significantly reduces the computation time. Several optimizations have been proposed to further speed up cDTW [55]. In the next section, we review clustering algorithms that can utilize these distance measures.

2.3 Time-Series Clustering Algorithms

Several methods have been proposed to cluster time series. All approaches generally modify existing algorithms, either by replacing the default distance measures with a version that is more suitable for comparing time series (raw-based methods), or by transforming the sequences into "flat" data so that they can be directly used in classic algorithms (feature- and model-based methods) [67]. Raw-based approaches can easily leverage the vast literature on distance measures (see Section 2.2), which has shown that invariances offered by certain measures, such as DTW, are general and, hence, suitable for almost every domain [15]. In contrast, feature- and model-based approaches are usually domain-dependent and applications on different domains require that we modify the features or models. Because of these drawbacks of feature- and model-based methods, in this paper we follow a raw-based approach.

The three most popular raw-based methods are agglomerative hierarchical, spectral, and partitional clustering [6]. For hierarchical clustering, the most widely used "linkage" criteria are the single, average, and complete linkage variants [33]. Spectral clustering [49] has recently started receiving attention [6] due to its success over other types of data [17]. Among partitional methods, k-means [42] and k-medoids [33] are the most representative examples. When partitional methods use distance measures that offer invariances to scaling, translation, and shifting, we consider them as shape-based approaches. From these methods, k-medoids is usually preferred [67]: unlike k-means, k-medoids computes the dissimilarity matrix of all data sequences and uses actual sequences as cluster centroids; in contrast, k-means requires the computation of artificial sequences as centroids, which hinders the easy adaptation of distance measures other than ED. However, from all these methods, only the k-means class of algorithms can scale linearly with the size of the datasets. Recently, k-means was modified to work with (i) DTW [54] and (ii) a distance measure that offers pairwise scaling and shifting of time-series sequences [69]. Both of these modifications rely on new methods to compute cluster centroids that we will review next.

2.4 Time-Series Averaging Techniques

The computation of an average sequence or, in the context of clustering, a centroid, is a difficult task that critically

depends on the distance measure used to compare time series. We now review the state-of-the-art methods for the computation of an average sequence.

With Euclidean distance, the arithmetic mean is used to compute an average sequence (e.g., as is the case in the centroid computation of the k-means algorithm). However, as DTW is more appropriate for many time-series tasks [37, 55], several methods have been proposed to average time-series sequences under DTW. Nonlinear alignment and averaging filters (NLAAF) [26] uses a simple pairwise method where each coordinate of the average sequence is calculated as the center of the mapping produced by DTW. This method is applied sequentially to pairs of sequences until only one pair is left. Prioritized shape averaging (PSA) [50] uses a hierarchical method to average sequences. The coordinates of an average sequence are computed as the weighted center of the coordinates of two time-series sequences that were coupled by DTW. Initially, all sequences have weight one, and each average sequence produced in the nodes of the tree has a weight that corresponds to the number of sequences it averages. To avoid the high computation cost of previous approaches, Ranking Shape-based Template Matching Framework (RSTMF) [44] approximates an ordering of the time-series sequences by looking at the distances of sequences to all other cluster centroids, instead of computing the distances of all pairs of sequences.

Several drawbacks of these methods have led to the creation of a more robust technique called Dynamic Time Warping Barycenter Averaging (DBA) [54], which iteratively refines the coordinates of a sequence initially picked from the data. Each coordinate of the average sequence is updated with the use of barycenter of one or more coordinates of the other sequences that were associated with the use of DTW. Among all these methods, DBA seems to be the most efficient and accurate averaging approach when DTW is used [54]. Another averaging technique that is based on matrix decomposition was proposed as part of K-Spectral Centroid Clustering (KSC) [69], to compute the centroid of a cluster when a distance measure for pairwise scaling and shifting is used. In our approach, which we will present in Section 3, we also rely on matrix decomposition to compute centroids.

2.5 Problem Definition

We address the problem of domain-independent, accurate, and scalable clustering of time series into k clusters, for a given value of the target number of clusters k.[1] Even though different domains might require different invariances to data distortions (see Section 2.1), we focus on distance measures that offer invariances to scaling and shifting, which are generally sufficient (see Section 2.2) [15]. Furthermore, to easily adopt such distance measures, we focus our analysis on raw-based clustering approaches, as we argued in Section 2.3. Next, we describe our k-Shape clustering algorithm.

3. K-SHAPE CLUSTERING ALGORITHM

Our objective is to develop a domain-independent, accurate, and scalable algorithm for time-series clustering that is invariant to scaling and shifting. We propose k-Shape, a clustering algorithm built on (i) a distance measure and (ii) a centroid computation method that can preserve the shapes of time series. We first discuss our distance measure, which is based on the cross-correlation measure (Section 3.1). Based on this distance measure, we propose a method to compute centroids of time-series clusters (Section 3.2). Finally, we describe k-Shape, our centroid-based clustering algorithm, which relies on an iterative refinement procedure that scales linearly in the number of sequences and generates homogeneous and well-separated clusters (Section 3.3).

3.1 Time-Series Shape Similarity

As discussed earlier, capturing shape-based similarity requires distance measures that can handle distortions in amplitude and phase. Unfortunately, the best performing distance measures offering invariances to these distortions, such as DTW, are computationally expensive (see Section 2.2). To circumvent this efficiency limitation, we adopt a normalized version of the cross-correlation measure.

