Querying Big Social Data

Wenfei Fan [⋆]
University of Edinburgh and Beihang University

Abstract. Big data poses new challenges to query answering, from computational complexity theory to query evaluation techniques. Several questions arise. What query classes can be considered tractable in the context of big data? How can we make query answering feasible on big data? What should we do about the quality of the data, the other side of big data? This paper aims to provide an overview of recent advances in tackling these questions, using social network analysis as an example.

1 Introduction

Big data refers to data that cannot be processed or analyzed using traditional processes or tools, *e.g.,* when the volume of data is "big" such as in PetaByte (PB, 10^{15} bytes) or ExaByte (EB, 10^{18} bytes). As an example, let us consider social networks, which are typically modeled as graphs. In such a graph, a node denotes a person, carrying attributes such as label, keywords, blogs, comments, rating. Its edges indicate relationships such as marriage, friendship, co-work, advise, support and recommendation. Social graphs are often "big". For example, Facebook has more than 1 billion users with 140 billion links[1].

Big data introduces challenges to query answering. As an example, consider graph pattern matching, which is commonly used in social network analysis. Given a social graph G and a pattern query Q, *graph pattern matching* is to find the set $M(Q, G)$ of all matches for Q in G, as illustrated below.

Example 1. Consider the structure of a drug trafficking organization [30], depicted as a graph pattern Q_0 in Fig. 1. In such an organization, a "boss" (`B`) oversees the operations through a group of assistant managers (`AM`). An `AM` supervises a hierarchy of low-level field workers (`FW`), up to 3 levels as indicated by the edge label 3. The `FW`s deliver drugs, collect cash and run other errands. They report to `AM`s directly or indirectly, while the `AM`s report directly to the boss. The boss may also convey messages through a secretary (`S`) to the top-level `FW`s as denoted by the edge label 1. A drug ring G_0 is also shown in Fig. 1 in which $A_1, \ldots, A_m$ are `AM`s, while A_m is both an `AM` and the secretary (`S`).

To identify all suspects in the drug ring, we want to find matches $M(Q_0, G_0)$ for Q_0 in G_0. Here graph pattern matching is traditionally defined as follows:

(1) subgraph isomorphism [35]: $M(Q_0, G_0)$ is the set of all subgraphs G' of G_0 isomorphic to Q_0, *i.e.,* there exists a *bijective function* h from the nodes of Q_0 to those of G' such that (u, u') is an edge in Q_0 iff $(h(u), h(u'))$ is an edge in G'; or

⋆ Fan is spported in part by EPSRC EP/J015377/1, UK, the RSE-NSFC Joint Project Scheme, and the 973 Program 2012CB316200 and NSFC 61133002 of China

1 *http://www.facebook.com/press/info.php?statistics*

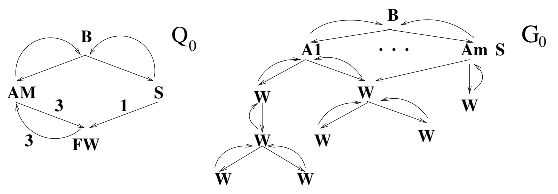

Fig. 1. Drug trafficking: Pattern and social graphs

(2) graph simulation [28]: $M(Q_0, G_0)$ is the maximum binary *relation* $S \subseteq V_Q \times V$, where V_Q and V are the set of nodes in Q_0 and G_0, respectively, such that

– for each node u in V_Q, there exists a node v in V such that $(u, v) \in S$, and
– for each pair $(u, v) \in S$ and each edge (u, u') in Q, there exists an edge (v, v') in G such that $(u', v') \in S$. □

No matter whether graph pattern matching is defined in terms of subgraph isomorphism or graph simulation, it involves *data-intensive* computation when graph G is "big". To develop effective algorithms for computing the set $M(Q, G)$ of matches for Q in big G, we need to answer the following questions.

(1) What query classes are tractable on big data? A class $\mathcal{Q}$ of queries is traditionally considered *tractable* if there exists an algorithm for answering its queries in time bounded by a polynomial (PTIME) in the size of the input, *e.g.,* a social graph and a pattern query [1]. That is, $\mathcal{Q}$ is considered *feasible* if its worst-case time complexity is PTIME. For graph pattern queries, it is NP-complete to determine whether there exists a match for Q in G when matching is defined with subgraph isomorphism, and it takes $O(|Q|^2 + |Q||G| + |G|)^2$ time to compute $M(Q, G)$ with graph simulation [21]. As will be seen shortly, however, PTIME or even linear-time algorithms are often beyond reach in the context of big data! This suggests that we revise the traditional notion of tractable queries, so that we can decide, given $\mathcal{Q}$, whether it is feasible to evaluate the queries of $\mathcal{Q}$ on big data.

(2) How can we make query answering feasible on big data? When a query class $\mathcal{Q}$ is not tractable on big data, we may be able to transform $\mathcal{Q}$ to an "equivalent" class $\mathcal{Q}'$ of queries that operate on smaller datasets. That is, we reduce the big data for $\mathcal{Q}$ to "small data" for $\mathcal{Q}'$, such that it is feasible to answer the queries of $\mathcal{Q}$. When querying big data, one often thinks of MapReduce [7] and Hadoop[2]. Nevertheless, MapReduce and Hadoop are not the only way to query big data. We will see that this is the case for graph pattern matching with simulation.

(3) In the context of big data it is often cost-prohibitive to compute exact answers to our queries. That is, algorithms for querying big data are often necessarily *inexact.* This is particularly evident when we want to find matches for our patterns in big social graphs based on subgraph isomorphism. Hence we may have to settle with *heuristics*, "quick and dirty" algorithms which return feasible answers. To this end, we naturally want *approximation algorithms*, *i.e.,* heuristics which find answers that are guaranteed to be not far from the exact answers [6, 36]. However, traditional approximation algorithms are mostly PTIME algorithms for NP

[2] *http://hadoop.apache.org/*

optimization problems (NPOs). In contrast, we need approximation algorithms for answering queries on big data rather than for NPOs, even when the queries are known in PTIME, such that the algorithms are tractable on big data.

