An Effective Syntax for Bounded Relational Queries

Citation for published version:
https://doi.org/10.1145/2882903.2882942

Digital Object Identifier (DOI):
10.1145/2882903.2882942

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
ACM SIGMOD Conference on Management of Data (SIGMOD 2016)

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
An Effective Syntax for Bounded Relational Queries

Yang Cao\textsuperscript{1,2} Wenfei Fan\textsuperscript{1,2} \\
\textsuperscript{1}School of Informatics, University of Edinburgh \{Y.Cao-17@sms, wenfei@inf\}.ed.ac.uk \\
\textsuperscript{2}RCBD and SLISDE, Beihang University

ABSTRACT

A query $Q$ is \textit{boundedly evaluable} under a set $\mathcal{A}$ of access constraints if for all datasets $D$ that satisfy $\mathcal{A}$, there exists a fraction $D_Q$ of $D$ such that $Q(D) = Q(D_Q)$, and the size of $D_Q$ and time for identifying $D_Q$ are both \textit{independent of} the size of $D$. That is, we can compute $Q(D)$ by accessing a bounded amount of data no matter how big $D$ grows. However, while desirable, it is undecidable to determine whether a query in relational algebra (RA) is boundedly evaluable.

In light of the undecidability, this paper develops an effective syntax for bounded RA queries. We identify a class of covered RA queries such that under $\mathcal{A}$, (a) every boundedly evaluable RA query is equivalent to a covered query, (b) every covered RA query is in relational algebra (RA), and (c) it takes $\text{PTIME}$ in $|Q|$ and $|\mathcal{A}|$ to check whether $Q$ is covered by $\mathcal{A}$. We provide quadratic-time algorithms to check the coverage of $Q$, and to generate a bounded query plan for covered $Q$. We also study a new optimization problem for minimizing access constraints for covered queries. Using real-life data, we experimentally verify that a large number of access constraints for covered queries. Using real-life data, we experimentally verify that a large number of RA queries in practice are covered, and that bounded query plans improve RA query evaluation by orders of magnitude.

Keywords

Boundedly evaluable; query evaluation; big data

1. INTRODUCTION

Querying big data is cost prohibitive. Given a query $Q$ and a dataset $D$, it is NP-complete to decide whether a tuple is in the query answer $Q(D)$ when $Q$ is an SP query (selection, projection and Cartesian product), and PSPACE-complete if $Q$ is in relational algebra (RA)\textsuperscript{[5]}. When $D$ is big, it is hard, if not impossible, to compute $Q(D)$ within constrained resources as time and available processors.

To tackle this problem, a notion of boundedly evaluable queries has recently been studied\textsuperscript{[13, 14, 18, 19]}. The idea is to compute $Q(D)$ by accessing only a fraction $D_Q$ of $D$ that suffices to answer $Q$ in $D$, instead of the entire $D$. To identify $D_Q$, it makes use of a set $\mathcal{A}$ of access constraints, which are a combination of simple cardinality constraints and their indices. Under $\mathcal{A}$, $Q$ is \textit{boundedly evaluable} if for all datasets $D$ that satisfy $\mathcal{A}$, there exists $D_Q$ such that

- $Q(D_Q) = Q(D)$, and
- the time for identifying $D_Q$ from $D$ and hence the size $|D_Q|$ of $D_Q$ are determined by $Q$ and $\mathcal{A}$ only.

That is, $Q(D)$ can be computed by accessing (identifying and fetching) a small $D_Q$ with the indices in $\mathcal{A}$, such that $|D_Q|$ is \textit{independent of} $|D|$, no matter how big $D$ grows.

The idea has proven effective on some real-life datasets, 77% of SP queries\textsuperscript{[14]} and 60% of graph pattern queries\textsuperscript{[13]} are found boundedly evaluable on average, under a few hundreds simple access constraints, outperforming conventional query evaluation approaches by orders of magnitude.

However, it is undecidable to determine whether an RA query $Q$ is boundedly evaluable under a set $\mathcal{A}$ of access constraints\textsuperscript{[19]}. Set difference (universal quantification) of RA queries states that each person $p_0$ denoting "me":

- $Q_0(\text{cid}) = Q_1(\text{cid}) - Q_2(\text{cid})$, where
- $Q_1(\text{cid}) = \pi_{\text{cid}}(\text{friend}(p_0, \text{fid}) \cap \text{dine}(\text{pid}, \text{cid}, \text{May}, 2015))$
- $Q_2(\text{cid}) = \pi_{\text{cid}}(\text{dine}(p_0, \text{cid}, \text{May}, 2015))$

Dataset $D_0$ may be big, with billions of users and trillions of friend links\textsuperscript{[21]}. It is costly to compute $Q_0(D_0)$ directly.

Now consider a set $\mathcal{A}_0$ of real-life cardinality constraints:

- $\psi_1$: friend($p_0$ $\rightarrow$ $\text{fid}$, 5000);
- $\psi_2$: dine($\text{pid}$, $\text{year}$, $\text{month}$) $\rightarrow$ $\text{cid}$, 31);
- $\psi_3$: dine($\text{pid}$, $\text{cid}$) $\rightarrow$ $\text{pid}$, 1);
- $\psi_4$: cafe($\text{cid}$ $\rightarrow$ $\text{city}$, 1).

Here $\psi_i$ specifies a constraint imposed by Facebook\textsuperscript{[16]}: a limit of 5000 friends per user; $\psi_2$ states that each person dines in at most 31 restaurants each month; $\psi_3$ says that (pid, cid) is a "key" of the pair, and $\psi_4$ states that each restaurant id is associated with a single city. Indices can be built on $D_0$ based on $\psi_2$ such that given a person, it
returns all the ids of her friends by accessing at most 5000 friend tuples; similarly for $\psi_2$, $\psi_3$ and $\psi_4$. These indices and constraints together are called access constraints [19].

Given the access constraints, we can compute $Q_1(D_0)$ by accessing at most 315000 tuples from $D_0$, instead of trillions. (1) We identify and fetch $T_1$ of at most 5000 fid's of friend tuples with pid = $p_0$, by using the index built for $\psi_1$. (2) For each fid value $f$ in $T_1$, we fetch $T_2$ of at most 31 cid's of dine tuples with fid $= f$, year = 2015 and month = MAY, leveraging the index for $\psi_4$. (3) For each cid in $T_2$, we fetch its cafe tuple by using the index for $\psi_3$, and return a set $T_3$ of cid's from these tuples with city = NYC. The query plan fetches at most 5000 + 5000 $\times$ 2 tuples only, all using indices, to compute $Q_1(D_0)$ no matter how big $D_0$ is. Therefore, $Q_1$ is boundedly evaluable under $A_0$. However, query $Q_2$ is not boundedly under $A_0$: we cannot make use of any indices above when accessing the (possibly huge) dine relation given pid = $p_0$ alone. Since the set difference operator in $Q_2$ forces us to check all tuples in $Q_2(D_0)$, one might think that $Q_2$ is not boundedly either.

Nonetheless, observe that $Q_0$ is equivalent to $Q_0'(\text{cid}) = Q_1(\text{cid}) - Q_2(\text{cid})$, where $Q_1(\text{cid}) = Q_1(\text{cid}) \cap_{\text{cid} = \text{cid}} Q_2(\text{cid})$. Moreover, $Q_2$ is boundedly evaluable. Indeed, for each cid value returned by $Q_1(D_0)$ (i.e., $T_3$ above), we can check whether ($p_0$, cid) is a pair occurring in relation dine, by accessing one tuple via the index for $\psi_4$. We return all those cid's that pass the check. Thus we can answer $Q_2(D_0)$ by accessing $5000 \times 31$ tuples. Therefore, $Q_0$ is equivalent to boundedly $Q_0'$, with a query plan consisting of the plan for $Q_1$, followed by the plan for $Q_2$; it accesses at most 470000 tuples only, no matter how big $D_0$ grows. This shows that $Q_0$ is actually boundedly evaluable under $A_0$.

This example tells us that to decide whether an RA (SQL) query is bounded, it is often necessary to check query equivalence, which is undecidable for RA queries in the presence of set difference [5]. An open question [14, 18, 19] asks whether it is still possible to make practical use of bounded evaluability for answering RA queries, given the undecidability?

**Contributions.** This paper is to answer the open question. We approach the problem by identifying an effective syntax for boundedly evaluable RA queries. That is, a class $\mathcal{L}$ of RA queries such that under a set $A$ of access constraints,

(a) every boundedly evaluable RA query is equivalent to a query in $\mathcal{L}$, i.e., $\mathcal{L}$ expresses all bound RA queries;
(b) every query $Q$ in $\mathcal{L}$ is boundedly evaluable; and
(c) it takes \textsc{Ptime} (polynomial time) in $|Q|$ and $|A|$ to syntactically check whether $Q$ is in $\mathcal{L}$.

That is, $\mathcal{L}$ identifies the core subclass of boundedly evaluable RA queries, without sacrificing their expressive power.

The study of bounded evaluability is analogous, to an extent, to the study of safe relational calculus queries, which are also undecidable. Effective syntax was first studied 30 years ago [20, 32, 34], to express all safe queries up to equivalence. As observed in [20], “several commercial database query systems give intuitively unexpected results on such queries” (unsafe queries); this is evidenced by a real-life example tested with SQL and QUEL [35]. Effective syntax imposes syntactical restrictions on undecidable safe queries, such that the restricted class is efficiently decidable.

Along the same lines, effective syntax allows us to make practical use of bounded evaluability. (1) It provides us with a guideline for formulating boundedly evaluable queries, just like its counterpart for safe queries. (2) As will be shown shortly, bounded evaluability analysis can be readily incorporated into commercial DBMSs. Given an input RA query $Q$, it first checks whether $Q$ is in $\mathcal{L}$, in \textsc{Ptime} by condition (c) above; if so, it generates a bound query plan for $Q$ by using indices in $A$, which is warranted to exist by (b). (3) By (a), if $Q$ is boundedly evaluable, it can be expressed in $\mathcal{L}$. Hence query rewriting rules can be implemented to transform $Q$ to an equivalent query in $\mathcal{L}$, to an extent.

More specifically, we provide theoretical results and practical methods for the bounded evaluability of RA as follows.

(1) We develop an effective syntax $\mathcal{L}$ for boundedly evaluable RA queries (Section 3), referred to as covered queries. In a nutshell, an RA query $Q$ is covered if for any relation in $Q$, its attributes needed for answering $Q$ can be fetched via the indices in $A$, in time bounded by the cardinality constraints of $A$. We prove that every boundedly evaluable RA query under $A$ is also covered by $A$ (i.e., property (a)).

(2) We develop an algorithm for checking covered queries (Section 4). Given an RA query $Q$ and a set $A$ of access constraints, the algorithm decides whether $Q$ is covered by $A$ in $O(|Q|^2 + |A|)$-time, where $|Q|$ is the size of $Q$ and $|A|$ is the total length of access constraints in $A$, independent of the size $|D|$ of dataset $D$. In practice, $|Q|$ and $|A|$ are typically much smaller than $|D|$. This proves property (c).

(3) We provide an algorithm to generate query plans for covered queries (Section 5). Given an RA query $Q$ covered by $A$, the algorithm generates a query plan $\xi$ of length $O(|Q| + |A|)$ such that for any dataset $D$ that satisfies $A$, $\xi$ computes $Q(D)$ by accessing a bounded amount of data determined by $Q$ and $A$. The algorithm is based on a nontrivial characterization of covered RA queries and takes $O(|Q|(|Q| + |A|))$ time, again independent of $|D|$. This proves property (b).

(4) We also study a new optimization problem (Section 6). Given a query $Q$ covered by $A$, it is to find a subset $A_m \subseteq A$ such that $Q$ remains covered by $A_m$, and the estimated data access via $A_m$ is minimized. We show that the problem is \textsc{NP}-complete and is not in \textsc{APX}, i.e., it has no \textsc{Ptime} constant-factor approximation algorithm. Nonetheless, we develop efficient heuristic algorithms with performance guarantees, some with reasonable approximation bounds.