Cross-correlation is a measure of similarity for time-lagged signals that is widely used for signal and image processing. However, cross-correlation, a measure that compares one-to-one points between signals, has largely been ignored in experimental evaluations for the problem of time-series comparison. Instead, starting with the application of DTW decades ago [7], research on that problem has focused on elastic distance measures that compare one-to-many or one-to-none points [9, 10, 37, 46, 64]. In particular, recent comprehensive and independent experimental evaluations of state-of-the-art distance measures for time-series comparison — 9 measures and their variants in [15, 66] and 48 measures in [22] — did not consider cross-correlation. Different needs from one domain or application to another hinder the process of finding appropriate normalizations for the data and the cross-correlation measure. Moreover, inefficient implementations of cross-correlation can make it appear as slow as DTW. As a consequence of these drawbacks, cross-correlation has not been widely adopted as a time-series distance measure. In the rest of this section, we show how to address these drawbacks. Specifically, we will show how to choose normalizations that are domain-independent and efficient, and lead to a shape-based distance measure for comparing time series efficiently and effectively.

Cross-correlation measure: Cross-correlation is a statistical measure with which we can determine the similarity of two sequences $\vec{x} = (x_1, \ldots, x_m)$ and $\vec{y} = (y_1, \ldots, y_m)$, even if they are not properly aligned.[2] To achieve shift-invariance, cross-correlation keeps $\vec{y}$ static and slides $\vec{x}$ over $\vec{y}$ to compute their inner product for each *shift* s of $\vec{x}$. We denote a shift of a sequence as follows:

$$\vec{x}_{(s)} = \begin{cases} (\overbrace{0, \ldots, 0}^{|s|}, x_1, x_2, \ldots, x_{m-s}), & s \geq 0 \\ (x_{1-s}, \ldots, x_{m-1}, x_m, \underbrace{0, \ldots, 0}_{|s|}), & s < 0 \end{cases} \quad (3)$$

When all possible shifts $\vec{x}_{(s)}$ are considered, with $s \in [-m, m]$, we produce $CC_w(\vec{x}, \vec{y}) = (c_1, \ldots, c_w)$, the cross-correlation sequence with length $2m-1$, defined as follows:

$$CC_w(\vec{x}, \vec{y}) = R_{w-m}(\vec{x}, \vec{y}), \quad w \in \{1, 2, \ldots, 2m-1\} \quad (4)$$

where $R_{w-m}(\vec{x}, \vec{y})$ is computed, in turn, as:

[1] Although the exact estimation of k is difficult without a gold standard, we can do so by varying k and evaluating clustering quality with criteria that capture information intrinsic to the data alone [33].

[2] For simplicity, we consider sequences of equal length even though cross-correlation can be computed on sequences of different length.

$$R_k(\vec{x},\vec{y}) = \begin{cases} \sum_{l=1}^{m-k} x_{l+k} \cdot y_l, & k \geq 0 \\ R_{-k}(\vec{y},\vec{x}), & k < 0 \end{cases} \quad (5)$$

Our goal is to compute the position w at which $CC_w(\vec{x},\vec{y})$ is maximized. Based on this value of w, the optimal shift of $\vec{x}$ with respect to $\vec{y}$ is then $\vec{x}_{(s)}$, where $s = w - m$.

Depending on the domain or the application, different normalizations for $CC_w(\vec{x},\vec{y})$ might be required. The most common normalizations are the biased estimator, NCC_b, the unbiased estimator, NCC_u, and the coefficient normalization, NCC_c, which are defined as follows:

$$NCC_q(\vec{x},\vec{y}) = \begin{cases} \frac{CC_w(\vec{x},\vec{y})}{m}, & q = \text{“}b\text{”}\ (NCC_b) \\ \frac{CC_w(\vec{x},\vec{y})}{m-|w-m|}, & q = \text{“}u\text{”}\ (NCC_u) \\ \frac{CC_w(\vec{x},\vec{y})}{\sqrt{R_0(\vec{x},\vec{x})\cdot R_0(\vec{y},\vec{y})}}, & q = \text{“}c\text{”}\ (NCC_c) \end{cases} \quad (6)$$

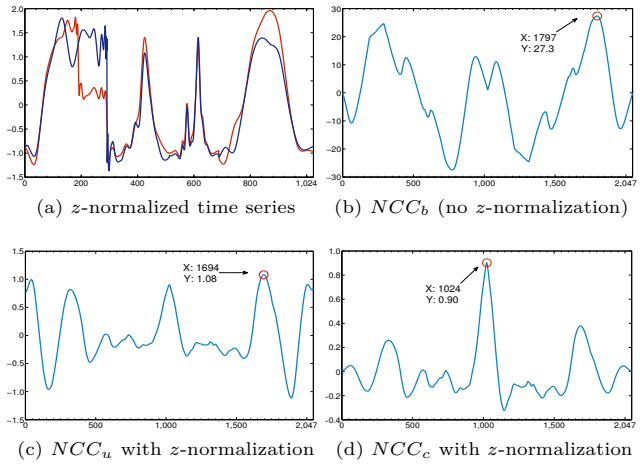
Beyond the cross-correlation normalizations, time series might also require normalization to remove inherent distortions. Figure 3 illustrates how the cross-correlation normalizations for two sequences $\vec{x}$ and $\vec{y}$ of length $m = 1024$ are affected by time-series normalizations. Independently of the normalization applied to $CC_w(\vec{x},\vec{y})$, the produced sequence will have length 2047. Initially, in Figure 3a, we remove differences in amplitude by z-normalizing $\vec{x}$ and $\vec{y}$ in order to show that they are aligned and, hence, no shifting is required. If $CC_w(\vec{x},\vec{y})$ is maximized for $w \in [1025, 2047]$ (or $w \in [1, 1023]$), one of $\vec{x}$ or $\vec{y}$ should be shifted by $i - 1024$ to the right (or $1024 - i$ to the left). Otherwise, if $w = 1024$, $\vec{x}$ and $\vec{y}$ are properly aligned, which is what we expect in our example. Figure 3b shows that if we do not z-normalize $\vec{x}$ and $\vec{y}$, and we use the biased estimator, then NCC_b is maximized at $w = 1797$, which indicates a shifting of a sequence to the left $1797 - 1024 = 773$ times. If we z-normalize $\vec{x}$ and $\vec{y}$, and use the unbiased estimator, then NCC_u is maximized at $w = 1694$, which indicates a shifting of a sequence to the right $1694 - 1024 = 670$ times (Figure 3c). Finally, if we z-normalize $\vec{x}$ and $\vec{y}$, and use the coefficient normalization, then NCC_c is maximized at $w = 1024$, which indicates that no shifting is required (Figure 3d).

