Querying Big Data: Bridging Theory and Practice

Wenfei Fan[a,b], Jinpeng Huai[b]

[a]*School of Informatics, University of Edinburgh, 10 Crichton Street, Edinburgh, EH8 9AB, United Kingdom*
[b]*International Research Center on Big Data, Beihang University, Beijing, No.37 XueYuan Road, 100083, Beijing, China*

Abstract

Big data introduces challenges to query answering, from theory to practice. A number of questions arise. What queries are "tractable" on big data? How can we make big data "small" so that it is feasible to find exact query answers? When exact answers are beyond reach in practice, what approximation theory can help us strike a balance between the quality of approximate query answers and the costs of computing such answers? To get sensible query answers in big data, what else do we necessarily do in addition to coping with the size of the data? This position paper aims to provide an overview of recent advances in the study of querying big data. We propose approaches to tackling these challenging issues, and identify open problems for future research.

Keywords: Big data, query answering, tractability, distributed algorithms, incremental computation, approximation, data quality.

1. Introduction

Big data is a term that is almost as popular as "internet" was back 20 years ago. It refers to a collection of data sets so large and complex that it becomes difficult to process using traditional database management tools or data processing applications [92]. More specifically, big data is often characterized with four V's: *Volume* for the scale of the data, *Velocity* for its streaming or dynamic nature, *Variety* for its different forms (heterogeneity), and *Veracity* for the uncertainty (poor quality) of the data [64]. Such data comes from social networks (*e.g.,* Facebook, Twitter, Sina Weibo), e-commerce systems (*e.g.,* Amazon, Taobao), finance (*e.g.,* stock transactions), sensor networks, software logs, e-government and scientific research (*e.g.,* environmental research), just to name a few, where data is easily of PetaByte (PB, 10^{15} bytes) or ExaByte (EB, 10^{18} bytes) size. The chances are that big data will generate as big impacts on our daily lives as internet has done.

New challenges. As big data researchers, we do not confine with the general characterization of big data. We are more interested in what specific technical problems or research issues big data introduces to query answering. Given a dataset D and a query Q, *query answering* is to find the answers $Q(D)$ to Q in D. Here Q can be an SQL query on relational data, a keyword query to search documents, or a personalized social search query on social networks (*e.g.,* Graph Search of Facebook [29]).

Example 1. *A fraction D_0 of an employee dataset of a company is shown in Figure 1. Each tuple in D_0 specifies the first-name (*FN*), last name (*LN*),* salary *and marital* status *of an employee, as well as the area code (*AC*) and* city *of her office. A query Q_0 is to find distinct employees whose first name is Mary.*

	FN	LN	AC	city	salary	status
t_1:	Mary	Smith	20	Beijing	50k	single
t_2:	Mary	Webber	10	Beijing	50k	married
t_3:	Mary	Webber	10	Beijing	80k	married
s_1:	Bob	Luth	212	NYC	80k	married
s_2:	Robert	Luth	212	NYC	55k	married

Figure 1: An employee dataset D_0

Such a query can be expressed in, e.g., relational algebra, written as $\sigma_{FN\,=\,\text{“Mary”}}R_0$ by using selection operator σ [1], where R_0 is the relation schema of D_0. To answer the query Q_0 in D_0, we need to find all tuples in D_0 that satisfy the selection condition: FN = *"Mary", i.e., tuples t_1, t_2 and t_3.* □

In the context of big data, query answering becomes far more challenging than what we have seen in Example 1. The new complications include but are not limited to the following.

<u>*Data*</u>. In contrast to a single traditional database D_0, there are typically multiple data sources with information relevant to our queries. For instance, a recent study shows that many domains have tens of thousands of Web sources [22], *e.g.,* restaurants, hotels, schools. Moreover, these data sources often have a large volume of data (*e.g.,* of PB size) and are frequently updated. They have different formats and may not come with a schema, as opposed to structured relational data. Furthermore, many data sources are *unreliable*: their data is typically "dirty".

<u>*Query*</u>. Queries posed on big data are no longer limited to our familiar SQL queries. They are often for document search, social search or even for data analysis such as data mining. Moreover, their semantics also differs from traditional queries. On one hand, it can be more flexible: one may want approximate answers instead of exact answers $Q(D)$. On the other hand, one could ask query answering to be ontology-mediated by coupling datasets with a knowledge base [12], or personalized and

Email addresses: `wenfei@inf.ed.ac.uk` (Wenfei Fan), `huaijp@buaa.edu.cn` (Jinpeng Huai)

context-aware [86] such that the same query gets different answers when issued by different people in different locations.

These tell us that query answering in big data is a departure from our familiar terrain of traditional database queries. It raises a number of questions. Does big data give rise to any new fundamental problems? In other words, do we need new theory for querying big data? Do we need to develop new methodology for query processing in the context of big data? What practical techniques could we use to cope with the sheer volume of big data? In addition to the scalability of query answering algorithms, what else do we have to pursue in order to find sensible or even correct query answers in big data?

Querying big data. This paper presents an overview of recent advances in the study of these problems. It is a progress report of the International Research Center on Big Data at Beihang University [10], which was established in September 2012, and has been working on querying big data since then. We report how we tackle the problems mentioned above.

BD-tractability. The first question we need to answer is what queries are *tractable* on big data. Given a query Q and a big dataset D, we want to know whether we can compute $Q(D)$ within our available resources such as time and space. As found in most textbooks (*e.g.,* [1, 81]), a class of queries is traditionally considered *tractable* if there exists an algorithm for answering its queries in time bounded by a polynomial in the size of the input (PTIME), *i.e., a database and a query*. In other words, a class of queries is *feasible* from a theoretical perspective if its worst-case time complexity is PTIME, while a class is considered difficult to solve when it is NP-hard. This notion of time complexity dates back to 1965 [60] and is almost 50-years old.

When it comes to big data, however, PTIME queries may no longer be feasible. For instance, consider the query Q_0 and dataset D_0 given in Example 1. To compute $Q_0(D_0)$ in the absence of any indices, one may need to scan D_0. Assuming the fastest Solid State Drives (SSD) with disk scanning speed of 6GB/s [85], a linear scan of D_0 takes 166,666 seconds when D_0 consists of 1PB of data; that is, 2,777 minutes, 46 hours, or 1.9 days! When D_0 has 1EB of data, we have to wait 5.28 years for a linear scan of D_0. That is, even *linear-time* ($O(n)$) queries become *infeasible* in the context of big data.

This suggests that we revise the classical computational complexity theory for querying big data. To this end, we propose a notion of *BD-tractable queries* [38], to help us determine what queries are tractable or feasible on big data.

Making queries BD-tractable. It is not surprising that many query classes are not BD-tractable. The next question naturally asks whether we can make these query classes BD-tractable? We approach this by studying both its fundamental problems and practical techniques, by making big data "small".

To understand what it takes to compute answers $Q(D)$ of a query Q in a dataset D, we want to identify *a core of D for answering Q, i.e.,* a minimum subset D_Q of D such that $Q(D) = Q(D_Q)$. Indeed, it often suffices to fetch a small or even a bounded subset D_Q of D for computing $Q(D)$, no matter how large the underlying dataset D is. For instance, when Q is a Boolean conjunctive query (*a.k.a.* SPC query [1]), we need at most $\|Q\|$ tuples from D to answer Q, *independent of the size of D*, where $\|Q\|$ is the number of tuples in the tableau representation of Q. This is also the case for many personalized social search queries. Intuitively, if a core D_Q of D for answering Q has a bounded size, then Q is *scale independent* in D [36], *i.e.,* we can efficiently compute $Q(D)$ no matter how big D is. This suggests that we study how to determine whether a query is scale independent in a dataset.

In addition, we develop several practical techniques for making big data "small". These include (a) distributed query processing by partial evaluation [47], with provable performance guarantees on both response time and network traffic; (b) query-preserving data compression [45]; (c) view-based query answering [50]; and (d) bounded incremental computation [49, 82]. All these techniques allow us to compute $Q(D)$ with a cost that is *not* a function of the size of D, and have proven effective in querying social networks. The list is not exclusive: there are many other techniques for making big data "small" and hence, making queries feasible on big data.

Query-driven and data-driven approximation. Some queries neither are BD-tractable nor can be made BD-tractable. An example is graph pattern matching by subgraph isomorphism. Here query Q is a graph pattern, dataset D is a graph, and the answer $Q(D)$ is the set of all subgraphs of D that are isomorphic to Q. Such queries are expensive: it is NP-complete even to decide whether there exists a subgraph of D that is isomorphic to Q! It is beyond reach in the context of big D to compute exact answers $Q(D)$. In light of this, algorithms for processing such queries on big data are *necessarily inexact*. We may have to settle with *heuristics*, "quick and dirty" algorithms which return approximate answers that are not necessarily optimal [81].

This highlights the need for studying the next question: how can we develop *approximation algorithms*, *i.e.,* heuristics which find answers that are guaranteed to be not far from the exact query answers? We propose two types of approximation.

(1) Query-driven approximation. For certain queries we can relax their semantics and reduce the complexity of query processing. One example is the class of graph pattern queries mentioned above, for social network analysis. Instead of adopting subgraph isomorphism for graph pattern matching, we can use (revisions of) graph simulation [41, 42, 74]. This reduces the complexity of graph pattern matching from intractability by subgraph isomorphism to quadratic-time or cubic-time by (revised) graph simulation! Better still, the revised notions of graph simulation allow us to catch more sensible matches in social data analysis than subgraph isomorphism can find.

(2) Data-driven approximation. In some applications we may not be able to relax query semantics. To this end, we propose a notion of *resource-bounded approximation* in this paper. In contrast to traditional approximation algorithms that directly operate on a given big dataset D, we first reduce D to "small data" D_Q with a "lower resolution" $\alpha \in (0, 1]$, such that $|D_Q| \leq \alpha |D|$. We then compute $Q(D_Q)$ as approximate query

answers to Q, such that $Q(D_Q)$ is within a *performance ratio* η to the exact answer $Q(D)$. We explore the *connection* between the resolution α and the quality bound η, to strike a *balance* between the computation cost and the quality of the approximate answers. Our preliminary study [52] has shown that for personalized social search queries, the performance ratio remains 100% even when the resolution α is as small as 0.0015% $(15*10^{-6})$. That is, we can reduce D of 1PB to D_Q of 15GB, while still retaining exact answers for such queries!

Big data = quantity + quality. To compute high-quality query answers from big data, it is often insufficient just to develop scalable algorithms to cope with large volume of the data. To illustrate this, let us consider the following example.

Example 2. *Recall query Q_0 and dataset D_0 from Example 1. Suppose that we have efficient techniques in place to compute $Q_0(D_0)$ for big D_0. As remarked earlier, $Q_0(D_0)$ consists of three tuples t_1, t_2 and t_3. The question is: can we trust $Q_0(D_0)$ to be the correct answer to what the user wants to find?*

Unfortunately, there are at least three reasons that discredit our trust in $Q_0(D_0)$. (1) In tuple t_1, attribute $t_1[\mathsf{AC}]$ is 20 and $t_1[\mathsf{city}]$ is Beijing, while the area code of Beijing is 10. In light of this, tuple t_1 is "inconsistent" and hence, its quality is in question. (2) The chances are that all three tuples t_1, t_2 and t_3 refer to the same person; in other words, they do not represent distinct *employees. (3) Furthermore, the dataset D_0 may be incomplete: for some employees whose first name is also Mary, their records are not included in D_0. In light of these, we do not know whether the answer $Q_0(D_0)$ is correct or not!* □

From the example we can see that when the datasets are dirty, we cannot trust the answers to our queries in those datasets. In other words, no matter how big datasets we can handle and how fast our query processing algorithms are, the query answers computed may not be correct and hence may be useless! Unfortunately, real-life data is often dirty [33], and the scale of data quality problems is far worse in the context of big data, since real-life data sources are often unreliable. Therefore, the study of the quality of big data is as important as techniques for coping with its quantity; that is, *big data = quantity + quality*!