(4) When we talk about the challenges introduced by big data, we often refer to the difficulty of coping with the sheer size of the data only. Nonetheless, the quality of the data is as important and challenging as its quantity. When the quality of the data is poor, answers to our queries in the data may be *inaccurate* or even *incorrect*! Indeed, one of the dimensions of big data is its *veracity*, "as 1 in 3 business leaders don't trust the information they use to make decisions"[3]. Referring to Example 1, poor data may lead to false accusation against innocents or letting go of real drug dealers. Already challenging even for "small" relational data, data quality management is far more difficult for big data.

This paper provides an overview of recent advances in tackling these questions. We present a revision of tractable query classes in the context of big data [10] (Section 2), and a set of effective techniques beyond MapReduce for graph pattern matching with simulation [12–17, 27] (Section 3). We revisit traditional approximation algorithms for querying big data [5] (Section 4). Finally, we highlight the need for studying the quality of big data (Section 5).

2 Tractable Query Classes on Big Data

We start with an examination of query evaluation on big data, including but not limited to graph pattern matching. To develop algorithms for answering a class $\mathcal{Q}$ of queries on big data, we want to know whether $\mathcal{Q}$ is *tractable*, *i.e.,* whether its queries can be evaluated on the big data within our available resources such as time. Traditionally $\mathcal{Q}$ is considered (a) "good" (tractable) if there exists a PTIME algorithm for evaluating its queries, (b) "bad" (intractable) if it is NP-hard to decide, given a query $Q \in \mathcal{Q}$, a dataset D and an element t, whether $t \in Q(D)$, *i.e.,* t is an answer to Q in D; and (c) "ugly" if the membership problem is EXPTIME-hard. This is, however, no longer the case when it comes to big data.

Example 2. Consider a dataset D of 1PB. Assuming the fastest Solid State Drives (SSD) with disk scanning speed of 6GB/s[4], a linear scan of D will take at least 166,666 seconds or 1.9 days. That is, even linear-time algorithms, a special case of PTIME algorithms, may no longer be feasible on big data! □

There has been recent work on revising the traditional computational complexity theory to characterize data-intensive computation on big data. The revisions are defined in terms of computational costs [10], communication (coordination) rounds [20, 25], or MapReduce steps [24] and data shipments [2] in the MapReduce framework [7]. Here we focus on computational costs [10].

One way to cope with big data is to separate offline and online processes. We preprocess the dataset D by, *e.g.,* building indices or compressing the data, yielding D', such that all queries of $\mathcal{Q}$ on D can be evaluated on D' *online* efficiently.

[3] *http://www-01.ibm.com/software/data/bigdata/*
[4] *http://www.fastestssd.com/featured/ssd-rankings-the-fastest-solid-state-drives/#pcie*

When the data is mostly static or when D' can be maintained efficiently, the preprocessing step can be considered as an *offline* process with a *one-time cost.*

Example 3. Consider a class $\mathcal{Q}_1$ of selection queries. A query $Q_1 \in \mathcal{Q}_1$ on a relation D is to find whether there exists a tuple $t \in D$ such that $t[A] = c$, where A is an attribute of D and c is a constant. A naive evaluation of Q_1 would require a linear scan of D. In contrast, we can first build a B^+-tree on the values of the A column in D, in a one-time preprocessing step offline. Then we can answer *all queries* $Q_1 \in \mathcal{Q}_1$ on D in $O(\mathsf{log}\,|D|)$ time using the indices. That is, we no longer need to scan D when processing *each* query in $\mathcal{Q}_1$. When D consists of 1PB of data, we can get the results in 5 seconds with the indices rather than 1.9 days. $\square$

The idea has been practiced by database people for decades. Following this, below we propose a revision of the traditional notion of tractable query classes.

To be consistent with the complexity classes of decision problems, we consider Boolean queries, such as Boolean selection queries given in Example 3. We represent a class $\mathcal{Q}$ of Boolean queries as a *language* S *of pairs* $\langle D, Q\rangle$, where Q is a query in $\mathcal{Q}$, D is a database on which Q is defined, and $Q(D)$ is true. In other words, S can be considered as a binary relation such that $\langle D, Q\rangle \in S$ if and only if $Q(D)$ is true. We refer to S as *the language for* $\mathcal{Q}$.

We say that a language S of pairs is *in complexity class* $\mathsf{C_Q}$ if it is in $\mathsf{C_Q}$ to decide whether a pair $\langle D, Q\rangle \in S$. Here $\mathsf{C_Q}$ may be the sequential complexity class P or the parallel complexity class NC, among other things. The complexity class P consists of all decision problems that can be solved by a deterministic Turing machine in PTIME. The parallel complexity class NC, *a.k.a.* Nick's Class, consists of all decision problems that can be solved by taking polynomial time in the logarithm of the problem size (parallel $\mathsf{polylog\text{-}time}$) on a PRAM (parallel random access machine) with polynomially many processors (see, *e.g.,* [22, 18]).

Π-tractable queries. Consider complexity classes $\mathsf{C_P}$ and $\mathsf{C_Q}$. We say that a class $\mathcal{Q}$ of queries is in $(\mathsf{C_P}, \mathsf{C_Q})$ if there exist a $\mathsf{C_P}$-computable preprocessing function Π and a language S' of pairs such that for all datasets D and queries $Q \in \mathcal{Q}$,

– $\langle D, Q\rangle$ is in the language S of pairs for $\mathcal{Q}$ if and only if $\langle \Pi(D), Q\rangle \in S'$, and
– S' is in $\mathsf{C_Q}$, *i.e.,* the language S' of pairs $\langle \Pi(D), Q\rangle$ is in $\mathsf{C_Q}$.

Intuitively, function $\Pi(\cdot)$ preprocesses D and generates another structure $D' = \Pi(D)$ offline, in $\mathsf{C_P}$. After this, for *all queries* $Q \in \mathcal{Q}$ that are defined on D, $Q(D)$ can be answered by evaluating $Q(D')$ online, in $\mathsf{C_Q}$. Here $\mathsf{C_P}$ indicates the cost we can afford for preprocessing, and $\mathsf{C_Q}$ the cost of online query processing. Depending on $D' = \Pi(D)$, we may let $\mathsf{C_Q}$ be P if D' is sufficiently small such that PTIME evaluation of $Q(D')$ is feasible, *i.e.,* if $\Pi(D)$ reduces big data D to "small data" D'. Otherwise we may choose NC for $\mathsf{C_Q}$, in parallel $\mathsf{polylog\text{-}time}$.

