Causal What-If and How-To Analysis Using HYPER

Fangzhu Shen*[1], Kayvon Heravi*[2], Oscar Gomez[1], Sainyam Galhotra[3], Amir Gilad[1], Sudeepa Roy[1], Babak Salimi[2]
[1]Duke University, [2] University of California San Diego, [3] University of Chicago
[1]{fangzhu.shen, oscar.gomez.quintero, amir.gilad, sudeepa.roy}@duke.edu, [2]{kheravi,bsalimi}@ucsd.edu, [3]sainyam@uchicago.edu
* Authors have contributed equally

***Abstract*—What-if and How-to queries are fundamental data analysis questions that provide insights about the effects of a hypothetical update without actually making changes to the database. Traditional systems assume independence across different tuples and non-updated attributes of the database. However, different attributes and tuples are generally dependent in real-world scenarios. We propose to demonstrate HYPER, a novel system to efficiently answer what-if and how-to queries while capturing causal dependencies among different attributes and tuples in the database. To compute the results, HYPER leverages a suite of optimizations along with techniques from causal inference to effectively estimate the answers. HYPER allows users to formulate complex hypothetical queries by using a novel SQL-like syntax and presents the output as interactive visualizations that can be explored and analyzed with ease.**

I. INTRODUCTION

We propose to demonstrate HYPER (**Hypo**th**e**tical **R**easoning)[1]a novel system for *hypothetical 'what-if' and 'how-to' reasoning* that accounts for complex probabilistic dependencies in the data and measures the collateral effects of hypothetical updates. Today, decision-making in many fields like business, healthcare, or real estate, is assisted by hypothetical reasoning over the data [2], [3]. What-if queries [4], [5] allow users to test assumptions by posing queries about hypothetical updates in the database and examining their effect on a query result. In contrast, how-to queries [6] have the opposite goal; users specify a target effect they want to achieve and the system computes the appropriate update that has to be performed in the database to fulfill the goal. We illustrate these queries with an example below.

Example 1: Consider a (simplified) products-review database [7] shown in Figure 1 containing Product and Review table. Suppose an analyst wants to examine "what would be the effect on the average ratings of Asus laptops by increasing their price by 10%?", which is a what-if query. or "How to increase average sentiment in the reviews for cameras by changing their price?" These queries are respective forms of what-if and how-to hypothetical reasoning that can assist analysts and decision-makers in gaining insights about their products and marketing strategies.

While several previous works in the database community have studied hypothetical reasoning, a substantial part of these [6], [8], [9] have focused on provenance updates and view manipulation as the main component for answering such queries. However, in many real-world situations, there are complex probabilistic causal dependencies between attributes of tuples and thus, updating an attribute of a tuple has collateral effects on other attributes of the same tuple, as well as attributes of other tuples. For instance, increasing the price of a product p_1 not only might affect the ratings of p_1 but also another competitor's product p_2 of the same category by comparison. Such dependencies cannot be expressed and captured by provenance. In Example 1, the provenance of the average rating of Asus laptops will not change if the price of the laptops is augmented. Similarly, for the how-to query, the provenance of the average sentiments for the reviews of cameras will not be affected by the change in price. Thus, previous work in databases fails to account for the collateral effect that increasing the price of a laptop may have on the user's ratings. Such dependencies between the attributes and tuples in the database can be captured by a causal model depicted as an intuitive graph [1] (such a graph is shown in the bottom-left corner of Figure 3).

PID	Category	Price	Brand
1	Laptop	999	Vaio
2	Laptop	529	Asus
3	Laptop	599	HP
4	DSLR	549	Canon
5	eBooks	15.99	T Press

(a) Product

PID	RID	Senti	Rating
1	1	-0.95	2
2	2	0.7	4
2	3	-0.2	1
3	3	0.23	3
4	5	0.7	4

(b) Review

Fig. 1: A product-review database

Extensions of HYPER for the demonstration We extend the HYPER methodology to capture database constraints and present an efficient implementation of the system with an intuitively designed interactive graphical user interface. HYPER supports a rich class of what-if and how-to queries that involve joins and aggregations taking into account causal dependencies among attributes of the same or different tuples. Users can easily formulate such queries by simply specifying the aggregate SQL query that calculates the output that the user wants to evaluate. For example, the average rating of Asus laptops in Example 1. HYPER then provides an SQL interfaces to specify hypothetical updates to the database. The user can choose any of these interfaces to visualize the effect of different updates to the database. HYPER provides the results of the hypothetical update queries along with an explanation of the variations across different subgroups in the database. HYPER further allows users to "zoom-in" to compare the effect of their chosen hypothetical update with other update

[1]The research paper that develops the approach used by HYPER appeared in SIGMOD 22' [1]

options. This comparison helps the user effectively explore the effect of different updates and choose the suitable option for their application. The inference mechanism in the backend is highly optimized by leveraging techniques from probabilistic databases [10], and recent advancements in causal inference over multi-table relational data [11]. We will demonstrate HYPER and enable users to explore HYPER themselves using real-world data from the Amazon Products database and other well-known datasets, showing its usefulness and effectiveness.