This motivates us to study the quality of big data. We consider five central issues of data quality: data consistency [34], data accuracy [16], information completeness [32], data currency [40] and entity resolution [31], from theory to practice. We study how to repair dirty data [20, 43, 46] and how to deduce true values of an entity [39], among other things, emphasizing new challenges introduced by big data.

Organization. The remainder of the paper is organized as follows. We start with BD-tractability in Section 2. We study scale independence and present several practical techniques for making queries BD-tractable in Section 3. When BD-tractable algorithms for computing exact query answers are beyond reach in practice, we study approximate query answering in Section 4, by proposing query-driven approximation and data-driven approximation. We study the other side of big data, namely, data quality, in Section 5. Finally, Section 6 concludes the paper.

The study of querying big data is still in its infancy, and it has raised as many questions as it has answered. In light of this, we also identify open research issues in this paper, and propose approaches to tackling them. We hope that the paper will incite interest in the study of querying big data, and we invite interested colleagues to join forces with us in the study.

2. Tractability Revised for Querying Big Data

This section studies the following problem: given a class $\mathcal{Q}$ of queries that we need to use, we want to determine whether $\mathcal{Q}$ is tractable in big data, *i.e.,* it is feasible to answer the queries of $\mathcal{Q}$ in big data within our available resources. As we have seen in Section 1, polynomial time can no longer provide a characterization for $\mathcal{Q}$ to be tractable in big data. This suggests that we revise the traditional notion of tractability, and define *BD-tractability*, *i.e.,* tractability for queries on big data.

Below we present a notion of BD-tractable queries. We encourage the interested reader to consult [38] for details.

Preliminaries. We start with a review of two well-studied complexity classes (see, *e.g.,* [58, 67] for details).

- The complexity class P consists of all decision problems that can be solved by a deterministic Turing machine in polynomial time (PTIME), *i.e.,* in $n^{O(1)}$ time, where n is the size of the input (dataset D and query Q in our case).
- The parallel complexity class NC, known as Nick's Class, consists of all decision problems that can be solved by taking $O(\log^{O(1)} n)$ time on a PRAM (parallel random access machine) with $n^{O(1)}$ processors.

In this paper we focus on query classes rather than decision problems. We use P to denote the set of all PTIME query classes. We say that a query class $\mathcal{Q}$ is in NC if all of its queries can be answered in parallel polylog-time, *i.e.,* polynomial time in *the logarithm* of the input using a PRAM with polynomially many processors. Such a query class is *highly parallel feasible*, *i.e.,* its queries can be efficiently answered on a *parallel* computer [58]. It is also known that a large class of NC algorithms can be implemented in the MapReduce framework [69], such that if an NC algorithm takes t time, than its corresponding MapReduce counterpart takes $O(t)$ rounds. We use NC to denote the set of all such parallel polylog-time query classes. It should be remarked that there have been revisions of the PRAM model by requiring $\log n$ processors instead of $n^{O(1)}$ [25].

BD-tractability. To make query answering feasible in big data, we adopt two ideas: (1) using parallel machines, and (2) separating offline and online processes. The second idea suggests that we preprocess a dataset D by, *e.g.,* building indices or compressing the data, which yields dataset D', such that all queries in $\mathcal{Q}$ on D can subsequently be processed on D' *online* efficiently. When the data is static or when D' can be incrementally maintained efficiently, the preprocessing step can be considered as an *offline* process with a *one-time cost*. Preprocessing has been a common practice of database people for decades.

Example 3. *Recall query Q_0 and dataset D_0 from Example 1. Extending Q_0, let us consider a class $\mathcal{Q}_0$ of* Boolean selection queries. *A query Q in $\mathcal{Q}_0$ is to find whether there exists a tuple $t \in D_0$ such that $t[A] = c$, where A is an attribute of D_0 and c is a constant. A naive evaluation of Q would require a linear scan of D_0. To efficiently answer queries of $\mathcal{Q}_0$ in D_0, we can first build B^+ trees on the values of the attributes of D_0, as a one-time preprocessing step offline. Then we can evaluate* all queries in $\mathcal{Q}_0$ *on D_0 in $O(\log|D_0|)$ time using the indices. That is, we no longer need to scan D_0 when processing* each *query in $\mathcal{Q}_0$. When D_0 consists of 1PB of data, we can get the results in 5 seconds with the indices rather than 1.9 days.* □

Based on these two ideas, below we propose a revision of the traditional notion of tractable query classes.

To be consistent with the complexity classes that are traditionally studied for decision problems [58, 67], we consider Boolean query classes $\mathcal{Q}$, and represent $\mathcal{Q}$ 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 $\langle D, Q\rangle \in S$, *i.e.,* $Q(D)$ is true. Here $\mathsf{C_Q}$ may be the sequential complexity class P or the parallel complexity class NC, among other things.

Complexity class BDT^0. We say that a class $\mathcal{Q}$ of queries is *BD-tractable* if there exist a PTIME-computable preprocessing function Π on datasets and a language S' of pairs such that for queries $Q \in \mathcal{Q}$ and all datasets D,

- $\langle D, Q\rangle$ 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 NC, *i.e.,* the language of pairs $\langle \Pi(D), Q\rangle$ is in NC.

We denote by BDT^0 the set of all BD-tractable query classes.

Intuitively, function $\Pi(\cdot)$ preprocesses D and generates another structure $D' = \Pi(D)$ offline, in PTIME. 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 NC, *i.e.,* in parallel polylog-time.

Observe the following. (a) As shown in Example 3, parallel polylog-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.,* [58]). (b) We consider PTIME preprocessing feasible since it is a *one-time* price and is performed *offline*. Note that the preprocessing step is also expected to be conducted using *parallel machines*, possibly by allocating more resources (*e.g.,* computing nodes) to it than to online query answering. Moreover, by requiring that $\Pi(\cdot)$ is in PTIME, the size of $\Pi(D)$ is bounded by a polynomial.

Example 4. *As we have seen in Example 3, the class $\mathcal{Q}_0$ of Boolean selection queries is in* BDT^0. *Indeed, function $\Pi(\cdot)$ preprocesses a dataset D_0 by building B^+-trees on attributes of D_0 in* PTIME. *After this, all queries in $\mathcal{Q}_0$ posed on D_0 can be answered in $O(\log|D|)$ time by using the indices in $\Pi(D_0)$. In fact, the class of all relational algebra queries extended with transitive closure is also in* BDT^0 *over ordered relational datasets, since those queries are in* NC *in this setting [88].* □

Making queries BD-tractable. Some query classes $\mathcal{Q}$ are not BD-tractable, but can be transformed to a BD-tractable query class by means of *re-factorizations*. A re-factorization re-partitions the data and query parts for $\mathcal{Q}$ and identifies a dataset for preprocessing, such that after the preprocessing, its queries can be subsequently answered in parallel polylog-time.

Complexity class BDT. More specifically, we say that a class $\mathcal{Q}$ of queries *can be made BD-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 $\mathcal{Q}$,

- $D' = \pi_1(D,Q)$, $Q' = \pi_2(D,Q)$, $\langle D, Q\rangle = \rho(D', Q')$, and
- the query class $\mathcal{Q}' = \{Q' \mid Q' = \pi_2(D,Q), \langle D, Q\rangle \in S\}$ is BD-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 $\pi_2(x)$. The data part D' is picked from x and will be preprocessed, such that after the preprocessing step, all the queries $Q' \in \mathcal{Q}'$ can then be answered in parallel polylog-time.

We use BDT to denote the set of all query classes that can be *made* BD-tractable. Obviously, BDT^0 is a subset of BDT, when $D = \pi_1(D,Q)$, $Q = \pi_2(D,Q)$, and ρ is the identity function. As will be seen next, BDT^0 is a proper subset of BDT unless P = NC, *i.e.,* there is a query class that is in BDT but not in BDT^0.

Example 5. *Consider Breadth-Depth Search (BDS) [58]:*

- *Input: An undirected graph $G = (V, E)$ with a numbering on the nodes, and a pair (u, v) of nodes in V.*
- *Question: Is u visited before v in the breadth-depth search of G induced by the vertex numbering?*

A breadth-depth search starts at a node s and visits all its children, pushing them onto a stack in the reverse order induced by the vertex numbering as the search proceeds. After all of s's children are visited, the search continues with the node on the top of the stack, which plays the role of s.

In the problem statement of BDS given above, the entire input, i.e., $x = (G, (u, v))$, is treated as a query, while its data part is empty. In this setting, there is nothing to be preprocessed. Moreover, it is known that BDS is P*-complete (cf. [58]), i.e., it is the hardest problem in the complexity class* P. *Unless* P = NC, *such a query cannot be processed in parallel* polylog-time. *In other words, this class of BDS queries is not in* BDT^0 *unless* P = NC. *It is also known that the question whether* P = NC *is as hard as our familiar open question whether* P = NP.

Nonetheless, there exists a re-factorization (π_1, π_2, ρ) of its instances $x = (G, (u, v))$ that identifies G as the data part and (u, v) as the query part. More specifically, $\pi_1(x) = G$, $\pi_2(x) = (u, v)$, and ρ maps $\pi_1(x)$ and $\pi_2(x)$ back to x. Given this, *we define preprocessing $\Pi(\cdot)$ as the function that performs breadth-depth search on G based on the ordering on the vertices, and returns a list M consisting of all the nodes in V in the same order as they are visited during the search. Then $\Pi(G)$ is clearly in* PTIME *in $|G|$. Let S' be the language of pairs $\langle M,(u,v)\rangle$ such that u appears before v in M. Obviously, one can decide whether $(M,(u,v)) \in S'$ by binary searches on M, in $O(\log |M|)$ time. Hence BDS is in* BDT. *In other words, while BDS is not BD-tractable, it can be made BD-tractable by means of a re-factorization. In light of this, BDS provides a witness that separates* BDT *and* BDT^0, *unless* P = NC. □

Fundamental issues. There are several important questions in connection with BD-tractability. What *reductions* can we use to transform one query class in BDT to another? Does there exist a natural class $\mathcal{Q}$ of queries that is *complete* for BDT, *i.e.,* $\mathcal{Q}$ is a class of the "hardest" queries in BDT? How large is BDT? In other words, is it a new complexity class or the same as P or NC? The same questions also arise for BDT^0. In fact, these are the "standard" questions one would have to answer for any complexity class, including our familiar P and NP.

These questions have been studied for BDT and BDT^0 [38].

- A form of NC-reductions $\leqslant_{fa}^{\mathsf{NC}}$ has been defined for BDT, 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 BDT (*i.e.,* if $\mathcal{Q}_1 \leqslant_{fa}^{\mathsf{NC}} \mathcal{Q}_2$ and $\mathcal{Q}_2$ is in BDT, then so is $\mathcal{Q}_1$). Similarly, NC-reductions have been defined for BDT^0 with these properties. In contrast to our familiar PTIME-reductions for NP problems (see, *e.g.,* [81]), these reductions require a pair of NC functions, *i.e.,* both are in parallel polylog-time.

