Gauss: Program Synthesis by Reasoning Over Graphs

ROHAN BAVISHI, University of California, Berkeley, USA

CAROLINE LEMIEUX, University of California, Berkeley, USA

KOUSHIK SEN, University of California, Berkeley, USA

ION STOICA, University of California, Berkeley, USA

While input-output examples are a natural form of specification for program synthesis engines, they can be imprecise for domains such as table transformations. In this paper, we investigate how extracting readily-available information about the user intent *behind* these input-output examples helps speed up synthesis and reduce overfitting. We present Gauss, a synthesis algorithm for table transformations that accepts partial input-output examples, along with *user intent graphs*. Gauss includes a novel conflict-resolution reasoning algorithm over graphs that enables it to learn from mistakes made during the search and use that knowledge to explore the space of programs even faster. It also ensures the final program is consistent with the user intent specification, reducing overfitting. We implement Gauss for the domain of table transformations (supporting Pandas and R), and compare it to three state-of-the-art synthesizers accepting only input-output examples. We find that it is able to reduce the search space by 56×, 73× and 664× on average, resulting in 7×, 26× and 7× speedups in synthesis times on average, respectively.

Additional Key Words and Phrases: program synthesis, graphs, reasoning, table transformations, data-science

1 INTRODUCTION

In a programming-by-example system, users provide input-output examples as specification, and the program synthesis engine finds a program that is able to transform the given input into the given output. The appeal of input-output examples as specification is clear: users likely already have some inputs for the program they want to synthesize, and can then simply craft the desired output. So, users need not learn a new specification language in order to use the synthesis engine.

After the success of Flashfill [Gulwani 2011], many works have explored the use of input-output specifications for increasingly complex input domains [Balog et al. 2016; Le and Gulwani 2014; Polozov and Gulwani 2015; Wang et al. 2019], including table transformations [Bavishi et al. 2019a; Feng et al. 2018, 2017]. Unfortunately, in the case of table transformations, as the input-output tables get more complex, so does the user effort in creating the input-output examples. Wang et al. [2019] allow users to provide a *partial* output table instead to reduce this effort.

However, in addition to the extra user effort that goes into creating these input-output tables, there is also something lost: a clear demonstration of user *intent*. Figure 1 illustrates what we mean by loss of user intent. The source of the number 102.5 in the first cell of the output is not immediately clear from the input-output example alone. However, the user who created this example *knows exactly* how they derived this number. It is simply the mean of all the numerical cells in rows labeled "Pants" (50, 70, 100, 190). The user could have easily demonstrated their intent given a suitable user-interface (UI). A clearer representation of the intent could not only help speed up synthesis, but prevent overfitted programs that are inconsistent with the intent.

In this paper, we present Gauss, a synthesis algorithm for table transformations that accepts *partial* input-output tables along with *user intent annotations*. Our key insight is that user intent is fundamentally a set of *relations* between elements of the input and elements of the output. Thus, we represent these user intent annotations as *graphs*. These graphs can be transparently constructed by, for example, the UI the user is using to produce the output. As a proof-of-concept, we created a Jupyter notebook extension which creates such a graph under the hood.

We focus on table transformations as they (compared to e.g. data structure transformations) are used widely across areas with non-traditional programmers, such as machine learning, data science,

Fig. 1. Input-output examples alone discards user intent information that was present while creating the output. In this example, it is not immediately clear that 102.5 is the mean of 50, 70, 100, and 190.

and business analytics. This is exemplified by the tremendous surge[1] in popularity of languages and APIs such as R and Pandas. Synthesis for table transformations has also been studied in several prior works on input-output example and natural language-based program synthesis [Bavishi et al. 2019a; Feng et al. 2018, 2017; Wang et al. 2019; Yu et al. 2018].

Gauss is an enumerative synthesis algorithm. It employs a novel reasoning procedure over graphs that helps it quickly prune away large classes of infeasible programs. At a high-level, Gauss adopts a divide-and-conquer approach: it breaks down the user intent graph into smaller subgraph specifications, and uses these as a measure of progress while enumerating the search space. Whenever it finds that a program or a class of similar programs do not satisfy these specifications, it detects a core subgraph that explains the failure of these programs in satisfying the specification. The algorithm then *learns* from this failure by performing inductive reasoning against a knowledge base of example programs and their graph abstractions, to rule out other programs in the search space. This algorithm presents important extension to the use of conflict-driven synthesis strategies in the realm of first-order logic [Feng et al. 2018]. In addition, Gauss reduces the likelihood of returning over-fitted solutions by ensuring that they are consistent with the user intent graph.

Gauss has multiple synthesis backends for different table transformation API. It has a backend for the Python `pandas` API supporting 33 functions as well as one for the R `tidyr/dplyr` API supporting the 10 `tidyr/dplyr` functions from the DSLs of Feng et al. [2017] and Wang et al. [2019]. These include popular reshaping and summarization operations on input tables. In addition, we built a prototype UI for table manipulations that transparently builds user intent graphs. So, users need only create input-output examples with our UI in order to reap the rewards of our algorithm.

We evaluate Gauss on two main fronts. First: do richer graph specifications enable significant pruning compared to synthesis techniques that only leverage input-output examples? We find that Gauss explores 56× and 73× and 664x fewer candidates than the state-of-the-art systems Morpheus [Feng et al. 2017], Viser [Wang et al. 2019] and Neo[Feng et al. 2018] on their respective benchmarks because of the additional graph specification. Second: can graph specifications allow the user to provide partial information, such as a partial output, as opposed to needing to construct a full input-output examples? We evaluate Gauss on benchmarks used in Viser and find that output size can be reduced by 33× on average while getting the correct answer just as quickly.

To summarize, our contributions are as follows:

- A synthesis algorithm for table transformation programs, taking partial input-output examples with graph intent annotations as specification.
- A conflict-driven pruning strategy for synthesis in the domain of graphs.
- Gauss, which implements this algorithm for the domain of table transformations, along with a UI that allows users to use it without ever writing graph specifications themselves. Both are released as open-source[2].