As illustrated by the example, normalizations of the data and the cross-correlation measure can have a significant impact on the cross-correlation sequence produced, which makes the creation of a distance measure a non-trivial task. Furthermore, as in Figure 3, cross-correlation sequences produced by pairwise comparisons of multiple time series will differ in amplitude based on the normalizations. Thus, a normalization that produces values within a specified range should be used to meaningfully compare such sequences.

Shape-based distance (SBD): To devise a shape-based distance measure, and based on the previous discussion, we use the coefficient normalization that gives values between -1 and 1, regardless of the data normalization. Coefficient normalization divides the cross-correlation sequence by the geometric mean of autocorrelations of the individual sequences. After normalization of the sequence, we detect the position w where $NCC_c(\vec{x},\vec{y})$ is maximized and we derive the following distance measure:

$$SBD(\vec{x},\vec{y}) = 1 - \max_w \left(\frac{CC_w(\vec{x},\vec{y})}{\sqrt{R_0(\vec{x},\vec{x}) \cdot R_0(\vec{y},\vec{y})}} \right) \quad (7)$$

which takes values between 0 to 2, with 0 indicating perfect similarity for time-series sequences.



(a) z-normalized time series (b) NCC_b (no z-normalization)

(c) NCC_u with z-normalization (d) NCC_c with z-normalization

Figure 3: Time-series and cross-correlation normalizations.

Up to now we have addressed shift invariance. For scaling invariance, we transform each sequence $\vec{x}$ into $\vec{x}' = \frac{\vec{x}-\mu}{\sigma}$, so that its mean μ is zero and its standard deviation σ is one.

Efficient computation of SBD: From Equation 4, the computation of $CC_w(\vec{x},\vec{y})$ for all values of w requires $\mathcal{O}(m^2)$ time, where m is the time-series length. The convolution theorem [32] states that the convolution of two time series can be computed as the Inverse Discrete Fourier Transform (IDFT) of the product of their individual Discrete Fourier Transforms (DFT). Cross-correlation is then computed as the convolution of two time series if one sequence is first reversed in time, $\vec{x}^{(t)} = \vec{x}^{(-t)}$ [32], which equals taking the complex conjugate in the frequency domain. However, DFT and IDFT still require $\mathcal{O}(m^2)$ time. By using a Fast Fourier Transform (FFT) algorithm [13], the time reduces to $\mathcal{O}(m \log(m))$. Data and cross-correlation normalizations can also be efficiently computed; thus the overall time complexity of SBD remains $\mathcal{O}(m \log(m))$. Moreover, recursive algorithms compute an FFT by dividing it into pieces of power-of-two size [20]. Therefore, to further improve the performance of the FFT computation, when $CC(\vec{x},\vec{y})$ is not an exact power of two we pad $\vec{x}$ and $\vec{y}$ with zeros to reach the next power-of-two length after $2m - 1$.

This section described effective cross-correlation and data normalizations to derive a shape-based distance measure. Importantly, we also discussed how the cross-correlation distance measure can be efficiently computed. Our experiments show that SBD is highly competitive, achieving similar results to cDTW and DTW while being orders of magnitude faster. We now turn to the critical problem of extracting a centroid for a cluster, to represent the cluster data consistently with the above shape-based distance measure.

3.2 Time-Series Shape Extraction

Many time-series tasks rely on methods that summarize a set of time series by only one sequence, often referred to as an *average sequence* or, in the context of clustering, as a *centroid*. The extraction of meaningful centroids is a challenging task that critically depends on the choice of distance measure. We now show how to determine such centroids for time-series clustering for the SBD distance measure, to capture shared characteristics of the underlying data.

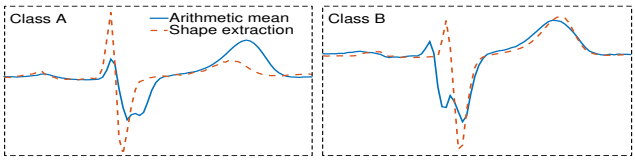


Figure 4: Examples of centroids for each class of the ECGFiveDays dataset, based on the arithmetic mean property (solid lines) and our shape extraction method (dashed lines).

The easiest way to extract an average sequence from a set of sequences is to compute each coordinate of the average sequence as the arithmetic mean of the corresponding coordinates of all sequences. This approach is used by k-means, the most popular clustering method. In Figure 4, the solid lines show such centroids for each class in the ECGFiveDays dataset of Figure 1: these centroids do not capture effectively the class characteristics (see Figures 1 and 4).

