Exploiting Hierarchies for Efficient Detection of Completeness in Stream Data

Simon Razniewski[1], Shazia Sadiq[2], and Xiaofang Zhou[2]

[1] Free University of Bozen-Bolzano, Bozen-Bolzano, Italy
razniewski@inf.unibz.it
[2] School of ITEE, The University of Queensland, Brisbane, Australia
{shazia, zxf}@itee.uq.edu.au

Abstract. In big data settings, the data can often be externally sourced with little or no knowledge of its quality. In such settings, users need to be empowered with the capacity to understand the quality of data sets and implications for use, in order to mitigate the risk of making investments in datasets that will not deliver. In this paper we present an approach for detecting the completeness of high volume stream data generated by a large number of data providers. By exploiting the inherent hierarchies within database attributes, we are able to devise an efficient solution for computing query specific completeness, thereby improving user understanding of implications of using query results based on incomplete data.

1 Introduction

Recent years have seen an unprecedented investment in big data initiatives. As companies intensify their efforts to get value from big data, the growth in the amount of data being managed continues at an exponential rate, leaving organizations with a massive footprint of unexplored, unfamiliar datasets. On February 8th, 2015, a group of global thought leaders from the database research community outlined the grand challenges in getting value from big data [2]. The key message was the need to develop the capacity to understand *how the quality of data affects the quality of the insight we derive from it.* Organizational big data investment strategies regarding what data to collect, clean, integrate, and analyze are typically driven by some notion of perceived value. However, even in traditional IS projects that follow carefully crafted analysis and design lifecycles, the misalignment between perceived and actual value has been a notoriously difficult problem leading to dismal return on investment [14]. In big data projects, where the data can often be externally sourced with little or no knowledge of its schemata, quality, or expected utility, the risk of making investments that will not deliver is greatly heightened.

It is widely known that the value of the dataset is inescapably tied to the underlying quality of the data. While data quality has been studied against a large number of so-called data quality dimensions [11], in this paper we focus on the completeness dimension. Completeness can be interpreted in multiple ways including (1) missing values, (2) missing records, and (3) sufficiency of the data for the task at hand. Unlike missing values which are easy to detect, missing records are not visible and can only be detected by using additional reference sources or meta data outside of a database e.g. from provenance information [10], master data or from business processes [18]. At the same time, when dealing with large and uncertain data sets, a user might experience long query processing times, only to realize that the provided result is based on incomplete records. Thus a key challenge in this regard is efficient assessment of record completeness in order to empower users with improved understanding of the implications of using query results based on incomplete data.

2 Motivating Scenario

Consider a fictional regional transportation agency, which we refer to by its acronym RTA in the following[3]. Use of RTA vehicles is by a smartcard, which passengers need to scan when boarding and alighting vehicles. RTA vehicles are fitted with readers that transmit scans in near real time to a central server. RTA uses the smartcard scan data in various decisions, such as:

1. Dynamic service scaling: If the demand for specific services is exceptionally high, the RTA tries to dispatch additional vehicles.
2. Connecting services: Especially at night time, when between service waiting times are long, managing the impact of delayed connections is an efficiency and safety issue.
3. Bus bunching detection and handling: The natural tendency of vehicles to cluster, called bus bunching, is a classical problem of public transport [1], which the RTA aims to predict or detect as early as possible.

For all three scenarios, if any part of the data is not complete (due to faults in data transmission, or technical failures that lead to card scans not being recorded at all), it may lead to incorrect query results and misinformation on the real status of the transport network. For example, the service control center might wrongly believe that a service is not full, and guaranteeing a delayed connection is not necessary. To avoid these misconceptions, knowledge of the completeness of data within a query result is necessary. The above example provides a high volume, high velocity big data setting. In this paper, we address the challenge of efficient data completeness assessment in such settings. Our approach is based on the notion of inherent hierarchies within data (such as RTA smartcard scan data) which we exploit to efficiently compute data completeness for specific queries. We will use RTA smartcard data as a running example throughout the paper to illustrate our approach.

Consider a single table *cardscan* in the RTA database, which contains the following attributes:

[3] There are actually 14 corporations in the transit sector with that acronym listed on Wikipedia.

VehicleType, Line, VehicleID, DateTime, Stop, SmartcardID.

