Towards Hypothetical Reasoning Using Distributed Provenance

Daniel Deutch, Yuval Moskovitch, Itay Polack Gadassi and Noam Rinetzky
Tel Aviv University

ABSTRACT

Hypothetical reasoning is the iterative examination of the effect of modifications to the data on the result of some computation or data analysis query. This kind of reasoning is commonly performed by data scientists to gain insights. Previous work has indicated that fine-grained data provenance can be instrumental for the efficient performance of hypothetical reasoning: instead of a costly re-execution of the underlying application, one may assign values to a pre-computed provenance expression. However, current techniques for fine-grained provenance tracking are ill-suited for large-scale data due to the overhead they entail on both execution time and memory consumption.

We outline an approach for hypothetical reasoning for large-scale data. Our key insights are: (i) tracking only relevant parts of the provenance based on an a priori specification of classes of hypothetical scenarios that are of interest and (ii) the distributed tracking of provenance tailored to fit distributed data processing frameworks such as Apache Spark. We also discuss the challenges in both respects and our initial directions for addressing them.

Cust

Num	Name	Zip	Plan
555-777	Bob	10001	Plan1
555-942	Alice	10002	Plan2
555-465	Dave	10003	Plan2

Plans

Plan	Price
Plan1	0.1
Plan2	0.2

Calls

Num	Mo	Dur
555-777	Jan	21
555-777	Jan	8
555-777	Jan	14
555-777	Feb	7
555-777	Feb	17
555-777	Feb	33
555-942	Jan	28
555-942	Jan	20
555-942	Jan	23
555-942	Jan	21
555-942	Feb	33
555-942	Feb	27
555-942	Feb	32
555-942	Feb	16
555-465	Jan	9
555-465	Jan	7
555-465	Jan	8
555-465	Feb	33
555-465	Feb	14
555-465	Feb	12

Figure 1: Example database

1 INTRODUCTION

Data analytics often involves *hypothetical reasoning*; repeatedly modifying the database according to specific scenarios, and observing the effect of such modifications on the result of some computation. A naive way to perform such an analysis is to create a copy of the data, modify it, and recompute the results for the inspected scenario. However, this approach can be very costly when the computation involves access to large-scale data. A more efficient method is to use *provisioning* [4, 6]: compute a symbolic *provenance expression* (PE) which encodes the result of the computation under any possible scenario. Then, the user interacts with the PE for efficient exploration of the scenarios. Creating the PE incurs a *one-time* overhead over the evaluation of a specific query. However, if the analyst inspects multiple scenarios, the creation of the PE may pay off in an inspection which is orders-of-magnitude more efficient than a naive recomputation.

Example 1.1 (Running example). Consider a database of a telephony company containing the number, name, zip code, and call plan of every customer, the *price per minute* (*ppm*) of every plan, and a log of the duration and date of every call (see Fig. 1). The following query computes the company revenues (this example is inspired by the one used in [6]):

```
SELECT Calls.Mo, SUM(Calls.Dur * Plans.Price)
FROM Calls, Cust, Plans
WHERE Cust.Plan = Plans.Plan
      AND Cust.Num = Calls.Num
GROUP BY Calls.Mo
```

The query computes the monthly revenues by summing the per-call-revenue, computed by multiplying the duration of every call by the *ppm* of the customer's plan and grouped by month. An analyst may be interested in the effect of possible changes to the call price on the company revenues. For example, the analyst may wish to compute the revenues under (some combination of) the following hypothetical scenarios:

HS1 What if the *ppm* is decreased by 10% in calling plan 1 and set to \$0.3 in calling plan 2?
HS2 What if all the customers with vanity phone numbers were subscribed to plan 1?
HS3 What if the *ppm* for costumers in Boston is set to \$0.3?
HS4 What if a 25% discount is given to calls which took less than ten minutes?

Computing a PE for the query of Example 1.1 and the scenario HS1 is relatively straightforward for small-scale data [4, 6] (see Section 2.1). Scenario HS2 is more challenging, since it involves changes to the parts of data that is in a comparison in the query. This may be handled in a c-table-like construction as mentioned in [6] (see Section 2.2). The two other scenarios involve changes that may not be directly applied to input data, but rather only to some views over it; we discuss the semantics of such scenarios and the questions that arise in Section 2.3. Further, for all scenarios, there is the concern of space and time overhead incurred by provenance tracking. We propose distributed provenance representations to address this, in Section 3. Specifically, we present two different methods for representing fine-grained provenance in a distributed manner: a *combined representation* that stores each polynomial as a single tuple in the output table comprised of a collection of monomials, and a *separated representation* that stores separately each monomial. Our preliminary results, based on an implementation in *Apache Spark* [12, 14, 15], indicate that the combined approach allows for more efficient exploration, provided that the provenance expression is relatively small, while the separated approach scales better as it is more suitable to the

Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org.

distributed model, however it incurs a non-negligible overhead over the combined approach where the latter can be used.