- There exists a *complete query class* $\mathcal{Q}_m$ for BDT under $\leqslant_{fa}^{\mathsf{NC}}$ reductions, *i.e.,* $\mathcal{Q}_m$ is in BDT and moreover, for all query classes $\mathcal{Q} \in \mathsf{BDT}$, $\mathcal{Q} \leqslant_{fa}^{\mathsf{NC}} \mathcal{Q}_m$. However, the question whether there exists a complete query class for BDT^0 is as hard as the open question whether P = NC.

- $\mathsf{NC} \subseteq \mathsf{BDT} = \mathsf{P}$. That is, all PTIME query classes can be made BD-tractable via proper re-factorizations, or in other words, by transforming them to a query class in BDT via $\leqslant_{fa}^{\mathsf{NC}}$ reductions. In contrast, unless P = NC, $\mathsf{BDT}^0 \subset \mathsf{P}$, *i.e.,* BDT^0 is indeed a proper subset of P, and hence, not all PTIME queries are BD-tractable.

These results are not only of theoretical interest, but also provide guidance for us to answer queries in big data. For instance, given a query class $\mathcal{Q}$, we can conclude that it can be made BD-tractable if we can find a $\leqslant_{fa}^{\mathsf{NC}}$ reduction to a complete query class $\mathcal{Q}_m$ of BDT. If so, we are warranted an effective algorithm for answering queries of $\mathcal{Q}$ in big data. Indeed, such an algorithm can be developed by simply composing the NC reduction and an NC algorithm for processing $\mathcal{Q}_m$ queries; then the algorithm remains in parallel polylog-time.

One may ask what query classes may not be made BD-tractable. The results above also tell us the following: unless P = NP, all query classes for which the membership problem is NP-hard are not in BDT. *The membership problem* for a query class $\mathcal{Q}$ is to decide, given a query $Q \in \mathcal{Q}$, a dataset D and an element e, whether $e \in Q(D)$, *i.e.,* e is in the answer to Q in D.

Open issues. There has been a host of recent work on revising the traditional complexity theory to characterize data-intensive computation on big data. The revisions are defined in terms of computational costs [38], communication (coordination) rounds [61, 71], or MapReduce steps [69] and data shipments [3] in the MapReduce framework [23]. Our notions of BD-tractability focus on computational costs [38]. The study is still preliminary, and a number of questions remain open.

(1) The first question concerns what complexity class precisely characterizes online query processing that is feasible on big data. As a starting point we adopt NC because (a) NC is considered highly parallel feasible [58]; (b) parallel polylog-time is feasible on big data; and (c) many NC algorithms can be implemented in the MapReduce framework [69], which is being used in cloud computing and data centers for processing big data. However, NC is defined in the PRAM model, which may not be accurate for real-life parallel frameworks such as MapReduce.

These call for a full treatment of parallel computation models that are more practical than PRAM for characterizing available resources in the real world. Such models should take into account *both computational complexity and communication costs*. Upon the availability of such models, the class BDT^0 of BD-tractable queries should then be revised accordingly.

(2) The second question concerns the complexity of preprocessing. Let us use $\mathsf{PQ}[\mathsf{C}_P, \mathsf{C}_Q]$ to denote the set of all query classes that can be answered by preprocessing the data sets in the complexity class C_P and subsequently answering the queries in C_Q. Then BDT^0 can be represented by $\mathsf{PQ}[\mathsf{P}, \mathsf{NC}]$. One may consider other complexity classes C_P instead of P. For instance, one may consider $\mathsf{PQ}[\mathsf{NC}, \mathsf{NC}]$ by requiring the preprocessing step to be conducted more efficiently; this is not very interesting since $\mathsf{PQ}[\mathsf{NC}, \mathsf{NC}]$ coincides with NC. On the other hand, one may want to consider C_P beyond P, *e.g.,* NP and PSPACE (*i.e.,* $\mathsf{PQ}[\mathsf{NP}, \mathsf{NC}]$ and $\mathsf{PQ}[\mathsf{PSPACE}, \mathsf{NC}]$). This is another debatable issue that demands further study. No matter what $\mathsf{PQ}[\mathsf{C}_P, \mathsf{C}_Q]$ we use, one has to strike a balance between its expressive power and computational cost in the context of big data.

(3) BD-tractability has only been studied for Boolean queries and decision problems, as people usually do in complexity theory. Nevertheless, BD-tractability for general queries, as well as for search and function problems, remains to be studied.

(4) There are a number of open issues in connection with query evaluation with preprocessing. Given a query class, how can we effectively *identify* a re-factorization that appropriately picks the right dataset to be preprocessed? What preprocessing strategies should we use? If a query class cannot be made BD-tractable, can we still answer its queries in big data? We will address some of these questions in the next a few sections.

(5) The last question concerns the existence of a complete query class for BDT^0. However, this is as hard as the problem whether P = NC, which is as hard as whether P = NP.

3. Making Big Data Small

Following up the notion of BD-tractability presented in the last section, we next investigate how we can make queries BD-tractable. There are many ways to do this, such as building up indices as we have seen in Example 3. In this section we focus on a particular approach, by *making big data small*. Suppose that we need to answer a class $\mathcal{Q}$ of queries in a big dataset D. We propose to reduce D to a dataset D' (or a number of fragments D') of *a manageable size*, such that (1) for all queries $Q \in \mathcal{Q}$, $Q(D) = Q(D')$, and (2) we can efficiently answer Q in D' within our available resources. In other words, as a preprocessing step, we reduce big D to small D' such that we can still compute exact answers $Q(D)$ by accessing only the small dataset D' instead of operating on the original big D directly.

The idea is simple. But to implement it, we need to settle several fundamental questions and develop practical techniques. Below we first study questions concerning whether it is possible at all to find a small dataset D' such that $Q(D) = Q(D')$. We then present several practical techniques to make big data small, which have been evaluated by using social network analysis as a testbed, and have proven effective in the application.

3.1. Scale Independence

We start with fundamental problems associated with the approach to making big data small. We first study the existence of a small subset D' of D such that we can answer Q in D by accessing only the data in D'. We then present effective methods for identifying such a D'. We invite the interested reader to consult [36] for a detailed report on this subject.

To simplify the discussion we consider relational queries. Let $\mathcal{R}$ be a relational schema (*i.e.*, $\mathcal{R} = (R_1, \ldots, R_n)$, where R_i is a relation schema [1]), D a database instance of $\mathcal{R}$, Q a query in query class $\mathcal{Q}$ such as relational algebra or conjunctive queries, and M a non-negative integer. Let $|D|$ denote the size of D, measured as the total number of tuples in relations of D.

The definition. We say that Q is *scale independent in* D w.r.t. M if there exists a subset $D_Q \subseteq D$ such that

- $|D_Q| \leq M$, and
- $Q(D_Q) = Q(D)$.

That is, to answer Q in D, we need only to fetch at most M tuples from D, *regardless of how big D is*. We refer to D_Q as a *core* for answering Q in D. Note that D_Q may not be unique. As will be seen shortly, we want to find a *minimum* core.

One step further, we say that Q is *scale independent for* $\mathcal{R}$ w.r.t. M if *for all instances* D of $\mathcal{R}$, Q is scale independent in D *w.r.t.* M, *i.e.*, one can always find a core D_Q with at most M tuples for answering Q in D.

The term "scale independence" is borrowed from [6, 7, 8]. The need for studying scale independence is evident in practice. It allows us to answer Q in big D by accessing a small dataset within our available resources. Moreover, if Q is scale independent for $\mathcal{R}$, we can answer Q without performance degradation when D grows, and hence, make Q scalable with $|D|$.

Example 6 [36] *Some real-life queries are actually scale independent. For example, below are (slightly modified) personalized search queries taken from Graph Search of Facebook [29].*

(1) Query Q_1 is to find all NYC *friends of a person p_0, from a dataset D_1. Here D_1 consists of two relations specified by* $\mathsf{person}(\mathsf{id}, \mathsf{name}, \mathsf{city})$ *and* $\mathsf{friend}(\mathsf{id}_1, \mathsf{id}_2)$, *which record the basic information of people (with a key* id*) and their friend relationships, respectively. Query Q_1 can be written as follows:*

$$Q_1(\mathsf{name}) = \exists \mathsf{id}\big(\mathsf{friend}(p_0, \mathsf{id}) \wedge \mathsf{person}(\mathsf{id}, \mathsf{name}, \mathit{NYC})\big).$$

Observe the following. (1) In personalized social searches we evaluate queries with a specified person, e.g., p_0 in Q_1. (2) Dataset D_1 is often big in real life. For instance, Facebook has more than 1 *billion users with* 140 *billion* friend *links [28]. A naive computation of the answer to Q_1, even if p_0 is known, may fetch the entire D_1, and is cost prohibitive.*

Nonetheless, we can compute $Q_1(D_1)$ by accessing only a small subset D_{Q_1} of D_1. Indeed, Facebook has a limit of 5000 friends per user (cf. [7]), and id *is a key of* person. *Thus by using indices on* id *attributes, we can identify D_{Q_1}, which consists of a subset D_f of* friend *including all friends of p_0, and a set D_p of* person *tuples t such that $t[\mathsf{id}] = t'[\mathsf{id2}]$ for some tuple t' in D_f. Then $Q_1(D_{Q_1}) = Q_1(D_1)$. Moreover, D_{Q_1} contains at most 10000 tuples of D_1, and is much smaller than D_1. Thus Q_1 is scale independent in D_1* w.r.t. $M \geq 10000$. *In fact, one can verify that Q_1 is scale independent in all instances of the schemas* person *and* friend *that satisfy the two constraints.*

(2) Consider another query Q_2, which is to find from a dataset D_2 all A-rated NYC *restaurants that were visited by* NYC *friends of p_0 in 2013. Here D_2 consists of four relations, specified by a relational schema $\mathcal{R}_2$ including* person *and* friend *as above, as well as* $\mathsf{restr}(\mathsf{rid}, \mathsf{name}, \mathsf{city}, \mathsf{rating})$ *(with* rid *as a key) and* $\mathsf{visit}(\mathsf{id}, \mathsf{rid}, \mathsf{yy}, \mathsf{mm}, \mathsf{dd})$ *(indicating that person* id *visited restaurant* rid *on a given date). Then Q_2 can be expressed as:*

$$\begin{aligned} Q_2(\mathsf{rn}, \mathsf{yy}) = \exists \mathsf{id}, \mathsf{rid}, \mathsf{pn}, \mathsf{mm}, \mathsf{dd}\big(&\mathsf{friend}(p_0, \mathsf{id}) \\ &\wedge \mathsf{visit}(\mathsf{id}, \mathsf{rid}, 2013, \mathsf{mm}, \mathsf{dd}) \wedge \mathsf{person}(\mathsf{id}, \mathsf{pn}, \mathit{NYC}) \\ &\wedge \mathsf{restr}(\mathsf{rid}, \mathsf{rn}, \mathit{NYC}, A)\big). \end{aligned}$$

Note that query Q_2 is also scale-independent. Indeed, (a) a year has at most 365 days; and (b) it is safe to assume that on a given day, each person id *dines out at most once. Putting these together with the constraints on* friend *and* person *(i.e., a person can have at most 5000 friends at Facebook, and* id *is a key of* person*), one can compute $Q_2(D_2)$ by accessing a bounded number of tuples, instead of scanning the entire D_2. Indeed, Q_2 is scale independent for all instances of schema $\mathcal{R}_2$ under these constraints.* $\square$

One can show that a query Q is scale independent for any schema $\mathcal{R}$ over which Q is defined when Q is either

- a Boolean conjunctive query if $\|Q\| \leq M$, or
- a top-k conjunctive query for a constant k and a scoring function f if $k\|Q\| \leq M$,

where $\|Q\|$ is the number of tuple templates in the tableau presentation of the conjunctive query Q [1]. Here Q is Boolean if for any instance D of $\mathcal{R}$, $Q(D)$ returns true if $Q(D)$ is nonempty and false otherwise; and Q is a top-k query if $Q(D)$ returns a subset $U \subseteq Q(D)$ such that (a) U consists of at most k tuples ($|U| = k$ if $|Q(D)| \geq k$), and (b) for all tuples $t \in Q(D) \setminus U$ and $s \in U$, $f(s) \geq f(t)$ [30].