II. CAUSAL MODEL, QUERIES, SOLUTIONS

We next give an overview of the hypothetical updates, causal model, and the HYPER algorithms to compute the output of the hypothetical update query.

Hypothetical updates. We consider a standard multi-relational database D. $A_i[t] \in \texttt{Dom}(A_i)$ denotes the value of the attribute A_i of the tuple t. Some attributes can not change values in a hypothetical update and are called *immutable* (denoted by $A_1, \cdots, A_m$), other attributes are *mutable* and can change values directly or indirectly (denoted by $B_1, \cdots, B_\ell$). For a tuple t in a relation R in D, a possible world of t is the set $PWD(t) = \{A_1[t], \cdots, A_m[t], v_1, \cdots, v_\ell \ : \ v_i \in \texttt{Dom}(B_i), i = 1 \text{ to } \ell\}$, where Dom denotes the domain, i.e, the set of all possible values the mutable attributes can assume. The set of possible worlds of relation R and database D are $PWD(R) = \times_{t \in R} PWD(t)$ and $PWD(D) = \times_{R \in D} PWD(R)$ respectively.

A hypothetical update $U = u_{R,B,f,S}$ on a database D is then a 4-tuple that includes a relation R in D containing the mutable update attribute $B \in \texttt{Attr}(R)$, a subset of tuples $S \subseteq R$ where the update will be applied, and a function $f : \texttt{Dom}(B) \rightarrow \texttt{Dom}(B)$ specifying the update for attribute $B[t]$ for tuples $t \in S$ to $f(B[t])$. A hypothetical update $u_{R,B,f,S}$ forces all tuples in set S in relation R to take the value $f(B[t])$ instead of $B[t]$. The state of the database after a hypothetical update is modeled by the post-update distribution, i.e., a probability distribution over possible worlds, i.e., $\Pr_{D,U} : PWD(D) \rightarrow [0, 1]$ such that $\sum_{I \in PWD(D)} \Pr_{D,U}(I) = 1$.

Causal model. The post-update distribution stems from a probabilistic relational causal model (PRCM) [11] expressed through a ground causal DAG G over the set of tuple attributes $A[t]$. Each node $A[t]$ represents a variable that is functionally determined by: (a) its parents $Pa(A[t])$ in G, and (b) some set of noise factors that do not appear in G. The post-update distribution is defined using a PRCM. Given a relation R in D, an update attribute $B \in \texttt{Attr}(R)$, a hypothetical update $U = u_{R,B,f,S}$ can be interpreted as an *intervention* that modifies the underlying PRCM and replaces the structural equation associated with the variables $B[t]$ for all $t \in S$ with the constant $f(B[t])$. Updating $B[t]$ propagates through all attributes of tuples in different relations according to the underlying PRCM. The uncertainty over unobserved noise variables induces uncertainty over post-update states of all tuples t' captured by the post-update distribution on the possible worlds: $\Pr_{D,U}(\tau)$ for $\tau \in PWD(t')$, and the post-update distribution of the entire database $\Pr_{D,U}(I), \forall I \in PWD(D)$.

TABLE I: **Operators of what-if queries**

Operator	Meaning
USE $Table(...)$OUTPUT $Agg(Y)$	SQL aggregate query that defines the output of interest
WHEN $g(A_l)$	Condition on the attribute A_l (expressed as the function g) that the tuples that will be updated must satisfy
UPDATE $B = f(B)$	Hypothetical update in the database. The operator changes the values in the attribute B to $f(B)$
POST(A_i)	The value of the attribute A_i after the update
PRE(A_j)	The value of the attribute A_j before the update

TABLE II: **Operators of how-to queries**

Operator	Meaning
USE $Table$, WHEN $g(A_l)$, FOR	See Table I
HOWTOUPDATE UPDATE B_1, B_2	Sets B_1 and B_2 as the attributes on which the updates can be performed
LIMIT $l_1 \leq$ POST$(B_1) \leq h_1$ AND $L1$(PRE(B_2), POST$(B_2)) \leq h_2$	In the update, B_1 can get values between l_1 and h_1 and B_2 can get values at most at h_1 distance from its original value (for each tuple)
TOMAXIMIZE Agg(POST(Y))	Defines the objective of the query: to maximize (or minimize) the post update value of an aggregate function on Y