To avoid such problems, we cast the centroid computation as an optimization problem where the objective is to find the minimizer of the sum of squared distances to all other time series sequences. However, as cross-correlation intuitively captures the similarity — rather than the dissimilarity — of time series, we can express the computed sequence as the maximizer of the squared similarities to all other time-series sequences. Such similarity (Equation 6) requires the computation of an optimal shift for every sequence. As this approach is used in the context of iterative clustering, we use the previously computed centroid as reference and align all sequences towards this reference sequence. This is a reasonable choice because the previous centroid will be very close to the new centroid. For this alignment, we use SBD, which identifies an optimal shift for every sequence. Subsequently, as sequences are already aligned towards a reference sequence, we can reduce this maximization to a well-known problem called maximization of the Rayleigh Quotient [25]. (See details of this reduction in [53].)

A desirable property of the above formulation is that we can extract the most representative shape from the underlying data in a few lines of code [53]. In Figure 4, the dashed lines show the centroids of each class in the ECGFiveDays dataset, extracted with our shape extraction method and using randomly selected sequences as reference sequences. This method for shape extraction can more effectively capture the characteristics of each class (Figure 1) than by using the arithmetic mean property (solid lines in Figure 4). We now show how our shape extraction method is used in a time-series clustering algorithm.

3.3 Shape-based Time-Series Clustering

We now describe k-Shape, our novel algorithm for time-series clustering. k-Shape relies on the SBD distance measure of Section 3.1 and the shape extraction method of Section 3.2 to efficiently produce clusters of time series.

k-Shape Clustering Algorithm: k-Shape is a partitional clustering method that is based on an iterative refinement procedure similar to the one used in k-means. Through this iterative procedure, k-Shape minimizes the sum of squared distances and manages to: (i) produce homogeneous and well-separated clusters, and (ii) scale linearly with the number of time series. Our algorithm compares sequences efficiently and computes centroids effectively under the scaling, translation, and shift invariances. k-Shape is a non-trivial instantiation of k-means and, in contrast to similar attempts in the literature [54, 69], its distance measure and centroid computation method make k-Shape the only scalable method that significantly outperforms k-means.

In every iteration, k-Shape performs two steps: (i) in the assignment step, the algorithm updates the cluster memberships by comparing each time series with all computed centroids and by assigning each time series to the cluster of the closest centroid; (ii) in the refinement step, the cluster centroids are updated to reflect the changes in cluster memberships in the previous step. The algorithm repeats these two steps until either no change in cluster membership occurs or the maximum number of iterations allowed is reached. In the assignment step, k-Shape relies on the distance measure of Section 3.1, whereas in the refinement step it relies on the centroid computation method of Section 3.2.

k-Shape expects as input the time series set and the number of clusters that we want to produce. (Please refer to [53] for the full algorithm.) Initially, we randomly assign the time series in to clusters. Then, we compute each cluster centroid with the shape extraction method (see Section 3.2). Once the centroids are computed, we refine the memberships of the clusters by using the SBD distance measure. We repeat this procedure until the algorithm converges or reaches the maximum number of iterations (usually a small number, such as 100). The output of the algorithm is the assignment of sequences to clusters and the centroids for each cluster.

We now turn to the experimental evaluation of k-Shape against the state-of-the-art time-series clustering approaches.

4. EXPERIMENTAL SETTINGS

In this section, we describe the experimental settings for the evaluation of both SBD and our k-Shape algorithm.

Datasets: We use 48 class-labeled time-series datasets, both synthetic and real, which span several different domains [1].

Platform: We ran our experiments on a cluster of 10 servers with identical configuration: Dual Intel Xeon X5550 processor with clock speed at 2.67 GHz and 24 GB RAM. Each server runs Ubuntu 12.04 and Matlab R2012b.

Implementation: We implemented our approach and all state-of-the-art approaches that we compare against under the same framework, in Matlab, for a consistent evaluation in terms of both accuracy and efficiency. For repeatability purposes, we make all datasets and source code available.[3]

Baselines: We compare SBD against the strongest state-of-the-art distance measures for time series (see Section 2.2 for a detailed discussion), namely, ED, DTW, and cDTW. Only cDTW requires setting a parameter, to constrain its warping window. We consider two cases from the literature: (i) cDTW^{opt}: we compute the optimal window by performing a leave-one-out classification step over the training set of each dataset; (ii) cDTW^{w}: we use as window 5%, for cDTW^{5}, of the length of the time series of each dataset. We compare k-Shape against the three strongest types of scalable and non-scalable clustering methods, namely, partitional, hierarchical, and spectral methods (see Section 2.3 for a detailed discussion), combined with the most competitive distance measures discussed previously (we denote them as Dist). As scalable methods, we consider the classic k-means algorithm with ED (k-AVG+ED) [42], and the following vari-

[3] http://www.cs.columbia.edu/~jopa/kshape.html

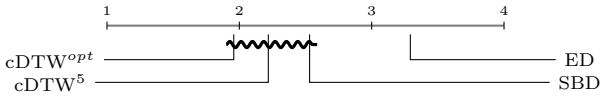


Figure 5: Ranking of distance measures based on the average of their ranks across datasets. The wiggly line connects all measures that do not perform statistically differently according to the Nemenyi test.

