An Effective Syntax for Bounded Relational Queries

Yang Cao[1,2] Wenfei Fan[1,2]

[1]School of Informatics, University of Edinburgh [2]RCBD and SLISDE, Beihang University
{Y.Cao-17@sms, wenfei@inf}.ed.ac.uk

ABSTRACT

A query Q is *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 *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 bounded under $\mathcal{A}$.

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 boundedly evaluable, and (c) it takes 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 RA queries in practice are covered, and that bounded query plans improve RA query evaluation by orders of magnitude.

Keywords

Bounded evaluability; 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 t is in the query answer $Q(D)$ when Q is an SPC query (selection, projection and Cartesian product), and PSPACE-complete if Q is in relational algebra (RA) [5]. When D is big, it is hard, if not impossible, to compute $Q(D)$ within constrained resources such as time and available processors.

To tackle this problem, a notion of boundedly evaluable queries has recently been studied [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 *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 *independent of* $|D|$, no matter how big D grows.

The idea has proven effective: on some real-life datasets, 77% of SPC queries [14] and 60% of graph pattern queries [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 [19]. Set difference (universal quantification) of RA makes the bounded evaluability analysis far more intriguing.

Example 1: Consider an example query Q_0 from Graph Search of Facebook [17]: *find me all restaurants in* NYC *which I have not been to, but in which my friends have dined in May, 2015.* The query is posed on dataset D_0, which consists of three relations: (a) $\mathsf{friend}(\mathsf{pid}, \mathsf{fid})$, stating that fid is a friend of pid, (b) $\mathsf{dine}(\mathsf{pid}, \mathsf{cid}, \mathsf{month}, \mathsf{year})$ indicating that a person pid dined in restaurant cid in month of year, and (c) $\mathsf{cafe}(\mathsf{cid}, \mathsf{city})$, stating that cid is located in city.

Query Q_0 is given in RA, with constant p_0 denoting "me":

$$
\begin{aligned}
Q_0(\mathsf{cid}) &= Q_1(\mathsf{cid}) - Q_2(\mathsf{cid}), \text{ where}\\
Q_1(\mathsf{cid}) &= \pi_{\mathsf{cid}}\big(\mathsf{friend}(p_0, \mathsf{fid}) \bowtie_{\mathsf{fid}=\mathsf{pid}} \mathsf{dine}(\mathsf{pid}, \mathsf{cid}, \text{MAY}, 2015)\\
&\qquad \bowtie_{\mathsf{cid}=\mathsf{cid}'} \mathsf{cafe}(\mathsf{cid}', \text{NYC})\big), \text{ and}\\
Q_2(\mathsf{cid}) &= \pi_{\mathsf{cid}}\ \mathsf{dine}(p_0, \mathsf{cid}, \mathsf{month}, \mathsf{year}).
\end{aligned}
$$

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

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

- ψ_1: $\mathsf{friend}(\mathsf{pid} \to \mathsf{fid}, 5000)$;
- ψ_2: $\mathsf{dine}((\mathsf{pid}, \mathsf{year}, \mathsf{month}) \to \mathsf{cid}, 31)$;
- ψ_3: $\mathsf{dine}((\mathsf{pid}, \mathsf{cid}) \to (\mathsf{pid}, \mathsf{cid}), 1)$;
- ψ_4; $\mathsf{cafe}(\mathsf{cid} \to \mathsf{city}, 1)$.

Here ψ_1 specifies a constraint imposed by Facebook [16]: a limit of 5000 friends per user; ψ_2 states that each person dines in at most 31 restaurants each month; ψ_3 says that (pid, cid) is a "key" of the pair, and ψ_4 states that each restaurant id is associated with a single city. Indices can be built on D_0 based on ψ_1 such that given a person, it

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2882942

returns all the ids of her friends by accessing at most 5000 friend tuples; similarly for ψ_2, ψ_3 and ψ_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 ψ_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 ψ_2. (3) For each cid in T_2, we fetch its cafe tuple by using the index for ψ_4, and return a set T_3 of cid's from these tuples with city $=$ NYC. The query plan fetches at most $5000 + 5000 \times 31 \times 2$ 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 $\mathcal{A}_0$.

However, query Q_2 is not bounded under $\mathcal{A}_0$: 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_0 forces us to check *all* tuples in $Q_2(D_0)$, one might think that Q_0 is not bounded either.

Nonetheless, observe that Q_0 is equivalent to $Q_0'(\mathsf{cid}) = Q_1(\mathsf{cid}) - Q_3(\mathsf{cid})$, where $Q_3(\mathsf{cid}) = Q_1(\mathsf{cid}) \bowtie_{\mathsf{cid}=\mathsf{cid}'} Q_2(\mathsf{cid}')$. Moreover, Q_3 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 ψ_3. We return all those cid's that pass the check. Thus we can answer $Q_3(D_0)$ by accessing 5000×31 tuples. Therefore, Q_0 is equivalent to bounded Q_0', with a query plan consisting of the plan for Q_1 above, followed by the plan for Q_3; it accesses at most 470000 tuples only, no matter how big D_0 grows. This shows that Q_0 is actually boundedly evaluable under $\mathcal{A}_0$. □

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 $\mathcal{A}$ of access constraints,

(a) every boundedly evaluable RA query is equivalent to a query in $\mathcal{L}$, *i.e.,* $\mathcal{L}$ expresses all bounded RA queries;

(b) every query Q in $\mathcal{L}$ is boundedly evaluable; and

(c) it takes PTIME (polynomial time) in $|Q|$ and $|\mathcal{A}|$ 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 bounded evaluable queries, just like its counterpart for safe queries. (2) As will be shown shortly, bounded evaluability analysis can be readily incorporated into commercial DBMS. Given an input RA query Q, it first checks whether Q is in $\mathcal{L}$, in PTIME by condition (c) above; if so, it generates a bounded query plan for Q by using indices in $\mathcal{A}$, 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 $\mathcal{A}$, in time bounded by the cardinality constraints of $\mathcal{A}$. We prove that every boundedly evaluable RA query under $\mathcal{A}$ is also covered by $\mathcal{A}$ (*i.e.,* property (a)).

(2) We develop an algorithm for checking covered queries (Section 4). Given an RA query Q and a set $\mathcal{A}$ of access constraints, the algorithm decides whether Q is covered by $\mathcal{A}$ in $O(|Q|^2 + |\mathcal{A}|)$-time, where $|Q|$ is the size of Q and $|\mathcal{A}|$ is the total length of access constraints in $\mathcal{A}$, independent of the size $|D|$ of dataset D. In practice, $|Q|$ and $|\mathcal{A}|$ 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 $\mathcal{A}$, the algorithm generates a query plan ξ of length $O(|Q||\mathcal{A}|)$ such that for any dataset D that satisfies $\mathcal{A}$, ξ computes $Q(D)$ by accessing a bounded amount of data determined by Q and $\mathcal{A}$. The algorithm is based on a nontrivial characterization of covered RA queries and takes $O(|Q|(|Q| + |\mathcal{A}|))$ 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 $\mathcal{A}$, it is to find a subset $\mathcal{A}_m \subseteq \mathcal{A}$ such that Q remains covered by $\mathcal{A}_m$ and the estimated data access via $\mathcal{A}_m$ is minimized. We show that the problem is NP-complete and is not in APX, *i.e.,* it has no 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 DBMS (Section 7). Given an RA query Q and a set $\mathcal{A}$ of access constraints, we check whether Q is covered by $\mathcal{A}$, and if so, we generate a bounded query plan for Q with minimal constraints in $\mathcal{A}$, 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 $\mathcal{A}$ 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 evaluability, 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 evaluability and effective syntax. The study of bounded evaluability 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 evaluability was proposed in [18], which extends [19] by ranging over all datasets that satisfy a set $\mathcal{A}$ of access constraints. A query Q is called *boundedly evaluable* under $\mathcal{A}$ if it has a *bounded query plan*; such a plan allows data access only via indices embedded in $\mathcal{A}$, 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 $\mathcal{A}$ of access constraints, it is decidable but EXPSPACE-hard to decide whether Q is boundedly evaluable under $\mathcal{A}$ [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 PTIME effective syntax for boundedly evaluable RA queries. It is radically different from the one for SPC [18], which becomes Π_2^p-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 evaluability is much harder to decide than effective boundedness. For SPC, bounded evaluability is EXPSPACE-hard [18], but effective boundedness is in PTIME [14]. (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 different from those of [14]. While effective boundedness for SPC is characterized by five syntactic rules [14], it is impossible to extend the rules to RA. It is much harder to decide which attributes to retrieve and how their values propagate for RA queries. (5) Our results substantially extend [14] and allow existing DBMS to support bounded evaluable RA queries. Even when SPC is concerned, we provide a PTIME solution that allows generic query plans, as opposed to the restriction of [14]. (6) We also study a new optimization problem for minimizing estimated data access, which is nontrivial (NP-complete and not in APX) and is not studied by [14].

Access constraints. Related to query answering under access constraints is the notion of access patterns. They require a relation to be only accessed by providing certain combinations of attributes [10–12, 25, 28]. Unlike access patterns, access constraints impose both cardinality constraints and "restricted" data accesses via indices. Moreover, we study how to answer RA queries by accessing a bounded amount of data. Hence the results on bounded evaluability are quite different from those for access patterns. In addition, we develop effective syntax for bounded RA queries, an issue that has not been studied for access patterns.

2. BOUNDED EVALUABILITY

In this section we review the notions of access constraints and bounded evaluability, following [14, 18, 19].

Access schema. Over a relational schema $\mathcal{R}$, an *access schema* $\mathcal{A}$ is a set of *access constraints* of the form [14, 19]:

$$\psi = R(X \to Y, N),$$

where R is a relation schema in $\mathcal{R}$, X and Y are sets of attributes of R, and N is a natural number.

A relation instance D of R *satisfies* the constraint if

- for any X-value $\bar{a}$ in D, $|D_Y(X = \bar{a})| \leq N$, where $D_Y(X = \bar{a}) = \{t[Y] \mid t \in D, t[X] = \bar{a}\}$; and
- there exists an *index on X for Y* that given an X-value $\bar{a}$, retrieves $D_Y(X = \bar{a})$ by accessing at most N tuples.

That is, for any given X-value, there exist at most N distinct corresponding Y values in D, and these Y values can be retrieved by using the index for ψ. Intuitively, ψ is a combination of a cardinality constraint and its index.

We say D *satisfies* access schema $\mathcal{A}$, denoted by $D \models \mathcal{A}$, if D satisfies all the constraints in $\mathcal{A}$.

For instance, denote by $\mathcal{R}_0$ the collection of relation schemas friend, dine and cafe. Then the set $\mathcal{A}_0$ consisting of $\psi_1 - \psi_4$ given in Example 1 is an access schema over $\mathcal{R}_0$.

Remark. (1) Traditional functional dependencies (FDs) are a special case of access constraints ($N = 1$). (2) Access constraint discovery will be discussed in Section 7.

Query plans under access schema $\mathcal{A}$. We study relational algebra (RA) queries, defined over a relational schema $\mathcal{R}$ with selection σ, projection π, Cartesian product $\times$, union $\cup$, set-difference $-$ and renaming ρ operators [5]. Following [18], we define a *query plan ξ for Q under $\mathcal{A}$* simply as a sequence of relational algebra operators and an additional $\mathsf{fetch}(X \in T, R, Y)$ operator returning $\bigcup_{\bar{a} \in T} D_{XY}(X = \bar{a})$ from D, where T is returned by some operation in the sequence prior to it, such that the plan retrieves data with constants in Q and these fetch operations only (see formal definition in Appendix A). Intuitively, query plans under access schema restrict traditional query plans to access data in an *controlled and quantified* manner, via fetch operations.

