Bounded Conjunctive Queries

Yang Cao[1,2] Wenfei Fan[1,2] Tianyu Wo[2] Wenyuan Yu[3]
[1]University of Edinburgh [2]RCBD and SKLSDE Lab, Beihang University [3]Facebook Inc.
{Y.Cao-17@sms, wenfei@inf}.ed.ac.uk woty@act.buaa.edu.cn wyu@fb.com

ABSTRACT

A query Q is said to be *effectively bounded* if for all datasets D, there exists a subset D_Q of D such that $Q(D) = Q(D_Q)$, and the size of D_Q and time for fetching D_Q are *independent of* the size of D. The need for studying such queries is evident, since it allows us to compute $Q(D)$ by accessing a bounded dataset D_Q, *regardless of* how big D is. This paper investigates effectively bounded conjunctive queries (SPC) under an access schema $\mathcal{A}$, which specifies indices and cardinality constraints commonly used. We provide characterizations (sufficient and necessary conditions) for determining whether an SPC query Q is effectively bounded under $\mathcal{A}$. We study several problems for deciding whether Q is bounded, and if not, for identifying a minimum set of parameters of Q to instantiate and make Q bounded. We show that these problems range from quadratic-time to NP-complete, and develop efficient (heuristic) algorithms for them. We also provide an algorithm that, given an effectively bounded SPC query Q and an access schema $\mathcal{A}$, generates a query plan for evaluating Q by accessing a bounded amount of data in any (possibly big) dataset. We experimentally verify that our algorithms substantially reduce the cost of query evaluation.

1. INTRODUCTION

Query answering is expensive. Consider the problem to decide, given a query Q, a dataset D and a tuple t, whether $t \in Q(D)$, *i.e.,* whether t is an answer to Q in D. This problem is NP-complete for conjunctive queries (*i.e.,* SPC, defined with selection, projection and Cartesian product operators); and it is PSPACE complete for queries in relational algebra ($\mathcal{RA}$, cf. [6]). When D is big, computing $Q(D)$ is cost-prohibitive. Indeed, even a linear-time query processing algorithm may take days on a dataset D of PB size (10^{15} bytes), and years when D is of EB size (10^{18} bytes) [21].

This motivates us to ask the following question: is it possible to compute $Q(D)$ by only accessing (visiting and fetching) a small subset D_Q of D? More specifically, we want to know whether a query Q has the following properties. For *all* datasets D, there exists a subset $D_Q \subset D$ such that

(a) $Q(D_Q) = Q(D)$,

(b) D_Q consists of no more than M tuples, and

(c) D_Q can be effectively identified by using access information, with a cost *independent of* $|D|$.

Here access information includes indices and cardinality constraints, specified as an access schema $\mathcal{A}$; and M is a bound determined by $\mathcal{A}$ and Q only. We say that Q is *effectively bounded under* $\mathcal{A}$ if it satisfies all the three conditions above, and *bounded* if it satisfies conditions (a) and (b) only.

If Q is effectively bounded, then we can find a bounded dataset D_Q and compute $Q(D)$ by using D_Q, *independent of* the size of possibly big D. Moreover, when D grows, the performance does not degrade. In other words, we can *reduce* big D to a "small" D_Q of a manageable size.

Many real-life queries are actually (effectively) bounded.

Example 1: Social networks, *e.g.,* Facebook, allow us to tag a photo and show who is in it. Such a tag is a link to the person "tagged". Consider the following.

(1) A query Q_0 is to find all photos from an album a_0 in which a person u_0 is tagged by one of her friends. The relations needed for answering Q_0 include the following:

- in_album(photo_id, album_id) for photo albums,
- friends(user_id, friend_id) for friends, and
- tagging(photo_id, tagger_id, taggee_id), indicating that taggee_id is tagged by tagger_id in photo_id.

We abbreviate these as in_album(pid_1, aid), friends(uid, fid) and tagging(pid_2, tid_1, tid_2), respectively.

Given these, Q_0 can be written as an SPC query as follows:

$$Q_0(\mathsf{pid}_1) = \pi_{\mathsf{pid}_1}\sigma_C\big(\mathsf{in\_album}(\mathsf{pid}_1,\mathsf{aid}) \times \mathsf{friends}(\mathsf{uid},\mathsf{fid}) \times \mathsf{tagging}(\mathsf{pid}_2,\mathsf{tid}_1,\mathsf{tid}_2)\big),$$

where the selection condition C is given as

$$\mathsf{aid} = a_0 \wedge \mathsf{uid} = u_0 \wedge \mathsf{pid}_1 = \mathsf{pid}_2 \wedge \mathsf{tid}_1 = \mathsf{fid} \wedge \mathsf{tid}_2 = \mathsf{uid}.$$

Observe the following. (a) A dataset D_0 consisting of these relations is possibly big; for instance, Facebook has more than 1 billion users with 140 billion friend links [18]. (b) Query Q_0 is *not* bounded: we can add new photos to album a_0, new friends of u_0 to friend, or new tuples to tagging, and Q_0 has to check these tuples when D_0 grows.

However, social networks often impose limits (cardinality constraints) on D_0, *e.g.,* (a) each album includes at most 1000 photos, (b) each person may claim up to 5000 friends, and (c) each person in a photo can only be tagged once [19].

Proceedings of the VLDB Endowment, Vol. 7, No. 11

Moreover, indices can be built on in_album(aid), friends(uid), and tagging($\mathsf{pid}_1, \mathsf{tid}_2$). As will be seen later, these indices and constraints make an *access schema* $\mathcal{A}_0$.

Under access schema $\mathcal{A}_0$, Q_0 is *effectively bounded*: we can compute $Q_0(D_0)$ by accessing at most 7000 tuples *no matter how large* D_0 *is*, as follows: (a) select a set T_1 of at most 1000 pid's from in_album with aid $= a_0$, by using the index on in_album(aid); (b) get a set T_2 of at most 5000 fid's from friends with user_id $= u_0$, using the index on friends(uid); (c) using $\mathsf{tid}_2 = u_0$ and pid_2's from T_1, fetch a set T_3 of at most 1000 ($\mathsf{pid}_2, \mathsf{tid}_1$) tuples from tagging via the index on tagging($\mathsf{pid}_2, \mathsf{tid}_2$); and (d) compute a join T_4 of T_2 and T_3. Then $Q_0(D_0) = \pi_{\mathsf{photo\_id}}(T_4)$. This query plan visits at most 7000 tuples in total. Moreover, these tuples can be efficiently identified and retrieved by using the indices.

(2) Queries like Q_0 are routinely posed on social networks. Thus we want a query Q_1, which is the same as Q_0 except that uid and aid are not constants, *i.e.,* values u_0 and a_0 are not given. Query Q_1 is *not* bounded even under $\mathcal{A}_0$.

However, Q_1 can be taken as a *parameterized query*, a template with parameters (uid, aid, fid, pid_2, tid_1, tid_2) such that some of them can be substituted with constants when Q_1 is executed. We identify a *minimum* subset X_P of parameters of Q_1, referred to as *dominating parameters*, such that when values of X_P are given, Q_1 is effectively bounded under $\mathcal{A}_0$. For instance, uid and aid make a set of dominating parameters: as shown above, when they are instantiated, the query on D_0 can be answered by accessing at most 7000 tuples. We can find X_P and suggest it to users for instantiation.

(3) As another example, consider an arbitrary *Boolean* SPC query Q_2 that, given an instance D of a relational schema $\mathcal{R}$, returns true if and only if $Q_2(D)$ is nonempty. It is known that Q_2 is bounded even in the absence of access schema [20]. More specifically, $Q_2(D)$ can be computed by accessing at most $|Q_2|$ amount of data no matter how big D is. Indeed, no matter $Q_2(D)$ is true or false, it needs a *witness* D_Q of size $|Q|$ such that $Q_2(D_Q) = Q_2(D)$. □

The idea of answering queries with a bounded dataset was first explored in [9–11], and was formalized in [20] (referred to as scale independence there). To make practical use of the idea, several questions have to be settled. Given a query Q and an access schema $\mathcal{A}$, can we determine whether Q is (effectively) bounded under $\mathcal{A}$? What is the complexity? If Q is not bounded, can we find a dominating-parameter set X_P of Q such that Q becomes effectively bounded under $\mathcal{A}$ when X_P is instantiated? Given a dataset D, how can we compute $Q(D)$ by efficiently fetching a bounded D_Q, by using access information in $\mathcal{A}$? These questions are *nontrivial*. It is known that it is *undecidable* to decide whether Q is bounded for Boolean $\mathcal{RA}$ queries [20]. The questions are open for SPC queries, which are considered "the most fundamental and the most widely used queries" in practice [23].

Contributions. This paper answers these questions for SPC queries. The main results are as follows.

(1) We formulate bounded SPC queries (Section 2). Following [20], we use an access schema $\mathcal{A}$ to specify indices and cardinality constraints for databases of a relational schema $\mathcal{R}$. We revise the notions of scale independence studied in [20]. We say that an SPC query Q is bounded if for all instances D of $\mathcal{R}$, there exists a $D_Q \subset D$ such that $Q(D) = Q(D_Q)$, and the size of D_Q is *independent of the size of* D. If in addition, D_Q can be efficiently fetched by using $\mathcal{A}$, then Q is effectively bounded. We show that some queries are bounded but are *not* effectively bounded.

(2) We study the problems of determining boundedness and effective boundedness (Section 3). We provide a set of deduction rules to decide whether an SPC query Q is bounded under an access schema $\mathcal{A}$, and show that the rules provide *a sufficient and necessary condition* for the boundedness. We also provide a *characterization* of effectively bounded Q under $\mathcal{A}$. In contrast to $\mathcal{RA}$ queries [20], these results tell us that there are systematic methods to decide whether SPC queries are bounded or effectively bounded under $\mathcal{A}$.

(3) We study several problems in connection with the (effective) boundedness of SPC queries, establish their complexity, and develop algorithms for them (Section 4). Given an SPC query Q and an access schema $\mathcal{A}$, we study problems to decide (a) whether Q is bounded under $\mathcal{A}$, (b) whether Q is effectively bounded under $\mathcal{A}$, (c) if Q is not effectively bounded, whether there exists a set X_P of dominating parameters of Q to make Q effectively bounded under $\mathcal{A}$, and (d) if so, how to find a minimum set X_P? We show that these problems are in $O(|Q|(|\mathcal{A}| + |Q|))$-time, $O(|Q|(|\mathcal{A}| + |Q|))$-time, NP-complete and NPO-complete, respectively. We develop efficient (heuristic) algorithms for these problems.

(4) We give a PTIME (polynomial time) algorithm to generate query plans for answering effectively bounded SPC queries Q under $\mathcal{A}$ (Section 5). The query plans allow us to answer Q in any (possibly big) dataset D by accessing a subset D_Q of D. The evaluation scales with the size of D: the size $|D_Q|$ of D_Q is decided by $\mathcal{A}$ and Q only, and D_Q can be fetched by using indices in $\mathcal{A}$ in time *independent of* $|D|$. We also study the problem for identifying a minimum D_Q, and show that its decision problem is NP-complete.

(5) We experimentally verify the efficiency and effectiveness of our algorithms, using real-life and synthetic data (Section 6). We find that our algorithms are efficient: they take at most 2.1 seconds to decide whether Q is effectively bounded under $\mathcal{A}$, and to generate a query plan for Q, when Q is defined on a relational schema with 19 tables and 113 attributes, and $\mathcal{A}$ consists of 84 constraints. Moreover, our bounded query evaluation approach is effective: on a real-life dataset D of 21.4GB, our query plan only accesses 3800 tuples and gets answers in 9.3 seconds on average, while MySQL takes *longer than 14 hours*. That is, our approach is *3 orders of magnitude* faster than MySQL. The improvement is *more substantial* when D grows, since our approach accesses a bounded subset of D *no matter how large* D *is*!

These results suggest an approach to answering queries in big data D. Given an SPC query Q and an access schema $\mathcal{A}$, we first check in $O(|Q|(|\mathcal{A}| + |Q|))$-time whether Q is effectively bounded under $\mathcal{A}$. If so, we compute $Q(D)$ by accessing a bounded $D_Q \subset D$, *independent of* $|D|$. If not, we may either identify a minimum set of dominating parameters and invite users to supply their values, or suggest users to extend their access schema, such that Q becomes effectively bounded. Only when none of these is possible, we pay the price of computing $Q(D)$ directly in big D.