(5) We show how bounded evaluability analysis can be integrated into existing DBMSs (Section 7). Given an RA query $Q$ and a set $A$ of access constraints, we check whether $Q$ is covered by $A$, and if so, we generate a bound query plan for $Q$ with minimal constraints in $A$, and compute $Q(D)$ by accessing a small fraction $D_Q$ of $D$, all by using the algorithms described above. We also show how access constraints can be discovered and incrementally maintained.

(6) We implement our approach on top of MySQL and PostgreSQL and experimentally evaluate its effectiveness using two real-life datasets and a commercial benchmark (query templates and datasets; Section 8). We find the following on the real-life data: under a set $A$ of at most 266 access constraints, on average (a) 67.5% of randomly generated RA queries are boundedly evaluable, among which 83.5% are covered; (b) our query plans outperform MySQL and PostgreSQL that use the same indices by at least 3 orders of magnitude, and the gap gets larger on bigger data; (c) our plans
access only 0.0019% of the data; that is, they “reduce” \( D \) from PB to GB; and (d) the indices account for 14.8% of the original data. We also find that (e) our algorithms for coverage checking, plan generation and minimizing access constraints are all efficient: they take at most 199ms in all cases.

These results settle the open question for the study of RA boundedly evaulability, from theory to practice. They suggest an approach to answering queries within bounded resources, by adding the functionality of bounded evaluation to existing DBMS. It is a common practice for decades in query evaluation to access as little data as possible, rather than the entire dataset, by making use of various indices. This work is an effort to formalize the idea, to decide when it is feasible to answer a query within bounded resources, and to provide a systematic method to achieve it.

Related work. We categorize previous work as follows. Bounded evaulability and effective syntax. The study of bounded evaulability was motivated by the idea of scale independence [6, 7], which is to guarantee that a bounded amount of work is required to execute all queries in an application, regardless of the size of the underlying data. The notion was formalized in [19], focusing on query answering in a particular given dataset. Bounded evaulability was proposed in [18], which extends [19] by ranging over all datasets that satisfy a set \( A \) of access constraints. A query \( Q \) is called boundedly evaluable under \( A \) if it has a boundedly query plan; such a plan allows data access only via indices embedded in \( A \), and interleaves data fetching and relational operations, to answer \( Q \) by accessing a bounded amount of data.

It is shown that for any SPCU query \( Q \) and any set \( A \) of access constraints, it is decidable but \( \text{EXPSPACE-hard} \) to decide whether \( Q \) is boundedly evaluable under \( A \) [18]; but it is undecidable when \( Q \) is in RA [19]. Effective syntax was explored for bounded SPC queries [18]. However, it was left open in [18] whether there exists an effective syntax for boundedly evaluable RA (i.e., FO, first-order logic) queries.

This work answers the open question in positive by providing a \( \text{PTIME} \) effective syntax for boundedly evaluable RA queries. It is radically different from the one for SPC [18], which becomes \( \Sigma_2 \)-complete when extended to SPCU, and undecidable for RA. The result of this work allows existing DBMS to support boundedly evaluable RA queries.

Effective boundedness. A notion of effective boundedness was studied for SPC [14], based on a restricted form of query plans in which data fetching must be completed before any relational operations can start. It was also studied for graph pattern queries via simulation and subgraph isomorphism [13], which are quite different from relational queries.

This work differs from [14] in the following. (1) We study effective syntax for RA, while [14] focuses on checking and answering SPC queries of a special form. Our main result is an effective syntax for boundedly evaluable RA queries, which is nontrivial since not every query class has an effective syntax [32]. This issue is not considered in [14]. (2) Bounded evaulability is much harder to decide than effective boundedness. For SPC, bounded evaulability is \( \text{EXPSPACE-hard} \) [18], but effective boundedness is in \( \text{PTIME} \). (3) We study RA, in contrast to SPC [14]. RA is equivalent to FO on relations [5], while SPC is a conjunctive fragment of FO, and does not support disjunction (union) and universal quantification (set difference). (4) Our methods for checking covered RA queries and for generating query plans are differ-
Bounded evaluability [18]. A query plan \( \xi \) is **boundedly evaluable under an access schema** \( \mathcal{A} \) (or simply **bounded**) if

1. for each operation \( \text{fetch}(X \in T, R, Y) \) in \( \xi \), there exists an access constraint \( R(X \rightarrow Y, N) \in \mathcal{A} \); and
2. the length of \( \xi \) is determined by \(|R|, |A| \) and \(|Q|\) only, which are the sizes of \( R, \mathcal{A} \), and \( Q \), respectively.

An RA query \( Q \) is **boundedly evaluable under \( \mathcal{A} \) (or bounded)** if it has a boundedly evaluable query plan under \( \mathcal{A} \).

Intuitively, if \( Q \) is bounded, then there exists a bounded query plan \( \xi \) for \( Q \) such that for all instances \( D \) of \( R \) that satisfy \( \mathcal{A} \), it fetches \( D_Q \) from \( D \) via the indices in \( \mathcal{A} \) such that \( Q(D) = Q(D_Q) \). Moreover, \(|D_Q|\) is determined by \( Q \) and constants in \( \mathcal{A} \) only, independent of \(|D|\), where \(|D|\) denotes the total number of tuples in \( D \). The time for identifying \( D_Q \) (checking indices) and fetching \( D_Q \) is also independent of \(|D|\) (assuming that given an \( X \)-value \( a \), it takes \( O(N) \) time to fetch \( DXY(X = a) \) in \( D \) with the index in \( R(X \rightarrow Y, N) \)).

**Example 2:** Recall \( Q_0 \) and \( A_0 \) from Example 1. A boundedly evaluable query plan for \( Q_0 \) under \( A_0 \) as follows.

The analysis of bounded evaluability is more intriguing for RA queries than for SPC, as illustrated below.

**Example 3:** Consider an access schema \( A_1 \) and RA query \( Q_3 \) defined on relation schemas \( R(A, B, E) \) and \( S(F, G, H) \):

1. \( A_1 = \{R(AB \rightarrow E, N), S(F \rightarrow GH, 2), S(GH \rightarrow GH, 1)\} \),
2. \( Q_3 = Q_3^1 \cup Q_3^2 \), where \( Q_3^1 = \pi_x(R(1, x, y) \bowtie S(w, x, y) \bowtie S(w, 1, x) \bowtie S(w, x, x)) \) and \( Q_3^2 = \pi_x(R(1, x, y) \bowtie S(w, 1, x) \bowtie S(w, x, x)) \), where \( \bowtie \) denotes natural join.

At a first glance, \( Q_3 \) seems not boundedly evaluable, since we cannot retrieve \( x \) and \( w \) values using indices in \( A_1 \) and thus cannot get \( y \) for \( Q_3^1 \). Similarly, we cannot get \( w \) and \( x \) values for \( Q_3^2 \). However, under \( S(F \rightarrow GH, 2) \in A_1 \), observe that \((x, y)\) must be equal to either \((1, x)\) or \((x, x)\) in all tuples retrieved from instance of \( S \) by any query plan for \( Q_3^1 \). In other words, under \( A_1 \), the SPC sub-query \( Q_3^1 \) reduces to SPCU \( Q_{31}^1 \cup Q_{31}^3 \), where \( Q_{31}^1 = \pi_x(R(1, 1, x) \bowtie S(w, 1, x) \bowtie S(w, x, x)) \) and \( Q_{31}^3 = \pi_x(R(1, x, y) \bowtie S(w, 1, x) \bowtie S(w, x, x)) \), which are made distinct via renaming. For an access constraint \( \phi = R(X \rightarrow Y, N) \) and a renaming \( S \) of \( R \) in \( Q \), we refer to \( S(X \rightarrow Y, N) \) as the **actualized constraint** of \( \phi \) on \( S \), and to the set of all actualized constraints of \( \mathcal{A} \) as the **actualized access schema** of \( \mathcal{A} \) on \( Q \). We consider \( w.l.o.g. \) normalized \( Q \) and actualized \( \mathcal{A} \) only, based on the lemma below.

**Lemma 1:** Given any RA query \( Q \) and access schema \( \mathcal{A} \) over relational schema \( R \), one can compute the actualized access schema \( \mathcal{A}' \) from \( Q \) and \( \mathcal{A} \) in \( O(|Q| \cdot |A|) \)-time such that

1. for any instance \( D \) of \( R \), \( \mathcal{A}' \subseteq \mathcal{A} \); and
2. \( Q \) is boundedly evaluable under \( \mathcal{A} \) iff \( Q' \) is boundedly evaluable under \( \mathcal{A}' \) (iff for if and only if).

**3. AN EFFECTIVE SYNTAX**

Essential to practical use of bounded evaluability is the following problem. Given a query \( Q \) and an access schema \( \mathcal{A} \), it is to decide whether \( Q \) is boundedly evaluable under \( \mathcal{A} \). The problem is undecidable for RA queries \( Q \) [19].

The undecidability motivates us to find an effective syntax \( \mathcal{L} \) for boundedly evaluable RA queries (see Section 1). We identify such an \( \mathcal{L} \), referred to as the class of **covered queries**.

**Covered queries.** We now define covered queries, starting with SPC. Intuitively, an SPC query \( Q \) is covered if for any relation \( S \) in \( Q \), all the attributes of \( S \) needed to answer \( Q \) can be fetched via indices in \( \mathcal{A} \) and moreover, their sizes are bounded by the cardinality constraints of \( \mathcal{A} \).

Consider an SPC query \( Q = \pi_2 \sigma_{c}(S_1 \times \ldots \times S_n) \) defined over a relational schema \( R \), where \( Z \) is a set of attributes of \( R, C \) is the selection condition of \( Q \), and \( S_1 \)'s are distinct relations after renaming (Lemma 1). We use \( \Sigma_Q \) to denote the set of all equality atoms \( \sigma = A' \lor A = c \) derived from \( C \) by the transitivity of equality. For any sets \( X \) and \( X' \) of attributes of \( Q \), we write \( \Sigma_Q \models X = X' \) if \( X = X' \) can be derived from \( \Sigma_Q \), which can be checked in \( O(\max(|X|, |X'|)) \) time after an \( O(|Q|^2) \)-time preprocessing of \( Q \).

**Coverage.** The set of **covered attributes** of \( Q \) by an access schema \( \mathcal{A} \), denoted by \( \text{cov}(Q, \mathcal{A}) \), includes attributes that can be accessed via indices in \( \mathcal{A} \). It is defined as follows:

- if \( \Sigma_Q \models \sigma_{c,r} \), then \( A \in \text{cov}(Q, \mathcal{A}) \);
- if \( R(\emptyset \rightarrow Y, N) \in \mathcal{A} \), then \( R[X] \subseteq \text{cov}(Q, \mathcal{A}) \);
- if \( R[X] \subseteq \text{cov}(Q, \mathcal{A}) \) and \( \Sigma_Q \models R[X] = S[Y] \), then \( S[Y] \subseteq \text{cov}(Q, \mathcal{A}) \); and
- if \( R(\emptyset \rightarrow Y, N) \in \mathcal{A} \) and \( R[X] \subseteq \text{cov}(Q, \mathcal{A}) \), then \( R[Y] \subseteq \text{cov}(Q, \mathcal{A}) \).

Here \( R(\emptyset \rightarrow Y, N) \) is an access constraint stating that there are at most \( N \) distinct \( X \) values in an instance of \( R \), e.g., there exist at most 12 distinct months per year.