Bounded evaluability [18]. A query plan ξ is *boundedly evaluable under an access schema* $\mathcal{A}$ (or simply *bounded*) if

(1) for each operation $\mathsf{fetch}(X \in T, R, Y)$ in ξ, there exists an access constraint $R(X \to Y, N)$ in $\mathcal{A}$; and
(2) the length of ξ is determined by $|\mathcal{R}|$, $|\mathcal{A}|$ and $|Q|$ only, which are the sizes of $\mathcal{R}$, $\mathcal{A}$, 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 ξ for Q such that for all instances D of $\mathcal{R}$ 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 $\bar{a}$, it takes $O(N)$ time to fetch $D_{XY}(X = \bar{a})$ in D with the index in $R(X \to Y, N)$).

Example 2: Recall Q_0 and $\mathcal{A}_0$ from Example 1. A boundedly evaluable query plan for Q_0 under $\mathcal{A}_0$ is as follows.

$T_1 = \{p_0\}$, $T_2 = \mathsf{fetch}(T_1, \mathsf{friend}, \mathsf{fid})$, $T_3 = \pi_{\mathsf{fid}}(T_2)$,
$T_4 = \{2015\}$, $T_5 = \{\text{MAY}\}$, $T_6 = T_4 \times T_5$, $T_7 = T_3 \times T_6$,
$T_8 = \mathsf{fetch}(X \in T_7, \mathsf{dine}, \mathsf{cid})$, $T_9 = \pi_{\mathsf{cid}}(T_8)$,
$T_{10} = \mathsf{fetch}(X \in T_9, \mathsf{cafe}, \mathsf{city})$, $T_{11} = \sigma_{\mathsf{city}=\text{NYC}}(T_{10})$,
$T_{12} = \pi_{\mathsf{cid}}(T_{11})$,
$T_{13} = T_1 \times T_{12}$, $T_{14} = \mathsf{fetch}(X \in T_{13}, \mathsf{dine}, (\mathsf{pid}, \mathsf{cid}))$,
$T_{15} = \pi_{\mathsf{cid}}(T_{14})$, $T_{16} = T_{13} \setminus T_{15}$.

Note that the sequence $T_1, \ldots, T_{12}$ forms a boundedly evaluable query plan for sub-query Q_1 of Q_0 under $\mathcal{A}_0$. □

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

Example 3: Consider an access schema $\mathcal{A}_1$ and RA query Q_4 defined on relation schemas $R(A, B, E)$ and $S(F, G, H)$:

- $\mathcal{A}_1 = \{R(AB \to E, N), S(F \to GH, 2), S(GH \to GH, 1)\}$.
- $Q_4 = Q_4^1 - Q_4^2$, where $Q_4^1 = \pi_x(R(1, x, y) \bowtie S(w, x, y) \bowtie S(w, 1, x) \bowtie S(w, x, x))$ and $Q_4^2 = \pi_x(R(1, x, x) \bowtie S(u, 1, x) \bowtie S(u, x, x))$, where $\bowtie$ denotes natural join.

At a first glance, Q_4 seems not boundedly evaluable, since we cannot retrieve x and w values using indices in $\mathcal{A}_1$ and thus cannot get y for Q_4^1. Similarly, we cannot get u and x values for Q_4^2. However, under $S(F \to GH, 2)$ in $\mathcal{A}_1$, 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_4^1. In other words, under $\mathcal{A}_1$, the SPC sub-query Q_4^1 reduces to SPCU $Q_4^{1'} \cup Q_4^{1''}$, where $Q_4^{1'} = \pi_x(R(1, 1, x) \bowtie S(w, 1, x) \bowtie S(w, x, x)$ and $Q_4^{1''} = Q_4^2$. Thus, under $\mathcal{A}_1$, Q_4 is equivalent to $Q_4^{1'}$, which is boundedly evaluable under $\mathcal{A}_1$. □

The presence of union ($\cup$) allows us to convert SPC to SPCU under $\mathcal{A}$ (*e.g.,* Q_4^1 to $Q_4^{1'} \cup Q_4^{1''}$), which may further interact with set difference ($-$) (*e.g.,* Q_4 and $Q_4^{1'}$).

Notations. We will use the following notations.

(1) Under an access schema $\mathcal{A}$, an RA query Q_1 is $\mathcal{A}$-*equivalent to* Q_2, denoted by $Q_1 \equiv_{\mathcal{A}} Q_2$, if for all instances D of $\mathcal{R}$ with $D \models \mathcal{A}$, $Q_1(D) = Q_2(D)$. That is, in all D satisfying $\mathcal{A}$, Q_1 and Q_2 return the same answer. This notion is stronger than the conventional equivalence $Q_1 \equiv Q_2$, which holds if for all instances D of $\mathcal{R}$, $Q_1(D) = Q_2(D)$ [5].

(2) To simplify the exposition, we consider RA queries Q in a *normal form* in which all occurrences of each relation name are made distinct via renaming. For an access constraint $\phi = R(X \to Y, N)$ and a renaming S of R in Q, we refer to $S(X \to Y, N)$ as the *actualized constraint* of ϕ 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* $\mathcal{R}$*, one can compute the actualized access schema* $\mathcal{A}'$ *from* Q *and* $\mathcal{A}$ *in* $O(|Q||\mathcal{A}|)$*-time such that*

(1) for any instance D *of* $\mathcal{R}$*,* $D \models \mathcal{A}$ *iff* $D \models \mathcal{A}'$*; 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 $Q = \pi_Z \sigma_C(S_1 \times \ldots \times S_n)$ defined over a relational schema $\mathcal{R}$, where Z is a set of attributes of $\mathcal{R}$, C is the selection condition of Q, and S_i's are distinct relations after renaming (Lemma 1). We use Σ_Q to denote the set of all equality atoms $A = A'$ or $A = c$ derived from C by the transitivity of equality. For any sets X and X' of attributes of Q, we write $\Sigma_Q \vdash X = X'$ if $X = X'$ can be derived from Σ_Q, which can be checked in $O(\mathsf{max}(|X|, |X'|))$ 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 $\mathsf{cov}(Q, \mathcal{A})$, includes attributes that can be accessed via indices in $\mathcal{A}$. It is defined as follows:

- if $\Sigma_Q \vdash \sigma_{A=c}$, then $A \in \mathsf{cov}(Q, \mathcal{A})$;
- if $R(\emptyset \to X, N) \in \mathcal{A}$, then $R[X] \subseteq \mathsf{cov}(Q, \mathcal{A})$;
- if $R[X] \subseteq \mathsf{cov}(Q, \mathcal{A})$ and $\Sigma_Q \vdash R[X] = S[Y]$, then $S[Y] \subseteq \mathsf{cov}(Q, \mathcal{A})$; and
- if $R(X \to Y, N) \in \mathcal{A}$ and $R[X] \subseteq \mathsf{cov}(Q, \mathcal{A})$, then $R[Y] \subseteq \mathsf{cov}(Q, \mathcal{A})$.

Here $R(\emptyset \to X, N)$ 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 $X_Q \subseteq \mathsf{cov}(Q, \mathcal{A})$; and
- *indexed by* $\mathcal{A}$ if for each relation name S in Q, there is an actualized constraint $S(X \to Y, N)$ of $\mathcal{A}$ such that
  - $S[X] \subseteq \mathsf{cov}(Q, \mathcal{A})$, 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 (syntax) 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* $\mathcal{A}$ if for all max SPC sub-queries Q_s of Q, Q_s is covered by $\mathcal{A}$. Similarly, Q is *fetchable via* $\mathcal{A}$ (resp. *indexed by* $\mathcal{A}$) if each max sub-SPC sub-query is fetchable via $\mathcal{A}$ (resp. indexed by $\mathcal{A}$).

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 $\mathcal{A}_0$ of Example 1, Q_1 and Q_3 are covered by $\mathcal{A}_0$, but Q_2 is not. Indeed, $X_{Q_1} = \{x_{p_0}, \mathsf{fid}, \mathsf{pid}, \mathsf{cid}, x_{\text{MAY}}, x_{2015}, \mathsf{cid}', x_{\text{NYC}}\} = \mathsf{cov}(Q_1, \mathcal{A}_0)$, where x_d denotes the attribute corresponding to a constant d in Q_1. Hence Q_1 is fetchable via $\mathcal{A}_0$; moreover, Q_1 is indexed by $\mathcal{A}_0$ since friend, dine and cafe are indexed by ψ_1, ψ_2 and ψ_4, respectively; similarly for Q_3. However, Q_2 is not fetchable via $\mathcal{A}_0$ since $\mathsf{cov}(Q_2, \mathcal{A}_0) = \{x_{p_0}\}$ but $X_{Q_2} = \{x_{p_0}, \mathsf{cid}\}$, and relation dine is not indexed by any constraint in $\mathcal{A}_0$ for Q_2. As a result, Q'_0 is covered by $\mathcal{A}_0$ since both of its max SPC sub-queries Q_1 and Q_3 are covered by $\mathcal{A}_0$. In contrast, Q_0 is not covered by $\mathcal{A}_0$ since Q_2 is not. □

The main result of the paper is as follows.

Theorem 2: *Under access schema $\mathcal{A}$, for any* RA *query Q,*

(1) if Q is boundedly evaluable under $\mathcal{A}$, then Q is $\mathcal{A}$-equivalent to an RA *query Q' that is covered by $\mathcal{A}$;*

(2) if Q is covered, then Q is boundedly evaluable; and

(3) it takes PTIME *to check whether Q is covered by $\mathcal{A}$.* □

That is, we reduce the problem of deciding RA bounded evaluability to syntactic checking of covered queries, *without losing the expressive power*. Indeed, all boundedly evaluable RA queries have an $\mathcal{A}$-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 prove Theorem 2(3) and (2) in Sections 4 and 5, respectively. For (1), we show that for any RA query Q, if it has a bounded query plan ξ under $\mathcal{A}$, then ξ can be converted to a query Q_ξ covered by $\mathcal{A}$ such that $Q_\xi \equiv_{\mathcal{A}} Q$ (see Appendix B). □

4. CHECKING COVERED QUERIES

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

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

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

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

Algorithm CovChk

Input: An RA query Q and an access schema $\mathcal{A}$.
Output: "yes" if Q is covered by $\mathcal{A}$ and "no" otherwise.

1. identify the set $\mathcal{S}_Q$ of all max SPC sub-queries of Q
2. **for each** max SPC sub-query Q_s in $\mathcal{S}_Q$ **do**
3. **if** Q_s is not indexed under $\mathcal{A}$ **then return** "no";
4. construct induced FDs $\Sigma_{Q_s,\mathcal{A}}$ for Q_s and $\mathcal{A}$;
5. **if** $\Sigma_{Q_s} \not\models \hat{X}_C^{Q_s} \to \hat{X}_{Q_s}$ **then return** "no";
6. **return** "yes";

Figure 1: Algorithm CovChk

Identifying max SPC **sub-queries**. CovChk computes the set $\mathcal{S}_Q$ 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 $\mathcal{A}$, *i.e.,* indexed by $\mathcal{A}$ and fetchable via $\mathcal{A}$ (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* ρ_U is an *attribute* renaming function: for all attributes A and A' in $\mathcal{S}_Q$, $\rho_U(A) = \rho_U(A')$ (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 $\phi = R(A \to B, N)$ in $\mathcal{A}$, we call $\rho_U(R[A]) \to \rho_U(R[B])$ an *induced* FD from Q and ϕ. We denote the set of all induced FDs from Q and constraints in $\mathcal{A}$ by $\Sigma_{Q,\mathcal{A}}$.