We use $\mathsf{\Pi T^0_Q}$ to denote the set of all $(\mathsf{P}, \mathsf{NC})$ query classes, referred to as the set of Π-*tractable* query classes, *i.e.,* when $\mathsf{C_P}$ is P and $\mathsf{C_Q}$ is NC. We are particularly interested in $\mathsf{\Pi T^0_Q}$ for the following reasons. (a) As shown in Example 3, parallel $\mathsf{polylog\text{-}time}$ is feasible on big data. Moreover, NC is robust and well-understood. It is one of the few parallel complexity classes whose connections with classical sequential complexity classes have been well studied (see, *e.g.,* [18]). Further, a large class of NC algorithms can be implemented in the MapReduce framework, which is widely used in cloud computing and data centers for processing big data, such that if an NC algorithm takes t time, than its corresponding MapReduce counterpart takes $O(t)$ MapReduce rounds [24]. (b) We consider PTIME preprocessing feasible since it is a *one-time* price and is performed *off-line*. In addition, P is robust and well-studied [18]. Moreover, by requiring that $\Pi(\cdot)$ is in PTIME, the size of the preprocessed data $\Pi(D)$ is bounded by a polynomial. When PTIME is too big a price to pay, we may preprocess D with parallel processing, by allocating more resources (*e.g.,* computing nodes) to it than to online query answering. Here we simply use PTIME for $\mathsf{C_P}$ to focus on the main properties of query answering with preprocessing.

Example 4. As shown in Example 3, query class $\mathcal{Q}_1$ is in $\mathsf{\Pi T_Q^0}$. Indeed, function $\Pi(\cdot)$ preprocesses D by building B^+-trees on attributes of D in PTIME. After this, for any (A, c) denoting a query in $\mathcal{Q}_1$, whether there exists $t \in D$ such that $t[A] = c$ can be decided in $O(\mathsf{log}|D|)$ time by using the indices $\Pi(D)$. □

Making query classes $\mathsf{\Pi}$-tractable. Many query classes $\mathcal{Q}$ are not $\mathsf{\Pi}$-tractable. For instance, unless $\mathsf{P} = \mathsf{NC}$, we are not aware of any NC algorithm for graph pattern matching even when matching is defined in terms of graph simulation. Nonetheless, some $\mathcal{Q}$ that is not in $\mathsf{\Pi T_Q^0}$ can actually be transformed to a $\mathsf{\Pi}$-tractable query class by means of *re-factorizations*, which re-partition the data and query parts of $\mathcal{Q}$ and identify a data set for preprocessing, such that after the preprocessing, its queries can be subsequently answered in parallel $\mathsf{polylog\text{-}time}$.

More specifically, we say that a class $\mathcal{Q}$ of queries *can be made* $\mathsf{\Pi}$*-tractable* if there exist three NC computable functions $\pi_1(\cdot)$, $\pi_2(\cdot)$ and $\rho(\cdot,\cdot)$ such that for all $\langle D, Q\rangle$ in the language S of pairs for $\mathcal{Q}$,

- $D' = \pi_1(D, Q)$, $Q' = \pi_2(D, Q)$, and $\langle D, Q\rangle = \rho(D', Q')$, and
- the query class $\mathcal{Q}' = \{Q' \mid Q' = \pi_2(D, Q), \langle D, Q\rangle \in S\}$ is $\mathsf{\Pi}$-tractable.

Intuitively, $\pi_1(\cdot)$ and $\pi_2(\cdot)$ re-partition $x = \langle D, Q\rangle$ into a "data" part $D' = \pi_1(x)$ and a "query" part $Q' = \pi_2(x)$, and ρ is an inverse function that restores the original instance x from $\pi_1(x)$ and $\pi_2(x)$. Then the data part D' can be preprocessed such that the queries $Q' \in \mathcal{Q}'$ can then be answered in parallel $\mathsf{polylog\text{-}time}$. We denote by $\mathsf{\Pi T_Q}$ the set of all query classes that can be *made* $\mathsf{\Pi}$-tractable.

A form of NC-reductions $\leqslant_{fa}^{\mathsf{NC}}$ is defined for $\mathsf{\Pi T_Q}$, which is transitive (*i.e.,* if $\mathcal{Q}_1 \leqslant_{fa}^{\mathsf{NC}} \mathcal{Q}_2$ and $\mathcal{Q}_2 \leqslant_{fa}^{\mathsf{NC}} \mathcal{Q}_3$ then $\mathcal{Q}_1 \leqslant_{fa}^{\mathsf{NC}} \mathcal{Q}_3$) and compatible with $\mathsf{\Pi T_Q}$ (*i.e.,* if $\mathcal{Q}_1 \leqslant_{fa}^{\mathsf{NC}} \mathcal{Q}_2$ and $\mathcal{Q}_2$ is in $\mathsf{\Pi T_Q}$, then so is $\mathcal{Q}_1$). The following results are known [10]:

- $\mathsf{NC} \subseteq \mathsf{\Pi T_Q^0} \subseteq \mathsf{P}$.
- Unless $\mathsf{P} = \mathsf{NC}$, $\mathsf{\Pi T_Q^0} \subset \mathsf{P}$, *i.e.,* not all PTIME queries are $\mathsf{\Pi}$-tractable.
- There exists a *complete query class* $\mathcal{Q}$ for $\mathsf{\Pi T_Q}$ under $\leqslant_{fa}^{\mathsf{NC}}$ reductions, *i.e.,* $\mathcal{Q}$ is in $\mathsf{\Pi T_Q}$ and moreover, for all query classes $\mathcal{Q}' \in \mathsf{\Pi T_Q}$, $\mathcal{Q}' \leqslant_{fa}^{\mathsf{NC}} \mathcal{Q}$.
- All query classes in P can be made $\mathsf{\Pi}$-tractable by transforming them to a query class in $\mathsf{\Pi T_Q}$ via $\leqslant_{fa}^{\mathsf{NC}}$ reductions.