---

[1]https://insights.stackoverflow.com/trends?tags=r%2Cpandas%2Cdplyr%2Ctidyr

[2]https://github.com/rbavishi/gauss-oopsla-2021
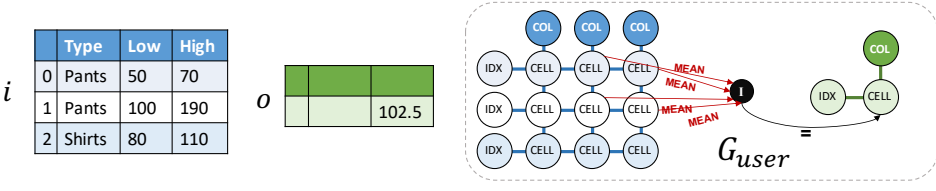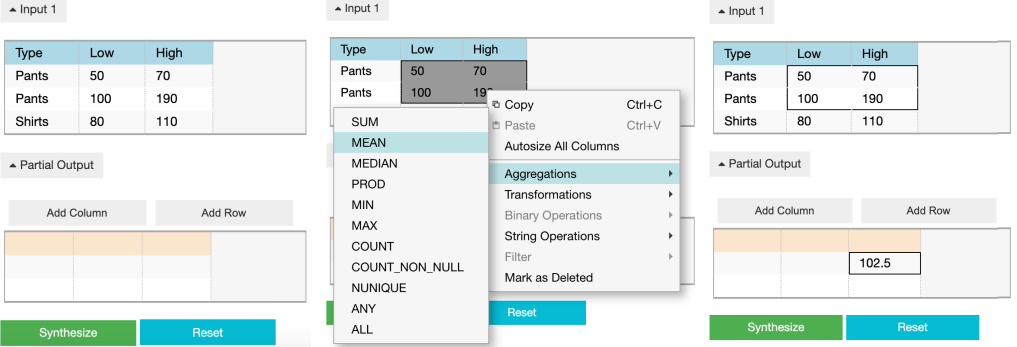
Fig. 2. An input (i), partial output (o) example, as well as a graph abstraction of user intent (G_{user}).

(a) First, the user loads the input table into the UI.

(b) The user selects the group of cells labelled "pants", right clicks, and selects the aggregation operation "MEAN".

(c) After the aggregated value is copied to clipboard, the user pastes it into the partial output.

Fig. 3. A user interaction with the UI that builds the graph abstraction of user intent from Figure 2.

2 OVERVIEW

We begin with a high-level overview of Gauss's algorithm for synthesizing table transformations. Figure 1 shows input and output tables describing a table transformation that involves a two-dimensional aggregation — the average of all Low and High values having the same Type category.

Figure 2 shows a synthesis specification that a user might supply to Gauss to synthesize code for this transformation. The specification consists of an input i, a *partial* output o and a graph abstraction of user intent G_{user}. The partial output in Figure 2 only contains the cell value 102.5, rather than the full output in Figure 1. G_{user} captures the core semantics of the transformation: the value 102.5 in the output is the mean of all the numbers in the Low and High columns with Pants in the Type column. Note that G_{user} is not provided directly by the user. We provide a user-interface (UI) that observes user interaction and automatically creates the graph G_{user}.

Figure 3 shows a series of interactions that create the graph abstraction on the right hand side of Figure 2. First in Figure 3a, the user loads the input dataframe into the UI. Then in Figure 3b, the user selects the quadrant of cells with values 50, 70, 100, and 190, and right clicks the selection. This brings up a menu of options, and the user selects the aggregation operation "MEAN": after clicking this operation, the mean of the selected values is copied to the clipboard. Finally in Figure 3c, the user pastes the value to a cell in the output section of the UI. At this point, note that the input-output example is identical to the partial input-output example given on the left hand side of Figure 2. Behind the scenes, the UI has constructed the graph G_{user}: first constructing the input table part of the graph on load (Figure 3a), then adding the intermediate computation and output nodes on paste (Figure 3c) thanks to the information provided by the user in their selection in Figure 3b.

```
148   t₁ = gather(i, "Low", "High", -"Type")
149   o = group_by(t₁, by="Type", Avg=mean("Low"))
```


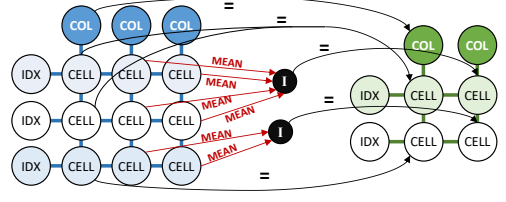|		Type	Var	Value
	0	Pants	Low	50
	1	Pants	High	70
t_1	2	Pants	Low	100
	3	Pants	High	190
	4	Shirts	Low	80
	5	Shirts	High	110

	Type	Avg
0	Pants	102.5
1	Shirts	95

o

(a) First, gather melts the input table into a "long" format, where High and Low are row values rather columns, producing t_1. Then, group_by can then average all values for each item type, producing *o*.

(b) The final graph abstraction of the solution program. Note that nodes corresponding to intermediate table cells do not appear; instead, the abstraction captures the direct relationship between input and output.

Fig. 4. The solution program for the synthesis problem in Figure 2, its intermediate and final output, and its graph abstraction.

The goal of GAUSS is to find a program which, when executed on the input table *i*, produces an output table that *contains* the partial output *o* provided by the user. Figure ?? shows the program synthesized by GAUSS. It first uses a reshaping operation gather, that "flattens" the Low and High columns into a single column (indicated by the arguments "Low", "High"), while keeping the Type column as its own column (indicated by the -"Type" argument). This call to gather results in the intermediate output t_1 in Figure ??. Then, the program performs a group_by operation, grouping on the Type column to compute the required averages. This results in the final output *o* in Figure ??.