Related Work. Data provenance has been studied for different data transformation languages, from relational algebra to Nested Relational Calculus, with different provenance models and applications and with different means for efficient storage. Particularly relevant are systems that support provenance for distributed systems [3, 5, 9, 11, 13]. We note that the Spark framework already supports a provenance-like logging mechanism that allows for fault tolerance. Unlike those existing works, we focus on cell-based provenance, where polynomials replace individual values, which is necessary fo answering what-if questions. The use of data provenance for hypothetical reasoning has also been studied in e.g., [4, 6, 7], but without targeting big data and addressing the scalability issues that consequently arise.

2 PROVISIONING IN A NUTSHELL

Provisioning, i.e., *hypothetical reasoning* using provenance, is a two step process: First, the analyst *parameterizes* some entries in the input database. Then, she explores the effect of hypothetical modifications by instantiating the parametrized entries with specific values [4, 6]. Technically, parametrization entails instrumenting entries using *symbolic variables*, e.g., by (symbolically) adding or multiplying them with the original numerical values. When the provisioning engine runs an SQL query, it treats the parameterized entries in a symbolic way, and creates a *provenance expression* (PE); an *output table* which may contain symbolic values. The symbolic representation allows the analyst to explore multiple scenarios by *evaluating* the PE using a specific *assignments* of *concrete values* to symbolic variables. Different types of scenarios lead to different challenges as we next explain.

2.1 Basic Provisioning

We start with the case where the parametrization involves a single table and the program does not perform selection or join over parameterized attributes. In this case, the semantics of provisioning is fairly straightforward. First, the parameterized data entries are annotated with variables which, roughly speaking, replace every parameterized data by a *multivariate polynomial.* Second, a query Q is executed according to its standard semantics except that summations and multiplications are executed symbolically to produce polynomials instead of concrete values. As for aggregation, we simply combine the polynomials in the provisioned attributes using a symbolic plus (+) operator. The analyst uses the generated provenance expression to explore a specific scenario by providing an *assignment* ρ which defines a (concrete) valuation for each symbolic variable and computing the value of the polynomial under ρ. It is straightforward to observe that this computation produces the same results as the re-execution of Q on a correspondingly modified database.

Example 2.1. Reconsider our running example query and scenario HS1 from the Introduction. To generate a PE for them (where we wish to support a generalized version of HS1), we first parameterize the input database as shown in the `Plans` table of Figure 3. In this example, the price for plan 1 is multiplied by a variable p_1 and the price for plan 2 is both multiplied by a variable p_2 and added a variable s_2, the latter allowing for its replacement by a different value. Evaluating the running example query using these symbolic values results in the output table and provisioning expressions shown in Figure 2.

Provenance expression

Mo	Revenues
Jan	$43 \cdot (0.1 \cdot p_1) + 116 \cdot (0.2 \cdot p_2 + s_2)$
Feb	$57 \cdot (0.1 \cdot p_1) + 167 \cdot (0.2 \cdot p_2 + s_2)$

Output

Mo	Rev.
Jan	38.67
Feb	55.23

Figure 2: A PE and the results of its evaluation according to scenario HS1 (i.e., $\rho = [p_1 \mapsto 0.9, p_2 \mapsto 0, s_2 \mapsto 0.3]$)

2.2 Conditional Provisioning

Hypothetical scenario HS2 in Example 1.1 is a particular member in a family of scenarios which enable computing the company's monthly revenues under different reassignments of customers to plans. A PE which allows inspecting the scenarios in this family can be generated by the techniques of [4, 6]. In our example, this entails parameterizing the `Plan` attribute in table `Cust` by replacing its content in every entry by a customer-unique symbolic variable, and then representing the join of `Cust` and `Plans` over the parameterized attribute using a c-table [10].