**Covered SPC.** Denote by \( X_Q \) the set of attributes in an SPC query \( Q \) that occur in either its selection condition \( C \) or the projection attributes \( Z \) of \( Q \). We say that \( Q \) is

- **fetchable via \( \mathcal{A} \)** if \( X_Q \subseteq \text{cov}(Q, \mathcal{A}) \); and
- **indexed by \( \mathcal{A} \)** if for each relation name \( S \) in \( Q \), there is an actualized constraint \( S(X \rightarrow Y, N) \) of \( A \) such that

  - \( S[X] \subseteq \text{cov}(Q, \mathcal{A}) \), and
  - \( S[XY'] \) includes all attributes of \( S \) that are in \( X_Q \), i.e., attributes \( XY \) come from the same tuple.

An SPC query \( Q \) is **covered by \( \mathcal{A} \)** if \( Q \) is both fetchable via \( \mathcal{A} \) and indexed by \( \mathcal{A} \). That is, all attributes needed by \( Q \) can be fetched using indices of \( \mathcal{A} \) and are bounded by \( \mathcal{A} \).
Covered RA. We represent an RA query $Q$ as its query (syntactic) tree $T^Q$ [5]. To simplify the discussion, we say that an RA query $Q'$ is a sub-query of $Q$ if $T^Q$ is a sub-tree of $T^{Q'}$.

A max SPC sub-query of $Q$ is a sub-query $Q_s$ such that

- $Q_s$ is an SPC query, and
- there exists no sub-query $Q'_s$ of $Q$ such that it is also in SPC, $Q_s \neq Q'_s$, and $Q_s$ is a sub-query of $Q'_s$.

An RA query $Q$ is covered by an access schema $A$ if for all max SPC sub-queries $Q_s$ of $Q$, $Q_s$ is covered by $A$. Similarly, $Q$ is fetchable via $A$ (resp. indexed by $A$) if each max sub-SPC sub-query is fetchable via $A$ (resp. indexed by $A$).

Intuitively, an RA query $Q$ is “normalized” by pushing set difference to the top level, on (unions of) max SPC sub-queries. These max SPC sub-queries characterize all relation attributes that need to be accessed when answering $Q$.

Example 4: For the queries and $A_0$ of Example 1, $Q_1$ and $Q_3$ are covered by $A_0$, but $Q_2$ is not. Indeed, $X_{Q_2} = \{x_{p_0}, fid, pid, cid, x_{3NYC}, x_{2015}, cid', x_{NYC}\} = \text{cov}(Q_1, A_0)$, where $x_{d}$ denotes the attribute corresponding to a constant $d$ in $Q_1$. Hence $Q_1$ is fetchable via $A_0$; moreover, $Q_1$ is indexed by $A_0$ since friend, dine and cafe are indexed by $\psi_1, \psi_2$ and $\psi_4$, respectively; similarly for $Q_2$. However, $Q_2$ is not fetchable via $A_0$ since $\text{cov}(Q_2, A_0) = \{x_{p_0}\}$ but $X_{Q_2} = \{x_{p_0}, cid\}$, and relation dine is not indexed by any constraint in $A_0$ for $Q_2$. As a result, $Q_0$ is covered by $A_0$ since both of its max SPC sub-queries $Q_1$ and $Q_3$ are covered by $A_0$. In contrast, $Q_0$ is not covered by $A_0$ since $Q_2$ is not.

The main result of the paper is as follows.

Theorem 2: Under access schema $A$, for any RA query $Q$, (1) if $Q$ is boundedly evaluable under $A$, then $Q$ is $A$-equivalent to an RA query $Q'$ that is covered by $A$; (2) if $Q$ is covered, then $Q$ is boundedly evaluable; and (3) it takes PTIME to check whether $Q$ is covered by $A$.

That is, we reduce the problem of deciding RA bounded evaluality to syntactic checking of covered queries, without losing the expressive power. Indeed, all boundedly evaluable RA queries have an $A$-equivalent covered version. For these RA queries, covered queries play the same role as range-safe RA queries for checking “the safety” SQL queries [5].

Proof sketch of Theorem 2(1). We will provide Theorem 2(2) and (3) in Sections 4 and 5, respectively. For (1), we show that for any RA query $Q$, if it has a bound query plan $\xi$ under $A$, then $\xi$ can be converted to a query $Q_1$ covered by $A$ such that $Q_1 \equiv_A Q$ (see Appendix B).

4. CHECKING COVERED QUERIES

We next give a constructive proof of Theorem 2(2) by providing an algorithm for checking covered queries, denoted by CovChk. Given an access schema $A$ and an RA query $Q$, CovChk returns “yes” if $Q$ is covered by $A$, and “no” otherwise. Below we show a result stronger than Theorem 2(3).

Proposition 3: Given an access schema $A$ and an RA query $Q$, algorithm CovChk determines whether $Q$ is covered by $A$ in $O(|Q|^2 + |A|)$ time.

Note that checking is conducted at the meta level on $Q$ and $A$ only, independent of (possibly big) datasets $D$.

The algorithm is shown in Fig. 1, consisting of two parts. It first finds the set $S_Q$ of all max SPC sub-queries of $Q$ (line 1). It then checks whether all queries in $S_Q$ are covered by $A$ (lines 2-5). It returns “yes” if and only if so (line 6).

Identifying max SPC sub-queries. CovChk computes the set $S_Q$ by a bottom-up scan of the query tree of $Q$. This is done in time linear in $|Q|$, since each relation of $Q$ occurs in only one max SPC sub-query of $Q$, by the assumption that relation names in $Q$ are distinct (see Section 2).

Checking coverage of SPC sub-queries. We next focus on how to check whether an SPC sub-query $Q_s$ is covered by $A$, i.e., indexed by $A$ and fetchable via $A$ (Section 3). While index checking is straightforward, the fetching condition is more involved, and is based on its connection with the implication analysis of functional dependencies (FDs) [5]. To establish the connection, we need the following notions.

Unification. A unification function $\rho_U$ is an attribute renaming function: for all attributes $A$ and $A'$ in $S_Q$, $\rho_U(A) = \rho_U(A')$ (assigned the same name) if and only if $\Sigma_Q \vdash A = A'$. For a set $X$, we denote by $\rho_U(X)$ the set $\{\rho_U(A) \mid A \in X\}$.

Induced FDs. For $\phi = R(A \rightarrow B, N) \in A$, we call $\rho_U(R[A]) \rightarrow \rho_U(R[B])$ an induced FD from $Q$ and $\phi$. We denote the set of all induced FDs from $Q$ and constraints in $A$ by $\Sigma_{Q,A}$.

Example 5: For $Q_1$ and $A_0$ of Example 1, define a unification function $\rho_U$ such that $\rho_U(\text{friend}[\text{pid}]) = \text{pid}$, $\rho_U(\text{friend}[\text{fid}]) = \text{fid}$, $\rho_U(\text{dine}[\text{pid}]) = \text{fid}$, $\rho_U(\text{dine}[\text{cid}]) = \text{cid}$, $\rho_U(\text{dine}[\text{year}]) = \text{year}$, $\rho_U(\text{dine}[\text{month}]) = \text{month}$, $\rho_U(\text{cafe}[\text{cid}]) = \text{cid}$ and $\rho_U(\text{cafe}[\text{city}]) = \text{city}$. Then $\Sigma_{Q_1,A_0}$ consists of the following induced FDs: $\text{pid} \rightarrow \text{fid}$, $\text{tid} \rightarrow \text{did}$, $\text{cid} \rightarrow \text{did}$, $\text{did} \rightarrow \text{cid}$, $\text{did} \rightarrow \text{cid}$.

We now give the connection between induced FDs and fetchable SPC queries. For an SPC query $Q_s$, let $X_{Q_s}$ be the set of all its attributes that occur in its selection condition or projection attributes, and $X_{Q_s}^C \subseteq X_{Q_s}$ be the set of attributes $A$ in $Q_s$ such that $\Sigma_{Q_s} \vdash A = c$ for some constant $c$. Let $X_{Q_s} = \rho_U(X_{Q_s})$ and $X_{Q_s}^C = \rho_U(X_{Q_s}^C)$. Then we have:

Lemma 4: An SPC query $Q_s$ is fetchable under $A$ if and only if $\Sigma_{Q_s,A} \vdash X_{Q_s}^C \rightarrow X_{Q_s}$.

Here $\Sigma \models \varphi$ denotes the standard FD implication: for all databases $D$, if $D$ satisfies $\Sigma$, then $D$ also satisfies $\varphi$ (see [5]).

Intuitively, $X_{Q_s}^C$ is the set of attributes whose values are already provided by $Q_s$, and $X_{Q_s}$ includes all the attributes whose values are needed for answering $Q_s$. The computation of $\text{cov}(Q, A)$ (Section 3) is a chasing process with $A$ to deduce $X_{Q_s}$ from $X_{Q_s}^C$. The process coincides with the implication analysis of $\Sigma_{Q_s,A} \vdash X_{Q_s}^C \rightarrow X_{Q_s}$ (see Appendix B).

Lemma 4 reduces the problem of checking whether an SPC query $Q_s$ is fetchable via $A$ to the implication analysis of FDs. Based on the lemma, algorithm CovChk checks
whether $Q_s$ is fetchable by firstly constructing the set $\Sigma_{Q_s,A}$ of induced FDs from $Q_s$ and $A$, and then checking whether $\Sigma_{Q_s,A} \models X_\xi^{Q_s} \rightarrow X_{Q_s}$ by invoking a linear-time FD implication algorithm [5] (lines 4–5). It checks whether $Q_s$ is indexed under $A$ simply by definition (line 3).

**Example 6:** Given $Q_0$, $Q'_0$ and $A_0$ of Example 1, by examining max SPC sub-queries, algorithm CovChk finds that $Q'_0$ is covered by $A_0$ but $Q_0$ is not (see Appendix C).

**Correctness & Complexity.** The correctness of CovChk follows from the definition of covered queries and Lemma 4. Algorithm CovChk can be implemented in $O(|Q|^2 + |A|)$ time (see Appendix D for more details). This completes the proof of Proposition 3 and Theorem 2(3).

The notations of the paper are summarized in Table 1.

### Table 1: Notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>(actualized) access schema</td>
</tr>
<tr>
<td>$</td>
<td>A</td>
</tr>
<tr>
<td>$</td>
<td>A</td>
</tr>
<tr>
<td>$Q_s$</td>
<td>a max SPC sub-query of $Q$</td>
</tr>
<tr>
<td>$\Sigma_{Q_s,A}$</td>
<td>equality $A = A'$ and $A = c$ derived from $Q_s$</td>
</tr>
<tr>
<td>$A \xi$</td>
<td>attributes in $\tau\xi$ or $\pi\xi$ of some max $Q_s$, of $Q$</td>
</tr>
<tr>
<td>$X_\xi^{Q_s}$</td>
<td>attributes of $S$ such that $\Sigma_{Q_s,A} \rightarrow A = c$ for a $Q_s$, of $Q$</td>
</tr>
<tr>
<td>$\rho_U(A)$</td>
<td>renaming of $A$ with unification function $\rho_U$</td>
</tr>
<tr>
<td>$\rho_U(X)$</td>
<td>$(\rho_U(A) : A \in X)$</td>
</tr>
<tr>
<td>$I_{\Sigma_{Q_s,A}}$</td>
<td>the set of induced FDs from $Q_s$ and $A$</td>
</tr>
<tr>
<td>$\xi_f(A)$</td>
<td>canonical bounded query plan</td>
</tr>
<tr>
<td>$\xi_f(S)$</td>
<td>unit fetching plan for attribute $A$</td>
</tr>
<tr>
<td>$\xi_f(S)$</td>
<td>$(Q,A)$-hyperpath for $Q$ and $A$</td>
</tr>
<tr>
<td>$\Pi_{U \cap V,A}$</td>
<td>hyperpath from set $V$ to node $u,A$</td>
</tr>
</tbody>
</table>

5. GENERATING BOUNDED PLANS

We now verify Theorem 2(2) by proving a stronger result.

**Theorem 5:** (1) For any RA query $Q$ covered by an access schema $A$, $Q$ has a canonical bounded query plan under $A$. (2) There exists an algorithm that given $Q$ covered by $A$, generates a canonical bounded query plan of length $O(|Q| + |A|)$ in $O(|Q|(|Q| + |A|))$ time.