We find that many real-life queries are effectively bounded under a simple access schema, such as (a) parameterized queries supported by e-commerce systems, where users issue queries via Web forms by instantiating parameters; and (b) social searches, *e.g.,* the one given in Example 1. Moreover, access schema can be deduced from our familiar functional dependencies (FDs), domain constraints and bounds on real-life data such as those imposed by Facebook (Example 1).

Detailed proofs of the results of the paper are given in [5].

Related work. We characterize related work as follows.

Scale independence. The notion of boundedness is a revision of scale independence proposed in [10], which aims to execute a bounded amount of work in an application regardless of the size of the underlying data. An extension to SQL was proposed in [9] to enforce scale independence, which allows users to specify bounds on the amount of data accessed and the size of intermediate results; when the data required exceeds the bounds, only top-k tuples are retrieved to meet the bounds. Scale independence was also studied in the presence of materialized views [11]. The study differs from our work as follows. (1) Its system [9] is based on key/value store with its own compiler, while we aim to directly improve traditional DBMS. (2) It does not consider effective boundedness and its characterizations. (3) It settles with approximate answers while we focus on exact answers.

The notion of scale independence was recently formalized in [20]. The notion of access schema was also proposed there. For a given bound M, [20] defines a scale independent query Q to be one that for all datasets D, there exists $D_Q \subseteq D$ such that $Q(D) = Q(D_Q)$ and $|D_Q| \leq M$. It studies several decision problems for scale independence. In particular, it shows that it is undecidable to check whether a Boolean $\mathcal{RA}$ query is scale independent. It also develops a set of rules as a sufficient condition for deciding whether an $\mathcal{RA}$ query is scale independent under an access schema.

This works extends [20] as follows. (1) We do not require the size of D_Q to be bounded by a predefined M. Indeed, if $|D_Q|$ is determined by $\mathcal{A}$ and Q only, its evaluation scales well with D. Hence, we define (effectively) bounded queries instead. (2) We provide *characterizations* for (effectively) bounded SPC queries Q under $\mathcal{A}$, which was not studied in [20]. As opposed to $\mathcal{RA}$ queries of [20], these give us *sufficient and necessary* conditions for deciding whether Q is (effectively) bounded. (3) We show that the (effective) boundedness of SPC queries can be decided in PTIME when M is not part of the input, but is NP-complete in the setting of [20] (when M is predefined), in contrast to the undecidability of the problem for $\mathcal{RA}$ queries. (4) None of the problems for dominating parameters was studied in [20]. (5) We give efficient (heuristic) algorithms for checking whether Q is (effectively) bounded, identifying dominating parameters, and for generating a query plan when Q is effectively bounded. No algorithms were provided in [20].

There has also been work on size bounds for join [12] and conjunctive queries [23]. Given a query Q and a dataset D, it is to decide bounds for $|Q(D)|$ in terms of $|Q|$ and $|D|$, possibly in the presence of keys and FDs [23]. Characterizations for deriving worst-case size bounds for these queries are presented there. That line of work differs from ours in both the problems studied and the approaches adopted (*e.g.,* coloring scheme of [23] vs. rule-based inference of ours).

Making big data small. There have been several data reduction schemes that, given a dataset D, find a small dataset D' such that one can evaluate queries posed on D by using D' instead. These include compression, summarization and data synopses such as histograms, wavelets, quantile summaries, clustering and sampling [7, 14, 17, 22, 24, 25, 27, 30]. Recently BlinkDB [8] has revised the idea to evaluate queries on big data. It adaptively samples data to find approximate query answers within a probabilistic error-bound and time constraints. Similar ideas were also explored in [9].

This work differs from the prior work as follows. (1) We aim to compute *exact answers* by using a bounded dataset whenever possible, rather than approximate query answers [8,9]. (2) The prior reduction schemes [7,14,17,22,24, 25,27,30] use *the same* dataset D' to answer *all queries* posed on D. In contrast, we adopt a *dynamic reduction scheme* that finds a small D_Q for each query Q. Here D_Q contains only the information needed for answering Q and hence, allows us to compute $Q(D)$ by using a small dataset D_Q.

Access schema. Cardinality constraints have been studied for relational data (*e.g.,* [26]). Following [20], this paper aims to identify a bounded dataset D_Q to answer a query by making use of available indices and cardinality constraints.

As remarked in [20], access schema is quite different from access patterns [15, 16, 28]. Access patterns require that a relation can only be accessed by providing certain combinations of attribute values. In contrast, access schemas combine indexing and cardinality constraints, and guide us to find a bounded dataset D_Q for query answering.

2. BOUNDED QUERIES UNDER AN ACCESS SCHEMA

Below we first review SPC queries, and then present access schemas. Based on these, we define bounded and effectively bounded SPC queries under an access schema.

SPC. Consider a relational schema $\mathcal{R} = (R_1, \ldots, R_l)$ in which each R_i is a relation schema. Recall that an SPC query over $\mathcal{R}$ has the following form (see, *e.g.,* [6]):

$$Q(Z) = \pi_Z \sigma_C (S_1 \times \ldots \times S_n).$$

Here S_j is a (renaming of a) relation schema in $\mathcal{R}$, Z is a set of attributes of $\mathcal{R}$, and C is the *selection condition* of Q, defined as a conjunction of equality atoms $x = y$ or $x = c$. where x, y are attributes and c is a constant. We refer to attributes that appear in Z or C as *the parameters of* Q.

To simplify the discussion, we consider Q defined over a single schema $R(A_1, \ldots, A_m)$. This does not lose generality due to the lemma below, in which we denote by $\mathsf{inst}(\mathcal{R})$ the set of all database instances of relational schema $\mathcal{R}$.

Lemma 1: *For any relational schema $\mathcal{R}$, there exist a single relation schema R, a linear-time function g_D from $\mathsf{inst}(\mathcal{R})$ to $\mathsf{inst}(R)$, and a linear-time query-rewriting function g_Q from* SPC *to* SPC *such that for any instance D of $\mathcal{R}$ and any* SPC *query Q over $\mathcal{R}$, $Q(D) = g_Q(Q)(g_D(D))$.* □

Access schema. An *access schema* $\mathcal{A}$ over relation schema R is a set of *access constraints* of the following form:

$$X \to (Y, N),$$

where X and Y are sets of attributes of R, and N is a natural number. A database D of R *satisfies* the constraint if

- for any X-value $\bar{a}$, $|D_Y(X = \bar{a})| \leq N$, where $D_Y(X = \bar{a}) = \{t[Y] \mid t \in D, t[X] = \bar{a}\}$; that is, for each X value, there exist at most N distinct corresponding Y values;

- there exists an *index on X for Y* such that given a X-value $\bar{a}$, it finds $D' \subseteq D$ such that $|D'| \leq N$ and $D'_Y(X = \bar{a}) = D_Y(X = \bar{a})$ with a cost measured in N.

Here D' is one of (possibly many) subsets of D with N tuples, one for each distinct value of Y, and N is independent of $|D|$. We say that D *satisfies* access schema $\mathcal{A}$, denoted by $D \models \mathcal{A}$, if D satisfies all the constraints in $\mathcal{A}$.

An access constraint is a combination of a cardinality constraint and an index. It tells us that for any given X-value. there exist a bounded number of corresponding Y values, and the Y values can be efficiently retrieved with the index.

Example 2: Recall from Example 1 the limit of 1000 photos per album. This can be expressed as an access constraint over schema $\mathsf{in\_album}$ with an index on $\mathsf{album\_id}$ for $\mathsf{photo\_id}$:

$$\mathsf{album\_id} \rightarrow (\mathsf{photo\_id}, 1000).$$

Another constraint over $\mathsf{tagging}$ enforces that each person is tagged at most once in a photo: $(\mathsf{photo\_id}, \mathsf{taggee\_id}) \rightarrow (\mathsf{tager\_id}, 1)$. Similarly, the limit of 5000 friends per person is expressed as $\mathsf{user\_id} \rightarrow (\mathsf{friend\_id}, 5000)$ over $\mathsf{friends}$. □

Observe the following. (a) Functional dependencies (FDs) $X \rightarrow Y$ (see [6]) are a special case of access constraints of the form $X \rightarrow (Y, 1)$ if an index is defined on X for Y. (b) Keys are a special form of access constraints $X \rightarrow (R, 1)$, where R denotes all the attributes of relation schema R. In general, given an access constraint $X \rightarrow (R, N)$, we can efficiently fetch the entire tuples when an X value is given.

In practice, access constraints can be deduced from the following: (1) FDs; mature techniques are already in place to automatically discover FDs, a special case of access constraints; moreover, the techniques can be extended to discover general access constraints; (2) attributes with bounded domains: if the domain of an attribute B is bounded by N (*e.g.,* each year has 12 months and at most 336 days), then $X \rightarrow (B, N)$ is an access constraint for any set X of attributes; and (3) the semantics of real-life data, *e.g.,* the number of vehicles involved in a road accident is at most 192 from 1979–2005 in the UK (see Section 6 for details).

Bounded and effectively bounded SPC queries. We say that an SPC query Q over relation schema R is *bounded under* an access schema $\mathcal{A}$ if for *all* instances D of R that satisfies $\mathcal{A}$, there exists a subset $D_Q \subseteq D$ such that

(a) $Q(D_Q) = Q(D)$; and
(b) the size $|D_Q|$ is *independent of* the size $|D|$ of D.

Here $|D|$ is measured as the total *number of tuples* in D.

We say that Q is *effectively bounded under* $\mathcal{A}$ if Q is bounded under $\mathcal{A}$ and there exists an algorithm that identifies D_Q in time determined by Q and $\mathcal{A}$, *not by* $|D|$.

Intuitively, Q is bounded under $\mathcal{A}$ if it can be answered in a bounded D_Q. It is effectively bounded if moreover, D_Q can be efficiently identified (assuming that given an X-value $\bar{a}$, it takes $O(N)$ time to identify $D_Y(X = \bar{a})$ in D via an access constraint $X \rightarrow (Y, N)$ in $\mathcal{A}$). For instance, as shown in Example 1, all Boolean SPC queries are bounded even in the absence of access schema, and query Q_0 is effectively bounded under the access schema $\mathcal{A}_0$ of Example 2.

The result below separates the class $\mathsf{SPC_b}$ of bounded queries from the class $\mathsf{SPC_{eb}}$ of effectively bounded queries under the same access schema, *i.e.,* $\mathsf{SPC_{eb}} \subset \mathsf{SPC_b}$.

Proposition 2: *There exists a query that is bounded but is not effectively bounded under the same access schema.* □

3. CHARACTERIZING EFFECTIVE BOUNDEDNESS

We now provide sufficient and necessary conditions for determining the (effective) boundedness of SPC queries Q under an access schema $\mathcal{A}$. The main result of the section is as follows. (1) There exists a set $\mathcal{I}_B$ of deduction rules such that Q is bounded *if and only if* it can be proven from Q and $\mathcal{A}$ using $\mathcal{I}_B$. (2) Similarly, there exists a set $\mathcal{I}_E$ of such rules for effectively boundedness. These yield *characterizations* of (effective) boundedness via symbolic computation. Moreover, they reveal insight into the boundedness analysis, which helps us develop checking algorithms in Section 4.

We give $\mathcal{I}_B$ and $\mathcal{I}_E$ in Sections 3.1 and 3.2, respectively.

3.1 Deduction Rules for Boundedness

Consider an SPC query $Q(Z) = \pi_Z \sigma_C (S_1 \times \ldots \times S_n)$, where S_i is a renaming of relation schema R. We use Σ_Q to denote the set of all equality atoms $S[A] = S'[A']$ or $S[A] = c$ derived from the selection condition C of Q by the transitivity of equality. We use X and X' to denote sets of attributes of Q. We write $\Sigma_Q \vdash X = X'$ if $X = X'$ can be derived from equality atoms in Σ_Q, which can be checked in $O(\max(|X|, |X'|))$ time by leveraging a list of attributes in Q that can be precomputed in $O(|Q|^2)$ time.

To simplify the discussion we assume *w.l.o.g.* that attributes in S_i's have distinct names via renaming; see, *e.g.,* query Q_0 of Example 1. We also assume *w.l.o.g.* that Q is satisfiable, *i.e.,* Σ_Q does not includes $S[A] = c$ and $S[A] = d$ when c and d are distinct constants.