3 Graph Pattern Matching in Big Social Data

We now study how to compute matches $M(Q, G)$ for a pattern Q in a big social graph G. We focus on matching defined in terms of graph simulation in this section, which is widely used in social data analysis such as detecting social communities and positions [4, 33]. As remarked earlier, it takes $O(|Q|^2 + |Q||G| + |G|)^2$ time to compute $M(Q, G)$ [21], a prohibitive cost when G is big. Nonetheless, we can leverage a variety of techniques commonly used by database people to reduce G to G' of smaller size via preprocessing, such that $M(Q, G')$ can subsequently be computed effectively for all patterns Q. Combinations of these techniques outperform direct implementation of simulation algorithms in MapReduce.

We first introduce a revision of graph simulation [28] for social data analysis (Section 3.1). We then present a set of matching techniques (Sections 3.2–3.6).

3.1 Bounded Simulation: Graph Simulation Revisited

Recall Example 1: we want to identify suspects involved in a drug ring by computing matches $M(Q_0, G_0)$ for pattern Q_0 in graph G_0. However, observe the following. (1) Nodes `AM` and `S` in Q_0 should be mapped to the *same* node A_m in G_0, which is not allowed by a bijection. (2) The node `AM` in Q_0 corresponds to *multiple* nodes $A_1, \ldots, A_m$ in G_0. This relationship cannot be captured by a function from the nodes of Q_0 to the nodes of G_0. (3) The edge from `AM` to `FW` in Q_0 indicates that an `AM` supervises `FW`s within 3 hops. It should be mapped to a *path* of a bounded length in G_0 rather than to an *edge*. Hence, neither subgraph isomorphism (for (1)–(3)) nor graph simulation (for (3)) is capable of identifying the drug ring G_0 as a match of Q_0. These call for revisions of the notion of graph pattern matching to accurately identify sensible matches in real-life social graphs.

To cope with this, a revision of graph simulation is proposed in [12], referred to as *bounded simulation*. To present this, we start with some notations.

Graphs and patterns. A *data graph* is a directed graph $G = (V, E, f_A)$, where (a) V is a finite set of nodes; (b) $E \subseteq V \times V$, in which (v, v') denotes an edge from v to v'; and (c) $f_A(\cdot)$ is a function that associates each v in V with a tuple $f_A(v) = (A_1 = a_1, \ldots, A_n = a_n)$, where a_i is a constant, and A_i is referred to as an *attribute* of v, written as $v.A_i$, carrying, *e.g.,* label, keywords, blogs, rating.

A *pattern query* is defined as $Q = (V_Q, E_Q, f_v, f_e)$, where (a) V_Q is a finite set of nodes and E_Q is a set of directed edges, as defined for data graphs; (b) $f_v(\cdot)$ is a function defined on V_Q such that for each node u, $f_v(u)$ is the *predicate* of u, defined as a conjunction of atomic formulas of the form A `op` a; here A denotes an attribute, a is a constant, and `op` is one of the comparison operators $<, \leq, =, \neq, >, \geq$; and (c) $f_e(\cdot)$ is a function defined on E_Q such that for each edge (u, u') in E_Q, $f_e(u, u')$ is either a positive integer k or a symbol $*$.

Intuitively, the predicate $f_v(u)$ of a node u specifies a *search condition*. We say that a node v in a data graph G *satisfies* the search condition of a pattern node u in Q, denoted as $v \sim u$, if for each atomic formula 'A `op` a' in $f_v(u)$, there exists an attribute A in $f_A(v)$ such that $v.A$ `op` a. We will allow an edge (u, u') in Q to be mapped to a path ρ in a data graph G, and $f_e(u, u')$ imposes a bound on the length of ρ. An example data graph (resp. pattern) is G_0 (resp. Q_0) of Fig. 1.

Bounded simulation. We now present bounded simulation. A data graph G *matches* a pattern Q via *bounded simulation*, denoted by $Q \trianglelefteq^{\mathsf{B}}_{\mathsf{sim}} G$, if there exists a binary relation $S \subseteq V_Q \times V$, referred to as a *match* in G for Q, such that

- for each node $u \in V_Q$, there exists a node $v \in V$ such that $(u, v) \in S$;
- for each pair $(u, v) \in S$, (a) $v \sim u$, and (b) for each edge (u, u') in E_Q, there exists a *path* ρ from v to v' in G such that $(u', v') \in S$, $\mathsf{len}(\rho) > 0$ and moreover, $\mathsf{len}(\rho) \leq k$ if $f_e(u, u') = k$. Here $\mathsf{len}(\rho)$ is the number of edges on ρ.

Intuitively, $(u, v) \in S$ if (1) node v in G satisfies the search condition specified by $f_v(u)$ in Q; and (2) each edge (u, u') in Q is mapped to a *path* ρ from v to v' in G ($\mathsf{len}(\rho) > 0$), such that v, v' match u, u', respectively; and moreover, when $f_e(u, u')$ is k, it indicates a bound on the length of ρ, *i.e.,* v is connected to v' within k hops. When it is $*$, ρ can be a path of an arbitrary length greater than 0.

For pattern Q_0 and graph G_0 given in Fig. 1, $Q_0 \trianglelefteq^{\mathsf{B}}_{\mathsf{sim}} G_0$: a match S_0 in G_0 for Q_0 maps `B` to `B`, `AM` to $A_1, \ldots, A_m$, `S` to A_m, and `FW` to all the `W` nodes.

As experimentally verified in [12], bounded simulation is able to accurately identify a number of communities in real-life social networks that its traditional counterparts fail to catch. In addition, the following is known.