Here canonical bounded query plans are bounded evaluability query plans that characterize covered RA queries. That is, every covered query $Q$ warrants a boundedly executable query plan $\xi$. Better still, $\xi$ can be generated in a bounded amount of time and has a bounded length, both determined by $Q$ and $A$, independent of the underlying databases.

The proof is nontrivial. Below we first introduce canonical bounded query plans (Section 5.1). We then provide an algorithm with the property of Theorem 5(2) (Section 5.2).

5.1 Capturing Covered Queries with Plans

We define canonical query plans and show that an RA query $Q$ is covered by $A$ if and only if $Q$ has a canonical bounded plan under $A$. From this Theorem 5(1) follows.

**Canonical bounded query plans.** For an RA query $Q$ under an access schema $A$, a canonical bounded query plan $\xi^c$ is a boundedly executable query plan for $Q$ that consists of a fetching plan $\xi^f_1$, followed by an indexing plan $\xi^i_1$ and then an evaluation plan $\xi^e_1$, as follows (see Appendix A).

Focusing plan $\xi^f_1$: A fetching plan $\xi^f_1$ is a sequence of unit fetching plans $\xi^f_1(A_1), \ldots, \xi^f_1(A_m)$, for all attributes $A_1, \ldots, A_m$ in $X_Q$ of $Q$. Here unit plan $\xi^f_1(A_i)$ fetches all the necessary values for $A_i$; it may use fetch by employing an access constraint $\phi$ of $A$, projections $(\pi)$ and Cartesian-product $(\times)$, but has no need for selection $(\sigma)$.

**Indexing plan $\xi^i_1$.** An indexing plan $\xi^i_1$ is a sequence of unit indexing plans $\xi^i_1(S_1), \ldots, \xi^i_1(S_m)$ for all relations $S_1, \ldots, S_m$ in $Q$. For each $S_i$, $\xi^i_1(S_i)$ ensures that the combinations of attributes fetched by $\xi^f_1(A_i)$ are from the same tuples in $S_i$. Let $Q_s$ be the max SPC sub-query in which $S_i$ occurs, $X^S_i = \{A_1, \ldots, A_K\}$ be the set of attributes of $S_i$ that are also in $X_Q$, and $S_i(X \rightarrow Y, N)$ be a constraint in $A$ that indexes $S_i$. Then $\xi^i_1(S_i)$ works in three steps: (1) join $\xi^f_1(A_i)$, $\ldots, \xi^f_1(A_k)$ together; (2) apply fetch under $S_i(X \rightarrow Y, N)$; and (3) return the intersection of (1) and (2).

**Evaluation plan $\xi^e_1.$** Plan $\xi^e_1$ is the RA expression of $Q$, in which each relation $S_i$ in $Q$ is replaced by $T_k$, where $T_k = \xi^f_1(S_i)$ is the output of the indexing plan $\xi^i_1(S_i)$ for $S_i$.

Intuitively, given a dataset $D$ with $D \models A$, a canonical bounded query plan $\xi^c$ first executes $\xi^f_1$ to fetch necessary data values from $D$ via indices in $A$. This is followed by $\xi^i_1$ to combine and filter partial tuples for each relation that is needed for answering max SPC sub-queries of $Q$. Finally, $\xi^e_1$ is executed against the fetched partial tuples instead of $D$ directly. That is, $\xi^c$ accesses data only via $\xi^f_1$ and $\xi^i_1$.

By the definitions of bounded evaluability and canonical bounded query plans, one can verify the lemma below (see Appendix B for a proof), from which Theorem 5(1) follows.

**Lemma 6:** For RA query $Q$ and access schema $A$, (1) $Q$ is fetchable via $A$ if $Q$ has a fetching plan under $A$; and (2) $Q$ is indexed by $A$ if $Q$ has an indexing plan under $A$.

5.2 Generating Canonical Bounded Plans

We next give a constructive proof of Theorem 5(2) by developing an algorithm that, given an access schema $A$ and an RA query $Q$ covered by $A$, returns a canonical bounded query plan $\xi^c$ of bounded length in $O(|Q|(|Q| + |A|))$ time. The idea of the algorithm is to encode $Q$ and $A$ in a hypergraph representation such that (i) there is a certain hyperpath in the hypergraph iff $Q$ is fetchable under $A$; and (ii) each such hyperpath encodes a canonical fetching plan for $Q$ under $A$.

Below we first introduce structures used by the algorithm. (see Table 1). We then present the algorithm.

$\xi^f_1$: A directed hypergraph $\Gamma_{Q,A}$ for $Q$ and $A$ is a directed hypergraph $(V,E)$ with a special node $r$, such that (1) $E$ encodes all the induced RHS-FDs; (2) for all induced RHS-FDs of form $\emptyset \rightarrow Y$, $\emptyset$ is encoded by $r$; and (3) for each attribute $A$ in $(\rho_U(A) : A \in X^Q_s)$, $Q$ is a max SPC sub-query of $Q$, there is a hyperedge from $r$ to the node encoding $A$ (see Appendix A).
Example 7: The \((Q,A)\)-hypergraph \(G_{Q,A}^t\) for \(Q_0\) and \(A_0\) of Example 1 is depicted in Fig. 2 (see Appendix C).

Hyperpath. A sub-hypergraph of \(H = (V,E)\) is a hypergraph \(H' = (V',E')\) such that \(V' \subseteq V, E' \subseteq E\), and \(E'\) is restricted to \(V'\). A hyperpath \([9]\) in \(H\) from a set \(S \subseteq V (S \neq \emptyset)\) to a target node \(t \in V\) is a sub-hypergraph \(\Pi_{S,t} = (V_{\Pi_{S,t}}, E_{\Pi_{S,t}})\) of \(H\) satisfying the following: if \(t \in S\), then \(E_{\Pi_{S,t}} = \emptyset\); otherwise its \(k \geq 1\) hyperedges can be ordered in a sequence \((e_1, \ldots, e_k)\) such that (a) for any \(e_i \in E_{\Pi_{S,t}}, \text{head}(e_i) \subseteq S \cup \{\text{tail}(e_1), \ldots, \text{tail}(e_{i-1})\}\); (b) \(t = \text{tail}(e_k)\); and (c) \(e_i\) is a hyperedge of \(\Pi_{S,t}\) other than itself is a hyperpath from \(S\) to \(t\) in \(H\). For example, a hyperpath \(\Pi_{\{r\},A_0[9]}\) from \(r\) to \(u_{A_0}\) in \(G_{Q_0,A_0}^t\) of Example 7 is highlighted in bold in Fig. 2.

We now establish the connection between hyperpaths and canonical fetching plans as follows.

Lemma 7: For any RA query \(Q\), access schema \(A\) and attribute \(A\) in \(X_Q\) of \(Q\), there exists a unit fetching plan \(\xi(A)\) for \(A\) under \(A\) if and only if there exists a hyperpath from \(r\) to \(u_A\) in the hypergraph \(G_{Q,A}\).

Lemma 7 tells us that to get a canonical fetching plan for \(Q\) under \(A\), it suffices to find hyperpaths from \(r\) to \(u_A\) in \(G_{Q,A}\) for all attribute \(A\) in \(X_Q\). Based on this we develop our algorithm for canonical bounded plan generation.

Algorithm. The algorithm, denoted by QPlan and shown in Fig. 3, takes as input an access schema \(A\) and an RA query \(Q\) covered by \(A\), and returns an optimal plan \(P_{Q,A}\) for \(Q\) and \(A\) (line 1). It generates \(P_{Q,A}\) in three steps: it first generates unit plans for attributes in \(X_Q\) (lines 1-6). It then builds an indexing plan \(\xi_0(S)\) for each relation name \(S\) that occurs in \(Q\) on top of the fetching plans (lines 7-9). Finally, it adds the evaluation plan \(\xi_E(Q)\) (line 10).

More specifically, it first constructs the \((Q,A)\)-hypergraph \(G_{Q,A}\) for \(Q\) and \(A\) (line 1), and initializes data structures for storing unit fetching plans \((L_F)\) and the final query plan \((P_{Q,A})\) (line 2). It then iteratively finds plans for attributes in \(X_Q\) (lines 3-6). For each attribute \(A\) in \(X_Q\), it finds a hyperpath \(\Pi(r,u_A)\) from \(r\) to \(u_A\) that encodes \(A\) in \(G_{Q,A}\), by invoking procedure \text{findHP} (line 4; not shown). Here \text{findHP} can be implemented in \(O(|G_{Q,A}|) = O(|Q| + |A|)\) time by traversing \(G_{Q,A}\) (9). It then converts the hyperpath into a unit fetching plan \(\xi_0(A)\) via procedure \text{transQP} (line 5) (see Appendix D for details), and adds the plan to \(P_{Q,A}\) (line 6). After these, algorithm QPlan generates indexing plans for all relations in \(Q\), by manipulating the unit fetching plans stored in \(L_F\), following the definition of indexing plan (lines 7-9). Finally, it adds the evaluation plan of \(Q\) to \(P_{Q,A}\) (line 10), and returns \(P_{Q,A}\) (line 11).

Example 8: Given \(Q_0\) and \(A_0\) of Example 1, QPlan first constructs the hypergraph \(G_{Q_0,A_0}\) shown in Fig 2 for \(Q_0\) and \(A_0\). It then iteratively finds unit fetching plans for attributes in \(X_{Q_0}\). Take \(cid'\) in sub-query \(Q_1\) of \(Q_0\) as an example.
6.1 Intractability & Approximation Hardness

The decision version of AMP, denoted dAMP(\(Q, A, K\)), is to decide, given access schema \(A\), RA query \(Q\) covered by \(A\) and a natural number \(K\), whether there exists \(A_m \subseteq A\) such that \(A_m\) covers \(Q\) and \(\sum_{R(X \rightarrow Y, N) \in A_m} N \leq K\). Its corresponding optimization problem, denoted by oAMP(\(Q, A\)), is to find the minimum \(K\) for dAMP(\(Q, A, K\)) to answer “yes”.

We also study two practical special cases. We say that a (\(Q, A\))-hypergraph \(G_{Q,A}\) is acyclic if \(G_{Q,A}\) is acyclic, where \(G_{Q,A}\) is a directed graph derived from \(G_{Q,A}\) by replacing each hyperedge \(e = (u_1, \ldots, u_l, v)\) with \(p\) edges \((u_1, v), \ldots, (u_p, v)\). Intuitively, \(G_{Q,A}\) is acyclic when the dependency relation on attributes of \(Q\) imposed by \(A\) is not "recursive".

We study the following two special cases:

\(\circ\) acyclic: when \(G_{Q,A}\) is acyclic; and
\(\circ\) elementary: for each \(\phi = R(X \rightarrow Y, N)\) in \(A\), either \(\phi\) is an indexing constraint, i.e., of the form \(R(X \rightarrow X, 1)\), or a unit constraint, i.e., when \(|X| = |Y| = 1\).

Both cases are quite common in practice: access constraints rarely incur recursive dependencies, and are often of the form of indexing or unit constraints. For example, (1) \(Q_0\) and \(A_0\) in Example 1 are an acyclic case since \(G_{Q_0, A_0}\) (Fig. 2) is acyclic; and (2) \(Q_6\) and \(A_0 \setminus \{\psi_2\}\) are an elementary case.

These problems are nontrivial, even their special cases.

**Theorem 9:**
(1) dAMP(\(Q, A, K\)) is NP-complete.
(2) oAMP(\(Q, A\)) is not approximable within \(c \cdot \log |X_Q \setminus X_P|\), for any constant \(c > 0\).
(3) When \((Q, A)\) is acyclic or elementary, dAMP(\(Q, A, K\)) remains NP-hard, and oAMP(\(Q, A\)) is not in APX.