Additionally, G_{user} must be a subgraph of the *graph abstraction* of the program synthesized by GAUSS when run on the input *i*. The graph abstraction of a program captures the relationship between its concrete inputs and output as a graph. For instance, Figure ?? shows the graph abstraction for the program at the top of Figure ?? (when using *i* from Figure 2 as input). This graph abstraction is obtained dynamically by applying a special function on the program and its inputs; the process is described in Section 4. For the purpose of this section, whenever we say a graph abstraction of a program, we assume the inputs to the program are the same as the input tables of the user-provided specification, i.e., *i*.

Enforcing that G_{user} is a subgraph of a solution's graph abstraction ensures that the program matches the user's intent. Given the spec in Figure 2, GAUSS returns the program shown in Figure 4.

We will now walk through Figure 5, which shows the steps followed by GAUSS to arrive at the solution in Figure 4. To synthesize this program, GAUSS employs enumerative search: it enumerates programs one-by-one, runs them, and checks their output against the specification. The key to GAUSS's performance is its ability to *prune* large parts of the search space of programs without enumerating them.

For simplicity in our walkthrough, we assume that we have only two table transformation components, gather and group_by, and that GAUSS only explores programs containing a maximum of two component calls. Note that every program synthesized by GAUSS is a linear sequence of component calls.

2.1 Extracting Query Graphs

To conduct this pruning, GAUSS uses *query graphs*, or simply *queries* from G_{user}. A **query** is a subgraph of G_{user} containing at least one input and one output node; GAUSS first extracts unit
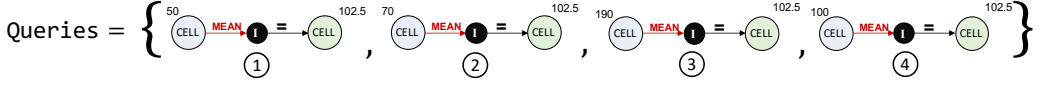
(a) Unit query graphs, or simply *queries*, of G_{user} in Figure 2.
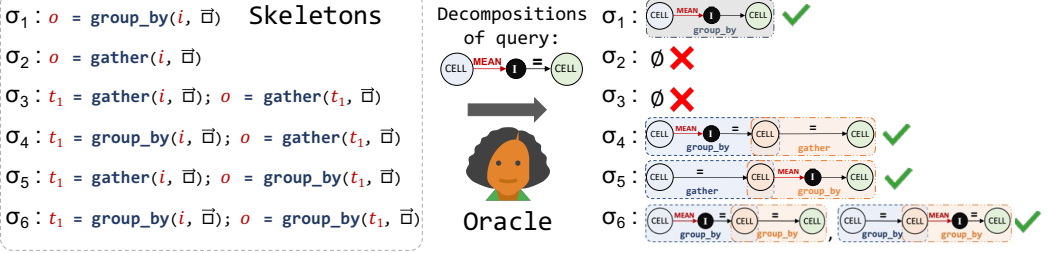
(b) For each skeleton, GAUSS uses the oracle to compute decompositions for each query. Because gather does not perform aggregation, there are no decompositions for σ_2 and σ_3. So GAUSS prunes skeletons σ_2, σ_3.
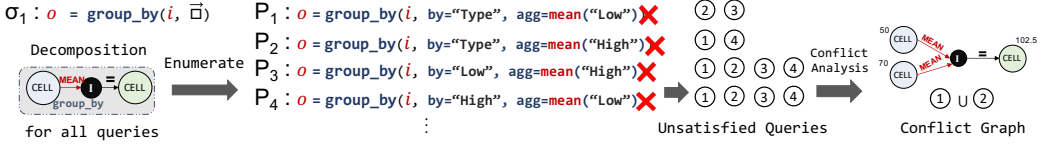
(c) While enumerating arguments for σ_1, GAUSS finds that no program realizes a decomposition for *all* queries. The smallest set of queries whose decompositions are not simultaneously realized form the *conflict query*.
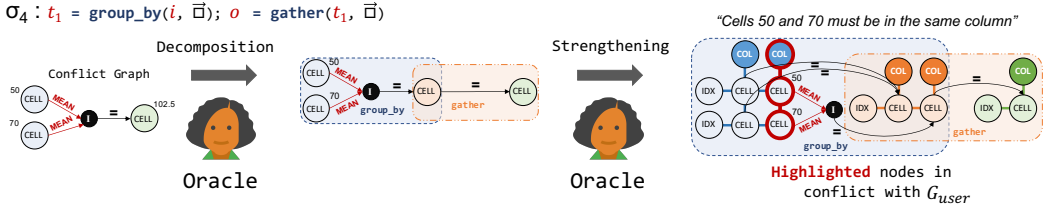
(d) GAUSS proves that no program with skeleton σ_4 is the solution. It finds that for a program with skeleton σ_4 to realize the decomposition of the conflict graph, the input cells 50 and 70 must be in the same column.
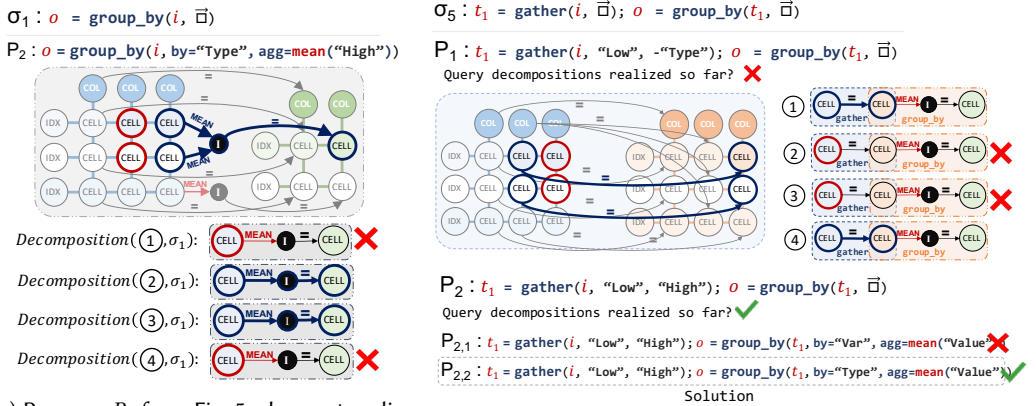
(e) Program P_2 from Fig. 5c does not realize the decompositions of, i.e. follow plans for, queries ① and ④.