Theorem 1 [12]: For any pattern $Q = (V_Q, E_Q, f_v, f_e)$ and graph $G = (V, E, f_A)$, (1) there exists a unique maximum match $M(Q, G)$ in G for Q, and (2) $M(Q, G)$ can be computed in $O(|V||E| + |E_Q||V|^2 + |V_Q||V|)$ time. $\Box$

As opposed to subgraph isomorphism, bounded simulation supports (a) simulation relations rather than bijective functions, (b) search conditions based on the contents of nodes, and (c) edge-to-path mappings instead of edge-to-edge. Graph simulation is a special case of bounded simulation, by only allowing simple patterns in which (a) node labels are the only attributes, and (b) all the edges are labeled with 1, *i.e.,* edge-to-edge mappings only. In contrast to the NP-hardness of subgraph isomorphism, the complexity of bounded simulation is in PTIME, comparable to that of graph simulation since in practice, $|Q| \ll |D|$.

There have also been revisions of (bounded) simulation by, *e.g.,* incorporating edge relationships [11] and imposing locality and duality on match relations [27].

3.2 Distributed Query Processing with Partial Evaluation

Although graph pattern matching with (bounded) simulation is in PTIME, when a social graph G is big, the cost of computing $M(Q, G)$ is still prohibitive. To cope with the sheer size of G, we next present a set of approaches to computing $M(Q, G)$ on big G. The key idea of these approaches is to reduce G to smaller G' via preprocessing, such that graph pattern matching in G' is feasible.

We start with distributed query processing, based on partial evaluation. Partial evaluation has proven useful in a variety of areas including compiler generation, code optimization and dataflow evaluation (see [23] for a survey). Intuitively, given a function $f(s, d)$ and part of its input s, partial evaluation is to specialize $f(s, d)$ with respect to the known input s. That is, it conducts the part of $f(s, \cdot)$'s computation that depends only on s, and generates a partial answer, *i.e.,* a residual function $f'(\cdot)$ that depends on the as yet unavailable input d.

This idea can be naturally applied to distributed graph pattern matching. Consider a pattern Q posed on a graph G that is partitioned into fragments $\mathcal{F} = (F_1, \ldots, F_n)$, where F_i is stored in site S_i. We compute $M(Q, G)$ as follows.

(1) The same query Q is posted to each fragment in $\mathcal{F}$.
(2) Upon receiving Q, each site S_i computes a *partial answer* of Q in fragment F_i, *in parallel*, by taking F_i as the known input s while treating the fragments in the other sites as yet unavailable input d.
(3) A coordinator site S_c collects partial answers from all the sites. It then assembles the partial answers and finds $M(Q, G)$ in the entire graph G.

The idea has proven effective for evaluating reachability queries defined in terms of regular expressions, which are a special case of pattern queries [15].

Theorem 2 [15]: On a fragmentation $\mathcal{F}$ of graph G, reachability queries Q can be answered (a) by visiting each site once, (b) in $O(|F_m||Q|^2 + |Q|^2|V_f|^2)$ time, and (c) with $O(|Q|^2|V_f|^2)$ communication cost, where F_m is the largest fragment in $\mathcal{F}$ and V_f is the set of nodes in G with edges to other fragments. $\Box$

That is, (1) the response time is dominated by the largest fragment in $\mathcal{F}$, *instead of* the size $|G|$ of G; (2) the total amount of data shipped is determined by the size of the query Q and how G is fragmented, *rather than* by $|G|$, and (3) the performance guarantees remain intact no matter how G is fragmented and distributed. As opposed to MapReduce [7], this approach does not require us to organize our data in $\langle \mathsf{key}, \mathsf{value} \rangle$ pairs or re-distribute the data. Moreover, it has performance guarantees on both response time and communication cost.

When G is not already partitioned and distributed, one may first partition G as preprocessing, such that the evaluation of Q in each fragment is feasible.

3.3 Query Preserving Graph Compression

Another approach to reducing the size of big graph G is by means of compressing G, relative to a class $\mathcal{Q}$ of queries of users' choice, *e.g.,* graph pattern queries. More specifically, a *query preserving graph compression* for $\mathcal{Q}$ is a pair $\langle R, P \rangle$, where $R(\cdot)$ is a *compression function*, and $P(\cdot)$ is a *post-processing function.* For any graph G, $G_c = R(G)$ is the *compressed graph* computed from G by $R(\cdot)$, such that (1) $|G_c| \leq |G|$, and (2) *for all queries* $Q \in \mathcal{Q}$, $Q(G) = P(Q(G_c))$. Here $P(Q(G_c))$ is the result of post-processing the answers $Q(G_c)$ to Q in G_c.

That is, we preprocess G by computing the compressed G_c of G offline. After this step, for any query $Q \in \mathcal{Q}$, the answers $Q(G)$ to Q in the big G can be computed by evaluating the same Q on the smaller G_c online. Moreover, $Q(G_c)$ can be computed *without decompressing* G_c. Note that the compression schema is *lossy*: we do not need to restore the original G from G_c. That is, G_c only needs to retain the information necessary for answering queries in $\mathcal{Q}$, and hence achieves *better* compression ratio than lossless compression schemes.

For a query class $\mathcal{Q}$, if G_c can be computed in PTIME and moreover, queries in $\mathcal{Q}$ can be answered using G_c in parallel $\mathsf{polylog\text{-}time}$, perhaps by combining with other techniques such as indexing, then $\mathcal{Q}$ is Π-tractable.

The effectiveness of the approach has been verified in [14], for graph pattern matching with (bounded) simulation, and for reachability queries a special case.

Theorem 3 [14]: There exists a graph pattern preserving compression $\langle R, P\rangle$ for bounded simulation, such that for any graph $G = (V, E, f_A)$, $R(\cdot)$ is in $O(|E| \log |V|)$ time, and $P(\cdot)$ is in linear time in the size of the query answer. $\square$

This compression scheme reduces the sizes of real-life social graphs by 98% and 57%, and query evaluation time by 94% and 70% on average, for reachability queries and pattern queries with (bounded) simulation, respectively. Better still, compressed G_c can be efficiently maintained. Given a graph G, a compressed graph $G_c = R(G)$ of G, and updates ΔG to G, we can compute changes ΔG_c to G_c such that $G_c \oplus \Delta G_c = R(G \oplus \Delta G)$, *without decompressing* G_c [14]. As a result, for each graph G, we need to compute its compressed graph G_c *once* for *all patterns*. When G is updated, G_c is incrementally maintained.