The class APX is the set of NP optimization problems that can be approximated by a constant-factor approximation algorithm, i.e., a PTIME algorithm within some constant.

6.2 Approximation Algorithms

Theorem 9 tells us that for AMP(\(Q, A\)), any efficient algorithm is necessarily heuristic. Below we provide an efficient heuristic that guarantees to find a minimal \(A_m \subseteq A\) that covers \(Q\), i.e., removing any constraint from \(A_m\) makes \(Q\) not covered by \(A_m\). Moreover, for the two special cases, there are approximation algorithms with accuracy bounds.

**Theorem 10:**
(1) There is an algorithm for AMP(\(Q, A\)) that finds minimal \(A_m\) in \(O(|Q|^2 + |A|(|Q| + |A|))\) time.
(2) For acyclic (\(Q, A\)), oAMP(\(Q, A\)) is approximable within \(O(1 + |X_Q \setminus X_P|)\) in \(O(|Q| + |A|)\) time.
(3) For elementary Q(\(Q, A\)), oAMP(\(Q, A\)) is approximable within \(O(1 + |X_Q \setminus X_P|^2)\) in \(O(|Q| + |A|)^2\) time, for any constant \(\epsilon > 0\).

As a proof, we outline the algorithms as follows.

**General case.** As a proof of Theorem 10(1), we give an algorithm for AMP(\(Q, A\)) for the general case, denoted by minA (not shown). It is based on the following heuristics: a constraint \(\phi = R(X \rightarrow Y, N)\), if it is not used to index a relation (Section 3), then it is less likely in the optimum solution \(A_m\), if \(Q\) remains covered by \(A \setminus \{\phi\}\), and moreover,

- (a) \(\text{cov}(Q, A \setminus \{\phi\})\) is small; and (b) \(N_\phi\) is large.

Based on the observation, algorithm minA works as follows. It first constructs the set \(\Sigma_{Q,A}\) of induced FDs of \(Q\) and \(A\). It then iteratively removes “redundant” FDs from \(\Sigma_{Q,A}\). In each iteration, it greedily selects an induced FD that corresponds to access constraint \(\phi\), such that (a) \(Q\) remains covered by \(A \setminus \{\phi\}\); and (b) \(w(\phi) = e_2 \cdot (\min(\psi_1, \psi_2, (\psi_1 + \psi_2)) / c_2)\) is maximum among all constraints \(\phi\), where \(c_1\) and \(c_2\) are user-tunable coefficients for normalizing the numbers. It returns all access constraints corresponding to the remaining FDs in \(\Sigma_{Q,A}\) when it cannot remove more FDs from \(\Sigma_{Q,A}\).

**Example 9:** Consider \(Q_1\) and \(A_0\) given in Example 1. Let \(A_1\) consist of constraints in \(A_0\) and an additional \(\psi_5\): dine(pid, year) \(\rightarrow\) cid, 366, i.e., each person dines out at most 366 times per year. For AMP(\(Q_1, A_1\)), algorithm minA returns \(A_m = \{\psi_1, \psi_2, \psi_5\}\) (see details in Appendix C).

**Algorithm.** Algorithm minA always returns minimal \(A_m \subseteq A\) for AMP(\(Q, A\)) since it keeps removing FDs until \(A_m\) is minimal, in \(O(|Q|^2 + |A|^2 (|Q| + |A|))\) time (see Appendix D).

**Acyclic case.** We prove Theorem 10(2) by giving an approximation algorithm, denoted by minA\_DAG (omitted). The algorithm capitalizes on the connection between hyperpaths and coverage (Lemma 7). It uses the following notion.

**Weighted \((Q, A)\)-hypergraph.** For an RA query \(Q\) and an access schema \(A\), the weighted \((Q, A)\)-hypergraph is a weighted hypergraph, where each hyperedge carries a weight defined as follows. Recall \(G_{Q,A}\) from Section 5.2. For each induced FD \(X \rightarrow Y\) in \(\Sigma_{Q,A}\) derived from an constraint \(R(X \rightarrow Y, N)\) in \(A\), the hyperedge encoding the induced RHS-FD \(X \rightarrow Y\) has weight \(N\), and hyperedges encoding the remaining induced \(Y \rightarrow Y_i\) (for \(Y_i \in Y\)) have weight 0. Moreover, all hyperedges emanating from the special node \(r\) have weight 0. For instance, for \(Q_1\) and \(A_1\) of Example 9, its weighted \((Q, A)\)-hypergraph \(G_{Q_1, A_1}\) is depicted in Fig. 7 in Appendix C (see Appendix A for a formal definition).

**Algorithm minA\_DAG.** Based on this notion, minA\_DAG simply computes the shortest hyperpaths from node \(r\) in \(G_{Q,A}\) to nodes encoding attributes in \(\langle X, X \rangle\) (recall \(X = \rho_X(Y)\) and Table 1) w.r.t. the sum of weights of hyperedges on it. It returns access constraints corresponding to the induced FDs encoded by edges of the hyperpaths, plus one constraint with the minimum \(N\) to index each relation \(S\) in \(Q\) (Section 3).

**Example 10:** For \(Q_1\) and \(A_1\) of Example 9, its weighted \((Q, A)\)-hypergraph \(G_{Q_1, A_1}\) is acyclic (see Fig. 7 in Appendix B). Given \(Q_1\) and \(A_1\), minA\_DAG computes shortest hyperpaths from \(r\) to \(u_{\text{pid}}, u_{\text{cit}}, u_{\text{cid}}, u_{\text{year}}\) and \(u_{\text{city}}\). For example, the shortest hyperpath from \(r\) to \(u_{\text{cid}}\) is the one containing edge \((\text{fid}, \text{year}, \text{month}, \text{cid})\) with weight 31. Algorithm minA\_DAG returns constraints \(\psi_1, \psi_2, \psi_4\) in \(A_1\) that correspond to the edges in those hyperpaths. As \(Q_1\) is already indexed by them, no more constraints are needed.

**Analysis.** By Lemma 7, minA\_DAG always returns \(A' \subseteq A\) that covers \(Q\). The approximation bound can then be proved based on the following (see details in Appendix B): the weight of the shortest hyperpaths from \(r\) to a node \(u_d\) denoting attribute \(A\) is the minimum “cost” to cover \(A\) with \(A\).

Observe that the search for shortest hyperpaths emanating from \(r\) can be conducted by BFS in \(G_{Q,A}\) in \(O(|G_{Q,A}|)\) time. Hence, algorithm minA\_DAG is in \(O(|Q| + |A|)\) time.

**Elementary case.** As a proof of Theorem 10(3), we develop an algorithm, denoted by minA\_E (omitted), for the elementary case \((Q, A)\) of AMP(\(Q, A\)). The idea is by reduction to the directed minimum steiner arborescence problem (dminSAP(\(G, u, V_T\)) (cf. [15]), which is to find the minimum
Figure 4: Bounded evaluability on DBMS

7. SUPPORTING BOUNDED QUERIES

We next present a framework for incorporating bounded evaluation of RA queries into existing DBMS, based on covered queries. To simplify the discussion, we use $I_A$ to denote the indices for all constraints in an access schema $A$.

A framework of bounded evaluation. The framework is shown in Fig. 4. Given an application for queries over instances of a relational schema $R$, it works as follows. As offline preprocessing (C1 in Fig. 4), it discovers an access schema $A$ from sample instances of $R$, builds indices $I_A$ for $A$ on the instance $D$ of $R$ in use, and maintains $I_A$ in response to updates to $D$. Given a user RA query posed on $D$, it first checks whether $Q$ is covered by $A$ (C2). If so, it picks a minimum set $A_m$ of $A$ that covers $Q$ (C3), generates a bounded query plan $\xi$ for $Q$ under $A_m$ (C4), and translates it into an SQL query $Q_\xi$ (C5). Query $Q_\xi$ can then be evaluated directly by the underlying DBMS on a bounded dataset $D_Q$ identified by the bounded plan $\xi$ (C6). If $Q$ is not covered, it is executed against $D$ by the DBMS. As will be seen shortly, a large fraction of RA queries are covered and hence, can be evaluated by accessing a small $D_Q$.

We next present its components in more details.

(1) Building and maintaining $(A, I_A)$. It has three parts.

(a) Discovering $A$. Like FDs, access constraints are defined on schema $R$. They can be mined by extending dependency discovery tools [26], e.g., TANE [23] for FDs. More specifically, on samples of a relation schema $R$, we search candidate attributes $X$ and $Y$ via revised FD mining, and use group_by on $X$ and aggregates count on $Y$ to form access constraint $R(X \rightarrow Y, N)$. These include those composed of attributes with a finite domain, e.g., $R(X \rightarrow \text{month}, 12)$, stating that a year has 12 months. These constraints hold on all instances of $R$, just like discovered FDs.

Discovered constraints also include those determined by policies and statistics, e.g., $\varphi_1$ of Example 1 imposing a limit of 5000 friends per person, and one stating that US airports host carriers of at most 28 airlines (see Section 8). Such constraints may change if Facebook changes their policy or some US airports expand, and are thus maintained (see below).

(b) Building indices $I_A$. For each discovered constraint $\varphi = R(X \rightarrow Y, N) \in A$, the index for $\varphi$ is constructed by creating a partial table $T_{XY} = \pi_{XY}(D)$ and building a hash index on $X$ over $T_{XY}$, where $D_X$ is the instance of $R$ in $D$. The index is no larger than $|D_R|$ and is constructed in $O(|D_R|)$ time. Thus, it takes $O(|A| |D|)$ time to build all indices in $A$, and the total size $I_A$ is at most $O(|A| |D|)$.

(c) Incremental maintenance of $(A, I_A)$. Now consider updates $\Delta D$ to $D$, i.e., sequences of tuple insertions and deletions (which can simulate value modifications). We show that in response to $\Delta D$, both constraints in $A$ and indices $I_A$ can be maintained by bounded incremental algorithms: their costs are determined by $A$ and the size $|\Delta D|$ of updates $\Delta D$ only, and are independent of $D$ and $I_A$. In practice, $\Delta D$ is typically small, and hence so are the costs.

Proposition 12: In response to updates $\Delta D$ to $D$, both $A$ and $I_A$ can be updated in $O(N_A |\Delta D|)$ time, where $N_A = \Sigma_{(X \rightarrow Y,N) \in A} N$.

(2) Checking whether $Q$ is covered by $A$. This can be carried out by algorithm CovChk of Section 4.

(3) Minimizing accessed data. This is conducted by the algorithms in Section 6 to minimize index access in $I_A$.

(4) Generating boundedly evaluable query plans $\xi_{(Q,A)}$. This is done by using algorithm QPlan of Section 5.

(5) Interpreting $\xi_{(Q,A)}$ as SQL query $Q_\xi$. We develop an algorithm, denoted by Plan2SQL (omitted), to translate a bounded plan $\xi$ into an SQL query $Q_\xi$, such that $Q_\xi$ can be directly executed by DBMS. Given $\xi$ and $A$, Plan2SQL returns $Q_\xi$ such that for any dataset $D |\subseteq A$, $Q_\xi$ returns $Q(D)$ by accessing the same amount of data in in index $I_A$ as $x$ does in $D$. For instance, recall $Q_1$ and $A_0$ of Example 1, $A'_0 = A_0 \setminus \varphi_2$, and the bounded query plan $\xi$ for $Q_1$, under $A'_0$ given in Example 2. Let the index relations in $I_A$ under $\varphi_1$, $\varphi_2$, and $\varphi_3$ in $A'_0$ be ind_friend, ind_dine, and ind_cafe, respectively. Plan2SQL($\xi, A'_0$) returns the following SQL query:

```sql
select distinct cid
from ind_cafe
```
Thus, bounded evaluation can be built on top of DBMS.

Added functionality. While indices and constraints are already employed by DBMS, their current mechanism stops short of taking advantage of bounded evaluation.