Example 2.2. Reconsider Figure 3, and now note that the `Cust` table is also parameterized, adding the parameters n_{777}, n_{942} and n_{465} for HS2. Each of these may be assigned to Plan1 or Plan2. The resulting PE for January in this example is

$$43 \cdot (0.1 \cdot p_1) \cdot [n_{777} = Plan1] + 43 \cdot (0.2 \cdot p_2 + s_2) \cdot [n_{777} = Plan2] +$$
$$92 \cdot (0.1 \cdot p_1) \cdot [n_{942} = Plan1] + 92 \cdot (0.2 \cdot p_2 + s_2) \cdot [n_{942} = Plan2] +$$
$$24 \cdot (0.1 \cdot p_1) \cdot [n_{465} = Plan1] + 24 \cdot (0.2 \cdot p_2 + s_2) \cdot [n_{465} = Plan2]$$

Expressions in brackets are mapped to 1 or 0 based on their truth value upon assignment of values to the variables. If we are still interested in exploring HS1, we simply assign to the n variables their original values, and proceed to assigning the other variables as in the previous example. But we can also, e.g., assign $Plan2$ to n_{777}, intuitively switching the plan of Bob to be $Plan2$; then the expression $[n_{777} = Plan1]$ is evaluated to 0 and $[n_{777} = Plan2]$ to 1, so that Bob is given the price of Plan2.

2.3 View-Based Provisioning

The parameterization of the database in Examples 2.1 and 2.2 is done over a single table for each hypothetical scenario. However, the symbolic representation needed for capturing scenarios HS3 and HS4 require the pre-generated views produced by joining tables `Cust` and `Plans` and tables `Cust` and `Calls`, respectively. The view is required because `Price` is parameterized based on the customer's `Zip` code in HS3 and the call's duration (`Dur`) in HS4. We can parameterize a view as if it is a single (albeit joined) table, and use it to generate a provenance expression as described in Section 2.1. However, the use of a parameterized view may render some queries as undefined for hypothetical reasoning as the following example shows.

Example 2.3. Retrieving the *ppm* of Plan1, using the parameterized view generated for scenario HS3 does not make sense because the price of the plan depends on the customer's zip code. Conversely, determining the monthly revenues over this view is

`Cust`

Num	Name	Zip	Plan
555-777	Bob	10001	n_{777}
555-942	Alice	10002	n_{942}
555-465	Dave	10003	n_{465}

`Plans`

Plan	Price
Plan1	$0.1 \cdot p_1$
Plan2	$0.2 \cdot p_2 + s_2$

Figure 3: Data with provenance annotation

well defined, and can be computed using, e.g., a rewrite of the query shown in Example 2.1.

The problem of using views for hypothetical reasoning is highly related to the problem of answering queries using views [8]. One notable difference between the two problems is that a query may be well defined under an hypothetical scenario also if it only relies on tables that are *not* used in the views; the problem arises only when the query uses the same pieces of data affected by the parameterization, but does so in a “different" way. Formalizing and studying the properties that make a query answerable under a definition of hypothetical scenarios over views is an intriguing problem for investigation.

3 DISTRIBUTED BASIC PROVISIONING

Provisioning large-scale data is challenging: computing and maintaining provenance expressions may lead to large overheads in terms of both memory and time. We propose a partial remedy to this problem via an adaptation of basic provisioning to the distributed setting, and report on initial promising experimental results obtained in the context of *Apache Spark* [12, 14, 15].

Simplifying assumptions. We focus on basic provisioning and further assume that only numerical attributes may be parameterized and that the parameterization is performed on each *record* separately—parameterization based on information located at multiple tables requires creating an appropriate (unparameterized) view at a preliminary stage.

3.1 Apache Spark

Apache Spark is a popular framework for writing large scale data processing applications. *Spark* provides operations such as *map*, *filter* and *fold* which can be seen as extensions to the standard database operations *project*, *select* and *aggregation*, respectively, with arbitrary UDFs applied. In addition, *Spark* provides a *natural join* operation between multisets indexed by a common (possibly non-unique) key and *foldByKey* operations which, analogously to the *groupByKey* operation in databases, aggregates together the values pertaining to the same key.

Spark programs are executed on a cluster comprised of a single *master* node, which coordinates one or more *worker* nodes. Roughly speaking, a spark program operates as follows: It first partitions the data across the cluster nodes. *map* and *filter* operations are executed by each node, and on each partition in parallel. *fold* operations are executed in two stages: Every worker node aggregates the values in its partitions and sends the partial result to the master node which aggregates them together. *foldByKey* and *join* operations may require a preliminary (expensive) *shuffle* stage where records are exchanged between working nodes according to a strategy determined by the driver. At the end of the shuffle stage, all the records pertaining to any key are located in the same partition. This allows to compute the (by-key) fold and join operations by every working node separately.

3.2 Distributed Provenance Expressions

Representing and interacting with a provenance expression can be computationally expensive when it is generated from large-scale data. Specifically, two challenges may arise: (i) the table may contain many polynomials, thus representing it would consume a lot of space, and (ii) every polynomial may be comprised of a large number of monomials, thus designing efficient ways to represent and evaluate large polynomials is required.