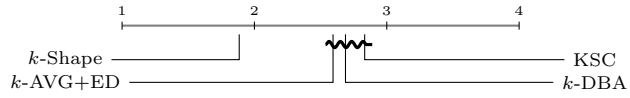


Figure 6: Ranking of k-means variants based on the average of their ranks across datasets. The wiggly line connects all techniques that do not perform statistically differently according to the Nemenyi test.

ants: (i) k-means with DTW as distance measure and the DBA method for centroid computation (k-DBA) [54] and (ii) k-means with a distance measure offering pairwise scaling and shifting of time series and computation of the spectral norm of a matrix for centroid computation (KSC) [69]. As non-scalable methods, among partitional methods we consider the Partitioning Around Medoids (PAM+Dist) implementation of the k-medoids algorithm [33]. Among hierarchical methods, we use agglomerative hierarchical clustering with single (H-S+Dist), average (H-A+Dist), and complete (H-C+Dist) linkage criteria [33]. Finally, among spectral methods, we consider the popular normalized spectral clustering method (S+Dist) [49]. Overall, we compared k-Shape against 20 clustering approaches.

Metrics: We compute CPU time utilization and report time ratios for our comparisons. We use the one nearest neighbor classification accuracy to evaluate the distance measures and the Rand Index [57] to evaluate clustering accuracy.

Statistical analysis: We use the Friedman test [19] followed by the post-hoc Nemenyi test [48] for comparison of multiple algorithms over multiple datasets and we report statistical significant results with a 95% confidence level.

5. EXPERIMENTAL RESULTS

We now provide highlights of the detailed experimental evaluation in [53]. First, we evaluate SBD against the state-of-the-art distance measures. Then, we compare k-Shape against scalable and non-scalable clustering approaches.

Evaluation of SBD: All distance measures, including SBD, outperform ED with statistical significance. The difference in accuracy between SBD and DTW is in most cases negligible: SBD performs at least as well as DTW in 30 datasets. Considering the constrained versions of DTW, we observe that SBD performs similarly to or better than cDTW^{opt} and cDTW^{5}in 22 and 18 datasets, respectively. To better understand the performance of SBD in comparison with cDTW^{opt} and cDTW^5, we evaluate the significance of their differences in accuracy when considered all together. Figure 5 shows the average rank across datasets of each distance measure. cDTW^{opt} is the top measure, with an average rank of 1.96, meaning that cDTW^{opt} performed best in the majority of the datasets. The Friedman test rejects the null hypothesis that all measures behave similarly, and, hence, we proceed with a post-hoc Nemenyi test, to evaluate the significance of the differences in the ranks. The wiggly line in the figure connects all measures that do not perform statistically differently according to the Nemenyi test. We observe that the ranks of cDTW^{opt}, cDTW^5, and SBD do not present a significant difference, and ED, which is ranked last, is significantly worse than the others. In terms of efficiency, SBD is only 4.4x slower than ED and remains one order of magnitude faster than cDTW^{opt} and cDTW^5. In conclusion, SBD is a very efficient, parameter-free distance measure that significantly outperforms ED and achieves similar results to both constraint and unconstraint versions of DTW.

Evaluation of k-Shape Against Other Scalable Methods: Figure 6 shows the average rank across datasets of each k-means variant. k-Shape is the top technique, with an average rank of 1.89, meaning that k-Shape was best in the majority of the datasets. The Friedman test rejects that all algorithms behave similarly, so we proceed with a post-hoc Nemenyi test, to evaluate the significance of the differences in the ranks. We observe that the ranks of KSC, k-DBA, and k-AVG+ED do not present a statistically significant difference, whereas k-Shape, which is ranked first, is significantly better than the others. Modifying k-means with inappropriate distance measures or centroid computation methods might lead to unexpected results. In terms of efficiency, k-Shape is one order of magnitude faster than KSC, two orders of magnitude faster than k-DBA, and one order of magnitude slower than k-AVG+ED.

Evaluation of k-Shape Against Non-Scalable Methods: To show the robustness of k-Shape in terms of accuracy beyond scalable approaches, we now ignore scalability and compare k-Shape against hierarchical, spectral, and k-medoids methods. Among all existing state-of-the-art methods that use ED or cDTW^5 as distance measures, only partitional methods perform similarly to or better than k-AVG+ED. In particular, PAM+cDTW^5 is the only method that outperforms k-AVG+ED. Figure 7 shows that k-Shape, PAM+SBD, PAM+cDTW^5, and S+SBD (i.e., all methods outperforming k-AVG+ED) do not present a significant difference in accuracy, whereas k-AVG+ED, which is ranked last, is significantly worse than the others.

In short, our experimental evaluation suggests that SBD is as competitive as state-of-the-art measures, such as cDTW and DTW, but faster, and k-Shape is the only method that is both accurate and efficient. In [53], we provide further details on these findings and on the performance of hierarchical and spectral methods as well.

6. CONCLUSIONS

We presented k-Shape, a partitional clustering algorithm that preserves the shapes of time series. k-Shape compares time series efficiently and computes centroids effectively under the scaling and shift invariances. We have identified many interesting directions for future work. For example, k-Shape currently operates over a single time-series representation and cannot handle multiple representations. Considering that several transformations (e.g., smoothing) can reduce noise and eliminate outliers in time series, an extension of k-Shape to leverage characteristics from multiple representations can significantly improve its accuracy. Another future direction is to explore the usefulness of k-Shape as a "subroutine" of other methods. For example, nearest centroid classifiers rely on effective clustering of time series and subsequent extraction of centroids for the clusters.

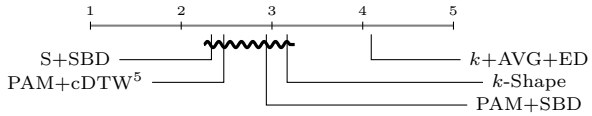


Figure 7: Ranking of methods that outperform k-AVG+ED based on the average of their ranks across datasets. The wiggly line connects all techniques that do not perform statistically differently according to the Nemenyi test.