What-if queries. In HYPER, users can specify several pre-conditions, update mechanisms, and constraints to formulate what-if and how-to queries with the help of a GUI (no new query language has to be learnt). The syntax of what-if queries is detailed in table I. HYPER takes as input an update attribute and the manner in which the user wishes to update it (part 3 in Figure 2). It then converts it to an UPDATE operator that performs a causal intervention, where the PRE value of the attribute is the default. In addition, the user can define the subset of the aggregate view such that the update will be computed only considering these tuples. This is done using the next line of the template (part 3 in Figure 2) that is converted to the FOR operator specified in [1]. The optional 'WHEN' clause is mapped to the WHEN operator [1] and specifies the set of tuples that will be updated; any valid SQL predicate can be used, e.g., $A =< const >$, $A \in$ (SELECT $\cdots$ AS $A \cdots$) etc. If the WHEN operator is not specified the update will be performed on the entire aggregate view.

Given a what-if query Q and a database D, the result of $Q(D)$ is the expected value of $\texttt{val}_{\texttt{whatif}}(Q, D, I)$ over all possible worlds $I \in PWD(D)$, using the post-update probability distribution $\Pr_{D,U}$: $\texttt{val}_{\texttt{whatif}}(Q, D) = \mathbb{E}_{I \in PWD(D)}[\texttt{val}_{\texttt{whatif}}(Q, D, I)]$, where $\texttt{val}_{\texttt{whatif}}(Q, D, I) = aggr(\{Y_I[t] \ : \ \mu_{\text{FOR}}(t) = true\})$, where all t are the tuples in the query aggregate view and the aggregate $aggr$ over $Y_I[t]$ values and $\mu_{\text{FOR}}(t)$ denotes the value of the predicate in the FOR clause over the tuple t.

How-to queries. In addition to the aggregate query (same as the what-if case), a how-to query requires specification of a set of attributes that can be updated to achieve the desired result. The syntax of how-to queries is detailed in table II. The user specifies these as the HOWTOUPDATE operator, which consists of the set of mutable attributes that can be updated, and uses only PRE values. To ensure that the updates on these attributes are valid, we assume that, for any pair of the attributes mentioned in this operator B_1, B_2, there are no paths in the ground causal graph of the PRCM between $B_1[t]$ and $B_2[t']$ for any $t, t' \in D$. User can state the constraints for the updates using the optional LIMIT, i.e., this operator defines the conditions that restrict the post-update values of the updates attributes specified in the HOWTOUPDATE operator for tuples in the query aggregate view that satisfy the WHEN operator. In particular, if an attribute B is numeric, its updates can be bounded by numeric limits, e.g., $l \leq \text{POST}(B) \leq h$, $\text{POST}(B) \leq \text{PRE}(B) + < const >$, $\text{POST}(B) \leq \text{PRE}(B) \times < const >$, etc., and if B is categorical (or numeric), the user can specify the permissible values as a set, e.g., $\text{POST}(B)$ IN $\{v_1, v_2, v_3\}$. Furthermore, this operator allows users to specify the maximal or minimal $L1(\text{POST}(B), \text{PRE}(B))$ distance between the original attribute values and the updated ones.

The result of a how-to query is then defined as $argmax_{Q_{WI} \in \mathcal{Q}_{whatif}(Q_{HT})} \texttt{val}_{\texttt{whatif}}(Q_{WI}, D)$, where $\texttt{val}_{\texttt{whatif}}(Q_{WI}, D)$ denotes the result of the what-if query Q_{WI}, and $\mathcal{Q}_{whatif}(Q_{HT}$ denotes the set of all what-if queries that have the same FOR and WHEN clauses, and whose UPDATE clause specifies an update that satisfies the update constraints in Q_{HT}.

Optimizations for computing query results. To evaluate the query output, a naive approach requires the enumeration of all possible worlds along with their probability estimation. However, this is not practical due to the exponential dependence of the instances on the number of attributes. For what-if queries, we leverage the causal dependence between attributes to simplify the estimation procedure. We decompose the database into independent sets of tuples. This allows us to compute the values of queries more efficiently. We further draw a connection between the query results and estimating the post-update distribution in causality. This, in turn, allows us to use existing approaches from causal inference to evaluate the query results. For a how-to queries, we assume a linear model for the objective function and employ linear program solvers to find the solution. We refer the reader to [1] for the complete details of our solution.