(f) GAUSS prunes the partial program P_1 because it cannot realize the decompositions of queries ② and ③.

Fig. 5. Walkthrough of GAUSS run on the specification in Figure 2, with components gather and group_by.

queries that have exactly one input and one output node. Figure 5a shows the four unit queries extracted from G_{user}, one for each edge between the "input" and "output" parts of G_{user}. The numbers next to nodes indicate the table cell the node corresponds to.

Observe that if G_{user} is a subgraph of the final graph abstraction G_P of a program, the query graphs from Figure 5a *must* be subgraphs of G_P as well. This means GAUSS can reason about the simpler query graphs, rather than the potentially complex G_{user}, to prune programs.

2.2 Deciding Skeletons for Exploration

First, GAUSS prunes out *skeletons* which can be safely discarded. A **skeleton** is simply a program with constant arguments unfilled. Thus, it captures only the components of the program. Again, for simplicity in this example, GAUSS is only considering the two components `gather` and `group_by`, so only has six possible skeletons to explore. These six skeletons are enumerated in Figure 5b as $\sigma_1, \ldots, \sigma_6$. The symbol $\vec{\square}$ represents unfilled constant arguments.

First, for every skeleton, GAUSS determines all possible *decompositions* of each query graph. At a high level, a **decomposition** of a query graph G_q corresponds to a *plan* that a program P can follow to make sure that G_q is a subgraph of the final graph abstraction of P.

The right-hand-side of Figure 5b shows the decompositions of our query graphs for each skeleton $\sigma_1, \ldots, \sigma_6$. Since all the query graphs in Figure 5a are isomorphic, these decompositions are the same for all of them. Consider, for example, the decomposition of query graphs for skeleton σ_5. It lays out a plan that specifies that (1) the output table of the call to `gather` must contain a cell with the value of the input cell (captured by the edge with label "="), and (2) the call to `group_by` should perform aggregation with this cell, captured via the MEAN edge to ❶.

The skeleton σ_6 lists two possible decompositions: either of the `group_by` calls perform the aggregation. The skeletons σ_2 and σ_3 do not have any associated decompositions. This is because they only call `gather`, which is a reshaping operation. It cannot aggregate values, and thus cannot have a MEAN edge in its abstraction.

So, GAUSS discards skeletons σ_2 and σ_3, meaning it will not enumerate any programs with those underlying skeletons. GAUSS then goes through the remaining skeletons one-by-one: for each skeleton, it enumerates programs by populating the skeleton's constant arguments.

2.3 Learning from Failures

Figure 5c shows GAUSS exploring all programs with underlying skeleton σ_1. Unfortunately, none of these programs satisfy the specification. This is because they do not follow the "plan", i.e. **realize** the decomposition, for all the query graphs. Figure 5e shows this in detail for the program $P_2 = \text{group\_by(i,by="Type",agg=mean("High"))}$. Because the graph abstraction for `group_by` has the decomposition graphs for ② and ③ as subgraphs, we say P_2 *realizes* those decompositions. However, the graph does not have the decomposition graphs for queries ① and ④ as subgraphs. So, we say that P_2 satisfies the queries ② and ③, while ① and ④ are unsatisfied.

Similarly for all the other programs enumerated in Figure 5c, some queries remain unsatisfied. To zero in on what went wrong, GAUSS creates the *conflict set*. This conflict set is the smallest possible set of queries such that no program satisfied *all* the queries in set. As can be seen in Figure 5c, no program satisfies both query ① and ②. So, in our running example, the conflict set contains the queries ① and ②.[3] The graph union of the queries in the conflict set is shown on the far right of Figure 5c. This union, called the *conflict graph*, is the subgraph of G_{user} where the cells containing 50 and 70 are involved in aggregation.

---

[3]The choice of queries ③ and ④ for the conflict set would also have worked out.

Before exploring another skeleton, Gauss makes sure that the same failure will not occur again. Suppose the skeleton σ_4 is the next to be explored. As shown in Figure 5d, Gauss first uses the oracle to get the decomposition of the conflict graph with respect to σ_4.

Then, Gauss asks the oracle to *strengthen* this decomposition with respect to the skeleton σ_4. During strengthening, the oracle uncovers additional nodes and edges that must be present in any program with skeleton σ_4 realizing this decomposition. The right-hand-side of Figure 5d shows the strengthened decomposition of the query graph w.r.t σ_4. We see that the additional nodes and edges impose the condition that the cells with values 50 and 70 must be in the same column.

However, this condition is not satisfied in the user example! The cells with value 50 and 70 are not in the same column in the input i (Figure 2). Hence, Gauss can safely discard the skeleton σ_4.

2.4 Smart Enumeration

After ruling out the skeleton σ_4 (Figure 5d), Gauss moves on to the skeleton σ_5. This process is illustrated in Figure 5f. First, Gauss fills in the arguments for gather, resulting in partial program $P_1 : (t_1 = \text{gather}(i, \text{"Low"}, -\text{"Type"}); o = \text{group\_by}(t_1, \vec{\Box}_2))$. Before exploring further arguments to fill into $\vec{\Box}_2$, Gauss checks whether the program so far is on track to realize the query decompositions.

In particular, Gauss evaluates the call to gather, and computes its graph abstraction. The abstraction is on the left-hand side of Figure 5f. With this set of arguments, gather discards cells corresponding to the High column.