Acknowledgments: We thank Or Biran, Christos Faloutsos, Eamonn Keogh, Kathy McKeown, Taesun Moon, François Petitjean, and Kapil Thadani for invaluable discussions and feedback. We also thank Ken Ross for sharing computing resources for our experiments. This research was supported by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior National Business Center (DoI/NBC) contract number D11PC20153. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/NBC, or the U.S. Government. This material is also based upon work supported by a generous gift from Microsoft Research. John Paparrizos is an Alexander S. Onassis Foundation Scholar.

7. REFERENCES

[1] The UCR Time Series Classification/Clustering Homepage. http://www.cs.ucr.edu/~eamonn/time_series_data. Accessed: May 2014.

[2] R. Agrawal, C. Faloutsos, and A. N. Swami. Efficient similarity search in sequence databases. In *FODO*, pages 69–84, 1993.

[3] J. Alon, S. Sclaroff, G. Kollios, and V. Pavlovic. Discovering clusters in motion time-series data. In *CVPR*, pages 375–381, 2003.

[4] A. J. Bagnall and G. J. Janacek. Clustering time series from ARMA models with clipped data. In *KDD*, pages 49–58, 2004.

[5] Z. Bar-Joseph, G. Gerber, D. K. Gifford, T. S. Jaakkola, and I. Simon. A new approach to analyzing gene expression time series data. In *RECOMB*, pages 39–48, 2002.

[6] G. E. Batista, E. J. Keogh, O. M. Tataw, and V. M. de Souza. CID: An efficient complexity-invariant distance for time series. *Data Mining and Knowledge Discovery*, pages 1–36, 2013.

[7] D. J. Berndt and J. Clifford. Using dynamic time warping to find patterns in time series. In *AAAI Workshop on KDD*, pages 359–370, 1994.

[8] Y. Cai and R. Ng. Indexing spatio-temporal trajectories with Chebyshev polynomials. In *SIGMOD*, pages 599–610, 2004.

[9] L. Chen and R. Ng. On the marriage of Lp-norms and edit distance. In *VLDB*, pages 792–803, 2004.

[10] L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD*, pages 491–502, 2005.

[11] Q. Chen, L. Chen, X. Lian, Y. Liu, and J. X. Yu. Indexable PLA for efficient similarity search. In *VLDB*, pages 435–446, 2007.

[12] Y. Chen, M. A. Nascimento, B. C. Ooi, and A. K. Tung. Spade: On shape-based pattern detection in streaming time series. In *ICDE*, pages 786–795, 2007.

[13] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[14] G. Das, K.-I. Lin, H. Mannila, G. Renganathan, and P. Smyth. Rule discovery from time series. In *KDD*, pages 16–22, 1998.

[15] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. Keogh. Querying and mining of time series data: Experimental comparison of representations and distance measures. *PVLDB*, 1(2):1542–1552, 2008.

[16] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, pages 419–429, 1994.

[17] M. Filippone, F. Camastra, F. Masulli, and S. Rovetta. A survey of kernel and spectral methods for clustering. *Pattern Recognition*, 41(1):176–190, 2008.

[18] E. Frentzos, K. Gratsias, and Y. Theodoridis. Index-based most similar trajectory search. In *ICDE*, pages 816–825, 2007.

[19] M. Friedman. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the American Statistical Association*, 32:675–701, 1937.

[20] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[21] M. Gavrilov, D. Anguelov, P. Indyk, and R. Motwani. Mining the stock market: Which measure is best? In *KDD*, pages 487–496, 2000.

[22] R. Giusti and G. E. Batista. An empirical comparison of dissimilarity measures for time series classification. In *BRACIS*, pages 82–88, 2013.

[23] S. Goddard, S. K. Harms, S. E. Reichenbach, T. Tadesse, and W. J. Waltman. Geospatial decision support for drought risk management. *Communications of the ACM*, 46(1):35–37, 2003.

[24] D. Q. Goldin and P. C. Kanellakis. On similarity queries for time-series data: Constraint specification and implementation. In *CP*, pages 137–153, 1995.

[25] G. H. Golub and C. F. Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[26] L. Gupta, D. L. Molfese, R. Tammana, and P. G. Simos. Nonlinear alignment and averaging for estimating the evoked potential. *IEEE Transactions on Biomedical Engineering*, 43(4):348–356, 1996.

[27] M. Halkidi, Y. Batistakis, and M. Vazirgiannis. On clustering validation techniques. *Journal of Intelligent Information Systems*, 17(2-3):107–145, 2001.

[28] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 3rd edition, 2011.

[29] R. Honda, S. Wang, T. Kikuchi, and O. Konishi. Mining of moving objects from time-series images and its application to satellite weather imagery. *Journal of Intelligent Information Systems*, 19(1):79–93, 2002.

[30] B. Hu, Y. Chen, and E. Keogh. Time series classification under more realistic assumptions. In *SDM*, pages 578–586, 2013.

[31] K. Kalpakis, D. Gada, and V. Puttagunta. Distance measures for effective clustering of ARIMA time-series. In *ICDM*, pages 273–280, 2001.

[32] Y. Katznelson. *An introduction to harmonic analysis*. Cambridge University Press, 2004.

[33] L. Kaufman and P. J. Rousseeuw. *Finding groups in data: An introduction to cluster analysis*, volume 344. John Wiley & Sons, 2009.

[34] E. Keogh. A decade of progress in indexing and mining large time series databases. In *VLDB*, pages 1268–1268, 2006.