Furthermore, suppose that VehicleID functionally determines Line, and Line functionally determines VehicleType, i.e., every vehicle can belong to only one line, and every line can run vehicles of only one type (bus/train/ferry/..). We consider a time window of 10 days, with days labelled from 1 to 10, accordingly. Some sample data is shown below.

VehicleType	Line	VehicleID	DateTime	Stop	SmartcardID
Bus	1	1B	Day 4 - 1:37 PM	Dutton Park	48789397
Bus	1	1A	Day 7 - 9:11 AM	Fairfield	39150370
Bus	2	2A	Day 2 - 6:50 PM	Yeronga	61096215
Bus	1	1B	Day 9 - 2:07 PM	South Bank	93367832
...	...	...	...	...	...

For illustration purposes, let us assume that the RTA has only 4 vehicles, all buses, with IDs 1A, 1B, 2A, and 2B, organized in two lines, 1 (vehicles 1A and 1B) and 2 (vehicles 2A and 2B). We also assume that RTA has knowledge on completeness of the data such as "the data for all vehicles is complete on all days, except for vehicle 2A on Day 3". The assumption of availability of completeness information is reasonable given the increasing functionality of complex onboard electronics [3]. Suppose now that the RTA wants to know how many people used each vehicle on Day 3. They could issue the following SQL query $Q_{VehiclesDay3}$:

```
SELECT VehicleID, COUNT(*)
FROM cardScan
WHERE CAST(DateTime as DATE) = 'Day3'
GROUP BY VehicleID
```

A hypothetical result to this query shown below could be colored with completeness information as shown in Table 1 (left side). White stands for correct rows

VehicleID	Count(*)
1A	823
1B	712
2A	152
2B	357

Line	Count(*)
1	4823
2	2712

Table 1. Results for Queries $Q_{VehiclesDay3}$ and Q_{Lines}.

(based on complete data), and gray stands for possibly incorrect rows (based on possibly incomplete data). A third color, light-gray is also possible, as the next query shows. Suppose now the RTA also wants to know how many people were riding each line in the last 14 days. They could issue the query Q_{Lines}:

```
SELECT Line, COUNT(*)
FROM cardScan
GROUP BY Line
```

The result to this query could be presented as shown in Table 1 (right side). Clearly, the count for Line 1 is complete, because vehicles 1A and 1B are complete, thus this row is colored in white. For Line 2, the result is not complete, but we propose to color the row in light gray, because there are two possible specializations of that row that are complete: (1) The count for Line 2 on days other than Day 3, and (2) The count of Line 2 for rides on vehicle 2B.

Suppose now that instead of 4, there are 1000 vehicles each sending data and completeness statements every 5 seconds, and that there are 1000 queries that the operation control would like to pose in real-time. Naively, this scenario would require 1000×1000, that is, 1 million recomputations of query completeness every 5 seconds, or 200k per second. Timely computation of completeness information for query results as demonstrated in the above examples is the key aim of the approach presented in this paper.

3 Related Work

The problem of data completeness assessment over partially complete databases was studied in [15,12,17]. Motro [15] used views to describe complete parts of databases, while Levy [12] used local completeness statements, similar to views. Biswas et al. [4] discussed the tradeoff between completeness and resource consumption in sensor networks, while in [7], the influence of probabilistic keys on the completeness of probabilistic databases was discussed. In the present work we rely on patterns, a less expressive formalism of selections on tables, introduced in [17], which allows completeness statements to be expressed in the same schema as the base data. Subsequently, this work introduced an algebra for manipulating these statements, similar to relational algebra.

The idea of promoting completeness information was first discussed in [16], where conditional finite domain constraints (CFDCs) were used. Conditional functional dependencies (CFDs) are well-known database constraints that originated in data cleaning applications [6]. They are more specific than functional dependencies, as they apply only to records that satisfy certain selection conditions, for instance, only for customers in the UK, the ZIP code determines the street. In [16], it was also shown how CFDCs can be used in promoting completeness statements using disjunctive logic programming. Promotion was also discussed in [17], although the approach there is a database-instance dependent one: Information about join values actually present in the database is used to assess whether incomplete join results already cover all possible parts of a join result. However this will not provide a way to derive the conclusions made by hierarchical promotion as presented in Sec. 2.