Example 5: For Q_1 and $\mathcal{A}_0$ of Example 1, define a unification function ρ_U such that $\rho_U(\mathsf{friend}[\mathsf{pid}]) = \mathsf{pid}$, $\rho_U(\mathsf{friend}[\mathsf{fid}]) = \mathsf{fid}$, $\rho_U(\mathsf{dine}[\mathsf{pid}]) = \mathsf{fid}$, $\rho_U(\mathsf{dine}[\mathsf{cid}]) = \mathsf{cid}$, $\rho_U(\mathsf{dine}[\mathsf{year}]) = \mathsf{year}$, $\rho_U(\mathsf{dine}[\mathsf{month}]) = \mathsf{month}$, $\rho_U(\mathsf{cafe}[\mathsf{cid}]) = \mathsf{cid}$ and $\rho_U(\mathsf{cafe}[\mathsf{city}]) = \mathsf{city}$. Then $\Sigma_{Q_1,\mathcal{A}_0}$ consists of the following induced FDs: $\mathsf{pid} \to \mathsf{fid}$, $(\mathsf{fid}, \mathsf{year}, \mathsf{month}) \to \mathsf{cid}$, $(\mathsf{fid}, \mathsf{cid}) \to (\mathsf{fid}, \mathsf{cid})$, and $\mathsf{cid} \to \mathsf{city}$. □

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 $X_C^{Q_s} \subseteq X_{Q_s}$ be the set of attributes A in Q_s such that $\Sigma_{Q_s} \vdash A = c$ for some constant c. Let $\hat{X}_{Q_s} = \rho_U(X_{Q_s})$ and $\hat{X}_C^{Q_s} = \rho_U(X_C^{Q_s})$. Then we have:

Lemma 4: *An* SPC *query Q_s is fetchable under $\mathcal{A}$ if and only if $\Sigma_{Q_s,\mathcal{A}} \models \hat{X}_C^{Q_s} \to \hat{X}_{Q_s}$.* □

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

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

Lemma 4 reduces the problem of checking whether an SPC query Q_s is fetchable via $\mathcal{A}$ to the implication analysis of FDs. Based on the lemma, algorithm CovChk checks

Notation	Description
$\mathcal{A}$	(actualized) access schema
$\|\mathcal{A}\|$	the total length of access constraints in $\mathcal{A}$
$\|\!\|\mathcal{A}\|\!\|$	the number of constraints in $\mathcal{A}$
Q_s	a max SPC sub-query of Q
Σ_{Q_s}	equality $A = A'$ and $A = c$ derived from Q_s
X_Q	attributes in σ_C or π_Y of some max Q_s of Q
X_C^Q	attributes A such that $\Sigma_{Q_s} \vdash A = c$ for a Q_s of Q
X_Q^S	attributes in both S and X_{Q_s} for some Q_s of Q
$\rho_U(A)$	renaming of A with unification function ρ_U
$\rho_U(X)$	$\{\rho_U(A) \mid A \in X\}$
$\Sigma_{Q_s,\mathcal{A}}$	the set of induced FDs from Q_s and $\mathcal{A}$
ξ^c	canonical bounded query plan
$\xi_F^c(A)$	unit fetching plan for attribute A
$\mathcal{G}_{Q,\mathcal{A}}$	$\langle Q, \mathcal{A}\rangle$-hypergraph for Q and $\mathcal{A}$
Π_{V_S,u_A}	hyperpath from set V_S to node u_A

Table 1: Notations

whether Q_s is fetchable by firstly constructing the set $\Sigma_{Q_s,\mathcal{A}}$ of induced FDs from Q_s and $\mathcal{A}$, and then checking whether $\Sigma_{Q_s,\mathcal{A}} \models \hat{X}_C^{Q_s} \rightarrow \hat{X}_{Q_s}$ by invoking a linear-time FD implication algorithm [5] (lines 4–5). It checks whether Q_s is indexed under $\mathcal{A}$ simply by definition (line 3).

Example 6: Given Q_0, Q_0' and $\mathcal{A}_0$ of Example 1, by examining max SPC sub-queries, algorithm CovChk finds that Q_0' is covered by $\mathcal{A}_0$ 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 $O(|Q|^2+|\mathcal{A}|)$ 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.

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 $\mathcal{A}$, Q has a canonical bounded query plan under $\mathcal{A}$. (2) There exists an algorithm that given Q covered by $\mathcal{A}$, generates a canonical bounded query plan of length $O(|Q||\mathcal{A}|)$ in $O(|Q|(|Q| + |\mathcal{A}|))$ time.* □

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

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 $\mathcal{A}$ *if and only if* Q has a canonical bounded plan under $\mathcal{A}$. From this Theorem 5(1) follows.

Canonical bounded query plans. For an RA query Q under an access schema $\mathcal{A}$, a *canonical bounded query plan* ξ^c is a boundedly evaluable query plan for Q that consists of a *fetching plan* ξ_F^c, followed by an *indexing plan* ξ_I^c and then an *evaluation plan* ξ_E^c, as follows (see Appendix A).

Fetching plan ξ_F^c: A fetching plan ξ_F^c is a sequence of *unit fetching plans* $\xi_F^c(A_1)$, …, $\xi_F^c(A_m)$, for all attributes A_1, …, A_m in X_Q of Q. Here unit plan $\xi_F^c(A_i)$ fetches all the necessary values for A_i; it may use fetch by employing an access constraint ϕ of $\mathcal{A}$, projections (π) and Cartesian-product ($\times$), but has no need for selection (σ).

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

Evaluation plan ξ_E^c. Plan ξ_E^c is the RA expression of Q, in which each relation S_i in Q is replaced by T_k, where $T_k = \xi_I^c(S_i)$ is the output of the indexing plan $\xi_I^c(S_i)$ for S_i.

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

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 $\mathcal{A}$, (1) Q is fetchable via $\mathcal{A}$ iff Q has a fetching plan under $\mathcal{A}$; and (2) Q is indexed by $\mathcal{A}$ iff Q has an indexing plan under $\mathcal{A}$.* □

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 $\mathcal{A}$ and an RA Q covered by $\mathcal{A}$, returns a canonical bounded query plan ξ^c of bounded length in $O(|Q|(|Q| + |\mathcal{A}|))$ time. The idea of the algorithm is to encode Q and $\mathcal{A}$ in a hypergraph representation such that (i) there is a certain hyperpath in the hypergraph *iff* Q is fetchable under $\mathcal{A}$; and (ii) each such hyperpath encodes a canonical fetching plan for Q under $\mathcal{A}$.

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

$\langle Q, \mathcal{A}\rangle$-hypergraph $\mathcal{G}_{Q,\mathcal{A}}$. A *directed hypergraph* $\mathcal{H}$ (cf. [9]) is a pair (V, E), where V is a nonempty set of nodes and E is a set of hyperedges. A *hyperedge* e in E is an ordered pair (H, t), where $H \subseteq V$, $H \neq \emptyset$, and $t \in V \setminus H$. Here H and t are called the *head* and *tail* of e, and are denoted by $\mathsf{head}(e)$ and $\mathsf{tail}(e)$, respectively. The size $|\mathcal{H}|$ of $\mathcal{H}$ is the sum of the cardinality of its hyperedges, *i.e.,* $\sum_{e \in E} |\mathsf{head}(e)|$. Hypergraphs have been used to model FDs with single attribute on their RHS (called RHS-FD) by its hyperedges [9].

Given an RA query Q and an access schema $\mathcal{A}$, we use a hypergraph to encode the induced FDs for all max SPC sub-queries of Q. For each induced FD $X \rightarrow Y$, there are $|Y \setminus X| + 1$ induced RHS-FDs $X \rightarrow \tilde{Y}$ and $\tilde{Y} \rightarrow Y_i$ for each $Y_i \in Y$, where $\tilde{Y}$ is a new attribute name denoting Y. A *$\langle Q, \mathcal{A}\rangle$-hypergraph $\mathcal{G}_{Q,\mathcal{A}}$* for Q and $\mathcal{A}$ 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 $\emptyset \rightarrow \tilde{Y}$, $\emptyset$ is encoded by r; and (3) for each attribute A in $\{\rho_U(A) \mid A \in X_C^{Q_s}$, Q_s is a max SPC sub-query of $Q\}$, there is a hyperedge from r to the node encoding A (see Appendix A).

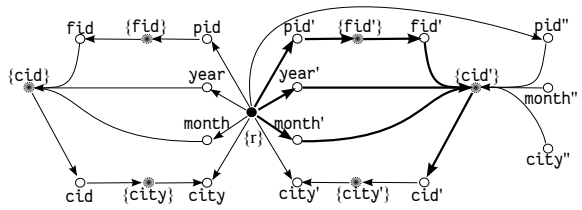


Figure 2: $\langle Q, \mathcal{A}\rangle$-hypergraph $\mathcal{G}_{Q'_0,\mathcal{A}_0}$ for Q'_0 and $\mathcal{A}_0$

Example 7: The $\langle Q, \mathcal{A}\rangle$-hypergraph $\mathcal{G}_{Q'_0,\mathcal{A}_0}$ for Q'_0 and $\mathcal{A}_0$ of Example 1 is depicted in Fig. 2 (see Appendix C). □

Hyperpath. A *sub-hypergraph* of $\mathcal{H} = (V, E)$ is a hypergraph $\mathcal{H}' = (V', E')$ such that $V' \subseteq V$, $E' \subseteq E$, and E' is restricted to V'. A *hyperpath* [9] in $\mathcal{H}$ 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 $\mathcal{H}$ 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 $\langle e_1, \ldots, e_k\rangle$ such that (a) for any $e_i \in E_{\Pi_{S,t}}$, $\mathsf{head}(e_i) \subseteq S \cup \{\mathsf{tail}(e_1), \ldots, \mathsf{tail}(e_{i-1})\}$; (b) $t = \mathsf{tail}(e_k)$; and (c) no sub-hypergraph of $\Pi_{S,t}$ other than itself is a hyperpath from S to t in $\mathcal{H}$. For example, a hyperpath $\Pi_{\{r\},u_{\mathsf{cid}'}}$ from r to $u_{\mathsf{cid}'}$ in $\mathcal{G}_{Q'_0,\mathcal{A}_0}$ 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 $\mathcal{A}$ and attribute A in X_Q of Q, there exists a unit fetching plan $\xi^c(A)$ for Q under $\mathcal{A}$* if and only if *there exists a hyperpath from r to $u_{\rho_U(A)}$ in the hypergraph $\mathcal{G}_{Q,\mathcal{A}}$.* □

Lemma 7 tells us that to get a canonical fetching plan for Q under $\mathcal{A}$, it suffices to find hyperpaths from r to u_A in $\mathcal{G}_{Q,\mathcal{A}}$ 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 $\mathcal{A}$ and an RA query Q covered by $\mathcal{A}$; it returns a canonical bounded query plan $\mathcal{P}_{Q,\mathcal{A}}$ for Q under $\mathcal{A}$. It generates $\mathcal{P}_{Q,\mathcal{A}}$ in three steps: it first generates unit fetching plans for attributes in X_Q (lines 1-6). It then builds an indexing plan $\xi^c_I(S)$ for each relation name S that occurs in Q on top of the fetching plans (lines 7-9). Finally it adds the evaluation plan ξ^c_E (line 10).