Rules. We present a set $\mathcal{I}_B$ of four deduction rules in Fig. 1. Given an SPC query Q and an access schema $\mathcal{A}$, we write

$$X \mapsto_{\mathcal{I}_B} (Y, N)$$

if $X \rightarrow (Y, N)$ can be deduced from $\mathcal{A}$ and Σ_Q by using the rules in $\mathcal{I}_B$. Here $X \mapsto_{\mathcal{I}_B} (Y, N)$ *extends* access constraints of Section 2 by allowing X and Y to be sets of attributes of Q from possibly multiple renamed relations of R in Q.

One can draw an analogy of $\mathcal{I}_B$ to our familiar Armstrong's Axioms for FD implication (see, *e.g.,* [6]).

(1) Reflexivity, Augmentation and Transitivity are immediate extensions of Armstrong's Axioms to access constraints. In particular, Transitivity allows us to propagate boundedness from one relation to another in a Cartesian product $S_1(X, Y_1) \times S_2(Y_2, W)$: if for any X-value $\bar{a}$, there exist at most N_1 distinct Y_1 values, then so do $S_2[Y_2]$ by $\Sigma_Q \vdash S_1[Y_1] = S_2[Y_2]$. Then from $Y_2 \rightarrow (W, N_2)$, it follows that given $\bar{a}$, there exist at most $N_1 * N_2$ distinct $S_2[W]$ values.

(2) Actualization is an application of some access constraint of $\mathcal{A}$ to a renaming S_i of R that appears in Q.

Example 3: Recall relation schemas $\mathsf{in\_album}(\mathsf{pid}_1, \mathsf{aid})$, $\mathsf{friends}(\mathsf{uid}, \mathsf{fid})$ and $\mathsf{tagging}(\mathsf{pid}_2, \mathsf{tid}_1, \mathsf{tid}_2)$ given in Example 1. Let X_0 be $(\mathsf{aid}, \mathsf{uid}, \mathsf{tid}_2, \mathsf{fid}, \mathsf{tid}_1)$.

We show below how $X_0 \mapsto_{\mathcal{I}_B} (y, N_y)$ is proven from query Q_0 of Example 1 and access schema $\mathcal{A}_0$ of Example 2 by using $\mathcal{I}_B$, for each parameter y in Q_0 (*i.e.,* σ_C or Z) and for some positive integer N_y determined by Q_0 and $\mathcal{A}_0$.

(1) $\mathsf{aid} \mapsto_{\mathcal{I}_B} (\mathsf{pid}_1, 1000)$ Actualization
(2) $\mathsf{pid}_2 \mapsto_{\mathcal{I}_B} (\mathsf{pid}_2, 1)$ Reflexivity

(**Reflexivity**) If $X' \subseteq X$, then $X \mapsto_{\mathcal{I}_B} (X', 1)$.
(**Actualization**) If $X \rightarrow (Y, N)$ is in $\mathcal{A}$, then $S_i[X] \mapsto_{\mathcal{I}_B} (S_i[Y], N)$ for each i in $[1, n]$.
(**Augmentation**) If $X \mapsto_{\mathcal{I}_B} (Y, N)$, then $X \cup W \mapsto_{\mathcal{I}_B} (Y \cup W, N)$.
(**Transitivity**) If $X \mapsto_{\mathcal{I}_B} (Y_1, N_1)$, $Y_2 \mapsto_{\mathcal{I}_B} (W, N_2)$, and $\Sigma_Q \vdash Y_1 = Y_2$, then $X \mapsto_{\mathcal{I}_B} (W, N_1 * N_2)$.

Figure 1: Deduction rules $\mathcal{I}_B$ for boundedness

(3) $\Sigma_{Q_0} \vdash \mathsf{pid1} = \mathsf{pid2}$	selection condition in Q_0
(3) $\mathsf{aid} \mapsto_{\mathcal{I}_B} (\mathsf{pid}_2, 1000)$	by (1), (2), (3) and Transitivity
(4) $X_0 \mapsto_{\mathcal{I}_B} (\mathsf{aid}, 1)$	Reflexivity
(5) $X_0 \mapsto_{\mathcal{I}_B} (\mathsf{pid}_2, 1000)$	(3)(2) and Transitivity

Similarly, $X_0 \mapsto_{\mathcal{I}_B} (\mathsf{tid}_1, 1)$, $X_0 \mapsto_{\mathcal{I}_B} (\mathsf{tid}_2, 1)$, $X_0 \mapsto_{\mathcal{I}_B} (\mathsf{uid}, 1)$ and $X_0 \mapsto_{\mathcal{I}_B} (\mathsf{fid}, 1)$ by Reflexivity. □

Characterization. We next show that $\mathcal{I}_B$ provides *a sufficient and necessary condition* for determining whether an SPC query $Q(Z)$ is bounded under an access schema $\mathcal{A}$.

We use the following notations: (a) X_B is the set of all parameters of Q that appear in the selection condition σ_C such that for any $S[A] \in X_B$ and any $z \in Z$, $\Sigma_Q \nvdash S[A] = z$, *i.e.,* attributes that involve in Boolean condition checking but are not part of the output; and (b) X_C is the set of all attributes such that for all $S[A] \in X_C$, $\Sigma_Q \vdash S[A] = c$ for some constant c, *i.e.,* already instantiated with constants.

Theorem 3: *An* SPC *query $Q(Z)$ is bounded under an access schema $\mathcal{A}$ if and only if for each parameter y in $X_B \cup Z$, $X_B \cup X_C \mapsto_{\mathcal{I}_B} (y, N_z)$, where N_z is a positive integer.* □

That is, Q is bounded under $\mathcal{A}$ iff for each "free variable" $z \in Z$ of Q, its boundedness can be deduced using $\mathcal{I}_B$ from (a) those parameters already instantiated in Q, and (b) those that only participate in condition checking and hence only need a witness for the truth value of the condition. The result is verified by using a notion of *access closures* (see [5]).

Example 4: For query $Q_0(Z)$ given in Example 1, $Z = \{\mathsf{pid}_1\}$, $X_B = \{\mathsf{tid}_1, \mathsf{fid}\}$, and $X_C = \{\mathsf{uid}, \mathsf{aid}, \mathsf{tid}_2\}$. By the deduction of $\mathcal{I}_B$ given in Example 3, $X_B \cup X_C \mapsto_{\mathcal{I}_B} (\mathsf{pid}_1, 1000)$, $X_B \cup X_C \mapsto_{\mathcal{I}_B} (\mathsf{tid}_1, 1000)$, $X_B \cup X_C \mapsto_{\mathcal{I}_B} (\mathsf{fid}, 1)$. Hence Q_0 is bounded under $\mathcal{A}_0$ by Theorem 3.

Now consider an arbitrary Boolean SPC query $Q(Z)$ under access schema $\mathcal{A}_\emptyset = \emptyset$. The set Z of parameters for projection is $\emptyset$, and $X_B \mapsto_{\mathcal{I}_B} (x, 1)$ for any $x \in X_B$ by Reflexivity. Thus Q is bounded under $\mathcal{A}_\emptyset$ by Theorem 3. □

3.2 Rules for Effective Boundedness

To decide whether an SPC query $Q(Z)$ is effectively bounded under $\mathcal{A}$, more needs to be done. When we propagate the boundedness from a set X of attributes to another set Y, we have to ensure that the values of Y can be efficiently retrieved *via available indices* in $\mathcal{A}$. Below we develop a set $\mathcal{I}_E$ of deduction rules by incorporating this condition.

Rules. Consider an access schema $\mathcal{A}$ over schema R and a set Y_R of attributes of R. We say that Y_R is *indexed* in $\mathcal{A}$ if there exists $X_R \subseteq Y_R$ such that (1) $X_R \rightarrow (W, N)$ is an access constraint in $\mathcal{A}$; and (2) $Y_R \subseteq X_R \cup W$.

If Y_R is indexed, given a value $\bar{b}$, we can check whether $Y_R = \bar{b}$ is in a dataset $D \models \mathcal{A}$ by using indices in $\mathcal{A}$. Otherwise, we cannot decide this without searching the entire D. Thus the condition is *necessary* for effective boundedness.

(**Reflexivity**) If $X' \subseteq X$, then $X \mapsto_{\mathcal{I}_E} (X', 1)$.
(**Actualization**) If $X \rightarrow (Y, N)$ is in $\mathcal{A}$, then $S_i[X] \mapsto_{\mathcal{I}_E} (S_i[Y], N)$ for each i in $[1, n]$.
(**Transitivity**) If $X \mapsto_{\mathcal{I}_E} (Y, N)$ and $Y \mapsto_{\mathcal{I}_E} (W, N')$, then $X \mapsto_{\mathcal{I}_E} (W, N * N')$.
(**Augmentation**) If $X \mapsto_{\mathcal{I}_E} (Y, N)$ and $X \cup Y$ is *indexed*, then $X \mapsto_{\mathcal{I}_E} (X \cup Y, N)$.
(**Combination**) If $X_1 \mapsto_{\mathcal{I}_E} (Y_1, N_1)$, ..., $X_k \mapsto_{\mathcal{I}_E} (Y_k, N_k)$, $\Sigma_Q \vdash Y_1 = Y_1'$, ..., $\Sigma_Q \vdash Y_k = Y_k'$, and $\bigcup_{i=1}^{k}(X_i \cup Y_i' \cup Y_i)$ is *indexed* in $\mathcal{A}$, then $X_1 \cup \cdots \cup X_k \mapsto_{\mathcal{I}_E} (Y_1' \cup \cdots \cup Y_k', N_1 * \cdots * N_k)$.

Figure 2: Rules $\mathcal{I}_E$ for effective boundedness

Consider an SPC query $Q(Z) = \pi_Z \sigma_C(S_1 \times \ldots \times S_n)$ and a set $Y = (Y_1, \ldots, Y_n)$ of *parameters in* Q (*i.e.,* in C or Z), where Y_i consists of attributes from S_i. We say that Y is *indexed in* $\mathcal{A}$ if each Y_i is indexed in $\mathcal{A}$.

Using these, we give a set $\mathcal{I}_E$ of five rules for deducing the effective boundedness of SPC queries, in Fig. 2. We define $X \mapsto_{\mathcal{I}_E} (Y, N)$ along the same lines as $X \mapsto_{\mathcal{I}_B} (Y, N)$, using $\mathcal{I}_E$. While Reflexivity, Actualization and Transitivity of $\mathcal{I}_E$ are the same as their counterparts in $\mathcal{I}_B$, the others are not.

(1) Augmentation in $\mathcal{I}_E$ revises its counterpart in $\mathcal{I}_B$ by allowing Y to be extended with only indexed attributes.

(2) Combination also restricts Augmentation of $\mathcal{I}_B$ by enforcing the indexing condition; *i.e.,* for any X_i-value $\bar{a}_i$, if $\bar{a}_i$ is in $\pi_{X_i}(D)$ for a dataset $D \models \mathcal{A}$, then the deduced Y-value must be in $\pi_Y(D)$ and can be retrieved via indices. Note that Augmentation is a special case of Combination; we opt to keep Augmentation in $\mathcal{I}_E$ as it is easier to use.

Characterization. Based on $\mathcal{I}_E$, we give *a sufficient and necessary condition* for effective boundedness. For an SPC query $Q(Z) = \pi_Z \sigma_C(S_1 \times \ldots \times S_n)$, we use the following notations: for all $i \in [1, n]$, (a) X_C^i is the set of all attributes of S_i already instantiated in Q, *i.e.,* $X_C^i = \{S_i[A] \in S_i \mid \Sigma_Q \vdash S_i[A] = c \text{ for a constant } c\}$, where S_i denotes the set of all attributes of S_i; (b) $X_C = X_C^1 \cup \cdots \cup X_C^n$; (c) X_Q^i denotes the set of all parameters of S_i that appear in either C or Z of Q; and (d) $\mathcal{X}^{\mathcal{A}}$ is the set of subsets $S_i[X]$ of attributes such that $X \rightarrow (Y, N)$ is in $\mathcal{A}$, for all $i \in [1, n]$.