Indices and query plans. Query plans generated by conventional query engines fetch entire tuples first and then filter tuples based on the query (see, e.g., [5]), by employing tuple-based indices, e.g., hash index and tree-based index [31]. In contrast, a boundedly evaluable query plan makes use of attribute-based indices. It identifies what attributes are necessarily needed, fetches values of the attributes, infers their connection with other attributes, composes attribute values into tuples and validates the tuples (via the indexing conditions of Section 3). However, existing DBMS stops short of exploring this, no matter what indices are provided.

This observation is verified by examining system logs of commercial DBMS, which shows excessive duplicated and unnecessary attributes in tuples fetched by DBMS, and the redundancies get inflated rapidly when joins are involved.

We also check whether a query Q is boundedly evaluable before Q is executed, as opposed to conventional DBMS.

(2) Constraints. Query optimization has been studied for reformulating a query Q as another query by “chasing” Q with constraints [5, 24, 30]. However, to the best of our knowledge, conventional query engines have made little use of it, partly because the chasing process is costly and may not even terminate. Moreover, cardinality constraints have not been explored for this purpose. In contrast, we use cardinality constraints to generate boundedly evaluable query plans, instead of query reformulation. These constraints are easy to reason about and can be readily supported by DBMS.

(3) Join ordering. Query engines may reorder joins in a query plan to minimize estimated data access [22, 27]. It is an effective optimization strategy complementary to this work. However, to comply with bounded data access via access constraints, some joins in a boundedly evaluable query plan cannot be reordered. It is an interesting topic to study what joins can be reordered in boundedly evaluable plans.

8. EXPERIMENTAL STUDY

Using real-life data, we conducted two sets of experiments to evaluate (1) the effectiveness of the RA-query evaluation approach based on the bounded evalability analysis, and (2) the efficiency of algorithms ChkCov, QPlan and minA.

Experimental setting. We used three datasets: two real-life (AIRCA and TFACC) and one benchmark (MCBM).


(2) UK traffic accident (TFACC) integrates the Road Safety Data [2] of road accidents that happened in the UK from 1979 to 2005, and National Public Transport Access Nodes (NaPTAN) [1]. It has 19 tables with 113 attributes, and over 89.7 million tuples in total, about 21.4GB of data.

(3) Mobile communication benchmark (MCBM) was generated by using a commercial benchmark from Huawei Technologies Co. Ltd. The dataset consists of 12 relations with 285 attributes, simulating mobile communication scenarios. We varied the number of tuples from $2^2$ to $360$ million, and used 360 million by default, about 90GB of data.

All of the three datasets were stored in MySQL.

Access schema. We extracted 266, 84 and 366 access constraints for AIRCA, TFACC and MCBM, respectively, by using the discovery method in Section 7. For example, a constraint on AIRCA is OnTimePerformance(Origin → AirlineID, 28), i.e., each airport hosted carriers of at most 28 airlines. On TFACC, we had Accident((data, police_force → accident_ID, 304), i.e., each police force in the UK had handled no more than 304 accidents within a single day from 1979 to 2005. In fact there are many more access constraints in the datasets, which were not used in our tests. We built indices for the constraints by using DBMS (see Section 7).

RA queries generator. We generated queries by using attributes that occurred in the access constraints and constants randomly extracted for those attributes. For MCBM, the query generation also complied with the provided query templates. We generated 300 RA queries Q on these datasets, 100 for each. The queries vary in the number of equality atoms in the selection conditions in the range of [4, 9], #-join of joins in the range of [0,5] and #-unidiff of set difference and union operators in the range of [0, 5].

Algorithms. We implemented the following algorithms in Python: (1) ChkCov (Section 4) to check whether an RA query is covered; (2) QPlan (Section 5) to generate canonical query plans for covered queries; (3) minA, minA, minA and minA (Section 6) to find minimum access constraints for covered queries; (4) Plan2SQL to interpret canonical query plans generated by QPlan as SQL queries (Section 7); (5) evalQP and evalQP to evaluate the translated queries QE with and without minimized $\Delta_0$ (via minA; by Plan2SQL) using DBMS, respectively; and (6) evalDBMS that directly uses DBMS engine for query evaluation, with a configuration in favor of DBMS, which is described as follows.

Configuration. For DBMS, we used MySQL 5.5.44 (MyISAM engine) and PostgreSQL 9.3.9, both with optimization enabled. Both original queries and query plans generated by our algorithms are executed on the same database server. In favor of MySQL and PostgreSQL, besides indices for access constraints, we also built extra indices when testing original queries on the DBMS, e.g., primary and foreign key indices and B-tree on numerical attributes, while these were all disabled when testing our query plans. The experiments were conducted on an Amazon EC2 d2.xlarge instance with 14 EC2 compute units and 30.5GB memory. All the experiments were run 3 times. The average is reported here.

Experimental Results. We next report findings. As results for PostgreSQL are even worse than MySQL when compared with ours, we mainly report MySQL to save space.

Exp1: Effectiveness of bounded evalability.

(1) Percentage of bounded evaluable and covered RA queries. Varying the number of access constraints, we tested the percentage of covered queries (via ChkCov) and boundedly evaluable queries (by manual examination). The results
(ii) ratio $P(D_Q)$ = $|D_Q|/|D|$, measuring the total amount of data $D_Q$ accessed by our query plans (the right y-axis), which is assessed by using MySQL’s `EXPLAIN` statement. Unless stated otherwise, we used the full-size datasets, all access constraints, and 5 covered queries randomly chosen.

1. Impact of $|D|$. To evaluate the impact of $|D|$, we varied the datasets by using scale factors from $2^{-5}$ to 1. As shown in Figures 5(a), 5(e) and 5(i), the results tell us the following.

(a) The evaluation time of evalQP is indifferent to the size of $D$, as expected for covered queries.

(b) Bounded query plans work well with large $D$. Indeed, evalQP took less than 5.9s, 8.3s, 6.5s with MySQL, and 5.5s, 9.0s, 7.0s with PostgreSQL on AIRCA, TFACC and MCBM, respectively, no matter how large the datasets were. In contrast, even on the smallest subsets with scale factor $2^{-5}$, evalDBMS took 2398s, 2759s, 5675s by MySQL and 3598s, 3851s, 7301s by PostgreSQL; it could not terminate within 2 hours for all larger subsets. This is why few points are reported for evalDBMS in the figures. In fact, evalDBMS could not finish within 14 hours on all three full-size datasets (both MySQL and PostgreSQL). That is, evalDBMS is at least $8.5 \times 10^3$, $6.1 \times 10^3$ and $7.8 \times 10^5$ times slower on AIRCA, TFACC and MCBM, respectively. The larger the dataset is, the bigger the gap between evalDBMS and evalQP is.

(c) Query plans generated by QPlan accessed a very small fraction of the data: $P(D_Q)$ is $1.7 \times 10^{-6}$, $3.7 \times 10^{-5}$, $2.2 \times 10^{-6}$ on full-size AIRCA, TFACC and MCBM, i.e., 0.00177%, 0.0037% and 0.0022% of these datasets, respectively.

Remark. As shown above, evalQP outperforms evalDBMS by at least 3 orders of magnitude, for reasons explained in Section 7. We also find that when queries $Q$ use key attributes only, evalDBMS is as fast as evalQP if extra key/foreign key

Figure 5: Effectiveness of bounded evaluability

Figure 6: Percentage of covered (bounded) queries

are shown in Figure 6, and tell us the following. (a) When all the discovered constraints are used, (i) at least 70, 65 and 48 out of 100 queries are boundedly evaluable, and (ii) 61, 52 and 42 are covered, on AIRCA, TFACC and MCBM, respectively. That is, at least 70%, 65% and 51% of the queries are boundedly evaluable, and among them 87%, 80% and 87.5% are covered. Hence, covered queries are indeed effective for determining the bounded evaluability of RA queries. (b) The more access constraints are used, the more queries are covered and boundedly evaluable, as expected. Nonetheless, among all the covered queries, 78.7%, 84.6% and 80.9% are already covered by only 25% of the discovered access constraints on AIRCA, TFACC and MCBM, respectively. That is, a large number of queries can be covered by a small number of constraints.

(II) Effectiveness of covered queries. We next evaluated the effectiveness of query plans generated by QPlan, by comparing the run time of evalQP and evalDBMS, both executed by MySQL. The results are reported in Figure 5, on datasets AIRCA, TFACC and MCBM, by varying $|D|$, $Q$ and $|A|$. We report (i) the average evaluation time (the left y-axis), and (ii) ratio $P(D_Q) = |D_Q|/|D|$, measuring the total amount
indices are built for MySQL and PostgreSQL, e.g., less than 3s with one join on full AIRCA. However, as long as \( Q \) involves non-key attributes, \texttt{evalDBMS} performs poorly on big tables, even provided with all indices. It gets worse when the number of non-key attributes increases. By looking into MySQL’s log and its \texttt{EXPLAIN} output, we verified that this is partially due to the following. Given an access constraint \( R(X \rightarrow Y; N) \), \texttt{evalQP} fetches only distinct values of the relevant \( XY \) attributes, but \texttt{evalDBMS} fetches entire tuples with irrelevant attributes of \( R \), although those attributes are not needed for answering \( Q \) at all, no matter what indices are provided. This led to duplicated \((X, Y)\) values when \( X \) is not a key, and the duplication got rapidly inflated by joins, e.g., \texttt{EXPLAIN} output shows that MySQL consistently accesses entire tables when there are no-key attributes.

(2) Impact of \( Q \). To evaluate the impact of queries, we varied \#-sel of \( Q \) from 4 to 9, \#-join from 0 to 5 and \#-unidiff of set operators (union and set-difference) from 0 to 5, while keeping the other factors unchanged.

The results are reported in Figures 5(b), 5(f), 5(j) for varying \#-sel and Figures 5(c), 5(g), 5(k) for varying \#-join. We find the following. (a) The complexity of \( Q \) has implications on the query plans generated by \texttt{QPlan}. The larger \#-sel or the smaller \#-join is, the faster the query plans are, and the smaller data \( D_Q \) is accessed. This is because with more selections or fewer joins, our plans generated by \texttt{QPlan} took less steps to fetch all attribute values needed. (b) Algorithm \texttt{evalQP} scales well with \#-sel and \#-join. It found answers for largest \( Q \) within 89.5s, on the three full-size datasets. (c) Algorithm \texttt{evalDBMS} is almost indifferent to \#-sel; in fact it only terminated within 3000s on extremely restricted (constant) selection queries, with at most one join on non-key attribute. But it is very sensitive to \#-join: it did the best when \#-join = 0, i.e., if there is no join (or Cartesian product) at all; but it cannot finish the job within 3000s for queries with 2 joins on all three datasets.

Our query plans are indifferent to \#-unidiff (hence the results are not shown). This is because our query plans fetch data via max SPC sub-queries, independent of the number of union and set-difference operations in the queries. We do not report the results of \texttt{evalDBMS} since it did not complete its computation within 3000s on all three datasets.

(3) Impact of \(#[A]\). To evaluate the impact of access constraints, we varied \(#[A]\) with scale factors from 0.2 to 1 in 0.2 increments, and tested the queries that are covered. Accordingly we varied the indices used by \texttt{evalDBMS}. We report \( P(D_Q) \) and run time of \texttt{evalQP}. As shown in Figures 5(d), 5(h) and 5(l), (a) more constraints help \texttt{QPlan} generate better query plans, as expected. For example, when all access constraints were used, \texttt{evalQP} took 5.8s, 8.5s and 6.3s for queries on AIRCA, TFACC and MCBM, respectively, while they took 10.2s, 20.1s and 9.6s given 20% of the constraints. (b) The more access constraints are used, the smaller \( D_Q \) is, as \texttt{QPlan} can find better plans given more options. (c) Algorithm \texttt{evalDBMS} did not produce results in any test within 3000s, even given the indices in full-size \( A \) of constraints.