However, the decompositions for queries ② and ③, shown on the right of Figure 5f, require the nodes highlighted in red be connected to cells in the intermediate output. This is not the case in the graph abstraction on the left, because the corresponding cells were discarded by the call to gather!

Thus, the decomposition for queries ② and ③ cannot be realized, regardless of the arguments for group_by. That is, any completion of P_1 will not satisfy the queries ② and ③, and thus can be safely pruned by Gauss even before exploring any arguments for group_by.

All these pruning strategies allow Gauss to quickly explore more promising arguments to σ_5, before arriving at the solution in the bottom right of Figure 5f.

3 PRELIMINARIES AND NOTATION

This section establishes common notation used throughout the formal description of the Gauss algorithm. The reader may want to refer back to it while reading Sections 4, 5, and 6.

3.1 Table Transformation Programs

Gauss synthesizes a linear **table transformation program**, say P. This program takes in a list of input table variables $\vec{v_{in}}$ and a program P of length k of the form:

$$(v_1 = C_1(\vec{p}_1, \vec{c}_1); \dots; v_k = C_k(\vec{p}_k, \vec{c}_k)),$$

where:

- each C_i is a table transformation component (e.g. an API function) with a list of table arguments $\vec{p}_i$ and a list of constant arguments $\vec{c}_i$,
- each v_i is a variable representing the table output of $C_i(\vec{p}_i, \vec{c}_i)$,
- each $p_i^j \in \vec{p}_i$ is either an input table variable in $\vec{v_{in}}$ or a table variable from the set $\{v_1, \dots, v_{i-1}\}$.

Let $\mathcal{D}$ be the domain of all such programs. The set of available components, $Components(\mathcal{D})$, consists of the standard *projection*, *selection* and *cross-product* relational algebra operators along with other operations such as gather, group_by, mutate, spread that allow a mix of common reshaping and summarization operations. The domain of constants, $Constants(\mathcal{D})$, consists of the (countably

infinite) set of column names, cell values, etc. Gauss borrows this set of components from Feng et al. [2017], which the reader can refer to for a more detailed discussion.

The **execution trace** of a program P on input tables $\vec{t_{in}}$ is:

$$\langle (C_1, \vec{t_1}, \vec{c_1}, o_1), \ldots, (C_k, \vec{t_k}, \vec{c_k}, o_k) \rangle$$

where for each component C_j:

- $\vec{t_j}$ is the vector of tables passed to C_j, and
- $o_j = C_j(\vec{t_j}, \vec{c_j})$ is the table produced by the execution of $C_j(\vec{t_j}, \vec{c_j})$.

We denote such a trace as $\tau(P, \vec{t_{in}})$. The output of P is the output of the last component: $P(\vec{t_{in}}) = o_k$.

A program skeleton, or just **skeleton**, is obtained by replacing all *constant* arguments of the program's components with holes. Precisely, a skeleton σ of length k is of the form:

$$\sigma = (v_1 = C_1(\vec{p_1}, \vec{\Box_1}); \ldots; v_k = C_k(\vec{p_k}, \vec{\Box_k})).$$

$Programs(\sigma)$ is the set of all programs sharing the skeleton σ and $Skeleton(P)$ is the skeleton of the program P. We use the shorthand $C_i(\sigma)$ to refer to the ith component of σ.

A **partial program** is a partially filled skeleton. That is, a partial program P_{part} with respect to some skeleton σ maps the first d holes of σ to appropriate constant argument vectors, i.e. $\sigma[\vec{\Box_1} \mapsto \vec{c_1}, \ldots, \vec{\Box_d} \mapsto \vec{c_d}]$.

The **partial execution trace** of P_{part} on input tables $\vec{t_{in}}$, denoted $\tau(P_{part}, \vec{t_{in}})$, is:

$$\langle (C_1, \vec{t_1}, \vec{c_1}, o_1), \ldots, (C_d, \vec{t_d}, \vec{c_d}, o_d) \rangle.$$

3.2 Graphs

A graph G consists of a set of nodes $N(G)$ and edges $E(G)$. Each node $n \in N(G)$ has a label $lbl(n)$ and an entity $entity(n)$. Entities define groups of nodes, where all $n \in N(G)$ with the same $e = entity(n)$ belong to the same group. We use $Entities(G)$ to denote the set of all entities in G, $\{entity(n) \mid n \in N(G)\}$. We use $N(G, x)$ to refer to the set of nodes in G with entity x.

Our edges are directed; we use $src(e)$ and $dst(e)$ to refer to the source and destination of the edge e, respectively. The label of an edge e is denoted $lbl(e)$. We say (n_1, n_2, ℓ) is an edge in G if there exists $e \in E(G)$ such that $src(e) = n_1 \wedge dst(e) = n_2 \wedge lbl(e) = \ell$. As an example, consider the graph in Figure 4. Node labels are COL, CELL and IDX while edge labels are = and MEAN. The nodes with the blue color scheme have the same entity i, while those in green have entity o.

We say G_1 is a **subgraph** of G_2, denoted by $G_1 \subseteq G_2$ if $N(G_1) \subseteq N(G_2) \wedge E(G_1) \subseteq E(G_2)$.

The subgraph **induced** in G by a set of nodes S_N contains only the nodes of G present in S_N and edges with end-points amongst this set of nodes.

The union of two graphs is the graph $G = G_1 \cup G_2$ such that $(N(G) = N(G_1) \cup N(G_2)) \wedge (E(G) = E(G_1) \cup E(G_2))$.

A graph G is a **unit graph** if there is exactly one node in G with entity ent for every entity $ent \in Entities(G)$.