Theorem 4: *An* SPC *query $Q(Z)$ is effectively bounded under an access schema $\mathcal{A}$ if and only if for each $i \in [1, n]$,*

(1) $X_C^i \subseteq W$ for some $W \in \mathcal{X}^{\mathcal{A}}$; and

(2) $X_C \mapsto_{\mathcal{I}_E} (X_Q^i, N_i)$ for some natural number N_i that is determined by Q and $\mathcal{A}$ only. □

That is, the instantiated attributes X_C^i can be checked using indices, as well as those attributes that participate in output or Boolean conditions of Q. We will use this characterization to generate query plans in Section 5. We show the result by using a notion of *effective access closures* [5].

Example 5: We show that $Q_0(\mathsf{pid}_1)$ of Example 1 is effectively bounded under access schema $\mathcal{A}_0$ of Example 2. First,

(1) $\mathsf{aid} \mapsto_{\mathcal{I}_E} (\mathsf{pid}_1, 1000)$	Actualization
(2) $\mathsf{aid} \mapsto_{\mathcal{I}_E} ((\mathsf{aid}, \mathsf{pid}_1), 1000)$	(1) and Augmentation
(3) $(\mathsf{aid}, \mathsf{uid}) \mapsto_{\mathcal{I}_E} (\mathsf{aid}, 1)$	Reflexivity
(4) $(\mathsf{aid}, \mathsf{uid}) \mapsto_{\mathcal{I}_E} ((\mathsf{aid}, \mathsf{pid}_1), 1000)$	(3)(2) and Transitivity
(5) $\mathsf{uid} \mapsto_{\mathcal{I}_E} (\mathsf{fid}, 5000)$	Actualization
(6) $\mathsf{uid} \mapsto_{\mathcal{I}_E} ((\mathsf{uid}, \mathsf{fid}), 5000)$	Augmentation
(7) $(\mathsf{aid}, \mathsf{uid}) \mapsto_{\mathcal{I}_E} (\mathsf{uid}, 1)$	Reflexivity

(8) $(\mathsf{aid}, \mathsf{uid}) \mapsto_{\mathcal{I}_E} ((\mathsf{uid}, \mathsf{fid}), 5000)$	(7)(6) and Transitivity
(9) $\mathsf{uid} \mapsto_{\mathcal{I}_E} (\mathsf{uid}, 1)$	Reflexivity
(10) $(\mathsf{aid}, \mathsf{uid}) \mapsto_{\mathcal{I}_E} ((\mathsf{pid}_2, \mathsf{tid}_2), 1000)$	(1)(9) and Combination
(11) $(\mathsf{pid}_2, \mathsf{tid}_2) \mapsto_{\mathcal{I}_E} (\mathsf{tid}_1, 1)$	Actualization
(12) $(\mathsf{pid}_2, \mathsf{tid}_2) \mapsto_{\mathcal{I}_E} ((\mathsf{pid}_2, \mathsf{tid}_1, \mathsf{tid}_2), 1)$	Augmentation
(13) $(\mathsf{aid}, \mathsf{uid}) \mapsto_{\mathcal{I}_E} ((\mathsf{pid}_2, \mathsf{tid}_1, \mathsf{tid}_2), 1000)$	(10)(12) and Transitivity

Then (a) condition (1) of Theorem 4 is satisfied since aid, uid and tid_2 are in subsets $\{\mathsf{aid}\}$, $\{\mathsf{udi}\}$ and $\{\mathsf{pid}_2, \mathsf{tid}_2\}$ of $\mathcal{X}^{\mathcal{A}_0}$, respectively. (b) Condition (2) is satisfied by deduction steps (4), (8) and (13) above, and as $(\mathsf{pid}_2, \mathsf{tid}_2)$ is indexed in $\mathcal{A}_0$. Thus Q_0 is effectively bounded under $\mathcal{A}_0$ by Theorem 4. $\square$

Remark. Note that we do not need "full and complete" access schema to achieve "boundedness". Instead, as will be shown in Section 6, in practice many queries on real-life data are effectively bounded, under only a small number of access constraints. Moreover, queries supported by real-life recommendation and e-commerce systems are typically parameterized queries: they are fixed query templates that only allow a few variables to be instantiated, *e.g.,* web forms supported by Amazon and Expedia. Under simple access constraints, these parameterized queries become effectively bounded and scalable with (possibly big) product databases.

4. BOUNDEDNESS: COMPLEXITY AND ALGORITHMS

We next study two issues in connection with the (effective) boundedness of SPC queries. (1) We study the complexity and algorithms for deciding whether an SPC query is (effectively) bounded under an access schema $\mathcal{A}$. (2) When Q is not effectively bounded, we study whether Q can be made effectively bounded under $\mathcal{A}$ by instantiating a set X_P of parameters of Q, and if so, how to compute a minimum X_P.

The main results of this section are as follows. (1) The boundedness of Q under $\mathcal{A}$ can be decided in quadratic time (Section 4.1). (2) The same complexity holds for effective boundedness (Section 4.2). (3) The decision problem for dominating parameters is NP-complete, and its optimization problem is NPO-complete. We provide an efficient heuristic algorithm to compute dominating parameters (Section 4.3).

4.1 Checking Boundedness

We start with the *boundedness problem* $\mathsf{Bnd}(Q, \mathcal{A})$:

- Input: A relation schema R, an SPC query Q over R, and an access schema $\mathcal{A}$ over R.
- Question: Is Q bounded under $\mathcal{A}$?

This is to decide whether for all datasets D that satisfy $\mathcal{A}$, there exists *at all* a subset D_Q such that $Q(D) = Q(D_Q)$ and $|D_Q|$ is independent of the size $|D|$ of the underlying D.

While this problem is undecidable for (Boolean) $\mathcal{RA}$ queries [20], it is decidable in PTIME for SPC.

Theorem 5: *For any* SPC *query Q and access schema $\mathcal{A}$,* $\mathsf{Bnd}(Q, \mathcal{A})$ *can be decided in $O(|Q|(|\mathcal{A}| + |Q|))$ time.* $\square$

Here $|\mathcal{A}|$ and $|Q|$ are the size of $\mathcal{A}$ and Q, respectively, and are typically small in practice, compared to datasets D.

As a constructive proof for Theorem 5, we next give such an algorithm for checking the boundedness of Q under $\mathcal{A}$.

Algorithm BCheck

Input: An SPC query Q, and an access schema $\mathcal{A}$.
Output: "yes" if Q is bounded under $\mathcal{A}$ and "no" otherwise.

1. $\Gamma := \mathsf{Actualize}(\mathcal{A}, Q)$; /*Initialization*/
2. $closure := X_B \cup X_C$; $\mathcal{B} := X_B \cup X_C$;
3. **for each** attribute A in $\mathcal{A}$ and Q and **each** ϕ in Γ **do**
4.   **if** $\mathsf{isIn}(\phi, A, Q)$ **then** /*suppose that ϕ is $X_\phi \mapsto_{\mathcal{I}_B} (Y_\phi, N_\phi)$*/
5.   add ϕ to $L[A]$; $n_\phi := |X_\phi|$;
6. **while** $\mathcal{B}$ is not empty **do** /*Computation*/
7. $A := \mathcal{B}.\mathsf{pop}()$;
8.   **for each** ϕ in $L[A]$ **do**
9.   decrease n_ϕ with 1;
10.   **if** $n_\phi = 0$ **do** /*suppose that ϕ is $X_0 \mapsto_{\mathcal{I}_B} (Y_0, N)$*/
11. $\mathcal{B} := \mathcal{B} \cup (Y_0 \setminus closure)$;
12.   **for each** attribute B_0 in Y_0 **do**
13.   **for** all B_0' such that $\Sigma_Q \vdash B_0 = B_0'$ **do**
14.   add B_0' to $closure$;
15. **if** $X_B \cup Z \subseteq closure$ **then return** "yes"; /*Checking*/
16. **return** "no";

Figure 3: Algorithm BCheck

Algorithm BCheck. The algorithm is denoted by BCheck and shown in Fig. 3. It is based on the characterization of $\mathcal{I}_B$ (Section 3). It computes $(X_B \cup X_C)^*$, stored in a variable *closure*, and concludes that Q is bounded under $\mathcal{A}$ if and only if $X_B \cup Z \subseteq closure$, *i.e.,* when all parameters of Q are covered by $(X_B \cup X_C)^*$ (see Theorem 3 and its proof).

More specifically, BCheck first actualizes access constraints of $\mathcal{A}$ in each renaming S_i of schema R in Q: for each $X \to (Y, N)$ in $\mathcal{A}$ and each S_i in Q, it includes $S_i[X] \mapsto_{\mathcal{I}_B} (S_i[Y], N)$ in a set Γ (line 1). Using Γ, it then computes *closure* (lines 2-14) such that if $X_B \cup X_C \mapsto_{\mathcal{I}_B} (y, N)$ for some N and attribute y, then y is included in *closure*. After this, it simply checks whether $X_B \cup Z$ is contained in *closure*; it returns "yes" if so and "no" otherwise (lines 15-16).

We next show how BCheck computes *closure*, starting with auxiliary structures used by BCheck.

Auxiliary structures. BCheck uses three auxiliary structures.

(1) BCheck maintains a set $\mathcal{B}$ of attributes in $\mathcal{A}$ and Q that are in *closure* but it remains to be checked what other attributes can be deduced from them via $\mathcal{I}_B$. Initially, $\mathcal{B} = X_B \cup X_C$ (line 2). BCheck uses $\mathcal{B}$ to control the **while** loop (lines 6–14): it terminates when $\mathcal{B} = \emptyset$, *i.e.,* when all necessary deduction checking via $\mathcal{I}_B$ has been completed.

(2) For each constraint ϕ: $X \mapsto_{\mathcal{I}_B} (Y, N)$ in Γ, BCheck maintains a counter n_ϕ to keep track of those attributes of X that are still in $\mathcal{B}$. Initially, n_ϕ is the number of attributes in X. When $n_\phi = 0$, *i.e.,* after all X attributes have been processed, the Y attributes can be added to $\mathcal{B}$ (lines 10-11).

(3) For each attribute A in Q and Γ, BCheck uses a list $L[A]$ to store all constraints $X \mapsto_{\mathcal{I}_B} (Y, N)$ in Γ such that either A is in X or there exists A' in X with $\Sigma_Q \vdash A = A'$. That is, $L[A]$ indexes constraints that are "applicable" to A.

Computing closure. With these structures, BCheck computes *closure* as follows. It first initializes the auxiliary structures as described above (lines 2-5). Here function $\mathsf{isIn}(\phi, A, Q)$ checks whether constraint ϕ: $X_\phi \mapsto_{\mathcal{I}_B} (Y_\phi, N_\phi)$ is "applicable" to attribute A, *i.e.,* whether there exists A' such that $\Sigma_Q \vdash A = A'$ and A' is in X_ϕ (line 5).

After this, BCheck processes attributes in $\mathcal{B}$ one by one (lines 6-14). For each attribute $A \in \mathcal{B}$ and each constraint $\phi : X_0 \mapsto_{\mathcal{I}_B} (Y_0, N)$ in $L[A]$, it decreases the counter n_ϕ

by 1. When $n_\phi = 0$, *i.e.,* all attributes in X_0 have been inspected, BCheck conducts deduction via $\mathcal{I}_B$ (lines 11-14). It adds to $\mathcal{B}$ attributes B_0 in Y_0 that are not yet in *closure* (line 11), and add to *closure* all those attributes B'_0 such that $\Sigma_Q \vdash B_0 = B'_0$ (lines 12–14). When $\mathcal{B}$ becomes empty, BCheck returns "yes" iff $X_B \cup Z \subseteq$ *closure* (lines 15-16).

Correctness & Complexity. The correctness of BCheck follows from Theorem 3. To see that BCheck is in $O(|Q|(|\mathcal{A}|+|Q|))$ time, observe the following. (1) The initialization steps take $O(|Q||\mathcal{A}|)$ time (lines 1-5). (2) The *closure* is computed in $O(|\mathcal{A}| + |\mathcal{A}||Q|)$ time (lines 6-14), since the counters are updated at most $O(|\mathcal{A}||Q|)$ times *in total*, and each ϕ in Γ is used at most once, in $O(|\phi|+|Q|)$ time (thus $O(|\mathcal{A}|+|\mathcal{A}||Q|)$ time in total). (3) The checking (line 15) can be done in $O(|Q|^2)$ time, since the size of *closure* is bounded by $O(|Q|)$.