[35] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Locally adaptive dimensionality reduction for indexing large time series databases. In *SIGMOD*, pages 151–162, 2001.

[36] E. Keogh and J. Lin. Clustering of time-series subsequences is meaningless: Implications for previous and future research. *Knowledge and Information Systems*, 8(2):154–177, 2005.

[37] E. Keogh and C. A. Ratanamahatana. Exact indexing of dynamic time warping. *Knowledge and Information Systems*, 7(3):358–386, 2005.

[38] C. Kin-pong and F. Ada. Efficient time series matching by wavelets. In *ICDE*, pages 126–133, 1999.

[39] F. Korn, H. V. Jagadish, and C. Faloutsos. Efficiently supporting ad hoc queries in large datasets of time sequences. In *SIGMOD*, pages 289–300, 1997.

[40] C.-S. Li, P. S. Yu, and V. Castelli. MALM: A framework for mining sequence database at multiple abstraction levels. In *CIKM*, pages 267–272, 1998.

[41] X. Lian, L. Chen, J. X. Yu, G. Wang, and G. Yu. Similarity match over high speed time-series streams. In *ICDE*, pages 1086–1095, 2007.

[42] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *BSMSP*, pages 281–297, 1967.

[43] R. N. Mantegna. Hierarchical structure in financial markets. *The European Physical Journal B-Condensed Matter and Complex Systems*, 11(1):193–197, 1999.

[44] W. Meesrikamolkul, V. Niennattrakul, and C. A. Ratanamahatana. Shape-based clustering for time series data. In *PAKDD*, pages 530–541. 2012.

[45] V. Megalooikonomou, Q. Wang, G. Li, and C. Faloutsos. A multiresolution symbolic representation of time series. In *ICDE*, pages 668–679, 2005.

[46] M. D. Morse and J. M. Patel. An efficient and accurate method for evaluating time series similarity. In *SIGMOD*, pages 569–580, 2007.

[47] A. Mueen, E. Keogh, and N. Young. Logical-shapelets: An expressive primitive for time series classification. In *KDD*, pages 1154–1162, 2011.

[48] P. Nemenyi. *Distribution-free Multiple Comparisons*. PhD thesis, Princeton University, 1963.

[49] A. Y. Ng, M. I. Jordan, and Y. Weiss. On spectral clustering: Analysis and an algorithm. In *NIPS*, pages 849–856, 2002.

[50] V. Niennattrakul and C. A. Ratanamahatana. Shape averaging under time warping. In *ECTI-CON*, pages 626–629, 2009.

[51] T. Oates. Identifying distinctive subsequences in multivariate time series by clustering. In *KDD*, pages 322–326, 1999.

[52] P. Papapetrou, V. Athitsos, M. Potamias, G. Kollios, and D. Gunopulos. Embedding-based subsequence matching in time-series databases. *TODS*, 36(3):17, 2011.

[53] J. Paparrizos and L. Gravano. k-Shape: Efficient and accurate clustering of time series. In *SIGMOD*, pages 1855–1870, 2015.

[54] F. Petitjean, A. Ketterlin, and P. Gançarski. A global averaging method for dynamic time warping, with applications to clustering. *Pattern Recognition*, 44(3):678–693, 2011.

[55] T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In *KDD*, pages 262–270, 2012.

[56] T. Rakthanmanon, E. J. Keogh, S. Lonardi, and S. Evans. Time series epenthesis: Clustering time series streams requires ignoring some data. In *ICDM*, pages 547–556, 2011.

[57] W. M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66(336):846–850, 1971.

[58] C. A. Ratanamahatana and E. Keogh. Making time-series classification more accurate using learned constraints. In *SDM*, pages 11–22, 2004.

[59] E. J. Ruiz, V. Hristidis, C. Castillo, A. Gionis, and A. Jaimes. Correlating financial time series with micro-blogging activity. In *WSDM*, pages 513–522, 2012.

[60] H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 26(1):43–49, 1978.

[61] Y. Shou, N. Mamoulis, and D. Cheung. Fast and exact warping of time series using adaptive segmental approximations. *Machine Learning*, 58(2-3):231–267, 2005.

[62] K. Uehara and M. Shimada. Extraction of primitive motion and discovery of association rules from human motion data. In *Progress in Discovery Science*, pages 338–348. 2002.

[63] M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, and E. Keogh. Indexing multidimensional time-series. *The VLDB Journal*, 15(1):1–20, 2006.

[64] M. Vlachos, G. Kollios, and D. Gunopulos. Discovering similar multidimensional trajectories. In *ICDE*, pages 673–684, 2002.

[65] H. Wang, Y. Cai, Y. Yang, S. Zhang, and N. Mamoulis. Durable queries over historical time series. *TKDE*, 26(3):595–607, 2014.

[66] X. Wang, A. Mueen, H. Ding, G. Trajcevski, P. Scheuermann, and E. Keogh. Experimental comparison of representation methods and distance measures for time series data. *Data Mining and Knowledge Discovery*, 26(2):275–309, 2013.

[67] T. Warren Liao. Clustering of time series data - a survey. *Pattern Recognition*, 38(11):1857–1874, 2005.

[68] Y. Xiong and D.-Y. Yeung. Mixtures of ARMA models for model-based time series clustering. In *ICDM*, pages 717–720, 2002.

[69] J. Yang and J. Leskovec. Patterns of temporal variation in online media. In *WSDM*, pages 177–186, 2011.

[70] L. Ye and E. Keogh. Time series shapelets: A new primitive for data mining. In *KDD*, pages 947–956, 2009.