More specifically, it first constructs the $\langle Q, \mathcal{A}\rangle$-hypergraph $\mathcal{G}_{Q,\mathcal{A}}$ for Q and $\mathcal{A}$ (line 1), and initializes data structures for storing unit fetching plans (L_F) and the final query plan ($\mathcal{P}_{Q,\mathcal{A}}$) (line 2). It then iteratively finds unit fetching 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 $\mathcal{G}_{Q,\mathcal{A}}$, by invoking procedure findHP (line 4; not shown). Here findHP can be implemented in $O(|\mathcal{G}_{Q,\mathcal{A}}|) = O(|Q| + |\mathcal{A}|)$ time by traversing $\mathcal{G}_{Q,\mathcal{A}}$ [9]. It then converts the hyperpath into a unit fetching plan $\xi^c_F(A)$ via procedure transQP (line 5) (see Appendix D for details), and adds the plan to $\mathcal{P}_{Q,\mathcal{A}}$ (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). It finally adds the evaluation plan of Q to $\mathcal{P}_{Q,\mathcal{A}}$ (line 10), and returns $\mathcal{P}_{Q,\mathcal{A}}$ (line 11).

Example 8: Given Q'_0 and $\mathcal{A}_0$ of Example 1, QPlan first constructs the hypergraph $\mathcal{G}_{Q'_0,\mathcal{A}_0}$ shown in Fig 2 for Q'_0 and $\mathcal{A}_0$. 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

Algorithm QPlan

Input: An access schema $\mathcal{A}$ and an RA query Q covered by $\mathcal{A}$.
Output: A canonical bounded query plan ξ^c for Q under $\mathcal{A}$.

1. construct the $\langle Q, \mathcal{A}\rangle$-hypergraph $\mathcal{G}_{Q,\mathcal{A}}$ for Q and $\mathcal{A}$;
2. $L_F[\,] := nil$; $\mathcal{P}_{Q,\mathcal{A}} = nil$;
3. **for each** attribute $A \in X_Q$ of Q **do**
   /* find a hyperpath from r to $u_{\rho_U(A)}$ in $\mathcal{G}_{Q,\mathcal{A}}$ for $\rho_U(A)$ */
4. $\quad \Pi_{\{r\},u_{\hat{A}}} := \mathsf{findHP}(r, u_{\hat{A}}, \mathcal{G}_{Q,\mathcal{A}})$; /* $\hat{A} = \rho_U(A)$ */
   /* translate hyperpath to a unit fetching plan for A */
5. $\quad L_F[A] := \mathsf{transQP}(\Pi_{\{r\},u_{\hat{A}}})$;
6. $\quad$ append $L_F[A]$ to $\mathcal{P}_{Q,\mathcal{A}}$;
7. **for each** relation S in Q **do**
8. $\quad$ construct indexing plan $\xi^c_I(S)$ with $L_F[A]$ for all A in X^S_Q;
9. $\quad$ append $\xi^c_I(S)$ to $\mathcal{P}_{Q,\mathcal{A}}$;
10. append evaluation plan ξ^c_E for Q to $\mathcal{P}_{Q,\mathcal{A}}$;
11. **return** $\mathcal{P}_{Q,\mathcal{A}}$;

Figure 3: Algorithm QPlan

(recall the setting of Example 7 in Appendix C). It finds a hyperpath $\Pi_{\{r\},u_{\mathsf{cid}'}}$ in $\mathcal{G}_{Q'_0,\mathcal{A}_0}$, marked in bold in Fig. 2. It then translates $\Pi_{\{r\},u_{\mathsf{cid}'}}$ into a unit fetching plan consisting of T_1-T_9 given in Example 2. Similarly, it generates unit fetching plans for all the other attributes in $X_{Q'_0}$. It then finds indexing plans for relations in Q'_0. For instance, an indexing plan for dine'' is as follows: $T'_1 = T_9$ (since T_1, …, T_9 form a unit fetching plan for cid'), $T'_2 = \{p_0\}$, $T'_3 = T'_1 \times T'_2$, $T'_4 = \mathsf{fetch}(X \in T'_3, \mathsf{dine}'', (\mathsf{pid}'', \mathsf{cid}'))$. Finally, it adds the evaluation plan. A complete canonical bounded query plan for Q'_0 under $\mathcal{A}_0$ is given in Appendix C, which is essentially the same as the one given in Example 2. □

Correctness & Complexity. The correctness of QPlan is ensured by Theorem 5(1) and Lemma 7. By Theorem 5(1), a covered Q has a canonical bounded query plan, including a unit canonical fetching plan $\xi^c_F(A)$ for each attribute A in X_Q of Q. By Lemma 7, there exists a hyperpath from r to u_A for each $A \in X_Q$, encoding $\xi^c_F(A)$. Hence QPlan is warranted to be able to find such a plan.

Algorithm QPlan can be implemented in $O(|Q|(|Q|+|\mathcal{A}|))$ time (See Appendix D for a detailed analysis).

The lemma below completes the proof of Theorem 5(2).

Lemma 8: *Given an* RA *query Q covered by $\mathcal{A}$,* QPlan *finds a canonical bounded query plan of length $O(|Q||\mathcal{A}|)$.* □

6. ACCESS MINIMIZATION

In this section, we study a new optimization problem for bounded evaluability, referred to as the *access minimization problem* and denoted by $\mathsf{AMP}(Q, \mathcal{A})$. It is stated as follows.

- Input: Access schema $\mathcal{A}$, RA query Q covered by $\mathcal{A}$.
- Output: A subset $\mathcal{A}_m \subseteq \mathcal{A}$ such that Q is also covered by $\mathcal{A}_m$ and $\mathcal{A}_m$ is *minimum*, *i.e.,* for any other subset $\mathcal{A}' \subseteq \mathcal{A}$, if Q is also covered by $\mathcal{A}'$, then $\sum_{R(X\to Y,N)\in\mathcal{A}_m} N \leq \sum_{R(X\to Y,N)\in\mathcal{A}'} N$.

That is, it is to identify a small set $\mathcal{A}_m$ of access constraints in $\mathcal{A}$ that covers Q and moreover, $\mathcal{A}_m$ estimates a "minimum" amount of data to be accessed for answering Q. It also suggests how many access constraints we need to cover a query, and the size of indices built for the constraints.

While useful, the problem is hard. We show that the problem is intractable (Section 6.1). Nonetheless, we provide efficient algorithms with performance guarantees (Section 6.2).

6.1 Intractability & Approximation Hardness

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

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

- *acyclic*: when $\mathcal{G}_{Q,\mathcal{A}}$ is acyclic; and
- *elementary*: for each $\phi = R(X \to Y, N)$ in $\mathcal{A}$, either ϕ is an *indexing* constraint, *i.e.,* of the form $R(X \to X, 1)$, 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 $\mathcal{A}_0$ in Example 1 are an acyclic case since $\mathcal{G}_{Q_0',A_0}$ (Fig. 2) is acyclic; and (2) Q_0' and $\mathcal{A}_0 \setminus \{\psi_2\}$ are an elementary case.

These problems are nontrivial, even their special cases.

Theorem 9: *(1)* $\mathsf{dAMP}(Q, \mathcal{A}, K)$ *is* NP*-complete.*
(2) $\mathsf{oAMP}(Q, \mathcal{A})$ *is not approximable within* $c * \log |X_Q \setminus X_C^Q|$ *for any constant* $c > 0$.
(3) When $(Q, \mathcal{A})$ *is acyclic or elementary,* $\mathsf{dAMP}(Q, \mathcal{A}, K)$ *remains* NP*-hard, and* $\mathsf{oAMP}(Q, \mathcal{A})$ *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 $\mathsf{AMP}(Q, \mathcal{A})$, any efficient algorithm is necessarily heuristic. Below we provide an efficient heuristic that *guarantees* to find a *minimal* $\mathcal{A}_m \subseteq \mathcal{A}$ that covers Q, *i.e.,* removing any constraint from $\mathcal{A}_m$ makes Q not covered by $\mathcal{A}_m$. Moreover, for the two special cases, there are approximation algorithms with accuracy bounds.

Theorem 10: *(1) There is an algorithm for* $\mathsf{AMP}(Q, \mathcal{A})$ *that finds minimal* $\mathcal{A}_m$ *in* $O(|Q|^2 + \|\mathcal{A}\|(|Q| + |\mathcal{A}|))$ *time.*
(2) For acyclic $(Q, \mathcal{A})$, $\mathsf{oAMP}(Q, \mathcal{A})$ *is approximable within* $O(1 + |X_Q \setminus X_C^Q|)$ *in* $O(|Q| + |\mathcal{A}|)$ *time.*
(3) For elementary $Q(Q, \mathcal{A})$, $\mathsf{oAMP}(Q, \mathcal{A})$ *is approximable within* $O(1 + |X_Q \setminus X_C^Q|^\epsilon)$ *in* $O((|Q| + |\mathcal{A}|)^{\frac{1}{\epsilon}} |X_Q \setminus X_C^Q|^{\frac{2}{\epsilon}}$ *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 $\mathsf{AMP}(Q, \mathcal{A})$ for the general case, denoted by minA (not shown). It is based on the following heuristics: a constraint $\phi = R(X \to Y, N_\phi)$, if it is not used to index a relation (Section 3), then it is less likely in the optimum solution $\mathcal{A}_m$ if Q remains covered by $\mathcal{A} \setminus \{\phi\}$, and moreover, (a) $\mathsf{cov}(Q, \mathcal{A}) \setminus \mathsf{cov}(Q, \mathcal{A} \setminus \{\phi\})$ is small; and (b) N_ϕ is large.

Based on the observation, algorithm minA works as follows. It first constructs the set $\Sigma_{Q,\mathcal{A}}$ of induced FDs of Q and $\mathcal{A}$. It then iteratively removes "redundant" FDs from $\Sigma_{Q,\mathcal{A}}$. In each iteration, it greedily selects an induced FD that corresponds to access constraint ϕ, such that (a) Q remains covered by $\mathcal{A} \setminus \{\phi\}$; and (b) $w(\phi) = \frac{c_1 \cdot N_\phi}{c_2 \cdot (|\mathsf{cov}(Q,\mathcal{A}) \setminus \mathsf{cov}(Q, \mathcal{A} \setminus \{\phi\})| + 1)}$ is maximum among all constraints $\mathcal{A}$, 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 $\Sigma_{Q,\mathcal{A}}$ when it cannot remove more FDs from $\Sigma_{Q,\mathcal{A}}$.

Example 9: Consider Q_1 and $\mathcal{A}_0$ given in Example 1. Let $\mathcal{A}_1$ consist of constraints in $\mathcal{A}_0$ and an additional ψ_5: $\mathsf{dine}((\mathsf{pid}, \mathsf{year}) \to \mathsf{cid}, 366)$, *i.e.,* each person dines out at most 366 times per year. For $\mathsf{AMP}(Q_1, \mathcal{A}_1)$, algorithm minA returns $\mathcal{A}_m = \{\psi_1\ \psi_2,\ \psi_4\}$ (see details in Appendix C). □

Analysis. Algorithm minA always returns minimal $\mathcal{A}_m \subseteq \mathcal{A}$ for $\mathsf{AMP}(Q, \mathcal{A})$ since it keeps removing FDs until $\mathcal{A}_m$ is minimal, in $O(|Q|^2 + \|\mathcal{A}\|^2(|Q| + |\mathcal{A}|))$ time (see Appendix D).