Example 6: We show how algorithm BCheck finds that query Q_0 of Example 1 is bounded under the access schema $\mathcal{A}_0$ of Example 2. Here $X_B \cup X_C = \{\text{aid, uid, tid}_2\text{, fid, tid}_1\}$. BCheck initializes Γ with aid $\mapsto_{\mathcal{I}_B}$ $(\text{pid}_1, 1000)$ (ϕ_1), $(\text{pid}_2, \text{tid}_2) \mapsto_{\mathcal{I}_B} (\text{tid}_1, 1)$ (ϕ_2), and uid $\mapsto_{\mathcal{I}_B}$ $(\text{fid}, 5000)$ (ϕ_3). It assigns $X_B \cup X_C$ as the initial value of *closure* and $\mathcal{B}$, and sets counters $n_{\phi_1} = n_{\phi_3} = 1$, $n_{\phi_2} = 2$. After aid is popped off from $\mathcal{B}$, n_{ϕ_1} is decreased to 0 and BCheck updates *closure* and $\mathcal{B}$ with ϕ_1 (lines 11-14). Since $\Sigma_Q \vdash \text{pid}_1 = \text{pid}_2$, both pid_1 and pid_2 are added to *closure*, and pid_1 is added to $\mathcal{B}$. After this iteration, *closure* remains unchanged and $\mathcal{B}$ will be reduced to empty. Since $X_B \cup Z = \{\text{pid}_1, \text{pid}_2, \text{tid}_1, \text{fid}\}$ is a subset of *closure*, BCheck returns "yes". □

4.2 Checking Effective Boundedness

We next study the *effective boundedness problem*, denoted by $\text{EBnd}(Q, \mathcal{A})$ and stated as follows:

- Input: R, Q and $\mathcal{A}$ as in $\text{Bnd}(Q, \mathcal{A})$.
- Question: Is Q effectively bounded under $\mathcal{A}$?

It is to decide whether for any D that satisfies $\mathcal{A}$, we can fetch $D_Q \subseteq D$ via indices in $\mathcal{A}$ such that $Q(D) = Q(D_Q)$.

Problem EBnd is also decidable in quadratic-time.

Theorem 6: $\text{EBnd}(Q, \mathcal{A})$ *is in* $O(|Q|(|\mathcal{A}| + |Q|))$ *time.* □

We prove Theorem 6 by providing an algorithm for checking the effective boundedness of Q under $\mathcal{A}$.

Algorithm EBCheck. The algorithm, denoted by EBCheck, extends algorithm BCheck by leveraging Theorem 4 and the following connection between $\mathcal{I}_E$ and the *access closure* for boundedness: for any sets X and Y of attributes in Q such that $X \subseteq Y$, $X \mapsto_{\mathcal{I}_E} Y$ if and only if $Y \subseteq X^*$ and Y is indexed in $\mathcal{A}$. Based on this, EBCheck works as follows.

Step 1 (computing closure): Compute X_C^* by adopting the *closure* computation part of BCheck (lines 1-14, Fig. 3) except that it initializes *closure* to be X_C instead of $X_B \cup X_C$.

Step 2 (checking): Check (a) whether $\bigcup_{i=1}^n X_Q^i$ is a subset of X_C^* and (b) whether $\bigcup_{i=1}^n X_Q^i$ is indexed in $\mathcal{A}$. If so, Q is effectively bounded under $\mathcal{A}$. Note that the condition (1) of Theorem 4 is implied by (b) here.

As both steps are in $O(|Q|(|\mathcal{A}|+|Q|)$ time, so is EBCheck.

Example 7: Consider again query Q_0 of Example 1 and access schema $\mathcal{A}_0$ of Example 2. The deduction analysis of Example 5 tells us that X_C^* of Q_0 covers parameters of in_album, friends and tagging; moreover, X_C^* is indexed by $\mathcal{A}_0$. That is, the conditions in Step 2 of EBCheck are satisfied. Hence, Q_0 is effectively bounded under $\mathcal{A}_0$. □

4.3 Computing Dominating Parameters

As illustrated in Example 1, when an SPC query Q is not effectively bounded under $\mathcal{A}$, we want to identify a minimum set X_P of parameters of Q such that if X_P is instantiated, Q becomes effectively bounded. We want to find and suggest such an X_P to users if it exists. When the users provide a value of X_P, Q can be answered in a big dataset D by accessing a bounded amount of data. We consider parameters of X_P that are not in X_C, *i.e.,* not yet instantiated in Q, and are not *trivial*, *i.e.,not* covering *all* attributes in Q.

More specifically, we use $Q(X_P = \bar{a})$ to denote the query obtained from Q when X_P is given a value $\bar{a}$. We call X_P a set of *dominating parameters* of Q under $\mathcal{A}$ *w.r.t.any fixed* fraction $\alpha \in (0, 1)$, if $|X_P|/|X_B| \leq \alpha$ and $Q(X_P = \bar{a})$ is effectively bounded under $\mathcal{A}$ *for all* given X_P values $\bar{a}$ (see Section 3.1 for X_B). Intuitively, by instantiating X_P, which contains at most $\alpha|X_B|$ attributes of Q, we can make $Q(X_P = \bar{a})$ effectively bounded under $\mathcal{A}$.

Problems and complexity. This suggests that we study the following decision and optimization problems.

The dominating parameter problem $\text{DP}(Q, \mathcal{A})$.

- Input: R, $Q(Z)$, $\mathcal{A}$ as in $\text{EBnd}(Q, \mathcal{A})$, any fixed α.
- Question: Does there exist a set of dominating parameters of Q under $\mathcal{A}$ *w.r.t.* α?

The minimum dominating parameter problem $\text{MDP}(Q, \mathcal{A})$.

- Input: R, $Q(Z)$, $\mathcal{A}$ as in $\text{EBnd}(Q, \mathcal{A})$, any fixed α.
- Output: A set of dominating parameters X_P of Q under $\mathcal{A}$ *w.r.t.* α with minimum cardinality, if it exists.

Problem $\text{DP}(Q, \mathcal{A})$ is to decide whether Q has a set of dominating parameters at all. Problem $\text{MDP}(Q, \mathcal{A})$ is to compute a minimum set of dominating parameters of Q.

Example 8: An SPC query may not have a set of dominating parameters under an access schema. As an example, consider query Q_0 of Example 1 and an access schema $\mathcal{A}_1$ that contains all access constraints in A_0 of Example 2 except $(\text{photo\_id}, \text{taggee\_id}) \to (\text{tagger\_id}, 1)$. Then Q_0 is not effectively bounded under $\mathcal{A}_1$, and worse still, no matter what parameters of Q_0 we instantiate, it is still not effectively bounded. This is because no index is built on tagging in $\mathcal{A}_1$, and hence we cannot verify, *e.g.,* whether $\text{tid}_2 = u_0$ is in a tagging instance without searching the entire D. □

While DP and MDP are important, they are hard.

Theorem 7: *For* SPC *query* Q *and access schema* $\mathcal{A}$*,*

(1) $\text{DP}(Q, \mathcal{A})$ *is* NP*-complete; and*

(2) $\text{MDP}(Q, \mathcal{A})$ *is* NPO*-complete.* □

NPO is the *class* of all NP optimization problems. NPO-complete problems are the hardest optimization problems in NPO: they do not even allow PTIME approximation algorithms with an exponential approximation ratio (cf. [13]).

Algorithm. In light of Theorem 7, we develop a heuristic algorithm that for any fixed $\alpha \in (0, 1)$, given Q and $\mathcal{A}$, checks whether there exists a set of dominating parameters for Q under $\mathcal{A}$ *w.r.t.* α; it finds and returns such a set X_P if so, and returns "no" otherwise. The algorithm, denoted by findDP_h, consists of three steps.

Step 1 (initial candidates): For each renaming S_i of R in Q and each parameter A of Q that is in S_i but is not in X_C, add A to a set X_P if there exists a constraint $X \to (Y, N)$ in $\mathcal{A}$ such that A is in $S_i[X] \cup S_i[Y]$.

Step 2 (checking): Check (a) whether $\bigcup_{i=1}^{n} X_Q^i$ is indexed in $\mathcal{A}$ and (b) whether for all X_Q^i, $X_Q^i \subseteq X_P$ (see Section 3.2 for the definition of X_Q^i). If not, return "no".

Step 3 (minimizing): We optimize X_P iteratively as follows. Each time we pick one attribute A of some S_i from X_P, and check whether there is $X \to (Y, N)$ in $\mathcal{A}$ such that $S_i[X] \subseteq X_P$, $A \notin S_i[X]$ and $A \in S_i[Y]$. If so, let $X_P = X_P \setminus \mathsf{ext_Q}(A)$ since X_P can be recovered from $X_P \setminus \{A\}$ via deduction of $\mathcal{I}_E$, where $\mathsf{ext_Q}(A)$ consists of all parameters x such that $\Sigma_Q \vdash A = x$. We then process the next attribute. We return X_P when it cannot be further reduced and $|X_P|/|X_B| \leq \alpha$.

Correctness & Complexity. One can verify that if $\mathsf{findDP_h}$ returns X_P, then X_P is a set of dominating parameters for Q under $\mathcal{A}$. Indeed, if X_P is instantiated, then for all S_i in Q, all parameters in X_Q^i can be deduced from X_P via $\mathcal{I}_E$ and are also indexed. Hence $Q(X_P = \bar{a})$ is effectively bounded under $\mathcal{A}$ by Theorem 4, for any X_P value $\bar{a}$.

Algorithm $\mathsf{findDP_h}$ is in $O(|Q|(|Q| + |\mathcal{A}|))$ time. Indeed, its step 1 is in $O(|\mathcal{A}||Q|)$ time; step 2 takes $O(|Q|^2)$ time since $|X_P|$ and $|X_Q|$ are both bounded by $|Q|$; and step 3 is in $O(|Q|(|\mathcal{A}| + |Q|))$ time because $|X_P| \leq |Q|$; hence it takes $O(|\mathcal{A}|)$ time to check whether an attribute A can be removed from X_P, and $O(|Q|)$ time to remove $\mathsf{ext_Q}(A)$.

Example 9: Recall that query Q_1 of Example 1 is not effectively bounded under access schema $\mathcal{A}_0$ of Example 2. Consider $\alpha = 3/7$. We show how $\mathsf{findDP_h}$ finds a set X_P of dominating parameters for Q_1. In step 1, it sets X_P = {$\mathsf{pid_1}$, aid, uid, fid, $\mathsf{pid_2}$, $\mathsf{tid_1}$, $\mathsf{tid_2}$}. In step 2, $\mathsf{findDP_h}$ finds X_Q^i contained in X_P for X_Q^i in $\mathsf{in\_album}$, $\mathsf{friends}$ and $\mathsf{tagging}$; hence there exists a set of dominating parameters for Q_1. In step 3, it reduces X_P. (a) It first finds that $\mathsf{album\_id} \to (\mathsf{photo\_id}, 1000)$ in $\mathcal{A}_0$, and removes $\mathsf{pid_1}$ and $\mathsf{pid_2}$ from X_P since $\Sigma_Q \vdash \mathsf{pid_1} = \mathsf{pid_2}$. (b) It then finds $\mathsf{user\_id} \to (\mathsf{friend\_id}, 5000)$ in $\mathcal{A}_0$, and removes fid and $\mathsf{tid_1}$ from X_P by $\Sigma_Q \vdash \mathsf{fid} = \mathsf{tid_1}$. Finally, $\mathsf{findDP_h}$ finds that it can remove no more parameters from X_P and $|X_P|/|X_B| \leq \alpha$, and thus returns X_P = {aid, uid, $\mathsf{tid_2}$}, which is exactly the set of instantiated parameters for Q_0 (by $\Sigma_{Q_0} \vdash \mathsf{tid_2} = \mathsf{u_0}$). □

5. ALGORITHM FOR EFFECTIVELY BOUNDED QUERIES

Algorithm $\mathsf{EBCheck}$ of Section 4.2 is able to determine the effective boundedness of SPC queries. However, it does not tell us *how to* identify a bounded amount of data to answer those queries. To bridge the gap, we next develop an algorithm that, given an effectively bounded SPC query $Q(Z) = \pi_Z \sigma_C(S_1 \times \ldots \times S_n)$ and an access schema $\mathcal{A}$, finds a query plan that, given a (big) dataset D, fetches a bounded $D_Q \subseteq D$ using indices in $\mathcal{A}$ such that $Q(D) = Q(D_Q)$.