III. GUI OVERVIEW OF HYPER

The back-end of HYPER is implemented in Python with the user interface in Flask. Figure 2 shows the graphical user interface consisting of the query interface and the output view.

Input interface. The input interface (left half of the screen) allows the user to choose the initial dataset, inspect the records and specify the hypothetical updates. This section allows the user to write a SQL aggregate query and specify various updates in a specified template. These different forms of inputs are processed by HYPER to identify different operators of what-if and how-to queries, which are used to evaluate the query output.

Results. The output of a what-if or a how-to query is divided into two different tabs: (i) Overall, and (ii) Vary updates. The first tab ("Overall") shows a bar chart consisting of the query output before (shown in green) and after (in orange) the update. Additionally, it shows a plot demonstrating the effect of the update on different subgroups identified by fixing the value of an attribute. The user can choose this attribute from a dropdown list of suggestions that is automatically generated according to the what-if query. For how-to queries, this tab additionally contains top-5 update recommendations that are returned by HYPER. The user can choose any of the recommendations to change the hypothetical update and the previously described plots update accordingly. The second tab ("Vary updates") shows the effect of varying the updated value of the attribute mentioned in the UPDATE operator on query output. This comparison helps the user understand the sensitivity of query output with respect to the update and thereby choose an appropriate value for further exploration.

IV. DEMONSTRATION SCENARIO

We demonstrate HYPER on the Amazon product database, which consists of two tables: (i) Product description, and (ii) Reviews (Figure 1 shows a sample). We will also provide three additional popular datasets: (i) Adult income dataset, (ii) German Credit dataset, and (iii) Academic review database. The user can interact with HYPER by writing different aggregate queries and varying updated attributes to try different types of hypothetical queries. Figure 2 shows a screenshot of the graphical user interface, where each step is annotated with a circle. In the first part of the demonstration, there will be a guided 'tour' of HYPER and in the second step, we will allow users to formulate their own hypothetical queries. We now discuss the different steps through which we guide the users for what-if and how-to analyses.

Step 0 (Guided tour of HYPER): We will begin by explaining the query interface using the what-if and how-to queries detailed in Example 1, and show their representation, while also allowing users to inspect the Amazon database itself. Users will be able to view the general results, and we will further show them how to zoom-in on different subsets of the data, using the drop-down menu.

Step 1 (Dataset selection and inspection): After guiding the users, they will choose a database from the drop-down menu for running their choice of hypothetical update query. Figure 2 shows the example with Amazon product database. Users can view the products and reviews tables in the output panel of the interface (right-half of the screen). The users can click on the 'specify constraints' button to modify the causal constraints of the database. The causal graph will be loaded with the database and users could choose to visualize it. If a DAG will not be

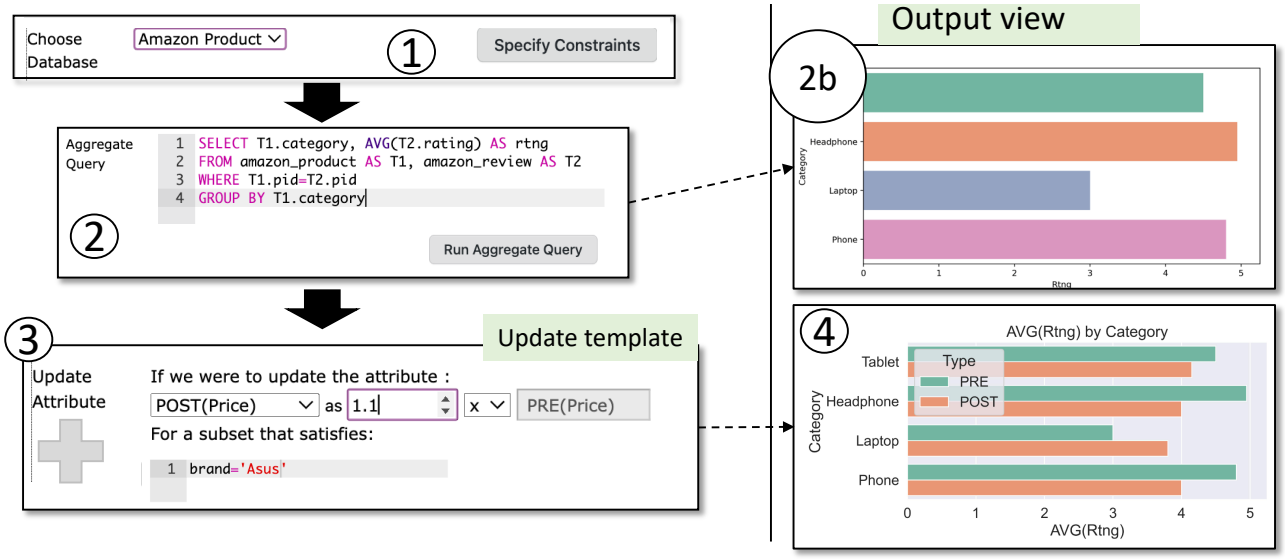

Fig. 2: What-if query that computes average rating for Asus laptops on increasing the price by 10%