Our work is also related to data streams because of the setting with a large number of data providers (i.e. RTA vehicles) emitting data at high frequency. Streams are time-evolving series of data [9]. Completeness of parts of data streams has received attention since early works by Tucker et al. introducing the notion of *punctuations*, which are special symbols in data streams that indicate that parts of the stream have been fully transmitted [19]. Later work by Johnson and Golab [8] investigates the semantics of queries over partially complete streams. The focus of this work is on stream warehouses, where base tables come from individual streams, tables are partitioned by time, and views may be built upon base tables. It then investigates which time partitions of views are already trustworthy, or which ones can be quickly recomputed, even if data in the base tables is still missing.

4 Foundation Concepts

Completeness Descriptions Incompleteness in databases is commonly understood as follows: The available database D is considered to be a subset of an unknown ideal database D^i, which describes the facts that hold in reality. For example in reality it might be the case that there were 823/712/902/357 card scans in vehicles 1A/1B/2A/2B on Day 3, while in the database D, only 823/712/*152*/357 are recorded. Completeness statements are then used to describe in which parts the available database D corresponds to the ideal database D^i. In the example above, we could (1) state that the data for Vehicle 1A is complete on Day 3, while stating (2) that the data for Vehicle 2A is complete would be incorrect, as 750 records are missing. Completeness statements are tuples that may contain wildcards, and where the constants describe selection conditions. Prior work [17] considered completeness statements of arbitrary shape, i.e., constants and wildcards could appear at arbitrary positions in the statement. This is not the case in our scenario.

Example 1. Vehicle 1A publishes statements such as

- *(*, *, 1A, Day 2 - 2pm-4pm, *, *)*,
- *(*, *, 1A, Day 3 - 9am-1pm, *, *)*.
- *(*, *, 1A, Day 8 - 7am-8pm, *, *)*.

Notably, the statements differ only in the *DateTime* field. In the following, we assume that all data and completeness statements come from a known set of objects, called *data-generating entities* (DGEs). Each DGE has a *descriptor pattern* that may consist of constants (“Bus”, “B1”), wildcards (“*”), and underscores (“_”). For all attributes in the DGE descriptor that are not underscores, any completeness statement produced by the DGE must have the same values as the descriptor pattern. For instance, the descriptor pattern for Vehicle 1A is *(*, *, 1A, _, *, *)*.

Example 2. Consider again the scenario from Section 2, where data from all vehicles besides Vehicle 2A on Day 3 was complete. We could describe this with the following completeness statements for the table *cardScan*:

VehicleType	Line	VehicleID	DateTime	Stop	SmartcardID	
Bus	2	2A	Day 4 - 1:37 PM	Dutton Park	48789397	Data
...	...	...	...	...	...	
*	*	1A	*	*	*	Completeness Statements
*	*	1B	*	*	*	
*	*	2A	Days 1-2, 4-14	*	*	
*	*	2B	*	*	*	

Query Completeness Reasoning Completeness statements can also be used to describe complete parts of query results. For $Q_{VehiclesDay3}$, the statements (1A,*), (1B,*) and (2B,*) hold, which we used before to color the corresponding rows in the result in white. A core problem is to find the statements that can be derived for a query result, given statements for database tables. This problem is called *query completeness reasoning.*

Example 3. Consider that the statements *(*, *, 1A, *, *, *)* and *(*, *, 1B, *, *, *)* hold for *cardScan*, i.e., that the table contains all scans for vehicles 1A and 1B. If this is the case, then, based on the information that 1A and 1B are the only vehicles of Line 1, the table contains also all card scans for it, i.e., the former statements logically entails (1,*) for Q_{Lines}, thus the white coloring of that row in Table 1.

Existing work reduces completeness reasoning to query rewritability [15], query independence of updates [12], query containment [17] and logic programming [16], but does not take into account hierarchic data, as discussed next.

Hierarchies Hierarchies within attributes can be visualized using trees, where each level corresponds to an attribute, each node in a level represents a possible attribute value, and edges link possible attribute value combinations. Fig. 1 shows the hierarchy for our use case.

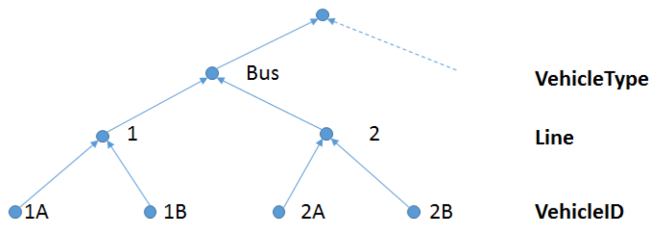

Fig. 1. Hierarchy of our use case.