The main results of the section are as follows. (1) There exists an $O(|Q|^2|\mathcal{A}|^3)$-time algorithm that generates query plans for effectively bounded SPC queries (Section 5.1). (2) We also study the problem to find a minimum bounded D_Q, and show that the problem is NP-complete (Section 5.2).

5.1 Determining and Computing D_Q

We find a query plan for Q by deducing a proof ρ_i for $X_C \mapsto_{\mathcal{I}_E} (X_Q^i, M_i)$ for all $i \in [1, n]$, following Theorem 4. Below we show that the proofs yield a query plan that, for any dataset D such that $D \models \mathcal{A}$, tells us how to find D_Q such that $Q(D) = Q(D_Q)$ and D_Q has at most $\sum_{i=1}^{n} M_i$ tuples.

Query plan from proofs. Suppose that $X_C \mapsto_{\mathcal{I}_E} (X_Q^i, M_i)$ is proven by $\rho_i = \varphi_1, \cdots, \varphi_m$, where φ_j denotes application of a rule in $\mathcal{I}_E$. We show that given D, ρ_i tells us how to find a list of subsets $T_1, \ldots, T_m$ of D such that

- $D_Q^i = \bigcup_{j=1}^{m} T_j$ and $D_Q = \bigcup_{i=1}^{n} D_Q^i$, and
- for all $j \in [1, m]$, $T_j \subseteq D$, T_j has at most N_j tuples and can be fetched by using indices in $\mathcal{A}$, where N_j is a number deduced from the proof, independent of $|D|$.

We can then compute $Q(D)$ by conducing joins and projections on these T_j's only, guided by conditions in σ_C of Q, as illustrated by how we get $Q_0(D_0)$ using T_1–T_4 in Example 1.

Below we show how to fetch T_j from D guided by rule φ_j, by giving two example rules (see [5] for other rules). Initially, $T_1 = \bigcup_{j=1}^{n} \sigma_{X_j = C_j}(D)$, and can be fetched by using indices in $\mathcal{A}$ on the constants of X_C (see Theorem 4 and its proof).

(a) When φ_j actualizes a constraint $X \to (Y, N)$ of $\mathcal{A}$, we fetch N tuples for T_j either from D by using index in $\mathcal{A}$ on X for Y, or from a bounded subset $T_{j'}$ of D ($j' < j$) deduced from previous steps in proof ρ_i, on which φ_j is applied.

(b) When φ_j is Combination, we get T_j as follows. Denote $\bigcup_{s=1}^{j-1} T_s$ by T. As indicated by the rule (Fig. 2), for $l \in [1, k]$, (i) all X_l and Y_l values are already fetched in T; and (ii) we can check whether these X_l and Y_l values appear in tuples of D, *i.e.*, they are contained in the projection of D on $\bigcup_{l=1}^{k} X_l \cup Y_l'$, by using the indices on the attributes. There are at most $N_1 * \ldots * N_k$ such tuples from T to be inspected in D, and T_j consists of these tuples.

Algorithm QPlan. We now present the algorithm, denoted by QPlan and shown in Fig. 4. Based on the connection between $\mathcal{I}_E$ proofs and query plans given above, QPlan focuses on finding a proof ρ_i for each $X_C \mapsto_{\mathcal{I}_E} (X_Q^i, M_i)$, based on the characterization of $\mathcal{I}_E$ of Section 3. It represents ρ_i as an object o_i, which consists of three components:

- $o_i.X$: parameters of X_Q^i deduced from the proof;
- $o_i.\mathcal{P}$: a proof for deducing $o_i.X$ from X_C; and
- $o_i.c$: the number of tuples that need to be fetched and inspected based on the query plan $o_i.\mathcal{P}$.

When o_i is completed, $o_i.\mathcal{P} = \rho_i$ and $o_i.c = M_i$.

Given an SPC query $Q(Z) = \pi_Z \sigma_C(S_1 \times \ldots \times S_n)$ that is effectively bounded under $\mathcal{A}$, QPlan returns a set $X_C^{\mathsf{min}+}$ of objects such that for $i \in [1, n]$, there exists $o_i \in X_C^{\mathsf{min}+}$ representing a proof for $X_C \mapsto_{\mathcal{I}_E} (X_Q^i, M_i)$.

More specifically, $X_C^{\mathsf{min}+}$ is a set of objects such that that Q is effective bounded under $\mathcal{A}$ if and only if for each $i \in [1, n]$, (1) $X_C^i \subseteq W$ for some W in $\mathcal{X}^{\mathcal{A}}$; and (2) $X_Q^i \subseteq o.X$ for some object o in $X_C^{\mathsf{min}+}$. It has a *coverage property*: for all Y, if $X \mapsto_{\mathcal{I}_E} (Y, N)$ and $X \subseteq Y$, then there exists some $o \in X_C^{\mathsf{min}+}$ such that $Y \subseteq o.X$. These suffice by Theorem 4.

We use the following notations. (a) A set S_2 of objects can be *deduced from* another set S_1 if there exists a proof from $\bigcup_{o \in S_1} o.X$ to $\bigcup_{o \in S_2} o.X$. (b) We use γ_1–γ_5 to denote the five rules in $\mathcal{I}_E$ (Fig. 2), respectively. For instance, γ_5 denotes Combination, and $\gamma_2(X \to (Y, N))$ indicates the application

Algorithm QPlan

Input: An SPC query Q, and an access schema $\mathcal{A}$.
Output: A set $X_C^{\mathsf{min}+}$ of objects representing a query plan.

1. $X_C^{\mathsf{min}+} := \{o_C\}$; $\mathcal{B} := X_C^{\mathsf{min}+}$; /*$o_C.X = X_C$, $o_C.\mathcal{P} = \emptyset$, $o_C.c = 0$*/
2. $\Gamma :=$ Actualize$(\mathcal{A}, Q)$; $\mathcal{T} := nil$; /*Initialization*/
3. **while** $\mathcal{B}$ is not empty **do** /*Computing set $X_C^{\mathsf{min}+}$*/
4. $\quad o := \mathcal{B}$.pop();
5. $\quad$ **for each** $\phi : W \mapsto_{\mathcal{I}_E} (Y, N)$ in Γ and $W \subseteq o.X$ **do**
6. $\quad\quad$ instantiate o_Y for *possibly* deducing $o.X \cup Y$ from X_C;
7. $\quad\quad$ add o_Y to sets $\mathcal{T}$; remove ϕ from Γ;
8. $\quad$ **for each** $\Sigma_Q \vdash W = X'$, $X' \subseteq o.X$ and $W' \not\subseteq o.X$ **do**
9. $\quad\quad$ **if** $o.X \cup W \not\subseteq o'.X$ for any o' in $X_C^{\mathsf{min}+}$ **do**
10. $\quad\quad\quad$ instantiate o_W for *possibly* deducing $o.X \cup W$ from X_C;
11. $\quad\quad\quad$ add o_W to $\mathcal{T}$ for checking the *indexing* condition of γ_5;
12. $\quad U :=$ chkComb$(\mathcal{T}, X_C^{\mathsf{min}+})$; /*Deduce with Combination*/
13. $\quad \mathcal{B} := \mathcal{B} \cup U$; $X_C^{\mathsf{min}+} := X_C^{\mathsf{min}+} \cup U$;
14. **return** $X_C^{\mathsf{min}+}$;

Procedure chkComb

Input: Sets $\mathcal{T}$ and $X_C^{\mathsf{min}+}$ of objects.
Output: Set U of objects that are deducible from $X_C^{\mathsf{min}+}$ by γ_5.

1. $U := \emptyset$; $u_i := \emptyset$ for each $X_i \to (Y_i, N_i)$ in $\mathcal{A}$;
2. **for each** $X_i \to (Y_i, N_i)$ in $\mathcal{A}$ **do**
3. $\quad$ **for each** $o \in \mathcal{T} \cup X_C^{\mathsf{min}+}$ **do**
4. $\quad\quad$ **if** $o.X \subseteq X_i \cup Y_i$ **then** add o to u_i;
5. $\quad$ **if** $X_i \subseteq \bigcup_{o \in u_i} o.X$ **do**
6. $\quad\quad$ instantiate o_i for deducing $\bigcup_{o \in u_i} o.X$ from X_C via γ_5;
7. $\quad$ **if** $o_i.X \not\subseteq o'.X$ for all $o' \in X_C^{\mathsf{min}+}$ **do**
8. $\quad\quad$ add o_i to U; $X_C^{\mathsf{min}+} := X_C^{\mathsf{min}+} \setminus u_i$;
9. **return** U;

Figure 4: Algorithm QPlan

of Actualization with access constraint $X \to (Y, N)$ in $\mathcal{A}$.

Algorithm QPlan also uses the following structures: (a) a set $\mathcal{B}$ of objects that are in $X_C^{\mathsf{min}+}$ but remain to be checked for other objects that can deduced from them, similar to its counterpart used in BCheck (Fig. 3); and (b) a set $\mathcal{T}$ of candidate objects deduced from equality atoms in Σ_Q, which is to be used when Combination rule is applied.

Using these structures, algorithm QPlan works as follows. It first collects in Γ all actualized constraints of $\mathcal{A}$ in the same way as BCheck (Fig. 3), and initializes both $X_C^{\mathsf{min}+}$ and $\mathcal{B}$ with the set consisting of only one object that represents the proof for X_C; it sets $\mathcal{T}$ empty (lines 1-2).

After these, QPlan iteratively finds objects that can possibly be deduced from $X_C^{\mathsf{min}+}$, by processing objects in $\mathcal{B}$ one by one (lines 3-13). For each object o in $\mathcal{B}$, it finds all possible *direct* deductions with the actualized constraints, and adds them to $\mathcal{T}$ (lines 5-7). More specifically, if there exists an actualized constraint ϕ: $W \mapsto_{\mathcal{I}_E} (Y, N)$ in Γ and if W is a subset of $o.X$, then $o.X \cup Y$ can possibly be deduced from X_C by first deducing $o.X$ using $o.\mathcal{P}$, and then deducing W by using Reflexivity (from $o.X$) followed by Transitivity (from X_C), with $o.c = N$, and possibly with Augmentation. Algorithm QPlan stores these single-step deductions in an object o_Y (line 6), and adds it to $\mathcal{T}$ for checking whether $o.X \cup Y$ is indexed in $\mathcal{A}$. It removes ϕ from Γ (line 7).

Intuitively, QPlan expands set $\mathcal{T}$ by including all new candidate objects that can *possibly* be deduced by γ_5 (*i.e.,* Combination rule), subject to the *indexing* condition of γ_5 to be checked (lines 8-11). It invokes procedure chkComb to identify combinations of objects in $\mathcal{T}$ to which γ_5 can be applied; chkComb returns a set U of new objects that encode new parameters of Q deduced by γ_5 (line 12; see details shortly). The objects of U are added to $X_C^{\mathsf{min}+}$ and $\mathcal{B}$ (line 17). The algorithm then proceeds to process the next object in $\mathcal{B}$ in the same way, until $\mathcal{B}$ becomes empty.

After the **while** loop, QPlan returns $X_C^{\mathsf{min}+}$ that contains proofs for each $X_C \mapsto_{\mathcal{I}_E} (X_Q^i, N_i)$ (line 14).

Procedure chkComb. Given $\mathcal{T}$ and $X_C^{\mathsf{min}+}$, chkComb finds all maximum subsets of $\mathcal{T} \cup X_C^{\mathsf{min}+}$ to which rule γ_5 can be applied, to deduce new parameters. More specifically, each subset satisfies the following conditions: (1) the union of their encoded attributes is indexed in $\mathcal{A}$; (2) it is *maximal*, *i.e.,* it cannot be expanded; and (3) no objects in it are already in $X_C^{\mathsf{min}+}$. Each of these subsets is encoded by a new object, representing all attributes covered by the subset.