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

Weighted $\langle Q, \mathcal{A}\rangle$*-hypergraph.* For an RA query Q and an access schema $\mathcal{A}$, the *weighted* $\langle Q, \mathcal{A}\rangle$*-hypergraph* is a weighted hypergraph, where each hyperedge carries a weight defined as follows. Recall $\mathcal{G}_{Q,\mathcal{A}}$ from Section 5.2. For each induced FD $X \to Y$ in $\Sigma_{Q,\mathcal{A}}$ derived from an constraint $R(X \to Y, N)$ in $\mathcal{A}$, the hyperedge encoding the induced RHS-FD $X \to \tilde{Y}$ has weight N, and hyperedges encoding the remaining induced $\tilde{Y} \to Y_i$ (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 $\mathcal{A}_1$ of Example 9, its weighted $\langle Q, \mathcal{A}\rangle$-hypergraph $\mathcal{G}_{Q_1,\mathcal{A}_1}$ is depicted in Fig. 7 in Appendix C (see Appendix A for a formal definition).

Algorithm $\mathsf{minA}_{\mathsf{DAG}}$. Based on this notion, $\mathsf{minA}_{\mathsf{DAG}}$ simply computes the shortest hyperpaths from node r in $\mathcal{G}_{Q,\mathcal{A}}$ to nodes encoding attributes in $(\hat{X_Q} \setminus \hat{X}_C^Q)$ (recall $\hat{X} = \rho_U(X)$ 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 $\mathcal{A}_1$ of Example 9, its weighted $\langle Q, \mathcal{A}\rangle$-hypergraph $\mathcal{G}_{Q_1,\mathcal{A}_1}$ is acyclic (see Fig. 7 in Appendix B). Given Q_1 and $\mathcal{A}_1$, $\mathsf{minA}_{\mathsf{DAG}}$ computes shortest hyperpaths from r to u_{pid}, u_{fid}, u_{cid}, u_{month}, u_{year} and u_{city}. For example, the shortest hyperpath from r to u_{cid} is the one containing edge $(\{\mathsf{fid}, \mathsf{year}, \mathsf{month}\}, \mathsf{cid})$ with weight 31. Algorithm $\mathsf{minA}_{\mathsf{DAG}}$ returns constraints ψ_1, ψ_2, ψ_4 in $\mathcal{A}_1$ 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, $\mathsf{minA}_{\mathsf{DAG}}$ always returns $\mathcal{A}' \subseteq \mathcal{A}$ 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_A denoting attribute A is the minimum "cost" to cover A with $\mathcal{A}$.

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

Elementary case. As a proof of Theorem 10(3), we develop an algorithm, denoted by $\mathsf{minA}_{\mathsf{E}}$ (omitted), for the elementary case $(Q, \mathcal{A})$ of $\mathsf{AMP}(Q, \mathcal{A})$. The idea is by reduction to the *directed minimum steiner arborescence* problem $(\mathsf{dminSAP}(G, u, V_T))$ (cf. [15]), which is to find the minimum

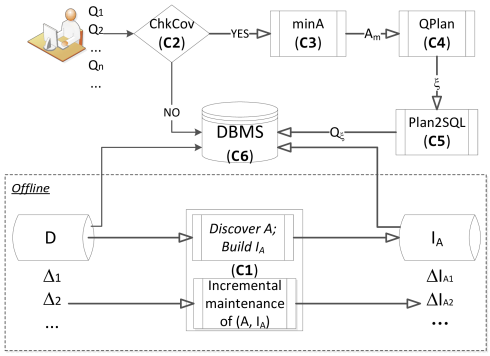


Figure 4: Bounded evaluability on DBMS

weighed arborescence rooted at node u spanning all nodes in a set V_T in a weighted directed graph G.

The reduction is as follows. Given an elementary case $(Q, \mathcal{A})$, we construct an instance (G, u, V_T) of dminSAP:

(a) G is the weighted $\langle Q, \mathcal{A}\rangle$-hypergraph $\mathcal{G}_{Q,\mathcal{A}_{ni}}$ for Q and $\mathcal{A}_{ni} \subset \mathcal{A}$, where $\mathcal{A}_{ni}$ contains all unit constraints in $\mathcal{A}$;
(b) u is the special node r in $\mathcal{G}_{Q,\mathcal{A}_{ni}}$; and
(c) V_T is the set of nodes in $\mathcal{G}_{Q,\mathcal{A}_{ni}}$ that correspond to attributes in $\hat{X}_Q \setminus \hat{X}_C^Q$ (see Table 1).

For elementary $(Q, \mathcal{A})$, $\mathcal{G}_{Q,\mathcal{A}_{ni}}$ is actually a weighted directed graph rooted at node r. Thus this is an instance of dminSAP(G, u). The reduction guarantees the following.

Lemma 11: *For elementary $(Q, \mathcal{A})$, oAMP$(Q, \mathcal{A})$ has a c-approximation algorithm that takes $O(f(Q, \mathcal{A}))$-time if dminSAP$(\mathcal{G}_{Q,\mathcal{A}_{ni}}, r, V)$ has a $(c-1)$-approximation algorithm in $O(f(|Q|, |\mathcal{A}_{ni}|) + |\mathcal{A}|)$-time.* □

Algorithm. Based on Lemma 11, algorithm minA$_E$ works as follows. (a) It first builds the reduction described above. (b) It then computes the minimum weighted arborescence DT_r rooted at r of $\mathcal{G}_{Q,\mathcal{A}_{ni}}$ that spans all nodes in V_T, for $(\mathcal{G}_{Q,\mathcal{A}_{ni}}, r, V_T)$ constructed in (a). (c) It returns the following constraints in $\mathcal{A}$: (i) constraints corresponding to edges in DT_r; and (ii) constraints that index relations in Q.

Analysis. It is known that for dminSAP$(\mathcal{G}_{Q,\mathcal{A}_{ni}}, r, V)$, there exists an $O(|V_T|^{\epsilon})$-approximation algorithm that takes at most $O(|\mathcal{G}_{Q,\mathcal{A}_{ni}}|^{\frac{1}{\epsilon}}|V_T|^{\frac{2}{\epsilon}})$-time for any constant $\epsilon > 0$ [15]. Moreover, observe that $|V_T| = |\hat{X}_Q \setminus \hat{X}_C^Q| \leq |X_Q \setminus X_C^Q|$. Thus minA$_E$ is the algorithm promised by Theorem 10(3).

This completes the proof of Theorem 10.

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 $\mathcal{I}_\mathcal{A}$ to denote the indices for all constraints in an access schema $\mathcal{A}$.

A framework of bounded evaluation. The framework is shown in Fig. 4. Given an application for queries over instances of a relational schema $\mathcal{R}$, it works as follows. As *offline* preprocessing (**C1** in Fig. 4), it discovers an access schema $\mathcal{A}$ from (sample) instances of $\mathcal{R}$, builds indices $\mathcal{I}_\mathcal{A}$ for $\mathcal{A}$ on the instance D of $\mathcal{R}$ in use, and maintains $\mathcal{I}_\mathcal{A}$ in response to updates to D. Given a user RA query posed on D, it first checks whether Q is covered by $\mathcal{A}$ (**C2**). If so, it picks a minimum set $\mathcal{A}_m$ of $\mathcal{A}$ that covers Q (**C3**), generates a bounded query plan ξ for Q under $\mathcal{A}_m$ (**C4**), and translates it into an SQL query Q_ξ (**C5**). Query Q_ξ can then be evaluated directly by the underlying DBMS on a bounded dataset D_Q identified by the bounded plan ξ (**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 $\langle \mathcal{A}, \mathcal{I}_\mathcal{A}\rangle$. It has three parts.

(a) Discovering $\mathcal{A}$. Like FDs, access constraints are defined on schema $\mathcal{R}$. 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 $R(X \to Y, N)$. These include those composed of attributes with a finite domain, *e.g.,* $R(X \to \mathsf{month}, 12)$, stating that a year has 12 months. These constraints hold on *all instances* of $\mathcal{R}$, just like discovered FDs.

Discovered constraints also include those determined by policies and statistics, *e.g.,* ψ_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 $\mathcal{I}_\mathcal{A}$. For each discovered constraint $\phi = R(X \to Y, N)$ in $\mathcal{A}$, the index for ϕ is constructed by creating a partial table $T_{XY} = \pi_{XY}(D_R)$ and building a hash index on X over T_{XY}, where D_R 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 $O(\|\mathcal{A}\||D|)$ time to build all indices in $\mathcal{A}$, and the total size $\mathcal{I}_\mathcal{A}$ is at most $O(\|\mathcal{A}\||D|)$.

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

Proposition 12: *In response to updates ΔD to D, both $\mathcal{A}$ and $\mathcal{I}_\mathcal{A}$ can be updated in $O(N_\mathcal{A}|\Delta D|)$ time, where $N_\mathcal{A} = \Sigma_{R(X\to Y,N)\in\mathcal{A}} N$.* □

(2) Checking whether Q is covered by $\mathcal{A}$. 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 $\mathcal{I}_\mathcal{A}$.

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

(5) Interpreting $\xi_{(Q,\mathcal{A})}$ as SQL query Q_ξ. We develop an algorithm, denoted by Plan2SQL (omitted), to translate a bounded plan ξ into an SQL query Q_ξ, such that Q_ξ can be directly executed by DBMS. Given ξ and $\mathcal{A}$, Plan2SQL returns Q_ξ such that for any dataset $D \models \mathcal{A}$, Q_ξ returns $Q(D)$ by accessing the same amount of data *in index* $\mathcal{I}_\mathcal{A}$ as ξ does in D. For instance, recall Q_1 and $\mathcal{A}_0$ of Example 1, $\mathcal{A}'_0 = \mathcal{A}_0 \setminus \{\psi_3\}$, and the bounded query plan ξ for Q_1 under $\mathcal{A}'_0$ given in Example 2. Let the index relations in $\mathcal{I}_\mathcal{A}$ under ψ_1, ψ_2 and ψ_4 in $\mathcal{A}'_0$ be ind_friend, ind_dine and ind_cafe, respectively. Plan2SQL$(\xi, \mathcal{A}'_0)$ returns the following SQL query:

```
select distinct cid
from ind_cafe
```

```
where city = NYC and cid in
    (select distinct cid
     from   ind_dine  /* no access to the underlying D */
     where month = MAY and year = 2015 and pid in
           (select distinct fid from ind_friend where pid = p0))
```

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 condition 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 evaluability 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).

(1) US Air carriers (AIRCA) records flight and statistic data of US air carriers from year 1987 to 2014. It consists of Flight On-Time Performance data [4] for departure and arrival data, and Carrier Statistic data [3] for airline market and segment data of the air carriers. It has 7 tables, 358 attributes, and over 162 million tuples, about 60GB of data.

(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 $2^{-5} \times 360$ 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 #-sel 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].