Technically, hierarchies can be seen as a stronger version of conditional finite domain constraints (CFDCs) [16]. CDFCs are database constraints used to assure that a certain value in a certain field limits the possible values of another field, e.g., that for records with *Line=1* it holds that *VehicleID* is either 1A or 1B. Our setting requires also the converse implication: If *VehicleID* is 1A or 1B, *Line* must be 1. We therefore introduce an extended formalism that we call *conditional two-way dependencies* (CTWDs). Formally, a CTWD has the form $R\{A_1 = v\}[A_2] = \mathcal{W}$, where $A_1 = v$ is a selection condition on the relation R, and $\mathcal{W}$ describes the possible values of attribute A_2 in records that satisfy $A_1 = v$. Their semantics is as follows: For any record r that has $r[A_1] = v$, it must hold that $r[A_2]$ is in $\mathcal{W}$ (like CFDCs), and conversely, if $r[A_2]$ is in $\mathcal{W}$, then $r[A_1]$ must have the value v. A hierarchy tree can then be decomposed into a set of CTWDs, with each parent-children-pair in the tree corresponding to one CTWD. For example, the parent-children-pair (1, (1A, 1B)) corresponds to the CTWD *cardScan*{Line=1}[VehicleID]={1A, 1B}.

5 Query Completeness Computation with Hierarchies

In this section we present our approach to data completeness assessment through efficient computation of query completeness. The hierarchical data model introduced in the previous section has a twofold impact on completeness computation:

Enabling Efficient Promotion. Promotion in [17] requires access to the database content, and to apply it in our use case we would need to normalize the database and to rewrite single-table queries into join queries, even if none of the attributes of the joined table is in the result, which may be counter-intuitive. Promotion in [16] requires to solve Π_2^P-complete logic programming problems, which is generally not scalable. Our first challenge is thus to perform efficient promotion on a single table.

Pruning Irrelevant DGEs. Consider again the 1000 vehicles sending statements every 5 seconds, and the 1000 real-time queries, which would naively require 200k recomputations of query completeness per second. Most likely, most statements do not affect the completeness of most queries, i.e., they are irrelevant for the respective queries. Our second challenge is thus to devise techniques to determine whether completeness statements from a DGE are relevant to a query or not.

5.1 Enabling Efficient Promotion

In this section we discuss how to do efficient promotion on a single table. Our idea is to use the CTWDs to promote completeness information, that a set of siblings has in common, recursively upward.

Definition 1 (Hierarchical Promotion). *Let R be a table and $F = R\{A_1 = v\}[A_2] = \mathcal{W}$ be a CTWD. Then if there exists a set S of completeness statements for R that is unifiable except for the attribute A_2, and if $\{\pi_{A_2}(s) \mid s \in S\} = \mathcal{W}$, we can assert the unifier of S, with attribute A_2 being promoted and attribute A_1 being replaced by v, for R.*

Example 4. Consider the CTWD *goCardRecords*{Line=1}[VehicleID]={1A, 1B} and the completeness statements *(*, *, 1A, Days 3-5, *, *)* and *(*, *, 1B, Days 4-10, *, *)*. Then, since these statements cover all possible values of the CTWD, and are unifiable, we can assert the unified statement *(*, 1, *, Days 4-5, *, *)* for Line 1.

If we repeat this procedure bottom-up, we will eventually derive all statements that hold for a table. As a preparation, we have to arrange completeness statements in a hierarchy, analogous to the hierarchy for the attribute values. Whenever a DGE transmits a new completeness statement s for a node n in this hierarchy, the method $\textit{processStmt}(n, s)$ notifies the parent, which in turn checks the applicability of promotion and recursively notifies further ancestors.The next theorem proves that the procedure *processStmt* is complete for all statements that can be inferred over a single table. We say that a set of completeness statements $\mathcal{S}$ is maximal, if all statements entailed by $\mathcal{S}$ are in $\mathcal{S}$.