The procedure works as follows. Assume *w.l.o.g.* that for each object o in $\mathcal{T} \cup X_C^{\mathsf{min}+}$, $o.X$ contains attributes from the same renaming S_i only. It associates a set u_i with each constraint $X_i \to (Y_i, A)$ in $\mathcal{A}$, initially empty (line 1). It collects in u_i all objects of $\mathcal{T} \cup X_C^{\mathsf{min}+}$ that can be combined using γ_5 and are indexed by $X_i \cup Y_i$ (lines 2-4). If X_i is covered by attributes encoded in the objects of u_i, then these attributes can be deduced by γ_5 and hence, a new object o_i is created to encode them (lines 5-6). If attributes in $o_i.X$ are not covered by existing objects in $X_C^{\mathsf{min}+}$, then it adds o_i to U, and removes objects of u_i from $X_C^{\mathsf{min}+}$ (lines 7-8). The process proceeds until all constraints in $\mathcal{A}$ are checked (line 2). After the loop, it returns set U.

Correctness & Complexity. The correctness of QPlan follows from Theorem 4 and the coverage property of $X_C^{\mathsf{min}+}$.

To see that QPlan is in $O(|Q|^2|\mathcal{A}|^3)$-time, observe the following. (1) At most $O(|Q||\mathcal{A}|)$ objects are added to $\mathcal{B}$. This is because each actualized constraint in Γ and each equality atom in σ_C of Q are processed *only once*; moreover, each equality atom yields at most $O(|\mathcal{A}|)$ objects. (2) The loop (lines 5-11) is executed at most $O(|Q||\mathcal{A}|)$ times *in total*. (3) Procedure chkComb is in $O(|Q||\mathcal{A}|^2)$ time; thus in the entire process, chkComb takes $O((|Q||\mathcal{A}| + |\mathcal{A}|) * |Q||\mathcal{A}|^2) = O(|Q|^2|\mathcal{A}|^3)$ time in total. Indeed, (a) its initialization is in $O(|\mathcal{A}|)$ time; (b) the total time taken by checking indexing (lines 3-4) is $O(|\mathcal{A}||\mathcal{T} \cup X_C^{\mathsf{min}+}|) = O(|Q||\mathcal{A}|^2)$; and (c) checking the conditions of line 5 and line 7 takes $O(|Q||\mathcal{A}|^2)$ time each. We remark that $|Q|$ and $|\mathcal{A}|$ are *typically small* in real-life, compared to the size of dataset D.

Example 10: We show how QPlan generates a query plan for Q_0 of Example 1 under access schema $\mathcal{A}_0$ of Example 2. Initially, both $X_C^{\mathsf{min}+}$ and $\mathcal{B}$ contain an object o_C encoding X_C such that $o_C.X = \{\mathsf{aid}, \mathsf{uid}, \mathsf{tid}_2\}$ and $o_C.\mathcal{P} = nil$. It then updates $X_C^{\mathsf{min}+}$ and $\mathcal{B}$ iteratively. At the beginning, o_C is popped off from $\mathcal{B}$. It constructs o_1 with $o_1.X = o_C.X \cup \{\mathsf{pid}_1\}$ $o_1.\mathcal{P} = [\gamma_1, \gamma_2(\mathsf{aid} \mapsto_{\mathcal{I}_E} (\mathsf{pid}_1, 1000), \gamma_4]$ and $o_1.c = 1000$; it puts o_1 in $\mathcal{T}$. Similarly, it adds o_2 to $\mathcal{T}$ with $o_2.X = o_C.X \cup \{\mathsf{fid}\}$, $o_2.\mathcal{P} = [\gamma_1, \gamma_2(\mathsf{uid} \mapsto_{\mathcal{I}_E} (\mathsf{fid}, 5000), \gamma_3]$ and $o_2.c = 5000$. After these, it invokes chkComb and finds $U = \{o_1, o_2\}$ since $o_1.X$ and $o_2.X$ are indexed in $\mathcal{A}_0$. It replaces o_C in $\mathcal{B}$ and $X_C^{\mathsf{min}+}$ with o_1 and o_2. After that, it pops off o_1 from $\mathcal{B}$ and finds that equality atom $\mathsf{pid}_1 = \mathsf{pid}_2$ in Σ_Q is applicable to o_1. Thus it adds o_3 to $\mathcal{T}$ with $o_3.X = o_1.X \cup \{\mathsf{pid}_2\}$, $o_3.\mathcal{P} = o_1.\mathcal{P}$ and $o_3.c = 1000$. By calling chkComb, o_4 is deduced using rule γ_5, with $o_4.X = o_3.X$, $o_4.\mathcal{P} = o_3.\mathcal{P} \oplus \gamma_5$ ($\oplus$ for appending), and $o_4.c = 1000$.

Note that the parameters of in_album, friends and tagging are covered by $o_1.X$, $o_2.X$ and $o_4.X$, respectively. Hence $o_1.\mathcal{P}$, $o_2.\mathcal{P}$ and $o_4.\mathcal{P}$ tell us how to fetch subsets T_1, T_2 and T_3 from any dataset $D_0 \models \mathcal{A}_0$, 7000 tuples in total. One can verify that T_1, T_2 and T_3 are precisely those described in Example 1. As shown there, we can fetch T_1, T_2 and T_3 from D_0 and compute $Q_0(D_0)$ by using these sets only. □

5.2 Minimum D_Q

One might be tempted to search for a minimum $D_Q \subseteq D$ such that $Q(D) = Q(D_Q)$ under $\mathcal{A}$. More formally, we say that Q is *M-bounded* if for all databases D of schema R, there exists a $D_Q \subseteq D$ such that $|D_Q| \leq M$ and $Q(D) = Q(D_Q)$. It is *effectively M-bounded* if in addition, D_Q can be identified in time independent of $|D|$. These notions were referred to (efficient) scale independence in [20]. The *decision problem* for finding minimum D_Q can be stated as follows:

- Input: R, Q and $\mathcal{A}$, and a natural number M.
- Question: Is Q (effectively) M-bounded under $\mathcal{A}$?

Unfortunately, when M is part of the input, the problem for deciding (effective) boundedness becomes intractable, as opposed to quadratic-time given in Theorems 5 and 6. Algorithms for deciding (effective) M-boundedness are in [5].

Theorem 8: *It is* NP*-complete to decide whether an* SPC *query is (a) M-bounded or (b) effectively M-bounded under an access schema.* □

Proof: We first extend rules in $\mathcal{I}_B$ (resp. $\mathcal{I}_E$) for (resp. effective) M-boundedness. We then give NP checking algorithms based on them, and show the NP-hardness by reductions from Vertex Cover, which is NP-complete [29]. □

6. EXPERIMENTAL STUDY

Using real-life and synthetic data, we conducted two sets of experiments to evaluate (1) the effectiveness of our query evaluation approach based on boundedness, and (2) the efficiency of algorithms BCheck, EBCheck, findDP_h and QPlan.

Experimental setting. We used three datasets: two real-life (TFACC and MOT) and one synthetic (TPCH).

(1) UK traffic accident (TFACC) was obtained by integrating the Road Safety Data [1], which records information about road accidents that happened in the UK from 1979 to 2005, and the National Public Transport Access Nodes data (NaPTAN) [2], with a fuzzy join on location attributes (latitude, longitude). It has 19 tables with 113 attributes, and over 89.7 million tuples in total. Its size is 21.4GB.

(2) The Ministry of Transport Test data (MOT [3]) records all MOT tests, including the makes and models of vehicles, odometer reading and reasons for failures, in year 2013. To make the data larger, we joined its 5 tables together. It is of 16.2GB size with 36 attributes and over 55 million tuples.

Synthetic data (TPCH) was generated by using TPC-H dbgen [4]. The dataset consisted of 8 relations. We varied the scale factor from 0.25 to 32 (32 by default) with the size of the data varying from 0.25GB to 32GB.

All of the three datasets were stored in MySQL.

Access schema. We manually extracted 84, 27 and 61 access constraints for TFACC, MOT and TPCH, respectively, by examining the size of their active domains and dependencies of their attributes. For example, on TFACC we had (1) $\texttt{date} \rightarrow (\texttt{aid}, 610)$ on relation $R_{\texttt{acc}}$, indicating that at most 610 accidents happened in the UK in a single day from 1979 to 2005; and (2) $\texttt{aid} \rightarrow (\texttt{vid}, 192)$ on $R_{\texttt{veh}}$, *i.e.,* at most 192 vehicles were involved in a single accident from 1979 to 2005. For each $X \rightarrow (Y, N)$ extracted, we built index by (a) creating a table by projecting the data on attributes $X \cup Y$, and (b) building an index on X for the new table, using MySQL. We found it easy to extract access constraints from real-life data as above. There are many more such constraints for our datasets, which we did not use in our tests.

SPC *queries*. We manually designed 45 SPC queries Q on these datasets, 15 for each. The queries vary in the number #-sel of equality atoms in the selection condition σ_C of Q, which is in the range of [4, 8], and the number #-prod of Cartesian products in Q, in the range of [0, 4].

Algorithms. We implemented the following algorithms, all in Python: (1) BCheck (Section 4.1) and EBCheck (Section 4.2) for checking boundedness and effective boundedness, respectively; (2) findDP_h (Section 4.3) to find dominating parameters; (3) QPlan to generate query plans that identify D_Q (Section 5.1), (4) evalDQ, a simple algorithm that evaluates effectively bounded SPC queries Q following the query plans generated by QPlan, *i.e.,* fetching D_Q from D and evaluating Q on D_Q, and (5) MySQL, which directly uses MySQL for query evaluation, with all the indices specified in $\mathcal{A}$.

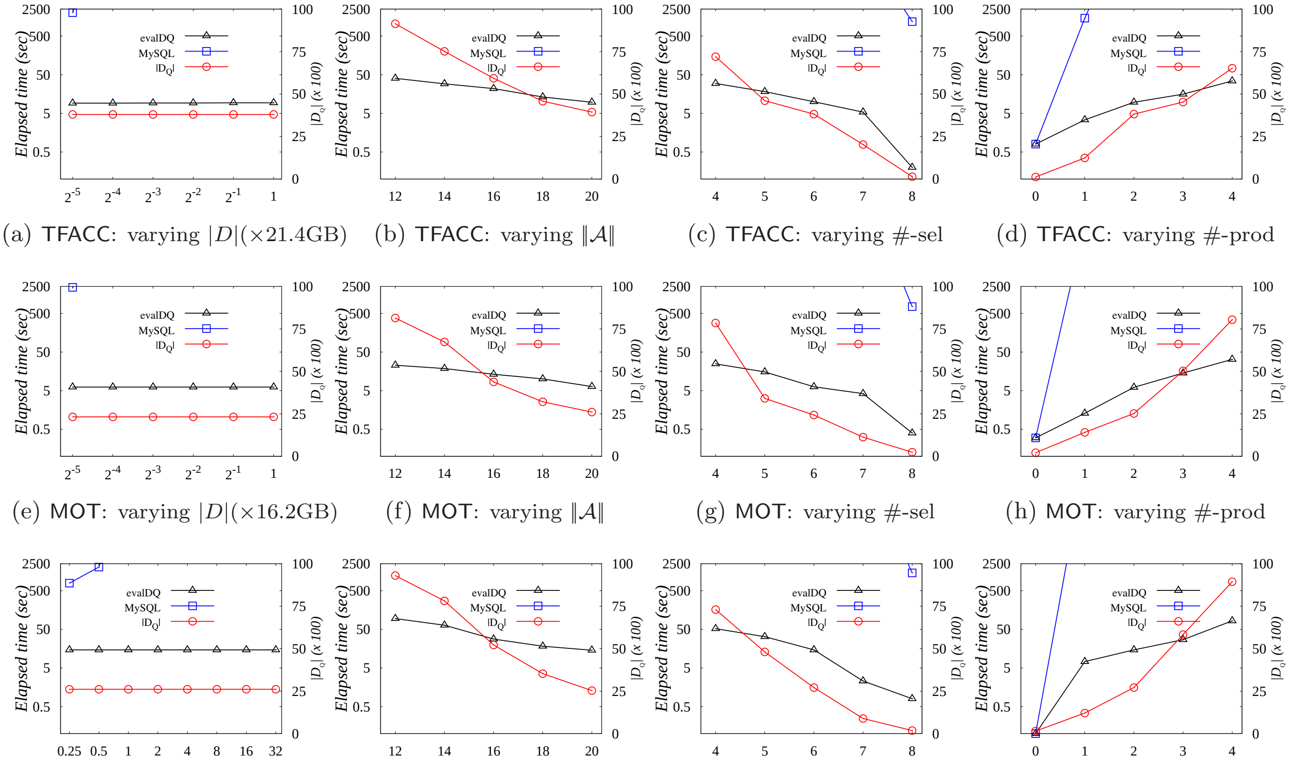
The experiments were conducted on an Amazon EC2 high-memory instance with 17GB memory and 6.5 EC2 compute units. We used MySQL 5.5.35 and MyISAM engine. All the experiments were run 3 times. The average is reported here.