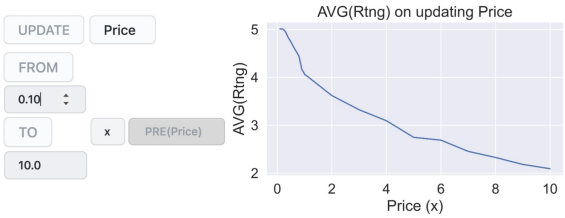

Fig. 3: Vary update tab in the query output view

loaded, the system could with a 'default' DAG, having an edge from each attribute to both the treatment and the outcome.

Step 2 (Aggregate query): Users will be able to specify the aggregate query that returns the output, which the user is interested to evaluate under hypothetical updates. Users could write, e.g., a SQL query to join the two tables in the database using product id as the key and return the average rating for different categories of products. HYPER shows a bar plot with the aggregated output for different values of the attribute mentioned in the group-by clause (labeled 2b in Figure 2).

Step 3 (Choose attributes to be updated): Users can specify the update in the form of a template. The update clause can contain complex conditions (similar to a WHEN clause in SQL) e.g. update the price of records with 'Brand=Asus' and 'Category $\neq$ Laptop'. Users can click on the plus button to specify multiple different updates that they want to compare.

Step 4 (Query Output view): On clicking "Run", the query output view will populate two different tabs with different types of plots, according to the user query. In Figure 2, the left plot shows the average rating for Asus products before and after the change in price. The lower-right plot shows the distribution of average ratings for different categories of Asus products. One conclusion from the results is that an increase in the price improves the average rating of tablets and laptops but reduces the rating of phones.

Step 5 (Varying updates view): The second tab in the output view shows a line plot that demonstrates the effect of the different updates of the chosen attribute on the output attribute. For the query shown in Figure 2, the plot will show varying price updates on the average rating of Asus products.

How-to query. Users will also be able to perform how-to queries. The output of how-to queries is slightly different, showing the top-5 updates for the query that will maximize/minimize the objective, and thus allowing users to choose any of the updates and visualize its effect.

In the subsequent iterations, users can modify the relevant view query, update and output attributes and repeat the above-mentioned steps to get additional insights about the dataset.

Acknowledgements: This work was supported by the NSF awards IIS-1703431, IIS-1552538, IIS-2008107, and an NSF Computing Innovation Fellowship 2030859.

REFERENCES

[1] S. Galhotra, A. Gilad, S. Roy, and B. Salimi, "Hyper: Hypothetical reasoning with what-if and how-to queries using a probabilistic causal approach," in *SIGMOD*, 2022, pp. 1598–1611.
[2] M. Golfarelli and S. Rizzi, "What-if simulation modeling in business intelligence," *Int. J. Data Warehous. Min.*, vol. 5, no. 4, pp. 24–43, 2009.
[3] B. Qureshi, "Towards a digital ecosystem for predictive healthcare analytics," in *MEDES*, 2014, pp. 34–41.
[4] L. V. S. Lakshmanan, A. Russakovsky, and V. Sashikanth, "What-if OLAP queries with changing dimensions," in *ICDE*, 2008, pp. 1334–1336.
[5] H. Herodotou and S. Babu, "Profiling, what-if analysis, and cost-based optimization of mapreduce programs," *PVLDB*, 2011.
[6] A. Meliou and D. Suciu, "Tiresias: the database oracle for how-to queries," in *SIGMOD*, 2012, pp. 337–348.
[7] R. He and J. McAuley, "Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering," in *WWW*, 2016.
[8] D. Deutch, Z. G. Ives, T. Milo, and V. Tannen, "Caravan: Provisioning for what-if analysis," in *CIDR*, 2013.
[9] B. S. Arab and B. Glavic, "Answering historical what-if queries with provenance, reenactment, and symbolic execution," in *USENIX*, 2017.
[10] N. N. Dalvi and D. Suciu, "Efficient query evaluation on probabilistic databases," *VLDB J.*, vol. 16, no. 4, pp. 523–544, 2007.
[11] B. Salimi, H. Parikh, M. Kayali, L. Getoor, S. Roy, and D. Suciu, "Causal relational learning," in *SIGMOD*, 2020, pp. 241–256.