Theorem 1. *Consider a maximal set of completeness statements $\mathcal{S}$ stored in a hierarchy of nodes N, and a completeness statement s for a node n in N. Then $\textit{processStmt}(n, s)$ produces all completeness statements that are entailed by $\mathcal{S} \cup \{s\}$.*

Instead of doing promotion repeatedly at query time (for the same or different queries), it is done here only once for every statement. Unlike in [17], where promotion requires repeated database access to retrieve relevant values, here no database access is needed as long as hierarchies can be kept in memory. Hierarchical promotion can also be applied to tables that satisfy multiple orthogonal hierarchies, e.g., if lines are associated with operators, and operators and vehicle types are orthogonal, that is, an operator may operate both buses and ferries, or trains can be operated by different operators. In such situations, the attributes would not be arranged in a tree but in a directed acyclic graphs with a single sink, and in the promotion, an OR-computation would take place: All vehicles would be complete if all operators OR all vehicle types were complete.

5.2 Pruning Irrelevant DGEs

Not all completeness statements are relevant to all queries. For instance, for a query for ferry traffic, completeness of the bus line 1 is clearly irrelevant. Recomputing query completeness only on the arrival of relevant statements may potentially yield huge savings. Recall the example at the beginning of this Section, where 1000 queries were posed and 1000 statements were published every 5 seconds. Let us assume that on average, each statement is relevant to only 5 queries (which makes sense if many queries concern few vehicles, and only few queries concern many vehicles). Then, ideally, it would be necessary to recompute only 5000 queries per 5 seconds, or 1000 queries per second, instead of 200k per second. To find out whether a query needs to be recomputed, we can check whether a source is relevant to a query, using query independent of update (QIU) techniques. Solutions for general QIU problems exist in [5,13]. In our case, what is needed to be checked is whether a query is independent of updates adhering to a DGE descriptor. If this is the case, statements produced by this DGE are relevant to the query.

Example 5. Consider a query Q_{F+B} for cards that were used both on a ferry and a bus. As a conjunctive query, this is written as

$$Q_{F+B}(c)\text{:-}\, cardScan(Ferry, x_1, x_2, x_3, x_4, c),\ cardScan(Bus, y_1, y_2, y_3, y_4, c).$$

Furthermore, consider the following two DGEs:

1. $d_1 =$*(Ferry, F2, 7B, _, * , *)*
2. $d_2 =$*(Train, T3, 9A, _, * , *)*

We can see that d_1 is relevant for this query, as its descriptor pattern syntactically matches the first atom of Q_{F+B}. In contrast, d_2 is not relevant for this query, because it matches neither atom of Q_{F+B}.

To check relevancy, we need to expand DGE descriptors using the CTWDs derived from the hierarchy. Although the descriptor *(*,*,9A,*,*)* is semantically equivalent to d_2 from above, it matches both atoms of Q_{F+B} and hence one might wrongly conclude that its statements are relevant to the query. Computationally, descriptor pattern expansion is linear in the height of the hierarchy, and relevancy checking is linear in the length of the query, as it involves only pairwise comparison of constants and variables. If we cannot find any atom in a query to which the source description can be mapped, we say that the source is not relevant for the query. The following proposition formalizes that statements deemed as irrelevant can be ignored in query completeness computation.

Proposition 1. *Let Q be a conjunctive query and p be a DGE descriptor that is not relevant to Q. Then*

1. *For no database, adding data records that match p changes the result to Q.*
2. *For no set of completeness statements, adding completeness statements that match p changes the completeness statements that hold for Q.*

5.3 Implementation of Query Completeness Reasoning

The core data component is the *completeness statement hierarchy*, which is both a data structure that allows to store completeness statements based on a hierarchy, and an object that implements the hierarchical promotion.For each database table, one such component would be deployed. DGEs would submit statements to this component, while queries would subscribe to the nodes in the hierarchy that are relevant to them. The completeness statement hierarchies would then be the bridge between DGEs and queries. On arrival of new relevant completeness statements, they would notify the subscribed queries. Once completeness of a query is recomputed (which need not happend immediately after notification, depending on load), the query can pull the completeness statements from the nodes of the completeness statement hierarchy that are relevant to it. The full interaction is illustrated in Fig. 2. In our motivating example, data from all DGEs is relevant to the queries $Q_{VehiclesDay3}$ and Q_{Lines}, hence, they would subscribe to all nodes in the completeness statement hierarchy.