A graph G_1 is **isomorphic** to G_2, denoted by $G_1 \simeq G_2$, if there exists a bijection $M : N(G_1) \rightarrow N(G_2)$ such that:

- $\forall n \in N(G_1). \ lbl(n) = lbl(M(n))$,
- $\forall (n_1, n_2) \in N(G_1). \ (entity(n_1) = entity(n_2)) \iff (entity(M(n_1)) = entity(M(n_2)))$, and
- (n_1, n_2, ℓ) is an edge in $G_1 \iff (M(n_1), M(n_2), \ell)$ is an edge in G_2

That is, there is a mapping between the nodes of G_1 and G_2 that preserves the edge structure along with the edge labels. The mapping also preserves the node labels and groupings. We use $G_1 \simeq_M G_2$ to explicitly specify a mapping M.
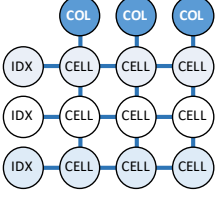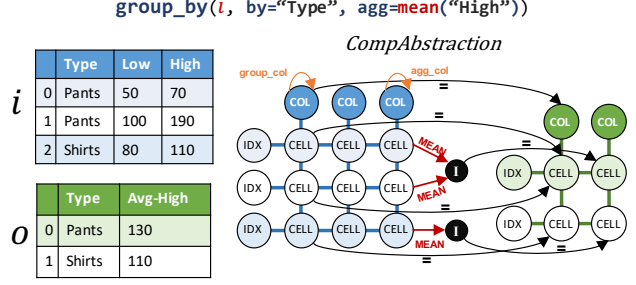
Fig. 6. Table abstraction for the input in Figure 2.

group_by(*i*, by="Type", agg=mean("High"))

Fig. 7. Component abstraction of a call to group_by. The constant arguments are embedded inside the call.

A graph G_s is **subgraph isomorphic** to G, denoted by $G_s \subseteq G$, if there exists an injective mapping $M : N(G_s) \mapsto N(G)$ and a subgraph G'_s of G ($G'_s \subseteq G$) such that $G_s \simeq_M G'_s$.

Throughout the paper, for ease of notation, whenever we use $G_1 \simeq G_2$ or $G_1 \subseteq G_2$, we enforce that the isomorphism mapping M has $\forall n \in N(G_1) \cap N(G_2). M(n) = n$. That is, M is the *identity mapping* for the nodes common to G_1 and G_2.

The $\star$ next to nodes in a sequence of graphs $\langle$, $\rangle$, indicate that the node is *shared* amongst the graphs. We use $\star_1$, $\star_2$, etc. to disambiguate multiple shared nodes. In figures, we use the notation to concisely denote sequences (e.g. Fig. 5b.).

4 GRAPH ABSTRACTIONS

We now define the concept of the *graph abstraction of a program*, which was alluded to in Section 2 but not formally defined. This concept is core to the entire GAUSS algorithm.

Suppose a program P produces an output table t_o when executed on input tables $\vec{t_{in}}$. The graph abstraction of a program P represents, as a graph, the relationship between (a) the input tables $\vec{t_{in}}$, (b) the constant arguments $\vec{c}$ embedded in P, and (c) the final output t_o. We denote the graph abstraction as $GraphAbstraction(P, \vec{t_{in}}, t_o)$.

There are two main ingredients required to define *GraphAbstraction*: (1) a *table abstraction* function $TableAbstraction(t)$ that represents table t as a graph and (2) a *component abstraction* function $CompAbstraction(C, \vec{i}, \vec{c}, o)$ that captures the relationship between input tables $\vec{i}$ and constants $\vec{c}$ and the output o of a single component C when executed with the inputs as a graph.

Our definition of the **table abstraction** function $TableAbstraction(t)$ returns a graph G_t with a node with label CELL for every table cell, a node with label COL for every column header and a node with label IDX for every row index. There is an edge with label COLUMN between the nodes corresponding to a column and every cell in that column. Similarly, there is an edge with label ROW between a row node and a cell node for every cell in that row. Thus G_t captures the structure of the table while disregarding the concrete values. Every node in G_t has the table t as the associated entity i.e. $\forall n \in N(G_t). entity(n) = t$. This captures the fact that nodes belong to the group of nodes associated with t. Figure 6 illustrates the table abstraction for the input table in Figure 2. The vertical and horizontal lines with color (▬) depict both the COLUMN and ROW edges.

Our definition of the **component abstraction** function $CompAbstraction(C, \vec{i}, \vec{c}, o)$ returns a graph G which contains the table abstractions of the inputs and output, i.e.,

$$(TableAbstraction(o) \subseteq G \land \forall i \in \vec{i}. TableAbstraction(i) \subseteq G),$$
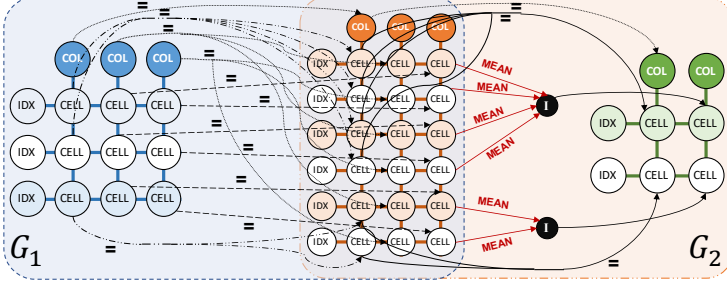
Fig. 8. Trace Abstraction for Program in Figure 4. The nodes and edges in the blue and orange boxes correspond to graphs G_1 and G_2 respectively.

along with nodes and edges that capture the relationship between the inputs and output, i.e. the semantics of the component. Figure 7 shows the graph for a call to `group_by` applied on the input table from Figure 2. Apart from the nodes and edges corresponding to the table abstractions, the equality edges (labelled '=") capture the grouping semantics, while the MEAN edges along with the *computation* nodes ❶ capture the aggregation. Additional self edges on the column nodes of the input table capture the interpretation of the constant arguments. As we shall see later in the section, these edges do not play a role in the final graph abstraction, but significantly speed up program enumeration during synthesis (Section 6.3).