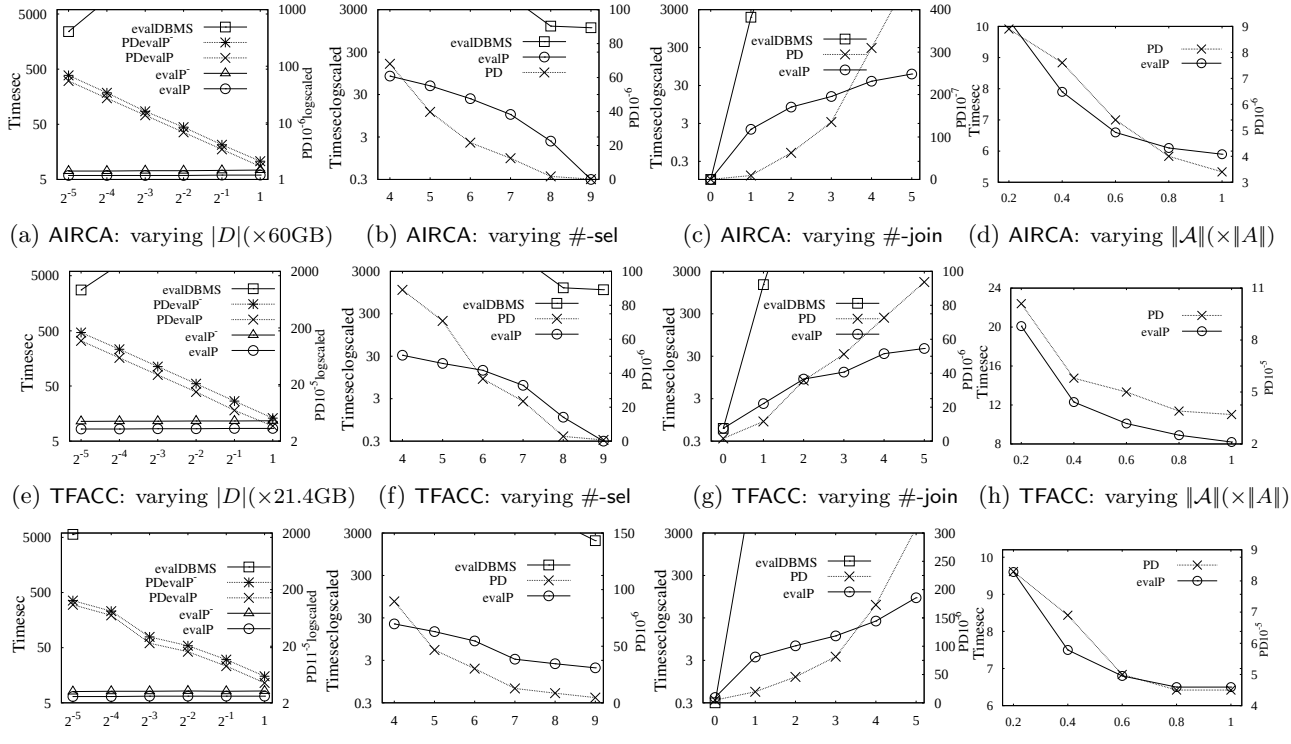
Algorithms. We implemented the following algorithms in Python: (1) ChkCov (Section 4) to check whether an RA query is covered; (2) QPlan (Section 5) to generate canonical query plans for covered queries; (3) minA, minA_{DAG} and minA_{E} (Section 6) to find minimum access constraints for covered queries; (4) Plan2SQL to interpret canonical query plans generated by QPlan as SQL queries (Section 7); (5) evalQP^{-} and evalQP to evaluate the translated queries Q_ξ with and without minimized $\mathcal{A}_m$ (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.

Exp-1: Effectiveness of bounded evaluability.

(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



(a) AIRCA: varying $|D|$(×60GB) (b) AIRCA: varying #-sel (c) AIRCA: varying #-join (d) AIRCA: varying $\|\mathcal{A}\|(\times\|A\|)$

(e) TFACC: varying $|D|$(×21.4GB) (f) TFACC: varying #-sel (g) TFACC: varying #-join (h) TFACC: varying $\|\mathcal{A}\|(\times\|A\|)$

(i) MCBM: varying $|D|$ (×90GB) (j) MCBM: varying #-sel (k) MCBM: varying #-join (l) MCBM: varying $\|\mathcal{A}\|(\times\|A\|)$

Figure 5: Effectiveness of bounded evaluability

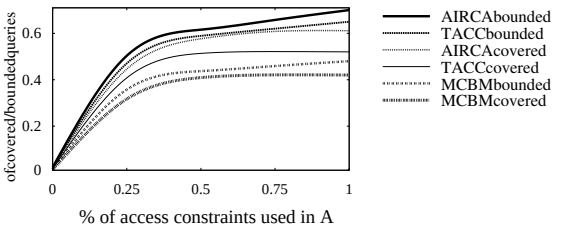


Figure 6: Percentage of covered (bounded) queries

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 $\|\mathcal{A}\|$. We report (i) the average evaluation time (the left y-axis), and (ii) ratio $P(D_Q) = |D_Q|/|D|$, measuring the total amount 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 8.5×10^3, 6.1×10^3 and 7.8×10^3 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 1.7×10^{-6}, 3.7×10^{-5}, 2.2×10^{-6} on full-size AIRCA, TFACC and MCBM. *i.e.,* 0.00017%, 0.0037% and 0.00022% 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

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, 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 EXPLAIN output, we verified that this is partially due to the following. Given an access constraint $R(X \to Y, N)$, evalQP fetches only distinct values of the relevant XY attributes, but 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.,* EXPLAIN output shows that MySQL consistently accesses entire tables when there are non-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 impacts on the query plans generated by 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 QPlan took less steps to fetch all attribute values needed. (b) Algorithm evalQP scales well with #-sel and #-join. It found answers for largest Q within 89.5s, on the three full-size datasets. (c) Algorithm 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 evalDBMS since it did not complete its computation within 3000s on all three datasets.

(3) Impact of $\|\mathcal{A}\|$. To evaluate the impact of access constraints, we varied $\|\mathcal{A}\|$ 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 evalDBMS. We report $P(D_Q)$ and run time of evalQP. As shown in Figures 5(d), 5(h) and 5(l), (a) more constraints help QPlan generate better query plans, as expected. For example, when all access constraints were used, 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 QPlan can find better plans given more options. (c) Algorithm evalDBMS did not produce results in any test within 3000s, even given the indices in full-size $\mathcal{A}$ of constraints.

(III) Effectiveness of minA. We also evaluated the effectiveness of minA for minimizing access schemas by comparing evalQP and evalQP$^-$. As reported in Figures 5(a), 5(e) and 5(i), (1) minA helps QPlan generate query plans that access less data; indeed, evalQP accessed much smaller D_Q than evalQP$^-$ in most cases; for example, $P(D_Q)$ is 0.0037% for evalQP on full-size TFACC, while it is 0.0051% for 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, evalQP used index no larger than 2.1% of the size of D while it was 3.3% for evalQP$^-$.

(IV) Size and creation time of indices. The total indices for all access constraints are of 7.7GB, 3.6GB and 9.5GB, accounting for 12.8%, 16.8% and 10.6% of $|D|$. They are smaller than the bound estimated in Section 7, since many constraints use attributes with small domains. 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 ChkCov, QPlan, minA, minA$_{\mathsf{DAG}}$ and minA$_{\mathsf{E}}$ on queries and access schemas for each of AIRCA, TFACC and MCBM. We found that ChkCov, QPlan, minA, minA$_{\mathsf{DAG}}$ and minA$_{\mathsf{E}}$ 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, evalQP answers queries in 5.9 seconds by accessing at most 0.00017% of the data on average, while evalDBMS is unable to find answers within 3000 seconds even on a dataset of 3.75 GB, with even more indices than evalQP can use. The performance gap between evalQP and 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 workload. While an approach is outlined in Section 7, the topic needs a full treatment. Another topic is to develop algorithms that, when a query is not boundedly evaluable, compute its approximate answers with provable accuracy bound, by accessing only a small fraction of data.

Acknowledgments. Fan and Cao are supported in part by 973 Program 2014CB340302 and 2012CB316200, NSFC 61421003 and 61133002, ERC 652976, EPSRC EP/J015377/1 and EP/M025268/1, NSF III 1302212, Shenzhen Peacock Program 1105100030834361, Guangdong Innovative Research Team Program 2011D005. Shenzhen Science and Technology Fund JCYJ20150529164656096 and Guangdong Applied R&D Program 2015B010131006.

10. REFERENCES

[1] *http://data.gov.uk/dataset/naptan*.

[2] *http://data.gov.uk/dataset/road-accidents-safety-data*.

[3] *www.transtats.bts.gov/DatabaseInfo.asp?DB_ID=110*.

[4] *www.transtats.bts.gov/DatabaseInfo.asp?DB_ID=120*.

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

[6] M. Armbrust, K. Curtis, T. Kraska, A. Fox, M. J. Franklin, and D. A. Patterson. PIQL: Success-tolerant query processing in the cloud. *PVLDB*, 5(3), 2011.

[7] M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh. SCADS: Scale-independent storage for social computing applications. In *CIDR*, 2009.

[8] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation,Combinatorial optimization problems and their approximability properties*. Springer, 1999.

[9] G. Ausiello, P. G. Franciosa, and D. Frigioni. Directed hypergraphs: Problems, algorithmic results, and a novel decremental approach. In *ICTCS*, 2001.

[10] M. Benedikt, P. Bourhis, and C. Ley. Analysis of schemas with access restrictions. *TODS*, 40(1):5, 2015.

[11] M. Benedikt, J. Leblay, and E. Tsamoura. Querying with access patterns and integrity constraints. *PVLDB*, 8(6), 2015.

[12] A. Calì and D. Martinenghi. Querying data under access limitations. In *ICDE*, 2008.

[13] Y. Cao, W. Fan, J. Huai, and R. Huang. Making pattern queries bounded in big graphs. In *ICDE*, 2015.

[14] Y. Cao, W. Fan, and W. Yu. Bounded conjunctive queries. *PVLDB*, 2014.

[15] M. Charikar, C. Chekuri, T. Cheung, Z. Dai, A. Goel, S. Guha, and M. Li. Approximation algorithms for directed steiner problems. In *SODA*, 1998.

[16] Facebook, 2013. *http://newsroom.fb.com*.

[17] Facebook. Introducing graph search. *https://en-gb.facebook.com/about/graphsearch*, 2013.

[18] W. Fan, F. Geerts, Y. Cao, T. Deng, and P. Lu. Querying big data by accessing small data. In *PODS*, 2015.

[19] W. Fan, F. Geerts, and L. Libkin. On scale independence for querying big data. In *PODS*, 2014.

[20] A. V. Gelder and R. W. Topor. Safety and translation of relational calculus queries. *TODS*, 16(2), 1991.

[21] I. Grujic, S. Bogdanovic-Dinic, and L. Stoimenov. Collecting and analyzing data from e-government facebook pages. In *ICT Innovations*, 2014.

[22] A. Gubichev and T. Neumann. Exploiting the query structure for efficient join ordering in SPARQL queries. In *EDBT*, 2014.

[23] Y. Huhtala, J. Kärkk ainen, P. Porkka, and H. Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *Comput. J.*, 42(2):100–111, 1999.

[24] I. Ileana, B. Cautis, A. Deutsch, and Y. Katsis. Complete yet practical search for minimal query reformulations under constraints. In *SIGMOD*, pages 1015–1026, 2014.

[25] C. Li. Computing complete answers to queries in the presence of limited access patterns. *VLDB J.*, 12(3), 2003.