3.4 Graph Pattern Matching Using Views

Another technique commonly used by database people is query answering using views (see [26, 19] for surveys). Given a query $Q \in \mathcal{Q}$ and a set $\mathcal{V}$ of view definitions, *query answering using views* is to reformulate Q into another query Q' such that (a) Q and Q' are equivalent, *i.e.,* for all datasets D, Q and Q' have the same answers in D, and moreover, (b) Q' refers only to $\mathcal{V}$ and its extensions $\mathcal{V}(D)$.

View-based query answering suggests another approach to reducing big data to small data. Given a big graph G, one may identify a set $\mathcal{V}$ views (pattern queries) and materialize the set $M(\mathcal{V}, G)$ of matches for patterns of $\mathcal{V}$ in G, as a preprocessing step offline. Then matches for patterns Q can be computed online by using $M(\mathcal{V}, G)$ only, *without accessing the original big* G. In practice, $M(\mathcal{V}, G)$ is typically much smaller than G, and can be incrementally maintained and adaptively adjusted to cover various patterns. For example, for graph pattern matching with bounded simulation, $M(\mathcal{V}, G)$ is no more than 4% of the size of G on average for real-life social graphs G. Further, the following has been shown [17].

Theorem 4 [17]: Given a graph pattern Q and a set $\mathcal{V}$ of view definitions, (1) it is in $O(|Q|^2|\mathcal{V}|)$ time to decide whether Q can be answered by using $\mathcal{V}$; and if so, (2) Q can be answered in $O(|Q||M(\mathcal{V}, G)| + |M(\mathcal{V}, G)|^2)$ time. $\square$

Contrast these with the complexity of graph pattern matching with bounded simulation. Note that $|Q|$ and $|\mathcal{V}|$ are sizes of pattern queries and are typically much smaller than G. Moreover, $|M(\mathcal{V}, G)|$ is about 4% of $|G|$ (*i.e.,* $|V| + |E|$) on average. As verified in [17], graph pattern matching using views takes no more than 6% of the time needed for computing $M(Q, G)$ directly in G on average.

3.5 Incremental Graph Pattern Matching

Incremental techniques also allow us to effectively evaluate queries on big data. Given a pattern Q and a graph G, as preprocessing we compute $M(Q, G)$ once. When G is updated by ΔG, instead of recomputing $M(Q, G \oplus \Delta G)$ starting from scratch, we incrementally compute ΔM such that $M(Q, G \oplus \Delta G) = M(Q, G) \oplus \Delta M$, to minimize unnecessary recomputation. In real life, ΔG is typically small: only 5% to 10% of nodes are updated weekly [31]. When ΔG is small, ΔM is often small as well, and is much less costly to compute than $M(Q, G \oplus \Delta G)$.

The benefit is more evident if there exists a bounded incremental matching algorithm. As argued in [32], incremental algorithms should be analyzed in terms of $|\mathsf{CHANGED}| = |\Delta G| + |\Delta M|$, the size of changes in the input and output, which represents the updating costs that are *inherent to* the incremental problem itself. An incremental algorithm is said to be *semi-bounded* if its cost can be expressed as a polynomial of $|\mathsf{CHANGED}|$ and $|Q|$ [13]. That is, its cost depends only on the size of *the changes* and the size of *pattern* Q, *independent of* the size of big graph G. A semi-bounded incremental algorithm often reduces big graph G to small data, since Q and $|\mathsf{CHANGED}|$ are typically small in practice.

Theorem 5 [13]: There exists a semi-bounded incremental algorithm, in $O(|\Delta G|(|Q||\mathsf{CHANGED}| + |\mathsf{CHANGED}|^2))$ time, for graph pattern matching defined in terms of bounded simulation. □

In general, a query class $\mathcal{Q}$ can be considered Π-tractable if (a) preprocessing $Q(D)$ is in PTIME, and (b) $Q(D \oplus \Delta D)$ can be incrementally computed in parallel polylog-time. If so, it is feasible to answer Q in response to changes to big data D.

3.6 Top-k Graph Pattern Matching

In social data analysis we often want to find matches of a designated pattern node u_o in Q as "query focus" [3]. That is, we just want those nodes in a social graph G that are matches of u_o in $M(Q, G)$, rather than the entire set $M(Q, G)$ of matches for Q. Indeed, a recent survey shows that 15% of social queries are to find matches of specific pattern nodes [29]. This is how graph search[5] of Facebook is conducted on its social graph. Moreover, it often suffices to find top-k matches of u_o in $M(Q, G)$. More specifically, assume a scoring function $s(\cdot)$ that given a match v of u_o, returns a non-negative real number $s(v)$. For a positive integer k, *top-k graph pattern matching* is to find a set U of matches of u_o in $M(Q, G)$, such that U has exactly k matches and moreover, for any k-element set U' of matches of u_o, $s(U') \leq s(U)$, where $s(U)$ is defined as $\Sigma_{v \in U} s(v)$. When there exist less than k matches of u_o in $M(Q, G)$, U includes all such matches.

This suggests that we develop algorithms to find top-k matches with *the early termination property* [8], *i.e.,* they stop as soon as a set of top-k matches is found, *without* computing the entire $M(Q, G)$. While the worst-case time complexity of such algorithms may be no better than their counterparts for computing the entire $M(Q, G)$, they may only need to inspect part of big G, without paying the price of full-fledged graph pattern matching. Indeed, for graph pattern matching with simulation on real-life social graphs, it has been shown that top-k matching algorithms just inspect 65%–70% of the matches in $M(Q, G)$ on average [16].

4 Approximation Algorithms for Querying Big Data

Strategies such as those given above help us make the evaluation of *some* queries tractable on big data. However, it is still beyond reach to find exact answers to many queries in big data. An example is graph pattern matching defined in terms of subgraph isomorphism: it is NP-complete to decide whether there exists

[5] *http://www.facebook.com/about/graphsearch*

a match. As remarked earlier, even for queries that can be answered in PTIME, it is often too costly and infeasible to compute their exact answers in the context of big data. As a result, we have to evaluate these queries by using *inexact* algorithms, preferably approximation algorithms with performance guarantees.