We implemented these base abstraction procedures—*TableAbstraction*, and a *CompAbstraction* for each of our API components—as imperative Python programs, each 50 LoC long on average. These procedures are key to defining the abstraction of an execution trace $TraceAbstraction(\tau(P, \vec{t_{in}}))$.

Given a trace $\tau(P, \vec{t_{in}}) = \langle (C_1, \vec{t_1}, \vec{c_1}, o_1), \ldots, (C_k, \vec{t_k}, \vec{c_k}, o_k) \rangle$, the **trace abstraction** is simply the sequence of component abstractions of its constituents i.e. $TraceAbstraction(\tau(P, \vec{t_{in}})) = \langle G_1, \ldots, G_k \rangle$ where $\forall i. G_i = CompAbstraction(C_i, \vec{t_i}, \vec{c_i}, o_i)$. Figure 8 shows the trace abstraction $\langle G_1, G_2 \rangle$ of the motivating example's solution on the input in Figure 2. The graphs G_1 and G_2 are the component abstractions of the calls to `gather` and `group_by`, respectively. Note how G_1 and G_2 share certain nodes and edges: this is because they both contain the table abstraction of `gather`'s output.

The trace abstraction captures the relationships between the inputs and output *for each of the constituent component invocations* of the program. However, the graph of user intent G_{user} captures only relationships between the input and output of the *entire program*. This means that to properly evaluate containment of G_{user}, the final graph abstraction for P must capture relationships between the original input table(s) and final output table directly.

Thus, in the final graph abstraction for a program P, we need to combine the input-output relationships of the individual components into an overall relationship between the input tables of the program and the final output. Consider the union $G_{12} = G_1 \cup G_2$ of 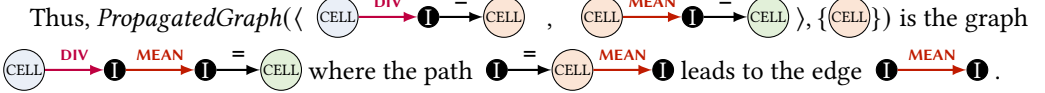graphs G_1 and G_2 in Figure 8. The path CELL $\xrightarrow{=}$ CELL $\xrightarrow{=}$ CELL appears in G_{12}. It essentially captures the fact that an input cell's value is equal to the value of a particular cell of the intermediate output table of `gather`, which in turn is equal to a cell of the final output table. We can use the transitivity of equality to conclude that the input cell's value is equal to the value of the final output cell. That is, we can add an edge as in CELL $\xrightarrow{=}$ CELL . Similarly we can simplify the path CELL $\xrightarrow{=}$ CELL $\xrightarrow{\text{MEAN}}$ ❶ $\xrightarrow{=}$ CELL in G_{12} to establish that the input cell is directly involved in an averaging operation: CELL $\xrightarrow{\text{MEAN}}$ ❶ $\xrightarrow{=}$ CELL .

We formalize this idea of *propagation* of relationships as follows:

Definition 4.1 (PropagatedGraph($\langle G_1, \ldots, G_k \rangle, S_N$)). Let $G_u = G_1 \cup \ldots \cup G_k$. The propagation graph of $\langle G_1, \ldots, G_k \rangle$ with respect to a set of nodes S_N is a graph G such that

- $N(G) = N(G_u) - S_N$ and
- there is an edge $e \in E(G)$ with $lbl(e) = \ell$ if and only if (a) its end-points are *not* in S_N and (b) there is a path between $src(e)$ and $dst(e)$ through nodes *in* S_N with at most one edge labelled with $\ell \neq$ "=" (only one non-equality edge).

Thus, *PropagatedGraph*($\langle$ CELL $\xrightarrow{\text{DIV}}$ ❶ $\xrightarrow{=}$ CELL * , * CELL $\xrightarrow{\text{MEAN}}$ ❶ $\xrightarrow{=}$ CELL $\rangle$, {CELL}) is the graph

CELL $\xrightarrow{\text{DIV}}$ ❶ $\xrightarrow{\text{MEAN}}$ ❶ $\xrightarrow{=}$ CELL where the path ❶ $\xrightarrow{=}$ CELL $\xrightarrow{\text{MEAN}}$ ❶ leads to the edge ❶ $\xrightarrow{\text{MEAN}}$ ❶ .

Now, the graph abstraction of a program is simply the propagated graph of the trace abstraction, augmented with the full table abstractions of the inputs and output.

Definition 4.2 (GraphAbstraction). Let P be a program with trace $\langle (C_1, \vec{t_1}, \vec{c_1}, o_1), \ldots, (C_k, \vec{t_k}, \vec{c_k}, t_o) \rangle$ when run on inputs $\vec{t_{in}}$ and output t_o and a corresponding trace abstraction $\langle G_1, \ldots, G_k \rangle$. Then,

$$GraphAbstraction(P, \vec{t_{in}}, t_o) = PropagatedGraph(\langle G_1, \ldots, G_k \rangle, S_N) \cup \bigcup_{t \in \vec{t_{in}} \vee t = t_o} TableAbstraction(t)$$

where $S_N = \bigcup_{i=1}^{k-1} N(TableAbstraction(t_i))$.

That is, the graph abstraction contains the abstraction of the inputs and output, as well as the relationships between inputs and output from the propagation graph. The graph abstraction must not contain any intermediate outputs, though it can contain intermediate computation nodes ❶. Hence S_N is the set of nodes belonging to the table abstractions of the intermediate output tables. Figure 4 shows the final graph abstraction of the solution program for the motivating example. Note how the user graph in Figure 2 is subgraph isomorphic to the graph in Figure 4. This is observation is the basis of the problem statement.

5 PROBLEM STATEMENT

We first formalize the synthesis problem using graph abstractions.