(III) Effectiveness of minA. We also evaluated the effectiveness of minA for minimizing access schemas by comparing \texttt{evalQP} and \texttt{evalQP}-. As reported in Figures 5(a), 5(e) and 5(i), (1) minA helps \texttt{QPlan} generate query plans that access less data; indeed, \texttt{evalQP} accessed much smaller \( D_Q \) than \texttt{evalQP} - in most cases; for example, \( P(D_Q) \) is 0.0037% for \texttt{evalQP} on full-size TFACC, while it is 0.0051% for \texttt{evalQP} -; and (2) minA also enables query plans to use indices of smaller size (i.e., index relations; not shown). For example, on full-size TFACC, \texttt{evalQP} used index no larger than 2.1% of the size of \( D \) while it was 3.3% for \texttt{evalQP} -.

(IV) Size and creation time of indices. The total indices for all access constraints are of \( 7.7 \) GB, \( 3.6 \) GB and \( 9.5 \) GB, accounting for 12.8%, 16.8% and 10.6% of \( |D| \). They are smaller than the bound estimated in Section 7, since many constraints cause attributes to be left out. They took 3.1, 2.2 and 4.2 hours to build offline for AIRCA, TFACC and MCBM, respectively, and were used to answer all queries.

Exp-2: Efficiency. The second set of experiments evaluated the efficiency of our algorithms \texttt{ChkCov}, \texttt{QPlan}, \texttt{minA}, \texttt{minApdag} and \texttt{minAg} on queries and access schemas for each of AIRCA, TFACC and MCBM. We found that \texttt{ChkCov}, \texttt{QPlan}, \texttt{minA}, \texttt{minApdag} and \texttt{minAg} took at most 65ms, 199ms, 86ms, 84 ms and 74 ms, respectively, for all queries on three datasets, with all the access constraints.

Summary. From the experiments we find the following. (1) Covered queries give us a practical effective syntax for boundedly evaluable RA queries. Over 80% of boundedly evaluable queries are covered. (2) Bounded evaluability is promising for querying large datasets. Indeed, (a) it is easy to find access constraints from real-life data, and many queries are covered under a small number of such constraints; and (b) for covered queries, the evaluation time and the amount of data accessed are independent of the size of the underlying dataset. As a result, on a real-life dataset of 60GB, \texttt{evalQP} answers queries in 5.9 seconds by accessing at most 0.00017% of the data on average, while \texttt{evalDBMS} is unable to find answers within 3000 seconds even on a dataset of 3.75 GB, with even more indices than \texttt{evalQP} can use. The performance gap between \texttt{evalQP} and \texttt{evalDBMS} gets bigger on larger datasets. (3) The size of the indices needed is 13.4% of \( |D| \) on average. (4) Our algorithms are efficient: they take at most 0.2 second in all cases tested.

9. CONCLUSION

We have answered an open question about the bounded evaluability of RA. Our solution consists of both fundamental results and practical algorithms. Our experimental results have shown that it is promising to make practical use of bounded evaluability. Indeed, a large number of RA queries are covered, and covered queries can be efficiently evaluated without worrying about the size of the underlying datasets.

One topic for future work is to develop algorithms for discovering a (minimum) set of access constraints to cover a topic needs a full treatment. Another topic is to develop algorithms that, when a query is not boundedly evaluable, is easy to find access constraints from real-life data, and many queries are covered under a small number of such constraints; and (b) for covered queries, the evaluation time and the amount of data accessed are independent of the size of the underlying dataset. As a result, on a real-life dataset of 60GB, \texttt{evalQP} answers queries in 5.9 seconds by accessing at most 0.00017% of the data on average, while \texttt{evalDBMS} is unable to find answers within 3000 seconds even on a dataset of 3.75 GB, with even more indices than \texttt{evalQP} can use. The performance gap between \texttt{evalQP} and \texttt{evalDBMS} gets bigger on larger datasets. (3) The size of the indices needed is 13.4% of \( |D| \) on average. (4) Our algorithms are efficient: they take at most 0.2 second in all cases tested.

APPENDIX A: Formal Definitions

Query plans (Section 2)

We define a query plan $\xi$ for $Q$ under $A$ as a sequence:

$$\xi(Q(R)) : T_1 = \delta_1, \ldots, T_n = \delta_n,$$

such that (1) for all instances $D$ of $R$, $T_n = Q(D)$; and (2) for all $i \in [1, n]$, $\delta_i$ is one of the following:

- $\{a\}$, where $a$ is a constant in $Q$;
- $\operatorname{fetch}(X \in T_j(R)), Y_j$, where $j < i$, and $T_j$ has attributes $X$; for each $\bar{a} \in T_j(D)$, it retrieves $D_{X,Y}(X = \bar{a})$ from $D$, and returns $\bigcup_{a \in T_j(D)} D_{X,Y}(X = \bar{a})$ or
- $\Pi_{Y_j}(T_j)$ or $\sigma_C(T_j)$, for $j < i$, a set $Y$ of attributes in $T_j$, and Boolean condition $C$ defined on $T_j$; or
- $T_i \times T_k$, $T_i \cup T_k$ or $T_i \setminus T_k$, for $j < i$ and $k < i$.

We denote $T_n$ by $\xi(D)$, as the result of applying $\xi$ to $D$.

Canonical bounded query plans (Section 5.1)

Faching plan $\xi^*_P$: A fetching plan $\xi^*_P$ is a sequence of unit fetching plans $\xi^*_P(A_i), \ldots, \xi^*_P(A_m)$, for all attributes $A_1, \ldots, A_m$, $X_Q$ of $Q$, where $\xi^*_P(A_i)$ is inductively defined as follows (assuming $A_i$ is in a max SPC sub-query $Q_s$ of $Q$):

(i) if $A_i \in X_Q$, then $\xi^*_P(A_i) = \{c\}$, where $\sigma_{A_i=c}$ is in $Q_s$;
(ii) if $A_i \rightarrow A'$ is in $Q_s$ and there exists a unit fetching plan $\xi^*_P(A')$ for $A'$, then $\xi^*_P(A_i) = \xi^*_P(A')$;
(iii) if there exists a constraint $R(W \rightarrow U, N)$ in $A$ such that $A_i \in R[U]$, and moreover, if for each $w_i \in R[W] = \{w_1, \ldots, w_m\}$, there exists a unit fetching plan $\xi^*_P(w_i)$ for $w_i$, then $\xi^*_P(A_i)$ is:

- $T_1 = \xi^*_P(w_1), \ldots, T_m = \xi^*_P(w_m)$,
- $T_{m+1} = T_1 \times \cdots \times T_m$. 

An indexing plan $\xi$ is a sequence of unit indexing plans $\xi^j(S_1), \ldots, \xi^j(S_n)$ for all relations $S_1, \ldots, S_n$ in $Q$. For each $S_i$, let $Q_i$ be the max SPC sub-query in which $S_i$ occurs, $X_{S_i} = \{A_1, \ldots, A_k\}$ be the set of attributes of $S_i$, also in $Q_i$, and $(X_i, Y_i)$ be a constraint in $A$ that indexes $S_i$. Then $\xi^j(S_i)$ is as follows:

- $T_1 = \xi^j(A_1), \ldots, T_k = \xi^j(A_k)$.
- $T_{k+1} = T_1 \times \cdots \times T_k$.
- $T_{k+2} = \sigma_{S_i[X]}(T_{k+1})$.
- $T_{k+3} = \text{fetch}(X \in T_{k+2}, S_i, Y_i)$, and
- $T_{k+4} = T_{k+3} \cap T_{k+3}$ (expressed in terms of $x, \sigma, \pi$).

That is, $\xi^j$ ensures that each $S_i$ in $Q$ is indexed.

**Appendix B: Proofs**

**Proof sketch of Theorem 2.1**

**Proof of Lemma 4**

Since $\Sigma_{Q_r, A} \models X_{Q_r} \rightarrow \hat{X}_{Q_r}$, iff $X_{Q_r} \subseteq (X_{Q_r})^*$ (cf. [5]), to show that $Q_r$ is fetchable via $A$ iff $\Sigma_{Q_r, A} \models X_{Q_r} \rightarrow \hat{X}_{Q_r}$, we just need to show that $X_{Q_r} \subseteq \text{cov}(Q_r, A)$ iff $X_{Q_r} \subseteq (X_{Q_r})^*$, where $(X_{Q_r})^*$ is the FD closure of $X_{Q_r}$ under $\Sigma_{Q_r, A}$. We prove this by showing the following:

1. $(X_{Q_r} \subseteq \text{cov}(Q_r, A))$ iff $\rho_U(X_{Q_r}) \subseteq \rho_U(\text{cov}(Q_r, A))$; and
2. $(\rho_U(\text{cov}(Q_r, A)))^* = (\rho_U(X_{Q_r}))^*$ (recall $\rho_U(X) = \hat{X}$).

Here (1) can be proved by contradiction based on the definitions. We prove (2) below. We first define a chasing procedure that computes $\text{cov}(Q_r, A)$ for any SPC query $Q_r$ under $A$. Based on it we then show $\rho_U(\text{cov}(Q_r, A))$ is equivalent to $(\rho_U(X_{Q_r}))^*$.

A chasing sequence of $\text{cov}(Q_r, A)$ for $Q_r$ is defined as $\text{cov}(Q_r, A) = \text{cov}_0(Q_r, A), \ldots, \text{cov}_{n}(Q_r, A)$, such that (1) $\text{cov}_0(Q_r, A) = X_{Q_r}^*$, and (2) for each $i \geq 0$, $\text{cov}_{i+1}(Q_r, A)$ is obtained by applying some rules given in the definition of coverage $\text{cov}(Q, A)$ so that $\text{cov}_0(Q_r, A) \neq \text{cov}_{i+1}(Q_r, A)$. Obviously such a chasing sequence is terminal; moreover, by the definition of $\text{cov}(Q, A)$, the result $\text{cov}_n(Q_r, A)$ of the chasing sequence (the last element) is exactly $\text{cov}(Q_r, A)$ for $Q_r$ and $A$. One can show $\rho_U(\text{cov}(Q_r, A)) = (\rho_U(X_{Q_r}^*))^*$ (thus $\rho_U(X_{Q_r}^*)^*$) by induction on the length $n$ of the chase.

**Proof of Lemma 6**

By the definition of indexed queries, $Q$ is indexed by $A$ iff $Q$ has an indexing plan. Below we show that $Q$ is fetchable via $A$ iff $Q$ has a fetch plan under $A$.

Assume that $Q$ is fetchable via $A$. Then each max SPC sub-query $Q_s$ is fetchable via $A$. Thus for each attribute $A \in X_{Q_s} \subseteq \text{cov}(Q_r, A)$, consider the chasing sequence $\text{cov}(Q_r, A) = \text{cov}_0(Q_r, A), \ldots, \text{cov}_{n}(Q_r, A)$ described in the proof of Lemma 4, where $\text{cov}_{i+1}(Q_r, A) = X_{Q_r}^*$ and $\text{cov}_n(Q_r, A) = \text{cov}(Q_r, A)$. There must exist $i \in [0, n]$ such that $A \in \text{cov}_{i+1}(Q_r, A)$ but $A \notin \text{cov}_{i+1}(Q_r, A)$ (if exists). We refer to $\text{cov}_{i+1}(Q_r, A)$, $\text{cov}_{i+1}(Q_r, A)$ as the deduced chasing for attribute $A$. One can verify that there exists a unit fetching plan for $A$ under $A$ by induction on the length of the deduced chasing for $A$.

This direction can be readily verified by induction on the length of $\xi^j(A)$, by the definitions of unit fetching plans and fetchable queries, similar to above.