Decision problems. To determine whether a query Q is scale independent, we need to study the following decision problems.

- The scale independence problem for $(\mathcal{Q}, D)$.
  - INPUT: A relational schema $\mathcal{R}$, an instance D of $\mathcal{R}$, a query $Q \in \mathcal{Q}$ over $\mathcal{R}$, and $M \geq 0$.
  - QUESTION: Is Q scale independent in D *w.r.t.* M?
- The scale independence problem for $\mathcal{Q}$.
  - INPUT: $\mathcal{R}$, a query $Q \in \mathcal{Q}$ over $\mathcal{R}$, and $M \geq 0$.
  - QUESTION: Is Q scale independent for $\mathcal{R}$ *w.r.t.* M?

That is, we want to find minimum cores for answering Q.

The complexity bounds of these problems have been established [36]. The problems are rather intriguing. For instance, the first one is Σ_3^p-complete ($\mathsf{NP}^{\mathsf{NP}^{\mathsf{NP}}}$) when $\mathcal{Q}$ is the class of conjunctive queries, and it is PSPACE-complete when $\mathcal{Q}$ is relational algebra (*i.e.*, first-order logic). Worse still, the second problem becomes undecidable for relational algebra. This is not surprising in database theory: for instance, the classical membership problem (see Section 2) is NP-complete for conjunctive queries, and PSPACE-complete for relational algebra [1].

Identifying a core. We have seen that it is rather expensive to determine whether a query Q is scale independent. Moreover, even after Q is found scale independent in a dataset D, it is non-trivial to identify a core D_Q for answering Q in D with a bounded size. As an example, consider a Boolean conjunctive query Q over a relational schema $\mathcal{R}$. As remarked earlier, we know that Q is scale independent for $\mathcal{R}$. The question is: is there an efficient algorithm that, given an instance D of $\mathcal{R}$, finds a core $D_Q \subseteq D$ such that $|D_Q| \leq \|Q\|$ and $Q(D_Q) = Q(D)$?

We approach this following the common practice of database people: we provide a sufficient condition for checking whether Q is scale independent and if so, for helping us efficiently compute a core for answering Q. This is formalized as follows.

Access schema. We define an *access schema* $\mathcal{A}$ over a relational schema $\mathcal{R}$ to be a set of tuples (R, X, N, T), where

- R is a relation schema in $\mathcal{R}$,
- X is a set of attributes of R, and
- N and T are natural numbers.

We say that a database instance D of $\mathcal{R}$ *conforms to the access schema* $\mathcal{A}$ if for each $(R, X, N, T) \in \mathcal{A}$:

- for each tuple of values $\bar{a}$ of attributes of X, the set $\sigma_{X=\bar{a}}(R)$ has at most N tuples *i.e.*, there exist at most N tuples t in R such that $t[X] = \bar{a}$; and
- $\sigma_{X=\bar{a}}(R)$ can be retrieved from D in time at most T.

That is, there exists an index on X that allows efficient retrieval of certain tuples from D, and there is a bound on the number of such tuples. Access schemas are *a combination* of indices and database dependencies, which are commonly used in practice.

Example 7. *Continuing with Example 6, we would have a tuple* $(\mathsf{friend}, \mathsf{id}_1, 5000, T)$ *for some value T in the access schema $\mathcal{A}$. That is, there exists an index on* id_1 *such that if* id_1 *is provided, at most 5000 tuples with such an id exist in* friend*, and it takes time T to retrieve those. In addition, we would have a tuple* $(\mathsf{person}, \mathsf{id}, 1, T')$ *in $\mathcal{A}$, indicating that* id *is a key for* person *with a known time T' for retrieving the tuple for a given* id. □

Computing a core by leveraging access schema. Given a relational schema $\mathcal{R}$, we say that a query Q is *scale independent under access schema* $\mathcal{A}$ if for *all* instances D of $\mathcal{R}$ that conform to $\mathcal{A}$, the answer $Q(D)$ can be computed in time that depends only on $\mathcal{A}$ and Q, but not on D. That is, Q is scale independent for $\mathcal{R}$ in the presence of $\mathcal{A}$, *independent of the size of the underlying* D. The following results are known.

- There is a set of syntactic rules for us to determine whether a relational algebra query Q is scale independent under $\mathcal{A}$; this provides us with a systematic method and a sufficient condition to check whether Q can be answered by accessing a bound number of tuples in all instances of D [36].
- For conjunctive queries Q, there exists a characterization, *i.e.*, a sufficient and necessary condition, to decide whether Q is scale independent under $\mathcal{A}$; better still, the decision problem is in polynomial time in the size of Q and $\mathcal{A}$ [17].
- If Q is scale independent under $\mathcal{A}$, then an efficient query plan can be worked out using the rules, such that we can find a core D_Q with a bounded size and $Q(D) = Q(D_Q)$. For conjunctive queries, there has been an experimental study with real-life data that shows such a query plan take 9 seconds as opposed to 14 hours by commercial system MySQL [17]! Moreover, it is easy to mine access constraints from real-life data, and a large percentage of queries are scale independent under simple access constraints. In other words, the approach by exploring scale independence is effective and practical.

3.2. Making Queries BD-tractable

We next turn to practical techniques for making big data small, and hence, BD-tractable. We take graph pattern matching in social graphs as our application domain, and present four data reduction strategies as examples, namely, distributed query processing via partial evaluation [47], query-preserving data compression [45], view-based query answering [50], and bounded incremental computation [49, 82]. The idea behind these approaches is simple. When our dataset D is a social graph G and Q is a pattern query, the complexity of computing query answer $Q(G)$ (the set of matches of Q in G) is measured by a function $f(|Q|, |G|)$. Since $f(\cdot, \cdot)$ may be the lower bound of the computation and cannot be further reduced, and $|Q|$ is typically small in practice, we reduce $|G|$, *i.e.*, by making big G small, to reduce the response time of query answering.

Graph pattern matching. We start with a review of graph pattern matching in social graphs, which typically represent social networks, *e.g.*, Facebook, Twitter, LinkedIn.

Social graphs. A social graph is a node-labeled 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 node v to v'; and (c) $f_A(\cdot)$ is a function that associates each node 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 $v.A_i$. In social graphs, each node denotes a person, and its attributes carry the contents of the node, *e.g.*, label, keywords, blogs, rating. An edge represents a relationship between two people.

Patterns. A *graph pattern* is given as $Q = (V_Q, E_Q, f_v)$, where

- V_Q is a finite set of nodes and E_Q is a set of directed edges, as defined for social graphs; and
- $f_v(\cdot)$ is a function defined on V_Q such that for each node u, $f_v(u)$ is the *search condition* for 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 $<, \leq, =, \neq, >, \geq$.

We say that a node v in a social 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 defined by $f_A(v)$ such that $v.A$ op a.

Graph pattern matching. Given a social graph G and a graph pattern Q, we want to compute the set $Q(G)$ of all matches in G for Q. In this section we consider a simple semantics for graph pattern matching, based on graph simulation [77], which has been widely used in Web site classification and social position detection, among other things (*e.g.,* [15, 19, 79, 97]).

We say that a social graph G *matches* a graph pattern Q via *graph simulation*, denoted by $Q \trianglelefteq_{\mathsf{sim}} G$, if there exists a binary relation $S \subseteq V_Q \times V$ that is inductively defined as follows:

- for each pattern node $u \in V_Q$, there exists a node $v \in V$ in the social graph such that $(u,v) \in S$; and
- for each $(u,v) \in S$, (a) $u \sim v$, and (b) for each edge (u,u') in E_Q, there is an edge (v,v') in E such that $(u',v') \in S$.

We refer to S as a *match* in G for Q.

It is known that if $Q \trianglelefteq_{\mathsf{sim}} G$, then there exists a unique *maximum* match S_o [62], *i.e.*, for any match S in G for Q, $S \subseteq S_o$. We define $Q(G)$ = S_o if $Q \trianglelefteq_{\mathsf{sim}} G$, and $Q(G)$ = $\emptyset$ otherwise.

It is known that it takes $O(|Q|^2 + |Q||G| + |G|^2)$ time to compute S_o [62], where $|G|$ denotes the size of G measured in the number of nodes and edges; similarly for the size $|Q|$ of Q. As remarked earlier, real-life social graphs are typically big, *e.g.*, Facebook graph has more than 1 billion nodes and 140 billion links [28]. Hence it is often prohibitively expensive to compute $Q(G)$ for social graphs G in the real world. These highlight the need for developing efficient techniques for graph pattern matching to cope with the sheer size of G.

Distributed query processing with partial evaluation. Distributed query processing is perhaps the most popular approach to querying big data, notably MapReduce [23]. Here we advocate distributed query processing with partial evaluation.

Partial evaluation has been used in a variety of applications including compiler generation, code optimization and dataflow evaluation (see [68] for a survey). 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 as much as possible 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 graph 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 $Q(G)$ as follows.

(1) The same pattern Q is posted to each fragment in $\mathcal{F}$.
(2) Upon receiving pattern Q, each site S_i computes a *partial answer* $Q(F_i)$ of Q in fragment F_i, *in parallel*, by taking F_i as the known input s while treating the fragments that reside 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 the answer $Q(G)$ to Q in the entire graph G.

The idea behind this is simple: we divide a big G into a collection $\mathcal{F}=(F_1,\ldots,F_n)$ of fragments, such that the response time is determined by the cost of computing $Q(F_m)$ (step 2), where F_m is the largest fragment in $\mathcal{F}$, and the cost of assembling partial answers (step 3). In other words, its parallel computational cost is dominated by the largest fragment F_m, rather than the original big graph G. In this way, we reduce a big G to small fragments F_i, and hence, reduce the response time. When G is not already partitioned and distributed, one may first partition G as *preprocessing*. In particular, when we can afford a number of processors, each F_i may have a manageable size and hence, the computation of $Q(F_i)$ is feasible at each site.