Previous work on this topic has mostly focused on developing PTIME approximation algorithms for NP-optimization problems (NPOs) [6, 18, 36]. An NPO A has a set I of instances, and for each instance $x \in I$ and each feasible solution y of x, there exists a positive score $m(x, y)$ indicating the quality measure of y. Consider a function $r(\cdot)$ from natural numbers to $(1, \infty)$. An algorithm T is called *a r-approximation algorithm for problem A* if for each instance $x \in I$, T computes a feasible solution y of x such that $R(x, y) \leq r(|x|)$, where $R(x, y)$ is the *performance ratio* of y *w.r.t.* x, defined as follows [6]:

$$R(x,y) = \begin{cases} \mathsf{opt}(x)/m(x,y) & \text{when } A \text{ is a maximization problem} \\ m(x,y)/\mathsf{opt}(x) & \text{when } A \text{ is a minimization problem} \end{cases}$$

where $\mathsf{opt}(x)$ is the optimal solution of x. That is, while the solution y found by $T(x)$ may not be optimal, it is not too far from $\mathsf{opt}(x)$ (*i.e.,* bounded by $r(|x|)$).

However, PTIME approximation algorithms that directly operate on the original instances of a problem may not work well when querying big data.

(1) As shown in Example 2, PTIME algorithms on x may be beyond reach in practice when x is big. Moreover, approximation algorithms are needed for problems that are traditionally considered tractable [18], not limited to NPO.

(2) In contrast to NPOs that ask for a single optimum, query evaluation is to find *a set* of query answers in a dataset. Thus we need to revise the notion of performance ratios to evaluate the quality of a set of feasible answers.

After the topic has been studied for decades, it is unlikely that we can expect soon to have a set of algorithms that on one hand, have low enough complexity to be tractable on big data, and on the other hand, have a nice performance ratio.

Data-driven approximation. To cope with this, we propose to develop algorithms that work on data with "resolution" lower than the original instances, and strike a *balance* between the efficiency (scalability) and the performance ratio [5]. Consider a pair $\langle D, Q\rangle$ that represents an instance x, where Q is a query and D is a dataset (see Section 2). When D is big, we reduce D to D' of manageable size, and develop algorithms that are feasible when operating on D'.

More specifically, consider a function $\alpha(\cdot)$ that takes $|D|$ as input, and returns a number in $(0, 1]$. We use a *transformation function* $f(\cdot)$ that given D, reduces D to $D' = f(D)$ with *resolution* $\alpha(|D|)$ such that $|D'| \leq \alpha(|D|) \cdot |D|$. We also use a *query rewriting function* $F : \mathcal{Q} \to \mathcal{Q}$ for a query class $\mathcal{Q}$ that, given any $Q \in \mathcal{Q}$, returns another query $F(Q)$ in $\mathcal{Q}$. Based on these, we introduce the following.

An algorithm T is called a (α, r)-*approximation algorithm* for $\mathcal{Q}$ if there exist functions $f(\cdot)$ and $F(\cdot)$ such that for any dataset D,

(1) $D' = f(D)$ and $|D'| \leq \alpha(|D|)|D|$; and

(2) for each query Q in $\mathcal{Q}$ defined on D, $Q' = F(Q)$, and algorithm T computes $Y = Q'(D')$ such that the performance ratio $R(\langle D, Q\rangle, Y) \leq r(|D|)$.

Intuitively, $f(\cdot)$ is an *offline* process that reduces big data D to small D' with *a lower resolution* $\alpha(|D|)$. After this, *for all* queries Q in $\mathcal{Q}$ posed on D, T is used to evaluate $Q' = F(Q)$ in D' as an *online* process, such that the feasible answers $Y = Q'(D')$ computed by T in D' are not too far from the *exact answers* $Q(D)$ in D. To evaluate the accuracy of Y, we need to extend the notion of performance ratio $R(\cdot,\cdot)$ to measure how close *a set* of feasible query answers Y is to the set $Q(D)$ of *exact* answers. There are a variety of choices for defining $R(\cdot,\cdot)$, depending on the application domain in which T is developed (see [5] for details).

Example 5. A weighted undirected graph is defined as $G = (V, E, w)$, where for each edge e in E, $w(e)$ is the weight of e. Given G and two nodes s, t in V, we want to compute the distance $\mathsf{dist}(s,t)$ between s and t in G, *i.e.,* the minimum sum of the weights of the edges on a path from s to t for all such paths in G.

There exist exact algorithms for computing $\mathsf{dist}(s,t)$ in $O(|E|)$-time (cf. [34]). However, when G is big, we need more efficient algorithms. It has been shown in [34] that for any constant $k \geq 1$, one can produce a data structure of size $O(k|V|^{1+1/k})$. After this offline process, all distance queries on G can be answered in $O(k)$ time (constant time) online by using the structure, with a constant performance ratio $2k-1$ [34]. That is, there exists a (α, r)-approximation algorithm for distance queries, with $\alpha(|G|) = |V|^{1+1/k}/(|V| + |E|)$ and $r = 2k - 1$. □

Data-driven approximation aim to explores the connection between *the resolution* of data and *the performance ratio* of algorithms, and speed up the *online* process. As remarked earlier, the choice of $f(\cdot)$ and T depends on what cost we can afford for offline preprocessing and what algorithms are tractable on big data. When $\alpha(|D|)$ is sufficiently small (*e.g.,* below a certain threshold ξ), $f(D)$ reduces "big data" D to "small data" D', on which a PTIME algorithm T is feasible. However, if D' remains "big", *i.e.,* when $\alpha(|D|) \geq \xi$, we may require T to be in NC. To cope with big D, the offline preprocessing step may require more resources such as computing nodes for parallel processing than online query evaluation.

5 Data Quality: The Other Side of Big Data

The study of querying big (social) data is still in its infancy. There is much more to be done. In particular, a complexity class that captures queries tractable on big data has to be identified, to characterize both computational and communication costs. Complete problems and reductions for the complexity class should be in place, so that we can effectively decide whether a class of queries is tractable on big data and whether a query class can be reduced to another one that we know how to process. In addition, more effective techniques for querying big data should be developed so that (combinations of) the techniques can improve query processing by MapReduce. Furthermore, the connection between data resolution and performance ratio needs a full treatment. Given a resolution, we should be able to determine what performance ratio we can expect, and vice versa.