[26] J. Liu, J. Li, C. Liu, and Y. Chen. Discover dependencies from data - A review. *TKDE*, 24(2), 2012.

[27] G. Moerkotte, P. Fender, and M. Eich. On the correct and complete enumeration of the core search space. In *SIGMOD*, 2013.

[28] A. Nash and B. Ludäscher. Processing first-order queries under limited access patterns. In *PODS*, 2004.

[29] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[30] L. Popa. Object/relational query optimization with chase and backchase. *IRCS Technical Reports Series*, 2001.

[31] R. Ramakrishnan and J. Gehrke. *Database management systems*. McGraw Hill, 2000.

[32] A. P. Stolboushkin and M. A. Taitslin. Finite queries do not have effective syntax. In *PODS*, 1995.

[33] R. R. Stoll. Set theory and logic. *W. H. Freeman and Co., San Francisco, Calif.-London*, 1961.

[34] J. D. Ullman. *Principles of Database Systems, 2nd Edition*. Computer Science Press, 1982.

[35] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*. Computer Science Press, 1988.

APPENDIX A: Formal Definitions

Query plans (Section 2)

We define a *query plan* ξ *for* Q *under* $\mathcal{A}$ as a sequence:

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

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

- $\{a\}$, where a is a constant in Q; or
- $\mathsf{fetch}(X \in T_j, R, Y)$, where $j < i$, and T_j has attributes X; for each $\bar{a} \in T_j(D)$, it retrieves $D_{XY}(X = \bar{a})$ from D, and returns $\bigcup_{\bar{a} \in T_j(D)} D_{XY}(X = \bar{a})$; or
- $\pi_Y(T_j)$ 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
- $T_i \times T_k$, $T_j \cup T_k$ or $T_j \setminus T_k$, for $j < i$ and $k < i$.

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

Canonical bounded query plans (Section 5.1).

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

(i) if $A_i \in X_C^{Q_s}$, then $\xi_F^c(A_i)$ is $\{c\}$, where $\sigma_{A_i=c}$ is in Q_s;

(ii) if $\sigma_{A_i=A'}$ is in Q_s and there exists a unit fetching plan $\xi_F^c(A')$ for A', then $\xi_F^c(A_i) = \xi_F^c(A')$; and

(iii) if there exists a constraint $R(W \to U, N)$ in $\mathcal{A}$ 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 $\xi_F^c(w_i)$ for w_i, then $\xi_F^c(A_i)$ is:

- $T_1 = \xi_F^c(w_1), \ldots, T_m = \xi_F^c(w_m)$,
- $T_{m+1} = T_1 \times \cdots \times T_m$,
- $T_{m+2} = \mathsf{fetch}(X \in T_{m+1}, R, U)$,
- $T_{m+3} = \pi_{A_i}(T_{m+2})$.

Indexing plan ξ_I^c. An indexing plan ξ_I^c is a sequence of *unit indexing plans* $\xi_I^c(S_1), \ldots, \xi_I^c(S_m)$ for all relations $S_1, \ldots, S_m$ in Q. For each S_i, let Q_s be the max SPC sub-query in which S_i occurs, $X_{Q_s}^{S_i} = \{A_1, \ldots, A_K\}$ be the set of attributes of S_i that also occur in X_{Q_s}, and $S_i(X \to Y, N)$ be a constraint in $\mathcal{A}$ that indexes S_i. Then $\xi_I^c(S_i)$ is as follows:

- $T_1 = \xi_F^c(A_1), \ldots, T_K = \xi_F^c(A_K)$,
- $T_{K+1} = T_1 \times \cdots \times T_K$,
- $T_{K+2} = \pi_{S_i[X]}(T_{K+1})$,
- $T_{K+3} = \mathsf{fetch}(X \in T_{K+2}, S_i, Y)$, and
- $T_{K+4} = T_{K+1} \cap T_{K+3}$ (expressed in terms of $\times$, σ, π).

That is, ξ_I^c ensures that each S_i in Q is indexed.

$\langle Q, \mathcal{A}\rangle$-hypergraph $\mathcal{G}_{Q,\mathcal{A}}$ (Section 5.2)

Given an RA query Q and an access schema $\mathcal{A}$, we use a hypergraph to encode the induced FDs for all max SPC sub-queries of Q. Let $\Sigma_{Q,\mathcal{A}}$ be the union of $\Sigma_{Q_s,\mathcal{A}}$ (the set of induced FDs for Q_s and $\mathcal{A}$) for all max SPC sub-queries Q_s in Q. We assume *w.l.o.g.* that for any two max SPC sub-queries Q_s and $Q_{s'}$ of Q, $\Sigma_{Q_s,\mathcal{A}} \cap \Sigma_{Q_{s'},\mathcal{A}} = \emptyset$. A $\langle Q, \mathcal{A}\rangle$-*hypergraph* (or simply hypergraph) $\mathcal{G}_{Q,\mathcal{A}}$ for Q and $\mathcal{A}$ is a directed hypergraph (V, E) derived from $\Sigma_{Q,\mathcal{A}}$ as follows.

(1) For each induced FD $X \to Y$ in $\Sigma_{Q,\mathcal{A}}$, with $X = \{x_1, \ldots, x_p\} (p \geq 1)$ and $Y \setminus X = \{y_1, \ldots, y_q\} (q \geq 1)$, there are $p + q + 1$ nodes $u_{x_1}, \ldots, u_{x_p}, u_{y_1}, \ldots, u_{y_q}$ and u_Y in V to encode $x_1, \ldots, x_p, y_1, \ldots y_q$ and the set Y, respectively, and there exist $q + 1$ hyperedges $e_1 = (\{u_{x_1}, \ldots, u_{x_p}\}, u_Y)$, $e_2 = (\{u_Y\}, u_{y_1})$, $\ldots$, $e_{q+1} = (\{u_Y\}, u_{y_q})$ in E.

(2) There is a dummy node r such that for each induced FD $\emptyset \to Y$ in $\Sigma_{Q,\mathcal{A}}$ with $Y = \{y_1, \ldots, y_q\}$ $(q \geq 1)$, there exist $q + 1$ nodes u_Y, u_{y_1}, $\ldots$, u_{y_q} in V and $q + 1$ hyperedges $(\{r\}, u_Y)$, $(\{u_Y\}, u_{y_1})$, $\ldots$, $(\{u_Y\}, u_{y_q})$ in E.

(3) For each attribute A in $\hat{X}_C^Q = \{\rho_U(A) \mid A \in X_C^{Q_s}$, Q_s is a max SPC sub-query of $Q\}$, there exist a node u_A in V and a hyperedge $(\{r\}, u_A)$ in E.

Weighted $\langle Q, \mathcal{A}\rangle$-hypergraph (Section 6.2).

For an RA query Q and an access schema $\mathcal{A}$, the *weighted* $\langle Q, \mathcal{A}\rangle$-*hypergraph* is a pair $(\mathcal{G}_{Q,\mathcal{A}}, w(\cdot))$, where $\mathcal{G}_{Q,\mathcal{A}} = (V, E)$ is the $\langle Q, \mathcal{A}\rangle$-hypergraph for Q and $\mathcal{A}$, and $w(\cdot) : E \to \mathbb{N}^+$ assigns an natural number $w(e)$ to each hyperedge e in E. More specifically, $w(\cdot)$ is defined as follows. Recall the definition of $\mathcal{G}_{Q,\mathcal{A}}$ given above. For each induced FD $X \to Y$ in $\Sigma_{Q,\mathcal{A}}$ with $X = \{x_1, \ldots, x_p\}$ and $Y \setminus X = \{y_1, \ldots, y_q\}$, suppose that $X \to Y$ is derived from $R(X \to Y, N)$ in $\mathcal{A}$. Then

(i) $w(e_1) = N$, where e_1 is the hyperedge $(\{u_{x_1}, \ldots, u_{x_p}\}, u_Y)$ in $\mathcal{G}_{Q,\mathcal{A}}$ *w.r.t.* $X \to Y$;

(ii) $w(e_2) = \ldots = w(e_{q+1}) = 0$, for $e_2 = (\{u_Y\}, u_{y_1})$, $\ldots$, $e_{q+1} = (\{u_Y\}, u_{y_q})$; and

(iii) $w(\{r\}, u_A) = 0$ for all hyperedges emanating from the dummy node r of $\mathcal{G}_{Q,\mathcal{A}}$.

APPENDIX B: Proofs

Proof sketch of Theorem 2(1)

It suffices to show that for any boundedly evaluable query plan ξ under $\mathcal{A}$, there exists a covered RA query Q_ξ such that $Q_\xi \equiv_{\mathcal{A}} \xi$, *i.e.,* $Q_\xi(D) = \xi(D)$ for any $D \models \mathcal{A}$. We show this in two steps. We first rewrite ξ into an equivalent RA query Q' by replacing every $\mathsf{fetch}(X \in T_j, R, Y)$ with $\pi_{S[XY]}\sigma_{Y_{Q_j}=S[X]}(Q_j \times R(X, Y, Z))$, where Q_j is the rewriting of the first j operations $T_1 = \delta_1, \ldots, T_j = \delta_j$ of ξ, and Y_{Q_j} is the set of attributes of the output relation of Q_j. We then show by induction on the length of Q_ξ that Q_ξ can be transformed into an equivalent Q'_ξ that is covered by $\mathcal{A}$, by equivalence-preserving rewriting laws of set algebra [33]. □

Proof of Lemma 4

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

(1) $X_{Q_s} \subseteq \mathsf{cov}(Q_s, \mathcal{A})$ iff $\rho_U(X_{Q_s}) \subseteq \rho_U(\mathsf{cov}(Q_s, \mathcal{A}))$; and

(2) $\rho_U(\mathsf{cov}(Q_s, \mathcal{A})) = (\rho_U(X_C^{Q_s}))^*$ (recall $\rho_U(X) = \hat{X}$).

Here (1) can be prove by contradiction based on the definitions. We prove (2) below. We first define a chasing procedure that computes $\mathsf{cov}(Q_s, \mathcal{A})$ for any SPC query Q_s under $\mathcal{A}$. Based on it we then show $\rho_U(\mathsf{cov}(Q_s, \mathcal{A})) = (\rho_U(X_C^{Q_s}))^*$.