There are many ways to develop distributed algorithms for graph pattern matching. To evaluate and assess these algorithms, we propose the following criteria. We say that a distributed algorithm $\mathcal{T}$ is *scalable parallel* if for all patterns Q, all graphs G and all fragmentations $\mathcal{F}$ of G,

- if its parallel computation cost is bounded by a polynomial in $|Q|$, $|F_m|$ and $|V_f|$, and
- the total data shipped is bounded by a polynomial in $|Q|$ and $|V_f|$,

where V_f is the set of nodes with edges across different fragments in $\mathcal{F}$. That is, the response time of $\mathcal{T}$ is dominated by the size of the query, the largest fragment in $\mathcal{F}$, and how $\mathcal{F}$ partitions G, *rather than* by the size of the underlying G; similarly for its network traffic. In practice $|V_f|$ is typically much smaller than $|G|$, and $|Q|$ is also small. Hence, if algorithm $\mathcal{T}$ has this property, then the more processors are available, the smaller the fragments tend to be, and therefore, the less parallel computation time and network traffic are needed,

Note that MapReduce algorithms require us to re-distribute the data in each round of Map and Reduce; hence, they are not scalable parallel. In contrast, there exist scalable parallel algorithms for distributed graph simulation based on partial evaluation. Part of the results has been reported in [47] for patterns defined in terms of regular expressions. It is shown that there exists a distributed algorithm to answer such pattern queries

- by visiting each site once,
- in $O(|F_m||Q|^2+|Q|^2|V_f|^2)$ time, and
- with $O(|Q|^2|V_f|^2)$ communication cost.

That is, it has performance guarantees on both response time and communication cost, as well as on site visits.

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 can achieve a *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 polylog-time, perhaps by combining with other techniques such as indexing and distributed processing, then $\mathcal{Q}$ is BD-tractable.

The effectiveness of this approach has been verified [45], for graph pattern matching based on graph simulation, and for reachability queries as a special case (*i.e.*, whether there exists a path from one node to another via social links). More specifically, the following has been reported in [45].

- There exists a query preserving compression $\langle R, P\rangle$ for graph pattern matching with 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.
- 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 graph 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 [45]. 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.

Graph pattern matching using views. This technique is commonly used (see [73, 59] 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' produce the same answers in D, and moreover, (b) Q' refers only to $\mathcal{V}$ and its extensions $\mathcal{V}(D)$, *without accessing the underlying* D.

View-based query answering suggests another approach to making big data to small. As an example, consider graph pattern queries for social network analysis. Given a big graph G, one may identify a set $\mathcal{V}$ of views (pattern queries) and materialize them with $\mathcal{V}(G)$ 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 $\mathcal{V}(G)$ only. In practice, $\mathcal{V}(G)$ is typically much smaller than G, and hence, this approach allows us to query big G by accessing small $\mathcal{V}(G)$. Better still, the views can be incrementally maintained in response to changes to G, and adaptively adjusted to cover various patterns. In light of this, this approach has generated renewed interest for querying big graphs as well as other forms of big data [8, 36, 50].

More specifically, for pattern queries based on graph simulation in social network analysis, we know the following [50]. Given a graph pattern Q and a set $\mathcal{V}$ of view definitions,

- it is in $O(|Q|^2|\mathcal{V}|)$ time to decide whether query Q can be answered by using views $\mathcal{V}$; and if so,
- $Q(G)$ can be computed in $O(|Q||\mathcal{V}(G)|+|\mathcal{V}(G)|^2)$ time;
- better still, $|\mathcal{V}(G)|$ is about 4% of $|G|$ (*i.e.*, $|V|+|E|$) on average for real-life social graphs; and as a result of these,
- the view-based approach takes no more than 6% of the time needed for computing $Q(G)$ directly in G on average.

Contrast these with the $O(|Q|^2+|Q||G|+|G|^2)$ complexity of graph simulation! Note that $|Q|$ and $|\mathcal{V}|$ are sizes of pattern queries and are typically much smaller than G in real life.

Incremental graph pattern matching. Given a pattern Q and a graph G, as *preprocessing* we compute $Q(G)$ once. When G is updated by ΔG, instead of recomputing $Q(G \oplus \Delta G)$ starting from scratch, we incrementally compute ΔM such that $Q(G \oplus \Delta G) = Q(G) \oplus \Delta M$, to minimize unnecessary recomputation. In real life, ΔG is typically small: only 5% to 10% of nodes are updated weekly [80]. When ΔG is small, ΔM is often small as well, and is much less costly to compute than $Q(G \oplus \Delta G)$. The idea has also been adopted for querying big data [8, 36, 49].

The benefit is more evident if there exists a bounded incremental matching algorithm. As argued in [82], 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 $|Q|$ [49]. That is, its cost depends only on the size of *the changes* and the size of *pattern* Q, *independent of* the size of big G. This effectively makes big G small, since $|\mathsf{CHANGED}| \ll |G|$, and Q is typically small in practice.

For graph pattern matching via graph simulation, it has been shown that there exists a semi-bounded incremental algorithm in $O(|\Delta G|(|Q||\mathsf{CHANGED}|+|\mathsf{CHANGED}|^2))$ time [49].

In general, a query class $\mathcal{Q}$ can be considered BD-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.

Remarks and open issues. We remark the following.

(1) There are a number of other effective techniques for querying big data, notably indexing we have seen earlier. These techniques and the strategies outlined above can be, and should be, combined together, when querying big data.

(2) View-based and incremental techniques can help us make queries scale independent [36]. More specifically, when a query Q is not scale independent, we may still make it feasible to query big data *incrementally*, *i.e.*, to evaluate Q incrementally in response to changes ΔD to D, by accessing a M-fraction of the dataset D. That is, we compute $Q(D)$, once and offline, and then incrementally answer Q on demand. We may also achieve *scale independence using views*, *i.e.*, when a set $\mathcal{V}$ of views is defined, we rewrite Q into Q' using $\mathcal{V}$, such that for any dataset D, we can compute $Q(D)$ by using Q', which accesses materialized views $\mathcal{V}(D)$ and fetches only a bounded amount of data from D. We refer the interested reader to [36] for details.

We conclude the section with several open issues.

(1) As we have seen in Section 3.1, access schemas help us determine whether a query is scale independent and if so, develop an efficient plan to evaluate the query. A practical question asks how to design an "optimal" access schema for a given query workload, such that we can answer as many given queries as possible by accessing a bounded amount of data.

(2) As remarked earlier, Boolean conjunctive queries are scale independent even in the absence of access schema. A natural question is: given a Boolean conjunctive query Q and a dataset D on which Q is defined, how can we efficiently identify a core of D for answering Q, in the absence of access schema?

(3) The third question concerns distributed pattern matching. Does there exist a distributed algorithm at all that, given a pattern query Q and a graph G that is partitioned into $\mathcal{F} = (F_1, \ldots, F_n)$, computes the matches $Q(G)$ of Q in G, such that its response time and data shipment depend on the size of Q and the largest fragment F_m of $\mathcal{F}$ only? This question asks about the possibility or impossibility of distributed query processing with certain performance guarantees. Recent work has shown that this is beyond reach for distributed graph simulation (although distributed simulation has certain performance guarantees) [51]. However, the question remains open for distributed pattern matching by, *e.g.*, subgraph isomorphism.

(4) A more general question asks about parallel scalability: for a query class, does there exist an algorithm for answering its queries such that the more processors are used, the less time it takes? That is, if we could afford "unlimited" resources, then a parallel scalable algorithm makes it feasible to answer the queries on big data, by using more computing facilities. There has been work on this issue. Unfortunately, the prior work focuses on either shared-memory architectures [72] or MapReduce [69, 89]. A "standard" notion of parallel scalability is not yet in place for general shared-nothing architectures, which are widely used in industry.

(5) As we have seen, view-based query answering provides us with an effective technique for querying big data. To make practical use of it, however, we need to answer the following question. Given a query workload, what views should we select to build and maintain, such that the queries can be efficiently answered by using views or better still, be scale independent?

4. Approximate Query Answering

The strategies we have seen in Section 3 help us make it feasible to answer *some* queries in big data. However, some queries may not be made BD-tractable. An example is graph pattern matching defined with subgraph isomorphism: it is NP-complete even to decide whether there exists a match (cf. [81]). For such queries, it is beyond reach to find exact answers in big data. Moreover, as remarked earlier, even for queries that can be answered in PTIME, it is sometimes too costly to compute their exact answers in big data. In light of this, we often have to evaluate these queries by using *inexact* algorithms, preferably approximation algorithms with performance guarantees.

This section proposes two approaches to developing approximation algorithms for answering queries in big data, referred to as query-driven and data-driven approximation.

4.1. Query Driven Approximation

For some query classes $\mathcal{Q}$ we can relax its semantics, such that it is less costly to answer queries Q of $\mathcal{Q}$ in a big dataset D under the new semantics, and moreover, the answer $Q(D)$ still gives users what they want. To illustrate this, we give two examples: graph pattern matching and top-k query answering.

Graph pattern matching revisited. We first review graph pattern matching defined in terms of subgraph isomorphism. Consider a social graph $G = (V, E, f_A)$ and a graph pattern $Q = (V_Q, E_Q, f_v)$ as defined in Section 3.2. Consider a subgraph $G' = (V', E', f'_A)$ of G, where V' is a subset of V, and E' and f'_A are restrictions of E and f_A on V', respectively.

We say that G' *matches* Q *by isomorphism*, denoted as $Q \trianglelefteq_{\mathsf{iso}} G'$, if there is a *bijective function* $h(\cdot) : V_Q \rightarrow V'$ such that

- $u \sim h(u)$ for each node $u \in V_Q$, and
- for each pair (u, u') of nodes in V_Q, $(u, u') \in E_Q$ if and only if $(h(u), h(u')) \in E'$.

Graph pattern matching *by subgraph isomorphism* is to compute, given a social graph G and a graph pattern Q, the set $Q(G)$ of all subgraphs G' of G such that $Q \trianglelefteq_{\mathsf{iso}} G'$. This semantics has been proposed for social graph analysis. However, it is intractable even in the classical computational complexity theory to compute $Q(G)$ based on subgraph isomorphism.

In light of the high complexity, we adopt graph simulation for graph pattern matching instead of subgraph isomorphism [42]. That is, we check $Q \trianglelefteq_{\mathsf{sim}} G$ (Section 3) rather than $Q \trianglelefteq_{\mathsf{iso}} G'$ for subgraphs G' of G. In fact, several revisions of graph simulation have been proposed, by allowing pattern edges to map to paths [42], incorporating edge labels [41], and retaining the topology of graph patterns [74]. These reduce the complexity of graph pattern matching from intractability (subgraph isomorphism) to low polynomial time (quadratic time or cubic time). Better still, it has been shown using real-life social networks that graph pattern matching with (revisions of) graph simulation is able to capture more sensible matches in social graph analysis than subgraph isomorphism can find. In other words, by relaxing the semantics of graph pattern matching from subgraph isomorphism to (revised) graph simulation, we can find high-quality matches for social data analysis in much less time.

Top-k graph pattern matching. As remarked earlier, even quadratic-time or cubic-time complexity may be too high when querying big data. In light of this, we may further relax the semantics of graph pattern matching defined with (revised) graph simulation and hence reduce the cost of the computation.

In social data analysis we often want to find matches of a particular pattern node u_o in Q as "query focus" [11]. That is, we just want those nodes in a social graph G that are matches of u_o in $Q(G)$, rather than the entire set $Q(G)$ of matches for Q. Indeed, a recent survey shows that 15% of social queries are to find matches of specific pattern nodes [78]. Moreover, it often suffices to find top-k matches of u_o in $Q(G)$. 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 $Q(G)$, 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 $Q(G)$, U includes all the matches (see, *e.g.,* [30], for top-k query answering).