Key	Value
(Jan, p_1)	$43 \cdot 0.1$
(Jan, p_2)	$116 \cdot 0.2$
(Jan, s_2)	116
(Feb, p_1)	$57 \cdot 0.1$
(Feb, p_2)	$167 \cdot 0.2$
(Feb, s_2)	167

Figure 4: Separated provenance representation.

Handling challenge (i). Using a *polynomial-level distributed representation*, which we refer to as the *combined representation.* In this representation, the provenance expression is spread across the cluster so that different nodes manage different polynomials (and each node manages a whole polynomial). Technically, the *combined representation* stores every polynomial as an additional attribute of each relation. The attribute is an array of *monomials*, where every monomial is a pair comprised of a coefficient and an array of symbolic variables. The symbolic execution of an aggregation operation amounts to combining the monomials coming from different polynomials by performing algebraic simplifications to compute the coefficients. For example, the provenance expression shown in Figure 2 is in the combined representation. Evaluating the provenance expression is done by an invocation of a map operation that evaluates every polynomial for the given assignment in the standard way.

Handling challenge (ii). Using *monomial-level distributed representation*, which we refer to as the *separated representation*: We split every polynomial into its constituent monomials and store each monomial in its own record. Technically, every monomial is represented as a key-value pair, where the value is the monomial’s coefficient and the key is a combination of a unique identifier of the polynomial that the monomial belongs together with the product of its variables. For example, Figure 4 depicts a separated representation of the PE shown in Figure 2. Under the separated representation, the (symbolic) execution of an aggregation operation does not combine different symbolic values into a single polynomial. Instead, when evaluating the provenance expression, a map operation evaluates the value of every monomial and a *foldByKey* operation computes the value of every polynomial.

3.3 Experimental Evaluation

We have performed a preliminary evaluation of our approach, and show two basic experiments based on our running example and a synthetically generated database. In the first experiment, we have joined the `Cust` and `Plans` tables, and parameterized the `Price` attribute of every customer record by multiplying it with a unique customer-unique variable. We ran the query using a `Cust` table that contained 2^{18} (262,144) customers, identified by their unique phone number, and uniformly distributed across the 41,668 US zip codes, and eight call plans. This resulted in a provenance expression comprised of 1536 polynomials. We considered different sizes of the input data by populating the `Calls` table with either 128, 256, 512, 1024, or 2048 calls for every customer. The resulting sizes of the `Calls` tables are in the range of 500 MB to 9.1 GB.

Creating the provenance expressions was up to ×2.1 slower than the computation of the non-provisioned query. Evaluating the queries, on the other hand, up to ×385 time faster than a naive recomputation. Figure 5 depicts the provenance evaluation time

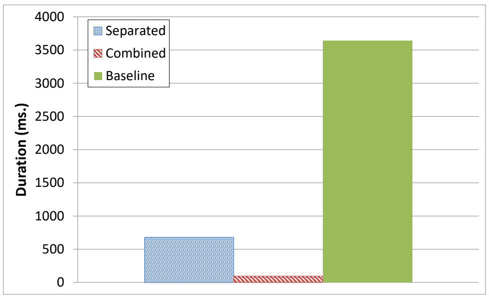

Figure 5: Experiment 1 (Assignment time)

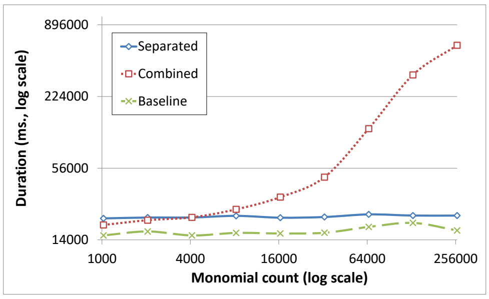

Figure 6: Experiment 2 (PE generation time)

for the *smallest* database[1]. We made sure that changing the size of the data only affected the value of the coefficients, specifically, it did not change the number of monomials in any of the polynomials. As a result, the time it took to evaluate the provenance expression was independent of the size of the database.

In the second experiment, we evaluated the efficacy of the different representations using polynomial of different sizes by generating a provenance expression that comprised of a single polynomial with different number of monomials. We joined the `Cust` and `Plans` tables, and parameterized the `Price` attribute of every customer by multiplying it with a symbolic variable. We used the same database as in the first experiment using a `Calls` table with 32 calls for every customer. We ran the experiment nine times, where in the ith experiment, for $i = 0..8$, we used the same variable for all the customers whose phone numbers ended with the same $10+i$ bits. Thus, the smallest polynomial contained 1024 monomials, and the biggest one had 262144. Figure 6 depicts the time it took to generate the provenance expressions. It shows that the Combined approach has the upper hand for small polynomials, yet it quickly becomes much slower than the Separated approach, up to the point of being prohibitively expensive.