A chasing sequence of $\mathsf{cov}(Q_s, \mathcal{A})$ for Q_s is defined as

$$\mathsf{cov}(Q_s, \mathcal{A}) = \mathsf{cov}_0(Q_s, \mathcal{A}), \ldots, \mathsf{cov}_n(Q_s, \mathcal{A}),$$

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

Proof of Lemma 6

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

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

$\boxed{\Leftarrow}$ This direction can be readily verified by induction on the length of $\xi_F^c(A)$, by the definitions of unit fetching plans and fetchable queries, similar to above. □

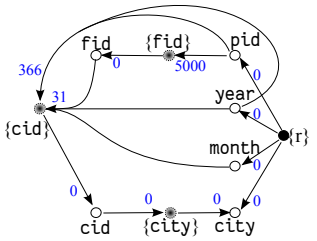


Figure 7: Weighted $\mathcal{G}_{Q_1,\mathcal{A}_1}$ for Q_1 and $\mathcal{A}_1$

Proof of Lemma 7

This is verified by giving translation algorithms Γ_ξ from $\xi^c(A)$ to $(\{r\}, u_{\rho_U(A)})$, and Γ_r from $(\{r\}, u_{\rho_U(A)})$ to $\xi^c(A)$. Below we outline Γ_r, which will be used later in our algorithm. Given a hyperpath $\Pi_{\{r\},u_A}$ from $\{r\}$ to u_A, Γ_r inductively generates fetching plans as follows: (a) if $\Pi_{\{r\},u_A}$ is a hyperedge $(\{r\}, u_A)$ constructed in case (3) of $\langle Q, \mathcal{A}\rangle$-hypergraph above, then return $T_1 = \{\mathrm{c}\}$; (b) if $\Pi_{\{r\},u_A}$ is a hyperedge $(\{r\}, u_Y)$ constructed in case (2) for induced FD $\emptyset \to Y$, then return $T_1 = \xi^c_F(A')$; and (c) if the last hyperedge of $\Pi_{\{r\},u_A}$ is a hyperedge (V_Y, u_A) constructed in case (1) of $\langle Q, \mathcal{A}\rangle$-hypergraph, and if for each u_{B_i} in $V_S = \{u_{Y_1}, \ldots, u_{Y_p}\}$, the unit fetching plan translated from hyperpath $\Pi_{\{r\},u_Y}$ is $\xi^c_F(Y_i)$, then return $T_1 = \xi^c_F(Y_1), \ldots, T_p = \xi^c_F(Y_p)$, $T_{p+1} = T_1 \times \cdots \times T_p$, $T_{p+2} = \mathsf{fetch}(X \in T_{p+1}, R, Y)$, and $T_{p+3} = \pi_A(T_{p+3})$. □

Proof of Theorem 9

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

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

(3) We show that $\mathsf{dAMP}(Q, \mathcal{A}, K)$ 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 $\mathsf{minA_E}$ maps feasible solutions to dminSAP with approximation ratio c to feasible solutions to oAMP with approximation ratio $c+1$ in the elementary case. This can be verified along the same lines as the performance bound analysis of $\mathsf{minA_{DAG}}$ outlined in Section 6.2, and the analysis of approximation bound of algorithm $\mathsf{minA_{DAG}}$ in Appendix D below. □

APPENDIX C: Examples

Example 6 (Section 4). Given Q_0 and $\mathcal{A}_0$ 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 $\mathcal{A}_0$ while Q_2 is not. It first computes the set $\Sigma_{Q_1,\mathcal{A}_0}$ of induced FDs from Q_1 and $\mathcal{A}_0$ (see Example 5), with $\hat{X}_{Q_1} = \{\mathsf{pid}, \mathsf{fid}, \mathsf{cid}, \mathsf{year}, \mathsf{month}, \mathsf{city}\}$ and $\hat{X}^{Q_1}_C = \{\mathsf{pid}, \mathsf{year}, \mathsf{month}, \mathsf{city}\}$. It verifies that Q_1 is covered by $\mathcal{A}_0$ since $\Sigma_{Q_1,\mathcal{A}_0} \models \hat{X}^{Q_1}_C \to \hat{X}_{Q_1}$, and Q_1 is indexed by $\mathcal{A}$ (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 $\mathcal{A}_0$, and thus so is Q'_0. □

Example 7 (Section 5). For Q'_0 and $\mathcal{A}_0$ of Example 1, its $\langle Q, \mathcal{A}\rangle$-hypergraph $\mathcal{G}_{Q'_0,\mathcal{A}_0}$ is depicted in Fig. 2, after the following conversions. We write $Q'_0 = Q_1 - Q_3$ ($Q_3 = Q_1(\mathsf{cid}) \bowtie_{\mathsf{cid}=\mathsf{cid}'} Q_2(\mathsf{cid}')$) 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 $Q_1(\mathsf{cid})$ of Q_3 to S' (*e.g.,* dine of $Q_1(\mathsf{cid})$ in Q_3 is renamed to dine'), and (b) each relation S in sub-query $Q_2(\mathsf{cid}')$ of Q_3 to S'' (*e.g.,* dine of $Q_2(\mathsf{cid})$ in Q_3 to dine''). In Fig. 2, we extend the unification function ρ_U given in Example 5. (a) For each attribute $S'[A]$ that occurs in sub-query $Q_1(\mathsf{cid}')$ of Q_3, if $\rho_U(S[A]) = A$ in Q_1, then $\rho_U(S'[A]) = A'$ for $Q_1(\mathsf{cid}')$ in Q_3. (b) For sub-query $Q_2(\mathsf{cid})$ of Q_3, $\rho_U(\mathsf{dine}''[\mathsf{pid}]) = \mathsf{pid}''$, $\rho_U(\mathsf{dine}''[\mathsf{cid}]) = \mathsf{cid}'$, $\rho_U(\mathsf{dine}''[\mathsf{month}]) = \mathsf{month}''$ and $\rho_U(\mathsf{dine}''[\mathsf{year}]) = \mathsf{year}''$. □

Example 8 (Section 5). A complete fetching plan for Q'_0 under $\mathcal{A}_0$ is as follows.

$T_1 = \{p_0\}$ $(\xi^c_F(\mathsf{pid}))$,
$T_2 = \mathsf{fetch}(X \in T_1, \mathsf{friend}, \mathsf{fid})$;
$T_3 = \pi_{\mathsf{fid}}(T_2)$ $(\xi^c_F(\mathsf{fid}))$;
$T_4 = \{2015\}$ $(\xi^c_F(\mathsf{year}))$;
$T_5 = \{\mathrm{MAY}\}$ $(\xi^c_F(\mathsf{month}))$;
$T_6 = T_3 \times T_4$;
$T_7 = T_6 \times T_5$;
$T_8 = \mathsf{fetch}(X \in T_7, \mathsf{dine}, \mathsf{cid})$;
$T_9 = \pi_{\mathsf{cid}}(T_8)$ $(\xi^c_F(\mathsf{cid}))$;
$T_{10} = \{\mathrm{NYC}\}$ $(\xi^c_F(\mathsf{city}))$;
$T_{11}, \ldots, T_{20}$; $T_{21} = \{p_0\}$ $(\xi^c_F(\mathsf{pid}''))$.

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 $\xi^c_I(\mathsf{dine}'')$ for relation dine'' is:

$T^I_1 = T_{19}$;
$T^I_2 = T_{20}$;
$T^I_3 = T_{19} \times T_{20}$;
$T^I_4 = \mathsf{fetch}(X \in T^I_3, \mathsf{dine}, (\mathsf{pid}, \mathsf{cid}))$;

similar for other relations. Finally, an evaluation plan for Q'_0 under $\mathcal{A}_0$ is exactly Q'_0 with each relation name S replaced by the T_S with $T_S = \xi^c_I(S)$. □

Example 9 (Section 6). For $\mathsf{AMP}(Q_1, \mathcal{A}_1)$, algorithm minA works as follows. It first finds that either ψ_2 and ψ_3, or ψ_3 and ψ_5 can be removed from $\mathcal{A}_1$ while keeping Q_1 covered. It then calculates $w(\psi_2) = \frac{c_1 \cdot 31}{c_2 \cdot 1}$ and $w(\psi_5) = \frac{c_1 \cdot 366}{c_2 \cdot 1}$. Suppose that $c_1 = c_2 = 1$. Then minA greedily picks ψ_5 instead of ψ_2. It finds that no more constraints can be removed while keeping Q_1 covered. Thus minA returns $\mathcal{A}_m = \{\psi_1\ \psi_2, \psi_4\}$. □

Example 10 (Section 6). The complete weighted $\langle Q, \mathcal{A}\rangle$-hypergraph $\mathcal{G}_{Q_1,\mathcal{A}_1}$ for Q_1 and $\mathcal{A}_1$ is shown in Fig. 7. □

APPENDIX D: Algorithm Analysis

Complexity of algorithm CovChk (Section 4)

We next show that CovChk can be implemented in $O(|Q|^2 + |\mathcal{A}|)$ time. (1) It takes $O(|Q|)$ time to compute the set $\mathcal{S}_Q$

of all max SPC sub-queries of Q. (2) Checking whether all Q_s's in $\mathcal{S}_Q$ are indexed by $\mathcal{A}$ can be implemented in $O(|Q|+|\mathcal{A}|)$ time, by building an index from relations in Q to constraints of $\mathcal{A}$ in $O(|Q|+|\mathcal{A}|)$ time before the iteration. (3) It takes $O(|Q_s|^2)$ time to construct induced FDs $\Sigma_{Q_s,\mathcal{A}}$, and the size of $\Sigma_{Q_s,\mathcal{A}}$ is bounded by $|\mathcal{A}_{Q_s}|$ for each Q_s. (4) FD implication checking can be done in linear time (cf. [5]), *i.e.,* $O(|\Sigma_{Q_s,\mathcal{A}}|+|X_C^{Q_s}|+|X_{Q_s}|) = O(|\mathcal{A}_{Q_s}|+|Q_s|)$ for each Q_s. Putting these together, CovChk is in $O(|Q|^2+|\mathcal{A}|)$ time.

Complexity of Algorithm QPlan (Section 5)

Algorithm QPlan can be implemented in $O(|Q|(|Q|+|\mathcal{A}|))$ time. Indeed, (1) constructing the $\langle Q,\mathcal{A}\rangle$-hypergraph $G_{Q,\mathcal{A}}$ takes $O(|Q|+|\mathcal{A}|)$ time. (2) In each iteration (lines 3-6), findHP takes $O(|Q|+|\mathcal{A}|)$ time to find hyperpath $\Pi_{\{r\},u_{\hat{A}}}$ (cf. [9]), and transQP takes $O(|\Pi_{\{r\},u_{\hat{A}}}|) = O(\|\mathcal{A}\|)$ time to translate P into a unit fetching plan, where $\|\mathcal{A}\|$ denotes the cardinality of $\mathcal{A}$. 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 $O(|Q|+|\mathcal{A}|) + O(|Q|(|Q|+|\mathcal{A}|+\|\mathcal{A}\|)) + O(|Q|) + O(|Q|) = O(|Q|(|Q|+|\mathcal{A}|))$ time in total.

Algorithm transQP (Section 5.2)

It is the translation algorithm Γ_r given in the proof of Lemma 7 in Appendix B.

Complexity of Algorithm minA (Section 6.2)

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

Approximation bound of Algorithm $\mathsf{minA}_{\mathsf{DAG}}$ (Section 6.2).

Let $c(\mathcal{A})$ denote the sum of the N's in all the constraints in $\mathcal{A}$. Let $\mathcal{A}'_I$ be the set of constraints in $\mathcal{A}'$ indexing a relation, and $\mathcal{A}'_{ni}$ be all the other constraints. Let $\mathcal{A}^{OPT}$ be the optimal solution to $\mathsf{AMS}(Q,\mathcal{A})$. We define $\mathcal{A}_I^{OPT}$ and $\mathcal{A}_{ni}^{OPT}$ analogous to $\mathcal{A}'_I$ and $\mathcal{A}'_{ni}$, respectively. One can verify that $\frac{c(\mathcal{A}')}{c(\mathcal{A}^{OPT})} = \frac{c(\mathcal{A}'_{ni})+c(\mathcal{A}'_I)}{c(\mathcal{A}_{ni}^{OPT})+c(\mathcal{A}_I^{OPT})} \leq \frac{k*c(\mathcal{A}_{ni}^{OPT})+c(\mathcal{A}'_I)}{c(\mathcal{A}_{ni}^{OPT})+c(\mathcal{A}_I^{OPT})} \leq k+1$, where $k = |\rho_U(X_Q)\setminus\rho_U(X)| \leq |X_Q \setminus X_C^Q|$.