This suggests that we develop algorithms to find top-k matches with *the early termination property* [30], *i.e.,* they stop as soon as a set of top-k matches is found, *without* computing the entire $Q(G)$. While the worst-case time complexity of such algorithms may be no better than their counterparts for computing the entire $Q(G)$, 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 defined in terms of graph simulation, we find that top-k matching algorithms just inspect 65%–70% of the matches in $Q(G)$ on average in real-life social graphs [48], even when diversity is taken into account to remedy the over-specification problem of retrieving too homogeneous answers [56], which makes top-k query answering a much harder bi-criteria optimization problem [24].

4.2. Data Driven Approximation

In some applications we may not be able to relax the semantics of our queries. To this end, we propose a data-driven approximation strategy, referred to as resource-bounded approximation. Below we first review traditional approximation schemes, and then introduce resource-bounded approximation.

Traditional approximation algorithms. Previous work on this subject has mostly focused on developing PTIME approximation algorithms for NP-optimization problems (NPOs) [21, 58, 90]. 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 $\eta(\cdot)$ from natural numbers to the range $(0,1]$.

An algorithm $\mathcal{T}$ is called *a η-approximation algorithm for problem A* if for each instance $x \in I$, $\mathcal{T}$ computes a feasible solution y of x such that $R(x,y) \geq \eta(|x|)$, where $R(x,y)$ is the *performance ratio* of y *w.r.t.* x, defined as follows [21]:

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

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

However, such PTIME approximation algorithms directly operate on the original instances of a problem, and may not work well when querying big data for the following reasons.

(1) As we have seen in Section 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 [58], not limited to NPO.

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

Resource-bounded approximation. To cope with this, below we propose resource-bounded approximation. In a nutshell, given a small ratio $\alpha \in (0,1)$ and a query Q posed on a dataset D, we extract a fraction D_Q of D such that $|D_Q| \leq \alpha|D|$, and compute *approximate answers* $Q(D_Q)$. Here α is called a *resource ratio* or a *resolution*. It is determined by our available resources for query evaluation, such as time and space.

Intuitively, the idea is the same as how we process our photos. When we cannot afford the time or storage for photos of high resolution, we settle with smaller images with lower resolution to reduce the cost, as long as such images are not too rough.

To formalize the idea, we first revise the notion of performance ratios for query answering. We then define resource-bounded approximation and demonstrate its effectiveness.

<u>*Accuracy of query answers*</u>. Consider a query Q and a dataset D. The exact answers to Q in D are typically a set $Q(D)$. Suppose that an algorithm $\mathcal{T}$ computes a set Y of approximate answers to Q in D. We define the *precision and recall of the set Y for* (Q,D) in the standard way, as follows:

$$\mathsf{precision}(Q,D,Y) = \frac{|Y \cap Q(D)|}{|Y|},$$

$$\mathsf{recall}(Q,D,Y) = \frac{|Y \cap Q(D)|}{|Q(D)|}.$$

That is, precision is the ratio of the number of correct answers in Y to the total number of answers in Y, while recall is the ratio of the number of correct answers in Y to the total number of exact answers in $Q(D)$. Based on these, we define the *accuracy of Y for (Q, D)* by adopting the usual F-measure [93]:

$$\mathsf{accuracy}(Q,D,Y) = 2\,\frac{\mathsf{precision}(Q,D,Y)\ \mathsf{recall}(Q,D,Y)}{\mathsf{precision}(Q,D,Y)+\mathsf{recall}(Q,D,Y)}$$

as the harmonic mean of precision and recall. Obviously, the larger $\mathsf{accuracy}(Q,D,Y)$ is, the more accurate Y is.

When both $Q(D)$ and Y are $\emptyset$, *i.e.,* no answer exists, we treat $\mathsf{accuracy}(Q,D,Y)$ as 1; we consider precision only if $Q(D)$ is $\emptyset$ but Y is not, and recall only if D is $\emptyset$ but $Q(D)$ is not.

Resource-bounded query answering. We now present resource-bounded approximation algorithms. Let $\alpha \in (0,1)$ be a *resource ratio* (or *resolution*), and $\mathcal{Q}$ be a class of queries.

Given a dataset D and a query Q in $\mathcal{Q}$, an algorithm $\mathcal{T}$ for $\mathcal{Q}$ queries *with resource-bound* α does the following:

- visits a fraction D_Q of D such that $|D_Q| \leq \alpha|D|$, and
- computes $Q(D_Q)$ as approximate answers.

We say that $\mathcal{T}$ has *accuracy ratio* η for $\mathcal{Q}$ if for all datasets D and all queries $Q \in \mathcal{L}_Q$, $\mathsf{accuracy}(Q,D,Q(D_Q)) \geq \eta$.

Note that the accuracy ratio η is in the range $(0,1]$. When $\eta = 1$, algorithm $\mathcal{T}$ finds *exact answers* for all datasets D and queries Q *i.e.,* the algorithm has 100% accuracy.

Algorithm $\mathcal{T}$ consists of two steps: it first reduces big D to a small D_Q, and then computes approximate query answers, both by accessing a bounded amount of data. Observe the following.

(1) *Dynamic reduction*. Recall that traditional data reduction schemes such as compression, summarization and data synopses, build *the same structure for all queries* [2, 9, 27, 53, 54, 65, 66, 70, 84, 91]. This is also how the strategies of Section 3.2 do. We refer to such strategies as *uniform reduction*.

In contrast, resource-bounded approximation adopts a *dynamic reduction strategy*, which finds a small dataset D_Q with only information needed for an *input query* Q, and hence, allows higher accuracy within the bound $\alpha|D|$ on data accessed. One can use any techniques for dynamic reduction, including those for data synopses such as sampling and sketching, as long as the process *visits a bounded amount of data in* D.

(2) *Approximate query answering*. Algorithm $\mathcal{T}$ computes $Q(D_Q)$ by accessing $\alpha|D|$ amount of data rather than the entire D. It aims to achieve the best performance ratio within $\alpha|D|$.

(3) *Scale independence*. When Q is scale independent in D *w.r.t.* some $M \geq \alpha|D|$, resource-bounded approximation achieves 100% accuracy, *i.e.,* with performance ratio $\eta = 1$.

(4) *Access schema*. The notion of resource-bounded approximation can be readily defined under an access schema $\mathcal{A}$ (see Section 3.1), to efficiently retrieve a bounded amount of data for query processing by leveraging indices and bounds in $\mathcal{A}$.

Personalized social search. To verify the effectiveness of the approach, we have conducted a preliminary study of personalized social search in real-life social graphs [52]. Such searches are supported by Graph Search of Facebook, *e.g.,* "find me all my friends in Beijing who like cycling" [29].

A personalized search is specified by a graph pattern Q in which a node u_p is designated to map to a particular node (person) v_p in a social graph G. As in the case for top-k graph pattern matching described earlier, the pattern Q also has a particular "output" pattern node u_o. The search is to compute $Q(G)$, the set of all matches of the output pattern node u_o of Q in graph G, while the "personalized" node u_p is mapped to v_p in G. Such searches are similar to what we have seen in Example 6. In contrast to queries given there, here we consider queries Q that are graph patterns rather than relational queries, and moreover, may not be scale independent in G.

For such patterns, we have developed resource-bounded approximation algorithms for graph pattern matching defined in terms of subgraph isomorphism and graph simulation (see Section 3.2). We have experimented with these algorithms using real-life social graphs. The results are very encouraging. We find that our algorithms are efficient: they are 135 and 240 times faster than traditional pattern matching algorithms based on graph simulation and subgraph isomorphism, respectively, Better still, the algorithms are accurate: even when the resource ratio α is as small as $15*10^{-6}$, the algorithms return matches with 100% accuracy! Observe that when G consists of 1PB of data, $\alpha|G|$ is down to 15GB, *i.e.,* resource-bounded approximation truly makes big data small, without paying too high a price of sacrificing the accuracy of query answers.

A similar idea has also been verified effective by BlinkDB [4]. BlinkDB adaptively samples data to find approximate answers to relational queries within a probabilistic error-bound and time constraints. In other words, it answers queries using data samples D_Q of a dataset D, instead of D.

Open issues. There is naturally more to be done.

(1) For a class $\mathcal{Q}$ of queries, the first problem is to find, given a resource ratio α, *the maximum provable accuracy ratio* η that resource-bounded algorithms can *guarantee* for $\mathcal{Q}$. A dual problem is to find, given an accuracy guarantee η, *the minimum resource ratio* α that resource-bounded algorithms can take.

(2) Another problem is to study, given an access schema $\mathcal{A}$, how can we develop a resource-bounded algorithm that makes maximum use of $\mathcal{A}$ to retrieve data efficiently, *i.e.,* it visits a minimum amount of data that is not covered by $\mathcal{A}$.

(3) The third topic is to develop resource-bounded approximation algorithms in various application domains. For instance, for social searches that are not personalized, *i.e.,* when no nodes in a graph pattern are designated to map to fixed nodes in a social graph G, can we develop effective resource-bounded approximation algorithms for graph pattern matching?

(4) Finally, approximation classes for resource-bounded approximation need to be defined, along the same lines as their counterparts for traditional approximation algorithms (*e.g.,* APX, PTAS, FPTAS [21]). Similarly, approximation-preserving reductions should be developed, and complete prob-

lems for those classes need to be identified for these classes.

5. Data Quality: The Other Side of Big Data

We have so far focused only on how to cope with the volume (quantity) of big data. Nonetheless, as remarked earlier, *big data = quantity + quality*. This section addresses data quality issues. We report the state of the art of this line of research, and identify challenges introduced by big data. The primary purpose of this section is to advocate the study of the quality of big data, which has been overlooked by and large, although data quality and data quantity are equally important.

5.1. Central Issues of Data Quality

We begin with an overview of central technical issues in connection with data quality. We then present current approaches to tackling these issues. We invite the interested reader to consult [33] for a recent survey on the subject.

Data quality problems. Data in the real world is often dirty. It is common to find real-life data inconsistent, inaccurate, incomplete, out of date and duplicated. Error rate of business data is approximately 1%–5%, and for some companies it is above 30% [83]. In most data warehouse projects, data cleaning accounts for 30%-80% of the development time and budget [87], for improving the quality of the data rather than for developing the systems. When it comes to incomplete information, it is estimated that "pieces of information perceived as being needed for clinical decisions were missing from 13.6% to 81% of the time" [76]. When data currency is concerned, it is known that "2% of records in a customer file become obsolete in one month" [26]. That is, in a database of 500 000 customer records, 10 000 records may go stale per month, 120 000 records per year, and within two years about 50% of all the records may be obsolete. As remarked earlier, the scale of the data quality problem is far worse in the context of big data.

Why do we care about dirty data? As shown in Example 2, we may not get correct query answers if our data is dirty. As a result, dirty data routinely leads to misleading analytical results and biased decisions, and accounts for loss of revenues, credibility and customers. For example, it is reported that dirty data cost US businesses 600 billion dollars every year [26].

Below we highlight five central issues of data quality.

Data consistency refers to the validity and integrity of data representing real-world entities. It aims to detect inconsistencies or conflicts in the data. For instance, tuple t_1 of Figure 1 is inconsistent: its area code is 20 while its city is Beijing.