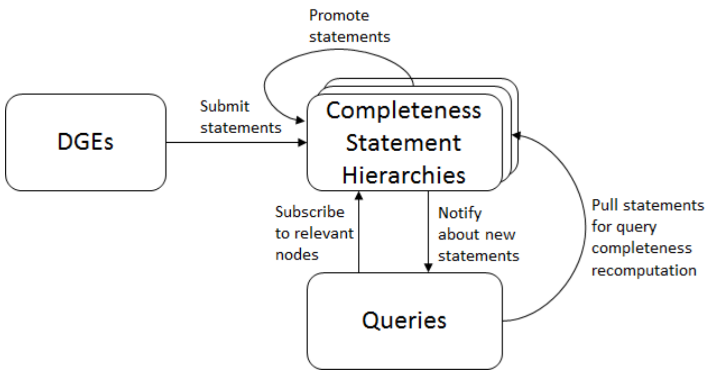
Promote statements
Submit statements
DGEs
Completeness Statement Hierarchies
Subscribe to relevant nodes
Notify about new statements
Pull statements for query completeness recomputation
Queries

Fig. 2. Interaction between DGEs, completeness statement hierarchies and queries.

6 Experiments

We have experimentally analyzed the scalability of completeness statement computation using our approach. The factor of interest is how fast query completeness can be recomputed based on changes to completeness statements. Three parameters are relevant:

1. The size of the hierarchy. We have chosen a hierarchy with 4 attributes, and vary its size by changing the branching factor (default branching factor 10, resulting in 10,000 DGEs).
2. The queries. We start with elementary queries that select data from a node in the hierarchy, one per node in the hierarchy, and obtain complex queries by joining elementary queries on the *SmartcardID* field.
3. The frequency with which completeness statements are submitted. As default we use 0.2 Hz (every 5 seconds).

From all the completeness statements generated in the experiments, we randomly dropped 1%, and set the arrival time of the others to a random timepoint in the next period (for instance, statements for the interval 10-15 seconds arrive randomly somewhere between second 15 and 20). Our experiments were performed on a desktop machine with an Intel i7 processor with 3.4 GHZ and 16 GB RAM. Our implementation was done as a single-threaded Java program.

In the first experiment, we compare completeness computation with a baseline, where for any query, completeness is recomputed whenever a new completeness statement arrives. The results are shown in Fig. 3 (left). The results indicate that the delay of the baseline method exceeds reasonable amounts already for less than 10,000 DGEs, while computation with hierarchies is below 2 seconds delay even with 40,000 DGEs. Second we increase the number of joins. As one can see in Fig. 3 (right), computation for queries with 3 joins is possible for up to 20,000 DGEs within a second. In the third experiment we compare the delay in completeness statement processing and query completeness recomputation for elementary queries. The results shown in Fig. 4 (left), indicate that

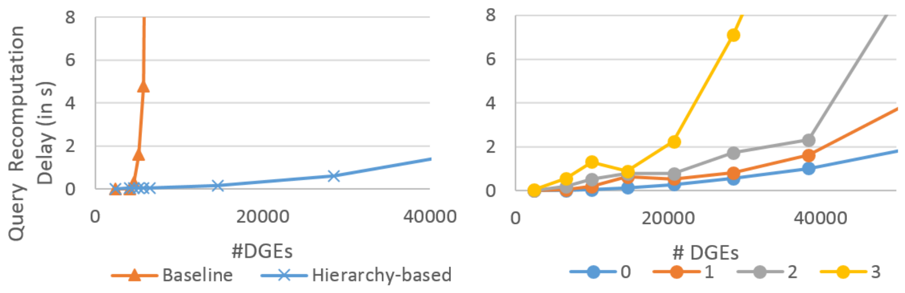

Fig. 3. Recomputation delay in our hierarchical approach compared with a naive baseline (left), and in relation to query length (right).

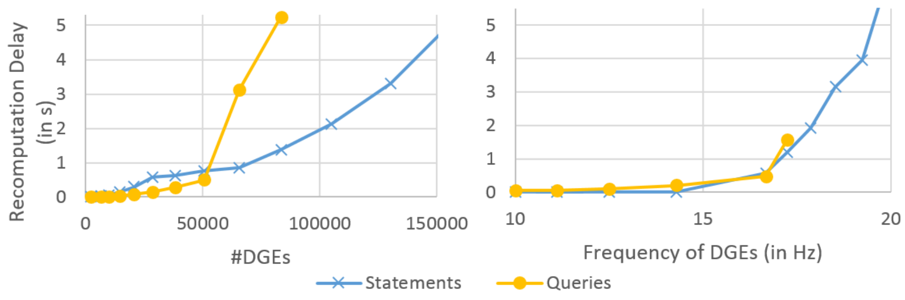

Fig. 4. Recomputation delay for queries and completeness statements, depending on the number of DGEs (left) and on the frequency of statement submission (right).