[71] J. Zakaria, A. Mueen, and E. Keogh. Clustering time series using unsupervised-shapelets. In *ICDM*, pages 785–794, 2012.

Changes to the TODS Editorial Board

Christian S. Jensen
csj@cs.aau.dk

It is of paramount importance for a scholarly journal such as *ACM Transactions on Database Systems* to have a strong editorial board of respected, world-class scholars. The editorial board plays a fundamental role in attracting the best submissions, in ensuring insightful and timely handling of submissions, in maintaining the high technical standards of the journal, and in maintaining the reputation of the journal. Indeed, the journal's Associate Editors, along with the reviewers and authors they work with, are the primary reason that TODS is a world-class journal.

Retiring Associate Editors

As of January 1, 2016, five Associate Editors have ended their terms:

- Wenfei Fan
- Chris Jermaine
- Tova Milo
- Lucian Popa
- Divesh Srivastava

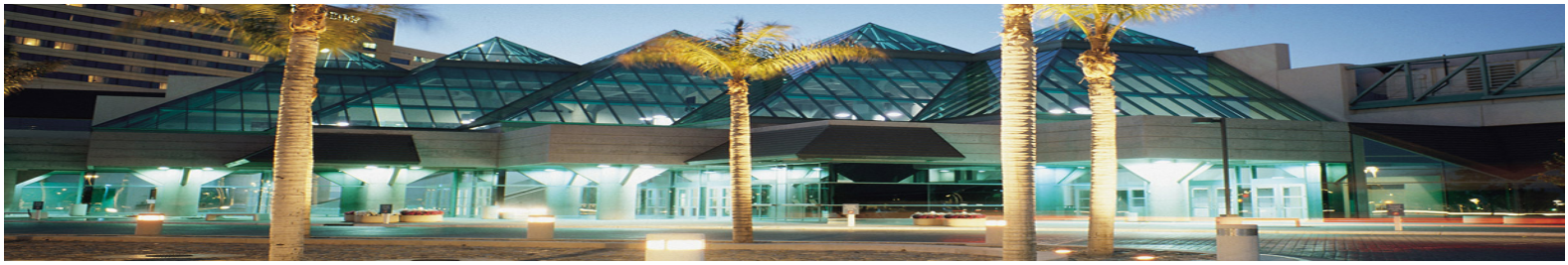
They have served on the editorial board for six, or between six and seven, years. In addition, they will stay on until they complete their current loads. They have each provided very substantial high-caliber service to the journal and the database community. Specifically, I have never seen them compromise on quality when handling submissions, and I believe that they have uniformly made sound technical decisions. We are all fortunate that they have donated their time and unique expertise during these years.

New Associate Editors

Also as of January 1, 2016, five new Associate Editors have joined the editorial board:

- Marcelo Arenas (`http://web.ing.puc.cl/~marenas`)
- Gao Cong (`http://www.ntu.edu.sg/home/gaocong/`)
- Torsten Grust (`http://db.inf.uni-tuebingen.de/team/grust`)
- Peter Haas (`http://researcher.watson.ibm.com/researcher/view.php?person=us-phaas`)
- Wim Martens (`http://www.theoinf.uni-bayreuth.de/en/team/Martens_Wim/`)

All five are highly regarded scholars in the field of database systems. We are very fortunate that these outstanding scholars are willing to volunteer their valuable time and indispensable expertise for handling manuscripts for the benefit of our scientific community. Indeed, I am gratified that they have committed to help TODS continue to evolve and improve, and I am looking forward to working with them.



Call for Papers SoCC 2016

ACM Symposium on Cloud Computing

5-7 October 2016, Santa Clara, California, USA

The annual ACM Symposium on Cloud Computing brings together researchers, developers, users, and practitioners interested in cloud computing. We solicit original contributions on all aspects of cloud computing. We particularly encourage submissions on the research, development, practice, and experience of cloud computing systems and data management. Specific topics of interest include but are not limited to the following, when related to cloud:

- Administration and Manageability
- Application, Analytics, Database as a Service
- Data Services Architectures
- Data Markets
- Distributed and Parallel Query Processing
- Distributed Systems
- Energy Efficiency and Management
- Fault Tolerance, High Availability, and Reliability
- IoT Infrastructure
- Large Scale Cloud Applications
- Multi-Tenancy
- Networking and SDNs
- Platform for Services
- Privacy of Data and Computation
- Programming Models
- Provisioning and Metering
- Query Optimization
- Resource Management
- Scientific Data Management
- Security of Infrastructure and Services
- Service Level Agreements
- Storage Systems and New Technologies
- Transactional Models
- Virtualization, Containers, VMs

Submission

Authors can submit a paper in one of the following categories:

- *Research Papers* describe original research, where novelty is a primary consideration.
- *Experience Papers* describe experiences with existing systems, where lessons from the experience are a primary consideration.
- *Vision Papers* describe speculative but well-reasoned and thought-provoking ideas, where insight is a primary consideration.

Submissions must be entered online at http://socc2016.org/submissions

SoCC 2016 also accepts poster submissions. Details and deadlines are available in the call for posters on the conference website: http://acmsocc.github.io/2016/

Important Dates

- Abstract submission: 17 May 2016
- Paper submission: 24 May 2016
- Acceptance notification: 29 July 2016
- Camera-ready deadline: 26 August 2016
- Conference dates: 5-7 October 2016

Conference Officers

General Chair: **Brian Cooper**

PC Chairs: **Marcos K. Aguilera**
Yanlei Diao