Inconsistencies are identified as violations of *data dependencies* (*a.k.a.* integrity constraints [1]). Errors in a single relation can be detected by intrarelation constraints such as conditional functional dependencies (CFDs) [34], while errors across different relations can be identified by interrelation constraints such as conditional inclusion dependencies (CINDs) [75]. An example CFD for the data of Figure 1 is: city = "Beijing" $\rightarrow \mathsf{AC} = 10$, asserting that for any tuple t, if $t[\mathsf{city}]$ = "Beijing", then $t[\mathsf{AC}]$ must be 10. As a data quality rule, this CFD catches the inconsistency in tuple t_1: $t_1[\mathsf{AC}]$ and $t[\mathsf{city}]$ violate the CFD.

Data accuracy refers to the closeness of values in a database to the true values of the entities that the database values represent. Observe that data may be consistent but not accurate. For instance, one may have a rule for data consistency: $\mathsf{age} \leq 120$, indicating that a person's age does not exceed 120. Consider a tuple t representing a high school student, with $t[\mathsf{age}] = 40$. While t is not inconsistent, it may not be accurate: a high school student is typically no older than 19 years old.

There has been recent work on data accuracy [16]: given tuples t_1 and t_2 pertaining to the same entity e, we decide whether t_1 is more accurate than t_2 *in the absence of* the true value of e. It is also based on integrity constraints as data quality rules.

Information completeness concerns whether our database has complete information to answer our queries. Given a database D and a query Q, we want to know whether the complete answer to Q can be found by using only the data in D. As shown in Example 2, when D does not include complete information for a query, the answer to the query may not be correct.

Information completeness has been a longstanding problem. A theory of relative information completeness has recently been proposed [32], to decide whether our database has complete information to answer our queries, and if not, how we can expand the database and make it complete, by including more data.

Data currency is also known as *timeliness*. It aims to identify the current values of entities, and to answer queries with the current values, in the absence of valid timestamps.

For example, recall the dataset D_0 from Figure 1. Suppose that we know that tuples t_1, t_2 and t_3 refer to the same person Mary. Note that these tuples have two distinct values for salary: 50k and 80k, one is current and the other is stale. We want to decide which one is current, when their timestamps are missing.

A data currency theory has recently been proposed in [40], to deduce data currency when temporal information is only partly known or not available at all. It is based on data quality rules defined in terms of temporal constraints. For instance, we can specify a rule asserting that the salary of each employee in a company does *not* decrease, as commonly found in the real world. Then we can deduce that Mary's current salary is 80k.

Data deduplication aims to identify tuples in one or more relations that refer to the same real-world entity. It is also known as entity resolution, duplicate detection, record matching, record linkage, merge-purge, database hardening, and object identification (for data with complex structures such as graphs).

For example, consider tuples t_1, t_2 and t_3 in Figure 1. To answer query Q_0 of Example 1, we want to know whether these tuples refer to the same employee Mary. The answer is affirmative if, *e.g.,* there exists another relation which indicates that Mary Smith and Mary Webber have the same email account.

The need for studying data deduplication is evident in data cleaning, data fusion and payment card fraud detection, among other things. No matter how important it is, data deduplication is nontrivial. Tuples pertaining to the same object may have different representations in various data sources. Moreover, the data sources may contain errors. These make it hard, if not impossible, to match a pair of tuples by simply checking whether their attributes pairwise equal. Worse still, it is often too costly to compare and examine every pair of tuples from big data.

Data deduplication is perhaps the most extensively studied topic of data quality. A variety of approaches have been proposed (see [63] for a survey). In particular, a class of dynamic constraints has been studied for data deduplication, known as matching dependencies (MDs), as data quality rules [31].

Improving data quality. We have seen that real-life data is often dirty, and dirty data is costly. In light of these, effective techniques have to be in place to improve data quality. To do this, a central question concerns how we can tell whether our data is dirty or clean. To this end, we need data quality rules to detect semantic errors in our data and fix those errors. A number of dependency (constraint) formalisms have been proposed as data quality rules, and are being used in industry, *e.g.,* CFDs, CINDs and MDs. Below we briefly describe the basic functionality of a rule-based system for data quality management.

Discovering data quality rules. To use dependencies as data quality rules, it is necessary to have efficient techniques in place that can *automatically discover* dependencies from data. Indeed, it is unrealistic to just rely on human experts to design data quality rules via an expensive and long manual process, or count on business rules that have been accumulated. This suggests that we learn informative and interesting data quality rules from (possibly dirty) data, and prune away insignificant rules.

More specifically, given a database D, the *discovery problem* is to find a *minimal cover* of all dependencies (*e.g.,* CFDs, CINDs, MDs) that hold on D, *i.e.,* a non-redundant set of dependencies that is logically equivalent to the set of all dependencies that hold on D. Several algorithms have been developed for discovering CFDs and MDs (*e.g.,* [18, 35, 55]).

Validating data quality rules. A given set Σ of dependencies, either automatically discovered or manually designed by domain experts, may be dirty itself. In light of this we have to identify "consistent" dependencies from Σ, *i.e.,* those rules that make sense, to be used as data quality rules. Moreover, we need to remove redundancies from Σ via the implication analysis of the dependencies, to speed up data cleaning process.

This problem is nontrivial. It is NP-complete to decide whether a given set of CFDs is satisfiable [34]. Nevertheless, there has been an approximation algorithm for extracting a set Σ' of consistent rules from a set Σ of possibly inconsistent CFDs, while guaranteeing that Σ' is within a constant bound of the maximum consistent subset of Σ (see [34] for details).

Detecting errors. After a validated set of data quality rules is identified, the next question concerns how to effectively catch errors in a database by using these rules. Given a set Σ of consistent data quality rules and a database D, we want to *detect inconsistencies* in D, *i.e.,* to find all tuples in D that violate some rule in Σ. When it comes to relative information completeness, we want to decide whether D has complete information to answer an input query Q, among other things.

For a centralized database D, given a set Σ of CFDs and CINDs, a fixed number of SQL queries can be *automatically* generated such that, when being evaluated against D, the queries return all and only those tuples in D that violate Σ [33]. That is, we can effectively detect inconsistencies by leveraging existing facility of commercial relational database systems.

Data repairing. After the errors are detected, we want to automatically localize the errors and fix the errors. We also need to identify tuples that refer to the same entity, and for each entity, determine its latest and most accurate values from the data in our database. When some data is missing, we need to decide what data we should import and where to import it from, so that we will have sufficient information for tasks at hand.

This highlights the need for *data repairing* [5]. Given a set Σ of dependencies and an instance D of a database schema $\mathcal{R}$, it is to find a candidate *repair* of D, *i.e.,* another instance D' of $\mathcal{R}$ such that D' satisfies Σ and D' *minimally differs* from the original database D. The data repairing problem is, nevertheless, highly nontrivial: it is NP-complete even when a fixed set of traditional functional dependencies (FDs) or a fixed set of inclusion dependencies (INDs) is used as data quality rules [14]. In light of these, several heuristic algorithms have been developed, to effectively repair data by employing FDs and INDs [14], CFDs [20, 96], CFDs and MDs [46] as data quality rules.

The data repairing methods mentioned above are essentially heuristic: while they improve the overall quality, they do not guarantee to find correct fixes for each error detected, *i.e.,* they do not warrant a precision and recall of 100%. Worse still, they may introduce new errors when trying to repair the data. Hence, they are not accurate enough to repair critical data such as clinical data, in which a minor error may have disastrous consequences. This highlights the quest for effective methods to find *certain fixes* that are guaranteed correct. Such a method has been developed in [43]. It guarantees that whenever it updates data, it correctly fixes an error without introducing new errors.

The rule discovery, rule validation, error detection and data repairing methods mentioned above have been supported by commercial systems and have proven effective in industry.

5.2. New Challenges Introduced by Big Data

Previous work on data quality has mostly focused on relational data residing in a centralized database. To improve the quality of big data and hence, get sensible answers to our queries in big data, new techniques have to be developed.

Repairing distributed data. Big data is often distributed. In the distributed setting, all the data quality issues mentioned above become more challenging. For example, consider error detection. As remarked earlier, this is simple in a centralized database system: SQL queries can be automatically generated so that we can execute them against our database and catch all inconsistencies and conflicts. In contrast, this is more intriguing in distributed data: it necessarily requires us to ship data from one site to another. In this setting, error detection with minimum data shipment or minimum response time becomes NP-complete [37], and the SQL-based techniques no longer work.

For distributed data, effective batch algorithms [37] and incremental algorithms [44] have been developed for detecting errors, with certain performance guarantees. However, rule discovery and data repairing algorithms remain to be developed for distributed data. These are highly challenging. For instance, data repairing for centralized databases is already NP-complete even when a fixed set of FDs is taken as data quality rules [14], *i.e.*, when only the size $|D|$ of datasets is concerned (*a.k.a.* data complexity [1]). When D is of PB size and D is distributed, its computational and communication costs are prohibitive.

Deducing the true values of entities. To answer a query in big data, we may have to use data from tens of thousands sources [22]. With this comes the need for data fusion and conflict resolution [13]. That is, for each entity e, we need to identify the set D_e of data items that refer to the same e from those sources, and moreover, deduce the true value of e from D_e.

Example 8. *Recall Figure 1. Suppose that t_1, t_2 and t_3 come from different sources. We need data deduplication methods to determine whether they refer to the same person Mary. If so, we want to find the true values of Mary. To do this, we may need to, e.g., reason about both data currency and consistency. As an example, for attribute* LN *(last name), Mary has two conflict values: Smith and Webber. We want to know what is the latest and correct value. To this end, 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 deduce that the true value of* LN *of Mart is Webber.*

As another example, suppose that s_1 and s_2 of Figure 1 refer to the same person. To deduce the true value of his FN *(first name), we may use a CFD:* FN = *"Bob"* $\rightarrow$ FN = *"Robert". This rule for data consistency allows us to normalize the* FN *attribute and change nickname Bob to Robert.* □

From the example we can see that to deduce the true values of an entity, we need to *combine* several techniques: data deduplication, data consistency and data currency, among other things. This can be done in a uniform logical framework based on data quality rules. There has been recent preliminary work on the topic [39]. Nonetheless, there is much more to be done.

Cleaning data with complex structures. Data quality techniques have been mostly studied for structured data with a regular structure and a schema, such as relational data. When it comes to big data, however, data typically has an irregular structure and does not have a schema. For example, an entity may be represented as a subgraph in a large graph, such as a person in a social graph. In this context, all the central issues of data quality have to be revisited. These are far more challenging than their counterparts for relational data, and effective techniques are not yet in place. Consider data deduplication, for instance. Given two graphs (without a schema), we want to determine whether they represent the same object. To do this, we need to extend data quality rules from relations to graphs.

Coupling with knowledge bases. A large part of big data comes from Web sources or social networks. To improve the quality of such data, we ultimately have to use knowledge bases and ontology. A number of knowledge bases are being developed, such as Knowledge Graph [57], Yago [95], and Wiki [94]. However, the quality of these knowledge bases needs to be improved themselves. This suggests that we study the following. How to detect inconsistencies and conflicts in a knowledge base? How to repair a knowledge base? How to make use of available knowledge bases to clean data from the Web?

6. Conclusion

We have reported an account of recent work of the International Research Center on Big Data at Beihang University, on querying big data. Our main conclusion is as follows.

- Query answering in big data is radically different from what we know about querying traditional databases.
- We need to revise complexity theory and approximation theory to characterize what we can do and what is impossible for computing exact or approximate query answers.
- Querying big data is challenging, but doable. It calls for a set of new effective query processing techniques.
- Big data = quantity + quality. These are the two sides of the same coin, and neither works well when taken alone.