the delays for both are below one second for up to 50,000 DGEs. Afterwards, the query computation delay increases steeply, while the completeness processing delay increases gradually, as completeness statements have priority in our implementation. In the last experiment, we vary the period of data submission of the DGEs between 100 ms and 50 ms (10-20 Hz), finding that the recomputation delays are reasonably small until a period of 60 ms (17 Hz). At higher frequencies, all compute time is used for hierarchical promotion, and queries are never recomputed (Fig. 4 right).

7 Discussion and Outlook

In the experiments, we have seen that the framework can be applied for elementary queries for up to 50,000 DGEs submitting statements every 5 seconds, or that up to 10,000 DGEs could submit data every 60 milliseconds. We believe that in practical applications, the frequency and the number of DGEs will be much lower[4], thus, the observed bounds validate the applicability of our framework.

[4] `http://goo.gl/x2kZD5`, for instance, cites 1200 buses in Brisbane in 2012.

We envisage that the efficient completeness annotation we have proposed is applicable in several domains besides transport, where a large number of DGEs submit data in real-time, and data loss occasionally happens. For example in *logistics* to manage inventory and delivery cycles; and similarly in *smart factories* where a variety of parameters such as state of processes and energy consumption needs continuous monitoring, and where deviations from desired states may require just-in-time interventions. In all such scenarios timely knowledge about incomplete data is critical to make accurate decisions.

References

1. `https://en.wikipedia.org/wiki/Bus_bunching`.
2. Serge Abiteboul, Luna Dong, Oren Etzioni, Divesh Srivastava, Gerhard Weikum, Julia Stoyanovich, and Fabian M Suchanek. The elephant in the room: getting value from big data. In *WebDB*, pages 1–5. ACM, 2015.
3. Kevin Ashton. That 'internet of things' thing. *RFiD Journal*, 22(7):97–114, 2009.
4. Jit Biswas, Felix Naumann, and Qiang Qiu. Assessing the completeness of sensor data. In *Database Systems for Advanced Applications*, pages 717–732, 2006.
5. José A. Blakeley, Neil Coburn, and Per-Åke Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. In *VLDB*, 1986.
6. Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for data cleaning. In *ICDE*, 2007.
7. Pieta Brown and Sebastian Link. Probabilistic keys for data quality management. In *Advanced Information Systems Engineering*, pages 118–132. Springer, 2015.
8. Lukasz Golab and Theodore Johnson. Consistency in a stream warehouse. In *CIDR*, pages 114–122, 2011.
9. Lukasz Golab and M Tamer Özsu. Issues in data stream management. *ACM Sigmod Record*, 32(2):5–14, 2003.
10. Olaf Hartig and Jun Zhao. Using web data provenance for quality assessment. CEUR Workshop Proceedings, 2009.
11. Vimukthi Jayawardene, Shazia Sadiq, and Marta Indulska. The curse of dimensionality in data quality. In *ACIS*, pages 1–11, 2013.
12. Alon Y. Levy. Obtaining complete answers from incomplete databases. In *VLDB*, pages 402–412, 1996.
13. Alon Y. Levy and Yehoshua Sagiv. Queries independent of updates. In *Proc. VLDB*, pages 171–181, 1993.
14. Andrew McAfee. Mastering the three worlds of information technology. *Harvard Business Review*, 84(11):141, 2006.
15. A. Motro. Integrity = Validity + Completeness. *ACM TODS*, 14(4), 1989.
16. Werner Nutt, Sergey Paramonov, and Ognjen Savkovic. Implementing query completeness reasoning. In *CIKM*, pages 733–742, 2015.
17. Simon Razniewski, Flip Korn, Werner Nutt, and Divesh Srivastava. Identifying the extent of completeness of query answers over partially complete databases. In *SIGMOD*, pages 561–576, 2015.
18. Simon Razniewski, Marco Montali, and Werner Nutt. Verification of query completeness over processes. In *BPM*, pages 155–170. Springer, 2013.
19. Peter Tucker, David Maier, Tim Sheard, Leonidas Fegaras, et al. Exploiting punctuation semantics in continuous data streams. *TKDE*, 15(3):555–568, 2003.