We have so far focused on how to cope with the volume (quantity) of big data. Nonetheless, *data quality* is as important as data quantity. As an example, consider tuples below that represent suspects for a secretary S in a drug ring, identified by graph pattern matching in social networks (recall Q_0 and G_0 of Fig. 1):

	FN	LN	AC	street	city	state	zip	status
t_1:	Mary	Smith	212	Mtn Ave	MH	NJ	10007	single
t_2:	Mary	Smith	908	Mtn Ave	MH	NJ	07974	single
t_3:	Mary	Luth	212	Broadway	NY	NY	10007	married

Each tuple in the table specifies a suspect: her name (FN and LN), area code AC, address (street, city, state, zip code), and marital status, extracted from social networks. Consider the following simple queries about the suspects.

(1) Query Q_1 is to find how many suspects are based in New Jersey. By counting those tuples t with $t[\mathsf{state}]$ = "NJ", we get 2 as its answer. However, the answer may be *incorrect*. Indeed, (a) the data in tuple t_1 is *inconsistent*: $t_1[\mathsf{AC}] = 212$ is an area code for New York, and it has conflict with $t_1[\mathsf{state}]$ (NJ). Hence NJ may not be the true value of $t_1[\mathsf{state}]$. (b) The data in the table may be *incomplete*. That is, some suspects may not use social networks and hence, are overlooked. (c) Tuples t_1, t_2 and t_3 may refer to the same person and hence, may be *duplicates*. In light of these data quality issues, we cannot trust the answer to query Q_1.

(2) Suppose that the table above is complete, t_1, t_2 and t_3 refer to the same person Mary, and all their attribute values were once the true values of Mary but some may have become obsolete. Now query Q_2 is to find Mary's current last name. We do not know whether it is Smith or Luth. However, we know that marital status can only change from single to married, and that her last name and marital status are correlated. From these we can conclude that the answer to Q_2 is Luth.

This example tells us the following. First, when the quality of the data is poor, we cannot trust answers to our queries no matter how big data we can handle and how efficient we can process our queries. Second, data quality analyses help us improve the quality of our query answers. However, already difficult for (small) relational data (see [9] for a survey), the study of data consistency, accuracy, currency, deduplication and information completeness is far more challenging for big data. Indeed, big data is typically heterogeneous (variety), time-sensitive (velocity), of low-quality (veracity) and big (volume). Despite these, data quality is a must for us to study if we want to make practical use of big data.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, 2010.
3. M. Bendersky, D. Metzler, and W. Croft. Learning concept importance using a weighted dependence model. In *WSDM*, 2010.
4. J. Brynielsson, J. Högberg, L. Kaati, C. Martenson, and P. Svenson. Detecting social positions using simulation. In *ASONAM*, 2010.
5. P. Buneman and W. Fan. Data driven approximation algorithms for querying big data. Unpublished manuscript, 2013.
6. P. Crescenzi, V. Kann, and M. Halldórsson. A compendium of NP optimization problems. `http://www.nada.kth.se/~ viggo/wwwcompendium/`.
7. J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.

8. R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *JCSS*, 66(4):614–656, 2003.
9. W. Fan and F. Geerts. *Foundations of Data Quality Management.* Morgan & Claypool Publishers, 2012.
10. W. Fan, F. Geerts, and F. Neven. Making queries tractable on big data with preprocessing. Unpublished manuscript, 2013.
11. W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding regular expressions to graph reachability and pattern queries. In *ICDE*, 2011.
12. W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractability to polynomial time. In *PVLDB*, 2010.
13. W. Fan, J. Li, Z. Tan, X. Wang, and Y. Wu. Incremental graph pattern matching. In *SIGMOD*, 2011.
14. W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *SIGMOD*, 2012.
15. W. Fan, X. Wang, and Y. Wu. Performance guarantees for distributed reachability queries. In *PVLDB*, 2012.
16. W. Fan, X. Wang, and Y. Wu. Diversified top-k graph pattern matching. Unpublished manuscript, 2013.
17. W. Fan, X. Wang, and Y. Wu. Graph pattern matching using views. Unpublished manuscript, 2013.
18. R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory.* Oxford University Press, 1995.
19. A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4), 2001.
20. J. M. Hellerstein. The declarative imperative: Experiences and conjectures in distributed logic. *SIGMOD Record*, 39(1):5–19, 2010.
21. M. R. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.
22. D. S. Johnson. A catalog of complexity classes. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A).* The MIT Press, 1990.
23. N. D. Jones. An introduction to partial evaluation. *ACM Comput. Surv.*, 28(3):480–503, 1996.
24. H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for MapReduce. In *SODA*, 2010.
25. P. Koutris and D. Suciu. Parallel evaluation of conjunctive queries. In *PODS*, 2011.
26. M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, 2002.
27. S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Capturing topology in graph pattern matching. *PVLDB*, 5(4), 2011.
28. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
29. M. Morris, J. Teevan, and K. Panovich. What do people ask their social networks, and why? A survey study of status message Q&A behavior. In *CHI*, 2010.
30. M. Natarajan. Understanding the structure of a drug trafficking organization: a conversational analysis. *Crime Prevention Studies*, 11:273–298, 2000.
31. A. Ntoulas, J. Cho, and C. Olston. What's new on the Web? The evolution of the Web from a search engine perspective. In *WWW*, 2004.
32. G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *TCS*, 158(1-2), 1996.
33. L. Terveen and D. W. McDonald. Social matching: A framework and research agenda. *ACM Trans. Comput.-Hum. Interact.*, 12(3), 2005.
34. M. Thorup and U. Zwick. Approximate distance oracles. *JACM*, 52(1):1–24, 2005.
35. J. R. Ullmann. An algorithm for subgraph isomorphism. *JACM*, 23(1):31–42, 1976.
36. V. V. Vazirani. *Approximation Algorithms.* Springer, 2003.