**Proof of Lemma 7**

This is verified by giving translation algorithms $\Gamma_\xi$ from $\xi^j(A)$ to $\{\hat{x}, u_{\rho(A)}\}$, and $\Gamma_r$ from $\{\hat{x}, u_{\rho(A)}\}$ to $\xi^j(A)$. Below we outline $\Gamma_r$, which will be used later in our algo-
Algorithm. Given a hyperpath $\Pi_{(r), u_A}$ from $\{r\}$ to $u_A$, $\Gamma_r$ inductively generates fetching plans as follows: (a) if $\Pi_{(r), u_A}$ is a hyperedge $\langle \{r\}, u_A \rangle$ constructed in case (3) of $(Q, A)$-hypergraph above, then return $T_1 = \{c\}$; (b) if $\Pi_{(r), u_A}$ is a hyperedge $\langle \{r\}, w_Y \rangle$ constructed in case (2) for induced FD $0 \rightarrow Y$, then return $T_1 = \xi_r^P(A')$; and (c) if the last hyperedge of $\Pi_{(r), u_A}$ is a hyperedge $\langle \{r\}, Y \rangle \rightarrow u_A$ constructed in case (1) of $(Q, A)$-hypergraph, and if for each $u_B$, in $V_S = \{u_{Y_1}, \ldots, w_Y\}$, the unit fetching plan translated from hyperpath $\Pi_{(r), u_Y}$ is $\xi_r^S(Y_1), \ldots, T_p = \xi_r^S(Y_p)$, then return $T_1 = \xi_r^S(Y_1), \ldots, T_p = \xi_r^S(Y_p), T_{p+1} = T_1 \times \cdots \times T_p, T_{p+2} = \text{fetch}(X \in T_{p+1}, R, Y)$, and $T_{p+3} = \pi_A(T_{p+3})$. □

Proof of Theorem 9

(1) For $\text{dAMP}(Q, A, K)$, we give an NP algorithm that guesses $A_m$ and checks whether $Q$ is covered by $A_m$ in PTIME. The lower bound follows from (3) below.

(2) We show the approximation-hardness of $\text{oAMP}(Q, A)$ by L-reduction from the Minimum Set Cover problem [29].

(3) We show that $\text{dAMP}(Q, A, K)$ is NP-hard for both special cases by reduction from the Vertex Cover problem, which is NP-complete (cf. [29]). We show the approximation-hardness of the special cases by AP-reduction from the Minimum Set Cover problem, which is not in APX [8]. □

Proof of Lemma 11

We prove Lemma 11 by showing step (c) of algorithm $\text{minAE}$ maps feasible solutions to $\text{diminSAP}$ with approximation ratio $c$ to feasible solutions to $\text{oAMP}$ with approximation ratio $c + 1$ in the elementary case. This can be verified along the same lines as the performance bound analysis of $\text{minADAG}$ outlined in Section 6.2, and the analysis of approximation bound of algorithm $\text{minADAG}$ in Appendix D below. □

APPENDIX C: Examples

Example 6 (Section 4). Given $Q_0$ and $A_0$ of Example 1, algorithm CovChk examines the max SPC sub-queries $Q_1$, and $Q_2$ of $Q_0$, and finds that $Q_1$ is covered by $A_0$ while $Q_2$ is not. It first computes the set $\Sigma_{Q_1, A_0}$ of induced FDs from $Q_1$ and $A_0$ (see Example 5), with $X_{Q_1} = \{\text{pid}, \text{fid}, \text{cid}, \text{year}, \text{month}, \text{city}\}$ and $X_{Q_2} = \{\text{pid}, \text{year}, \text{month}, \text{city}\}$. It verifies that $Q_1$ is covered by $A_0$ since $\Sigma_{Q_1, A_0} \mid X_{Q_1} \rightarrow X_{Q_1}$, and $Q_1$ is indexed by $A$ (see Example 4). Along the same lines, it finds that $Q_2$ is not covered, and concludes that $Q_0$ is not covered. In contrast, it finds that max SPC sub-queries $Q_1$ and $Q_3$ of $Q_0$ are both covered by $A_0$, and thus so is $Q_0$. □

Example 7 (Section 5). For $Q_0$ and $A_0$ of Example 1, its $(Q, A)$-hypergraph $G_{Q_0, A_0}$ is depicted in Fig. 2, after the following conversions. We write $Q_0 = Q_1 - Q_3$ ($Q_3 = Q_1(\text{cid}) \times_{\text{cid} \rightarrow \text{pid}} Q_2(\text{pid})$) in the normal form of Section 2 such that it keeps relation names of $Q_1$ unchanged, and renames (a) each relation $S$ in sub-query $Q_1(\text{cid})$ of $Q_3$ to $S'$ (e.g., dine of $Q_1(\text{cid})$ in $Q_3$ is renamed to dine'), and (b) each relation $S$ in sub-query $Q_2(\text{cid})$ of $Q_3$ to $S''$ (e.g., dine of $Q_2(\text{cid})$ in $Q_3$ to dine''). In Fig. 2, we extend the unification function $\rho_{UV}$ given in Example 5. (a) For each attribute $S'[A]$ that occurs in sub-query $Q_1(\text{cid})$ of $Q_3$, if $\rho_{UV}(S'[A]) = A$ in $Q_1$, then $\rho_{UV}(S'[A]) = A'$ for $Q_1(\text{cid})$ in $Q_3$. (b) For sub-query $Q_2(\text{cid})$ of $Q_3$, $\rho_{UV}(\text{dine'[pid]}) = \text{pid}'$, $\rho_{UV}(\text{dine'[month]}) = \text{cid}'$, $\rho_{UV}(\text{dine'[month]}) = \text{month}'$ and $\rho_{UV}(\text{dine'[year]}) = \text{year}'$. □

Example 8 (Section 5). A complete fetching plan for $Q_0'$ under $A_0$ is as follows.

\[
\begin{align*}
T_1 &= \{p_0\} (\xi_r^S(\text{pid})), \\
T_2 &= \text{fetch}(X \in T_1, \text{friend}, \text{fid}); \\
T_3 &= \pi_{\text{sid}}(T_2) (\xi_r^S(\text{fid})); \\
T_4 &= \{2015\} (\xi_r^S(\text{year})); \\
T_5 &= \{\text{MAy}\} (\xi_r^S(\text{month})); \\
T_6 &= T_3 \times T_4; \\
T_7 &= T_6 \times T_5; \\
T_8 &= \text{fetch}(X \in T_7, \text{dine}, \text{cid}); \\
T_9 &= \pi_{\text{sid}}(T_8) (\xi_r^S(\text{cid})); \\
T_{10} &= \{\text{NYC}\} (\xi_r^S(\text{city})); \\
T_{11} &\ldots, T_{20}; T_{21} = \{p_0\} (\xi_r^S(\text{pid}'')).
\end{align*}
\]

Here $T_{11} - T_{20}$ are unit fetching plans for attributes in $Q_3$, and are the same as $T_1 - T_{10}$ w.r.t. attribute renaming.

An indexing plan $\xi_{Q_0'}(\text{dine}[\text{month}])$ for relation dine' is:

\[
\begin{align*}
T_1' &= T_{19}; \\
T_2' &= T_{20}; \\
T_3' &= T_{19} \times T_{20}; \\
T_4' &= \text{fetch}(X \in T_3', \text{dine}, \text{(pid, cid)});
\end{align*}
\]

similar for other relations. Finally, an evaluation plan for $Q_0'$ under $A_0$ is exactly $Q_0'$ with each relation name $S$ replaced by the $T_S$ with $T_S = \xi_r(S)$. □

Example 9 (Section 6). For $\text{AMP}(Q_1, A_1)$, algorithm minA works as follows. It first finds that either $\psi_2$ or $\psi_3$ or $\psi_5$ can be removed from $A_1$ while keeping $Q_1$ covered. It then calculates $w(\psi_2) = \frac{11}{2}$ and $w(\psi_5) = \frac{11}{2}$ $\triangle$. Suppose that $c_1 = c_2 = 1$. Then minA greedily picks $\psi_5$ instead of $\psi_2$. It finds that no more constraints can be removed while keeping $Q_1$ covered. Thus minA returns $A_m = \{\psi_1, \psi_2, \psi_4\}$. □

Example 10 (Section 6). The complete weighted $(Q, A)$-hypergraph $G_{Q_1, A_1}$ for $Q_1$ and $A_1$ is shown in Fig. 7. □

APPENDIX D: Algorithm Analysis

Complexity of algorithm CovChk (Section 4)

We next show that CovChk can be implemented in $O(|Q|^2 + |A|)$ time. (1) It takes $O(|Q|)$ time to compute the set $\Sigma_0$ of all max SPC sub-queries of $Q$. (2) Checking whether all $Q_1$’s in $\Sigma_0$ are indexed by $A$ can be implemented in $O(|Q| + |A|)$ time, by building an index from relations in $Q$ to constraints of $A$ in $O(|Q| + |A|)$ time before the iteration. (3) It takes $O(|Q|^2)$ time to construct induced FDs $\Sigma_{Q_0, A}$, and the size of $\Sigma_{Q_0, A}$ is bounded by $|A_{Q_0}|$ for each $Q_0$. (4)
FD implication checking can be done in linear time (cf. [5]), i.e., 
\[ O(|\Sigma_{Q_s,A}| + |X^Q_C|) = O(|\Lambda_{Q_s}| + |Q_s|) \]
for each \( Q_s \). Putting these together, CovChk is in \( O(|Q|^2 + |A|) \) time.

**Complexity of Algorithm QPlan (Section 5)**

Algorithm QPlan can be implemented in \( O(|Q|(|Q| + |A|)) \) time. Indeed, (1) constructing the \( \langle Q,A \rangle \)-hypergraph \( G_{Q,A} \) takes \( O(|Q| + |A|) \) time. (2) In each iteration (lines 3-6), findHP takes \( O(|Q| + |A|) \) time to find hyperpath \( \Pi_{\{r\},u_A} \) (cf. [9]), and transQP takes \( O(|\Pi_{\{r\},u_A}|) = O(|A|) \) time to translate \( P \) into a unit fetching plan, where \(|A|\) denotes the cardinality of \( A \). There are no more than \(|Q|\) iterations. (3) Indexing plan generation takes \( O(|Q|) \) time in total. (4) The size of the evaluation plan is bounded by \(|Q|\). Putting these together, QPlan takes \( O(|Q| + |A|) + O(|Q|(|Q| + |A|)) \) + \( O(|Q|) + O(|Q|) = O(|Q|(|Q| + |A|)) \) time in total.

**Algorithm transQP (Section 5.2)**

It is the translation algorithm \( \Gamma_r \) given in the proof of Lemma 7 in Appendix B.

**Complexity of Algorithm minA (Section 6.2)**

Algorithm minA can be implemented in \( O(|Q|^2 + |A|^2(|Q| + |A|)) \) time. Indeed, (1) it takes \( O(|Q|^2 + |A|) \) time to construct \( \Sigma_{Q,A} \); (2) it iterates at most \(|A|\) times; and (3) in each iteration, it takes \( O(|A| \cdot |\Sigma_{Q,A}|) = O(|A|(|Q| + |A|)) \) time to update the scores of all constraints in \( A \) and check whether removing each of them will make \( Q \) not covered.

**Approximation bound of Algorithm minA\textsubscript{DAG} (Section 6.2).**

Let \( c(A) \) denote the sum of the \( N \)'s in all the constraints in \( A \). Let \( A_1 \) be the set of constraints in \( A \) indexing a relation, and \( A_{1i} \) be all the other constraints. Let \( A_{OP T} \) be the optimal solution to AMS(\( Q,A \)). We define \( A_{1i}^{OPT} \) and \( A_{0i}^{OPT} \) analogous to \( A_1 \) and \( A_{1i} \), respectively. One can verify that

\[
\frac{c(A_1^{OPT}) + c(A_{1i}^{OPT})}{c(A_1^{OPT}) + c(A_{1i}^{OPT})} = \frac{c(A_1^{OPT}) + c(A_{1i}^{OPT})}{c(A_1^{OPT}) + c(A_{1i}^{OPT})} \leq k + 1,
\]

where \( k = |\rho_U(X_Q) \setminus \rho_U(X)| \leq |X_Q \setminus X^Q_C| \).