Experimental Results. We next report our findings.

Exp-1: Effectiveness of bounded query evaluation. The first set of experiments evaluated the effectiveness of the bounded query evaluation approach. We first examined the queries generated by using algorithm EBCheck. We found that 35 out of 45 queries are effectively bounded under the access schemas, over 77%. We then evaluated the effectiveness of the query plans generated by QPlan, by comparing the running time of evalDQ with its counterpart of MySQL. The results are reported in Figures 5, on datasets TFACC, MOT and TPCH, by varying $|D|$, Q and $\|\mathcal{A}\|$ (we use $\|\mathcal{A}\|$ to denote the number of access constraints in $\mathcal{A}$). In each of them, we report (a) the average evaluation time (*the left y-axis*), and (b) the size $|D_Q|$ of datasets D_Q accessed by evalDQ (*the right y-axis*). Unless stated otherwise, the tests were conducted on all effectively bounded queries, all access constraints, and full-size datasets by default.

(1) Impact of $|D|$. To evaluate the impact of $|D|$, we varied the size of TFACC and MOT by using scale factors from 2^{-5} to 1, and varied TPCH from 0.25GB to 32GB.

The results are shown in Figures 5(a), 5(e) and 5(i), which tell us the following. (1) The evaluation time of evalDQ is *independent of the size of D*. This verifies our analysis in Section 5. (2) MySQL does not scale well with large D. Indeed, evalDQ consistently took 9.3s, 6.2s, 14.7s on TFACC, MOT and TPCH, respectively, no matter how large the parts of the datasets were used. In contrast, MySQL took 2024s, 2367s and 2045s on subsets of TFACC, MOT and TPCH of sizes $2^{-5} \times 21.4$GB, $2^{-5} \times 16.2$GB and 0.5GB, respectively, and *could not* finish its computation within 2500s for *all larger subsets*. For example, MySQL took longer than 14



(a) TFACC: varying $|D|(\times 21.4\text{GB})$ (b) TFACC: varying $\|\mathcal{A}\|$ (c) TFACC: varying #-sel (d) TFACC: varying #-prod

(e) MOT: varying $|D|(\times 16.2\text{GB})$ (f) MOT: varying $\|\mathcal{A}\|$ (g) MOT: varying #-sel (h) MOT: varying #-prod

(i) TPCH: varying $|D|$ (GB) (j) TPCH: varying $\|\mathcal{A}\|$ (k) TPCH: varying #-sel (l) TPCH: varying #-prod

Figure 5: Effectiveness of bounded query evaluation

Algorithm	TFACC	MOT	TPCH
BCheck	0.8s	0.3s	0.5s
EBCheck	0.8s	0.3s	0.5s
findDP_h	0.3s	0.1s	0.2s
QPlan	2.1s	0.9s	1.4s

Table 1: Elapsed Time

hours on the entire TFACC. That is why only a couple of points are reported for MySQL in the figures. That is, even on *the smallest subsets* we tested, MySQL was 10^2 *times slower* than evalDQ, and at least 5.4×10^3 *time slower* on full sized dataset. In fact, *the larger the datasets* are used, *the bigger the gap* between MySQL and evalDQ are. (3) The size $|D_Q|$ of data accessed evalDQ is also *independent on* $|D|$. Indeed, evalDQ accessed 3800, 2320, 2610 tuples on average, on TFACC, MOT and TPCH, respectively, on all subsets.

(2) Impact of $\|\mathcal{A}\|$. To evaluate the impact of access constraints, we varied $\|\mathcal{A}\|$ from 12 to 20 and tested the queries that are effectively bounded. Accordingly we varied the indices used by MySQL. The results are shown in Figures 5(b), 5(f) and 5(j). The results tell us the following. (1) More access constraints help QPlan get better query plans. For example, when 20 access constraints were used, evalDQ took 9.6s, 6.4s and 14.4s for queries on TFACC, MOT and TPCH, respectively, as opposed to 40.4s, 22.8s and 95s with 12 access constraints, although the queries are effectively bounded in both cases. (2) The more constraints are used, the smaller $|D_Q|$ is, as QPlan can find better proofs (query plans) given more options. (3) MySQL did not produce results *in any single test* within 2500s, no matter whether we used more or less indices embedded in access schemas.

(3) Impact of Q. To evaluate the impact of queries, we varied #-sel of Q from 4 to 8, and #-prod of Q from 0 to 4. We report the average evaluation time of evalDQ and the size $|D_Q|$ for all queries with the same #-sel or #-prod, in Figures 5(c), 5(g) and 5(k), and Figures 5(d), 5(h) and 5(l), respectively. They tell us the following. (1) The complexity of Q has impacts on the quality of query plans generated by QPlan. The larger #-sel or the smaller #-prod is, the better the evaluation time of evalDQ and the size $|D_Q|$ of data accessed by evalDQ are, as expected. (2) Algorithm evalDQ scales well with #-sel and #-prod. It finds answers in all cases within 90s, on the three full datasets. (3) MySQL is indifferent to #-sel. But it is sensitive to #-prod: it is as fast as evalDQ when #-prod $= 0$, *i.e.*, when there is *no* Cartesian product at all; but it cannot stop within 2500s for queries even with 1 Cartesian product, except one case of TFACC.

To understand the gap in performance between MySQL and ours, we examined the system logs and found the following. Given an access constraint $X \to (Y, N)$ on a relation R, evalDQ fetched only relevant (X, Y) attribute values; in contrast, MySQL fetched entire tuples with irrelevant attributes of R, even with the index on X; this led to duplicated (X, Y) values, and the duplications got rapidly inflated by Cartesian product; hence the gap in performance.

Exp-2: Efficiency. The second set of experiments evaluated the efficiency of our algorithms BCheck, EBCheck, findDP_h and QPlan on queries and access schemas for each of TFACC, MOT and TPCH. We used all access constraints, and report in Table 1 *the longest elapsed time* of each algorithm on all queries for each dataset. These results verify that all of our algorithms are efficient: for all queries, all of

Problem	M is not predefined	M is part of input
$\mathsf{Bnd}(Q, \mathcal{A})$	$O(\|Q\|(\|\mathcal{A}\| + \|Q\|))$ (Th 5)	NP-complete (Th 8)
$\mathsf{EBnd}(Q, \mathcal{A})$	$O(\|Q\|(\|\mathcal{A}\| + \|Q\|))$ (Th 6)	NP-complete (Th 8)
$\mathsf{DP}(Q, \mathcal{A})$	NP-complete (Th 7)	NP-complete [5]
$\mathsf{MDP}(Q, \mathcal{A})$	NPO-complete (Th 7)	NPO-complete [5]

Table 2: Complexity bounds

our algorithms took no more than 2.1 seconds, even QPlan, the one with the highest complexity (see Section 5). These confirm our complexity analyses of these algorithms.

Summary. From the experimental results we find the following. (1) The notion of effective boundedness is practical. It is rather easy to find sufficiently many access constraints in real-life data, and many practical queries are actually effectively bounded. (2) The bounded query evaluation approach allows us to query big data. Its evaluation time and amount of data accessed are *independent of* the size of the underlying dataset. For example, on a real-life dataset of 21.4GB, evalDQ finds answers to queries in 9.3 seconds by accessing no more than 3800 tuples on average. In contrast, MySQL is unable to get answers within 2500 seconds in almost all of the cases except for extremely restricted queries (without Cartesian products). Even on a dataset of $2^{-4} \times 21.4$GB (1.3GB), it took longer than 3 hours. The gap between evalDQ and MySQL is *more substantial* on larger datasets. (3) Our algorithms are efficient: they are able to check (effective) boundedness, identify dominating parameters, and generate query plans in 2.1 seconds for queries defined on large schemas and a variety of access constraints.

7. CONCLUSION

We have studied (effective) boundedness for SPC, a class of queries that are widely used in practice (cf. [23]). We have investigated fundamental problems to characterize what SPC query Q can be evaluated under an access schema $\mathcal{A}$, and to make Q effectively bounded under $\mathcal{A}$ by identifying a minimum set of parameters to instantiate. We have established their complexity bounds, as summarized in Table 2. We have also developed efficient (heuristic) algorithms to make practical use of effective boundedness. Our experimental results have verified that effective boundedness yields a promising approach to querying big data.

The study is still in its infancy. (1) It is undecidable to decide whether an $\mathcal{RA}$ query is (effectively) bounded [20]. Nonetheless, we can still find efficient heuristic algorithms to check the effective boundedness of $\mathcal{RA}$. (2) Given a set of parameterized queries, we want to study how to build an *optimal* access schema under which the queries are effectively bounded. (3) When a query is not effectively bounded, it may be effectively bounded *incrementally* or *using views*. A preliminary study of these issues has been reported in [11, 20]. However, effective algorithms remain to be developed.

Acknowledgment. Cao, Fan and Yu are supported in part by 973 Programs 2014CB340302 and 2012CB316200, NSFC 61133002, Guangdong Innovative Research Team Program 2011D005, Shenzhen Peacock Program 1105100030834361, China, and EPSRC EP/J015377/1, UK. Cao is also supported by the Fundamental Research Funds for the Central Universities YWF-14-RSC-018, China.

8. REFERENCES

[1] *http://data.gov.uk/dataset/road-accidents-safety-data*.
[2] *http://data.gov.uk/dataset/naptan*.
[3] *http://data.gov.uk/dataset/anonymised_mot_test*.
[4] *http://www.tpc.org/tpch/*.
[5] Full paper. *http://homepages.inf.ed.ac.uk/s1165433/bounded.pdf*.
[6] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
[7] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *SIGMOD*, 1999.
[8] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.
[9] M. Armbrust, K. Curtis, T. Kraska, A. Fox, M. J. Franklin, and D. A. Patterson. PIQL: Success-tolerant query processing in the cloud. In *VLDB*, 2011.
[10] 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.
[11] M. Armbrust, E. Liang, T. Kraska, A. Fox, M. J. Franklin, and D. Patterson. Generalized scale independence through incremental precomputation. In *SIGMOD*, 2013.
[12] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. In *FOCS*, 2008.
[13] G. Ausiello. *Complexity and approximation: Combinatorial optimization problems and their approximability properties*. Springer, 1999.
[14] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *SIGMOD*, 2003.
[15] V. Bárány, M. Benedikt, and P. Bourhis. Access patterns and integrity constraints revisited. In *ICDT*, 2013.
[16] A. Deutsch, B. Ludäscher, and A. Nash. Rewriting queries using views with access patterns under integrity constraints. *TCS*, 371(3), 2007.
[17] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, 1996.
[18] Facebook. *http://newsroom.fb.com*.
[19] Facebook. Constraints on the number of photos, friends and tags. *https://www.facebook.com/help* {*/227794810567981*, */community/question/?id=364005057055660*}, 2014.
[20] W. Fan, F. Geerts, and L. Libkin. On scale independence for querying big data. In *PODS*, 2014.
[21] W. Fan, F. Geerts, and F. Neven. Making queries tractable on big data with preprocessing. In *VLDB*, 2013.
[22] M. N. Garofalakis and P. B. Gibbons. Wavelet synopses with error guarantees. In *SIGMOD*, 2004.
[23] G. Gottlob, S. T. Lee, and G. Valiant. Size and treewidth bounds for conjunctive queries. In *PODS*, 2009.
[24] Y. E. Ioannidis and V. Poosala. Histogram-based approximation of set-valued query-answers. In *VLDB*, 1999.
[25] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *VLDB*, 2009.
[26] P. C. Kanellakis. On the computational complexity of cardinality constraints in relational databases. *Inf. Process. Lett.*, 11(2):98–101, 1980.
[27] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley, 1990.
[28] C. Li. Computing complete answers to queries in the presence of limited access patterns. *VLDB J.*, 12(3), 2003.
[29] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
[30] P. Rösch and W. Lehner. Sample synopses for approximate answering of group-by queries. In *EDBT*, 2009.