Analysis. The Separated representation has an advantage for big polynomial whereas the Combined approach is better when the polynomials are small. We believe that the reason for this is that the Separated representation allows to store and evaluate a single polynomial concurrently using multiple nodes. Furthermore, this representation is particularly beneficial in the context of *Spark* which handles tables containing a large number of records very efficiently, but struggles when it is asked to process large records due to its in-memory representation of the data and its single-threaded record processing.

4 OPEN PROBLEMS AND FUTURE WORK

We briefly discussed in this paper the problem of provisioning for big data, highlighting semantic and scalability issues, and proposing partial solutions in the context of Apache Spark. Both facets of the problem require extensive investigation, which is the subject of our on-going work. On the semantic side, a notable omission in the current literature is a formal language for specifying hypothetical scenarios (the need for such language is also mentioned in [4]); then, given a formal specification, can we efficiently decide whether an hypothetical is "answerable" (as in the view-based examples we have shown, this is not a given) and if so efficiently generate the PE? In terms of efficiency, we plan to extend our *Spark*-based implementation to allow for conditional provisioning. In addition, we are interested in exploring the benefits of a *hybrid* method for representing distributed provenance which combines the methods Separated and Combined we have presented by maintaining every polynomial as a table of sub-polynomial instead of single monomials. Last, we plan to investigate possibilities of "approximate provisioning", where we lose some granularity of the possible assignments to variables but gain in performance.

ACKNOWLEDGEMENTS

This research has been partially funded by the Israeli Science Foundation, the Blavatnik Interdisciplinary Cyber Research Center (TAU ICRC), the Blavatnik Computer Science Research Fund and Intel. The contribution of Yuval Moskovitch is part of Ph.D. thesis research conducted at Tel Aviv University.

REFERENCES

[1] Amazon Inc. *Amazon Elastic Block Store (Amazon EBS).* https://aws.amazon.com/ebs/, 2017.
[2] Amazon Inc. *Amazon Elastic Compute Cloud (Amazon EC2)—Instance Types.* https://aws.amazon.com/ec2/instance-types/, 2017.
[3] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. Putting lipstick on pig: Enabling database-style workflow provenance. *PVLDB*, 5(4), 2011.
[4] S. Assadi, S. Khanna, Y. Li, and V. Tannen. Algorithms for Provisioning Queries and Analytics. In *(ICDT 2016)*, volume 48, 2016.
[5] C. Chen, H. T. Lehri, L. K. Loh, A. Alur, L. Jia, B. T. Loo, and W. Zhou. Distributed provenance compression. In *SIGMOD*, 2017.
[6] D. Deutch, Z. G. Ives, T. Milo, and V. Tannen. Caravan: Provisioning for what-if analysis. In *CIDR*, 2013.
[7] D. Deutch, Y. Moskovitch, and V. Tannen. Provenance-based analysis of data-centric processes. *VLDB J.*, 24(4), 2015.
[8] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4), 2001.
[9] R. Ikeda, H. Park, and J. Widom. Provenance for generalized map and reduce workflows. In *CIDR*, 2011.
[10] T. Imielinski and W. L. Jr. Incomplete information in relational databases. *J. ACM*, 31(4), 1984.
[11] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. D. Millstein, and T. Condie. Titian: Data provenance support in spark. *PVLDB*, 9(3), 2015.
[12] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia. *Learning Spark: Lightning-Fast Big Data Analytics.* O'Reilly Media, Inc., 1st edition, 2015.
[13] H. Park, R. Ikeda, and J. Widom. RAMP: A system for capturing and tracing provenance in mapreduce workflows. *PVLDB*, 4(12), 2011.
[14] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.
[15] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10*, 2010.

[1] We ran our experiments on Amazon EC2 public cloud [2] using a cluster comprised of nine `m4.xlarge` virtual machines. Eight machines were used as workers and one as the cluster's driver. Each virtual machine has four virtual CPUs, 16GB memory. (The physical CPU used for an `m4.xlarge` virtual machine is a 2.4GHz Intel Xeon E5-2676 v3 processor, where every virtual CPU is a hyperthread of an Intel Xeon core.) We ran *Spark* version 1.62 over Amazon Linux AMI 2016.03, with Linux kernel 4.4.11, Java runtime 1.8.0_101, and Scala version 2.10.5. The UDFs were implemented in Scala, and the data was stored in Amazon Simple Storage Service (S3) [1].