Summing up, we believe that the need for studying query answering in big data cannot be overstated, and that the subject is a rich source of questions and vitality. We reiterate our invitation to interested colleagues to join us in the study.

Acknowledgments. Fan and Huai are supported in part by 973 Program 2014CB340302. Fan is also supported in part by NSFC 61133002, 973 Program 2012CB316200, Guangdong Innovative Research Team Program 2011D005 and Shenzhen Peacock Program 1105100030834361, China, EPSRC EP/J015377/1, UK, and NSF III 1302212, US.

References

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

[2] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *SIGMOD*, pages 275–286, 1999.

[3] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.

[4] 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*, pages 29–42, 2013.

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

[6] M. Armbrust, K. Curtis, T. Kraska, A. Fox, M. J. Franklin, and D. A. Patterson. PIQL: Success-tolerant query processing in the cloud. *PVLDB*, 5(3):181–192, 2011.

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

[8] M. Armbrust, E. Liang, T. Kraska, A. Fox, M. J. Franklin, and D. Patterson. Generalized scale independence through incremental precomputation. In *SIGMOD*, pages 625–636, 2013.

[9] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *SIGMOD*, pages 539–550, 2003.

[10] Beihang University. International Research Center at Big Data. *http://rcbd.buaa.edu.cn/en/index.html*.

[11] M. Bendersky, D. Metzler, and W. Croft. Learning concept importance using a weighted dependence model. In *WSDM*, pages 31–40, 2010.

[12] M. Bienvenu, B. ten Cate, C. Lutz, and F. Wolter. Ontology-based data access: a study through disjunctive datalog, CSP, and MMSNP. In *PODS*, pages 213–224, 2013.

[13] J. Bleiholder and F. Naumann. Data fusion. *ACM Comput. Surv.*, 41(1), 2008.

[14] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, pages 143–154, 2005.

[15] J. Brynielsson, J. Högberg, L. Kaati, C. Martenson, and P. Svenson. Detecting social positions using simulation. In *ASONAM*, pages 48–55, 2010.

[16] Y. Cao, W. Fan, and W. Yu. Determining the relative accuracy of attributes. In *SIGMOD*, pages 565–576, 2013.

[17] Y. Cao, W. Fan, and W. Yu. Bounded conjunctive queries. *PVLDB*, pages 1231 – 1242, 2014.

[18] F. Chiang and R. J. Miller. Discovering data quality rules. *PVLDB*, 1(1):1166–1177, 2008.

[19] J. Cho, N. Shivakumar, and H. Garcia-Molina. Finding replicated Web collections. *SIGMOD Rec.*, 29(2):355–366, 2000.

[20] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *VLDB*, pages 315–326, 2007.

[21] P. Crescenzi, V. Kann, and M. Halldórsson. A compendium of NP optimization problems. *http://www.nada.kth.se/∼viggo/wwwcompendium/*.

[22] N. N. Dalvi, A. Machanavajjhala, and B. Pang. An analysis of structured data on the Web. *PVLDB*, 5(7):680–691, 2012.

[23] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[24] T. Deng and W. Fan. On the complexity of query result diversification. *PVLDB*, 6(8):577–588, 2013.

[25] R. Dorrigiv, A. López-Ortiz, and A. Salinger. Optimal speedup on a low-degree multi-core parallel architecture (LoPRAM). In *SPAA*, pages 185–187, 2008.

[26] W. W. Eckerson. Data quality and the bottom line: Achieving business success through a commitment to high quality data. Technical report, The Data Warehousing Institute, 2002.

[27] 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*, pages 226–231, 1996.

[28] Facebook. *http://newsroom.fb.com*.

[29] Facebook. Introducing Graph Search. *https://en-gb.facebook.com/about/graphsearch*.

[30] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *JCSS*, 66(4):614–656, 2003.

[31] W. Fan, H. Gao, X. Jia, J. Li, and S. Ma. Dynamic constraints for record matching. *VLDB J.*, 20(4):495–520, 2011.

[32] W. Fan and F. Geerts. Relative information completeness. *ACM Trans. on Database Systems*, 35(4), 2010.

[33] W. Fan and F. Geerts. *Foundations of Data Quality Management*. Morgan & Claypool Publishers, 2012.

[34] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Trans. on Database Systems*, 33(1), 2008.

[35] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *TKDE*, 23(5):683–698, 2011.

[36] W. Fan, F. Geerts, and L. Libkin. On scale independence for querying big data. In *PODS*, pages 51–62, 2014.

[37] W. Fan, F. Geerts, S. Ma, and H. Müller. Detecting inconsistencies in distributed data. In *ICDE*, pages 64–75, 2010.

[38] W. Fan, F. Geerts, and F. Neven. Making queries tractable on big data with preprocessing. *PVLDB*, 6(8):577–588, 2013.

[39] W. Fan, F. Geerts, N. Tang, and W. Yu. Inferring data currency and consistency for conflict resolution. In *ICDE*, pages 470–481, 2013.

[40] W. Fan, F. Geerts, and J. Wijsen. Determining the currency of data. *ACM Trans. on Database Systems*, 37(4), 2012.

[41] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding regular expressions to graph reachability and pattern queries. In *ICDE*, pages 39–50, 2011.

[42] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractability to polynomial time. *PVLDB*, 3(1):1161–1172, 2010.

[43] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Towards certain fixes with editing rules and master data. *VLDB J.*, 21(2):213–238, 2012.

[44] W. Fan, J. Li, N. Tang, and W. Yu. Incremental detection of inconsistencies in distributed data. *TKDE*, 2014.

[45] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *SIGMOD*, pages 157–168, 2012.

[46] W. Fan, S. Ma, N. Tang, and W. Yu. Interaction between record matching and data repairing. *ACM J. of Data and Information Quality*, 2014.

[47] W. Fan, X. Wang, and Y. Wu. Performance guarantees for distributed reachability queries. *PVLDB*, 5(11):1304–1315, 2012.

[48] W. Fan, X. Wang, and Y. Wu. Diversified top-k graph pattern matching. *PVLDB*, 6(13):1510–1521, 2013.

[49] W. Fan, X. Wang, and Y. Wu. Incremental graph pattern matching. *ACM Trans. on Database Systems*, 38(3), 2013.

[50] W. Fan, X. Wang, and Y. Wu. Answering graph pattern queries using views. In *ICDE*, pages 184–195, 2014.

[51] W. Fan, X. Wang, and Y. Wu. Distributed graph simulation: Impossibility and possibility. *PVLDB*, pages 1083 – 1094, 2014.

[52] W. Fan, X. Wang, and Y. Wu. Querying big graphs within bounded resources. In *SIGMOD*, pages 301–312, 2014.

[53] M. N. Garofalakis and P. B. Gibbons. Wavelet synopses with error guarantees. In *SIGMOD*, pages 476–487, 2004.

[54] P. B. Gibbons and Y. Matias. Synopsis data structures for massive data sets. In *SODA*, pages 909–910, 1999.

[55] L. Golab, H. Karloff, F. Korn, D. Srivastava, and B. Yu. On generating near-optimal tableaux for conditional functional dependencies. *PVLDB*, 1(1):376–390, 2008.

[56] S. Gollapudi and A. Sharma. An axiomatic approach for result diversification. In *WWW*, pages 381–390, 2009.

[57] Google. Knowledge Graph. *http://www.google.co.uk/insidesearch/features/ search/knowledge.html*.

[58] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, 1995.

[59] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[60] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Trans. American Mathematical Society*, 117:285–306, May 1965.

[61] J. M. Hellerstein. The declarative imperative: Experiences and conjectures in distributed logic. *SIGMOD Record*, 39(1):5–19, 2010.

[62] M. R. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, pages 453–462, 1995.

[63] T. N. Herzog, F. J. Scheuren, and W. E. Winkler. *Data Quality and Record Linkage Techniques*. Springer, 2009.

[64] IBM. IBM big data platform. *http://www-01.ibm.com/software/data/bigdata/*.

[65] Y. E. Ioannidis and V. Poosala. Histogram-based approximation of set-valued query-answers. In *VLDB*, pages 174–185, 1999.

[66] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *VLDB*, pages 275–286, 2009.

[67] D. S. Johnson. A catalog of complexity classes. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*. The MIT Press, 1990.

[68] N. D. Jones. An introduction to partial evaluation. *ACM Comput. Surv.*, 28(3):480–503, 1996.

[69] H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for MapReduce. In *SODA*, pages 938–948, 2010.

[70] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley, 1990.

[71] P. Koutris and D. Suciu. Parallel evaluation of conjunctive queries. In *PODS*, pages 223–234, 2011.

[72] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *TCS*, 71(1):95–132, 1990.

[73] M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, pages 233–246, 2002.

[74] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Strong simulation: Capturing topology in graph pattern matching. *ACM Trans. on Database Systems*, 39(1), 2014.

[75] S. Ma, W. Fan, and L. Bravo. Extending inclusion dependencies with conditions. *TCS*, pages 64–95, 1998.

[76] D. W. Miller Jr., J. D. Yeast, and R. L. Evans. Missing prenatal records at a birth center: A communication problem quantified. In *AMIA Annu Symp Proc.*, pages 535–539, 2005.

[77] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[78] 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*, pages 1739–1748, 2010.

[79] L. D. Nardo, F. Ranzato, and F. Tapparo. The subgraph similarity problem. *TKDE*, 21(5):748–749, 2009.

[80] 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*, pages 1 – 12, 2004.

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

[82] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *TCS*, 158(1-2):213–224, 1996.

[83] T. Redman. The impact of poor data quality on the typical enterprise. *Commun. ACM*, 2:79–82, 1998.

[84] P. Rösch and W. Lehner. Sample synopses for approximate answering of group-by queries. In *EDBT*, pages 403–414, 2009.

[85] G. Santos. SSD ranking: The fastest solid state drives. *http://www.fastestssd.com/ featured/ssd-rankings-the-fastest-solid-state-drives/#pcie*, Oct 2012.

[86] T. K. Sellis. Personalization in web search and data management. In *Model and Data Engineering*, pages 1–1. Springer, 2011.

[87] C. C. Shilakes and J. Tylman. Enterprise information portals. Technical report, Merrill Lynch, Inc., New York, NY, Nov. 1998.

[88] D. Suciu and V. Tannen. A query language for NC. *J. Comput. Syst. Sci.*, 55(2):299–321, 1997.

[89] Y. Tao, W. Lin, and X. Xiao. Minimal MapReduce algorithms. *SIGMOD*, pages 529–540, 2013.

[90] V. V. Vazirani. *Approximation Algorithms*. Springer, 2003.

[91] J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *SIGMOD*, pages 193–204, 1999.

[92] Wikipedia. Big data. *http://en.wikipedia.org/wiki/Big_data#cite_note-23*.

[93] Wikipedia. F-measure. *http://en.wikipedia.org/wiki/Precision_and_recall*.

[94] Wikipedia. Wiki. *http://en.wikipedia.org/wiki/Wiki*.

[95] Wikipedia. Yago. *http://en.wikipedia.org/wiki/YAGO_(database)*.

[96] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *PVLDB*, 4(5):279–289, 2011.

[97] L. Zou, L. Chen, and M. T. Özsu. Distance-join: Pattern match query in a large graph database. In *PVLDB*, pages 886–897, 2009.