Definition 5.1 (Synthesis Problem). Assume a user specification consisting of input tables $\vec{t_{in}}$, partial output table $t_{o_{part}}$, and a graph abstraction of the user intent G_{user} (captured automatically via UI). The table abstractions of $\vec{t_{in}}$ and $t_{o_{part}}$ are included in G_{user} i.e. $\forall t \in \vec{t_{in}}$. $TableAbstraction(t) \subseteq G_{user}$ and $TableAbstraction(t_{o_{part}}) \subseteq G_{user}$. The synthesis problem is to find a program P such that:

$$(t_{o_{part}} \text{ is contained in } t_o) \wedge (G_{user} \subsetneq GraphAbstraction(P, \vec{t_{in}}, t_o)) \text{ where } (t_o = P(\vec{t_{in}}))$$

The first clause is in line with a standard problem formulation in example-based synthesis: the output table t_o of program P when executed on $\vec{t_{in}}$, contains the user-provided partial output table $t_{o_{part}}$. The second clause enforces a match with the user's *intent*; the graph abstraction of P must be *consistent* with the user-provided graph i.e. G_{user} is *subgraph isomorphic* to G.

6 SYNTHESIS ALGORITHM

Gauss's synthesis algorithm is *enumerative* in nature. That is, it enumerates and checks programs against the specification one-by-one and stops when it finds a solution or has exhausted all programs. Clearly, simply enumerating all programs will be prohibitively expensive as the space of possible programs is very large. The key to Gauss's performance is how it exploits the user-provided intent graph to *prune* large parts of this space without explicit enumeration. Next, we provide the intuition for, and formalize the key idea behind, Gauss's pruning: graph decompositions.

6.1 Graph Decompositions

Consider the sequence of graphs $s = \langle$ (CELL) $\xrightarrow{=}$ (CELL) * , *(CELL) $\xrightarrow{\text{MEAN}}$ ❶ $\xrightarrow{=}$ (CELL) $\rangle$. Its propagated graph (Definition 4.1) with respect to the singleton set $\{$(CELL)$\}$ is $G =$ (CELL) $\xrightarrow{\text{MEAN}}$ ❶ $\xrightarrow{=}$ (CELL) . We can think of the sequence s as a decomposition of the resulting graph G. Intuitively, s divides the task of aggregation into two parts: first preserve the value of the cell in another cell, and then perform aggregation on it. We formalize this notion of decomposition below.

Definition 6.1 (Graph Decompositions). A sequence of graphs $\langle G_1, \ldots, G_k \rangle$ is a *decomposition* of a graph G if there exists a set of nodes $S_N \subseteq N(G_1 \cup \ldots \cup G_k)$ such that:

$$G = PropagatedGraph(\langle G_1, \ldots, G_k \rangle, S_N) \cup \bigcup_{x \in Entities(G)} G[N(G, x)]$$

The second term is a subgraph of G without any *inter-entity* edges: in particular, the union of the subgraphs induced by nodes with the same entity. A decomposition is *minimal* if no edges or nodes can be removed from the constituent graphs without violating the above property.

Combining the problem statement (Definition 5.1) and the graph abstraction formulation (Definition 4.2) leads us to the following observation, which is core to Gauss's pruning strategies.

Observation 6.2. *If P solves the synthesis problem $(\vec{t_{in}}, t_{o_{part}}, G_{user})$, then for all subgraphs $G_q \subseteq G_{user}$, including G_{user} itself, there exists a decomposition $\langle G_1, \ldots, G_k \rangle$ of G_q such that:*

$$(\forall j \in [1, k].\ G_j \subseteq G'_j) \text{ and } \langle G_1, \ldots, G_k \rangle \text{ is minimal.}$$
$$\text{where } \langle G'_1, \ldots, G'_k \rangle = TraceAbstraction(\tau(P, \vec{t_{in}}))$$

Intuitively, $\langle G_1, \ldots, G_k \rangle$ can be thought of as a "plan" followed by P to ensure that G_q is present in its overall graph abstraction. This is because each G_i is a subgraph of the corresponding component abstraction G'_i. We say that this decomposition is *realized* by a program P if $\forall j \in [1, k].\ G_j \subseteq G'_j$.

Definition 6.3 (Realized Decompositions). Let $\langle G'_1, \ldots, G'_k \rangle = TraceAbstraction(\tau(P, \vec{t_{in}}))$. The predicate $Realizes(\langle G_1, \ldots, G_k \rangle, P, \vec{t_{in}})$ returns true if $\forall j \in [1, k].\ G_j \subseteq G'_j$ and false otherwise.

For example, the decomposition s at the beginning of this section — for $G =$ (CELL) $\xrightarrow{\text{MEAN}}$ ❶ $\xrightarrow{=}$ (CELL) , a subgraph of G_{user} in Figure 2 — is realized by the solution program of the motivating example. The decomposition succinctly captures the plan executed by the program — the gather invocation is in charge of preserving the value in its reshaping operation, so that the group_by invocation can then use it in a averaging operation.

How does Observation 6.2 enable pruning? Suppose we want to enumerate and check programs in $Programs(\sigma)$ against the user specification $(\vec{t_{in}}, t_{o_{part}}, G_{user})$, where σ is a skeleton of length k. Given a graph $G_q \subseteq G_{user}$, suppose we have access to a set S_σ of decompositions of G_q satisfying the following property: if there exists a program $P_\sigma \in Programs(\sigma)$ that solves the synthesis problem for the given user spec, then there exists a decomposition in S_σ realized by P_σ. This immediately allows us to implement two straightforward pruning strategies:

(1) If S_σ is empty, there does not exist any solution in $Programs(\sigma)$. Thus we can prune the entire family of programs with skeleton σ at one go.

(2) If S_σ is non-empty and there is a partial program $P_{partial}$ with the first d constant argument holes of σ filled and corresponding trace-abstraction $\langle G_1, \ldots, G_d \rangle$, if there is no $\langle G'_1, \ldots, G'_k \rangle$ in S_σ such that $G_i = G'_i$ for all $i \in [1, d]$, we can prune all programs in $Programs(P_{partial})$.

One can think of S_σ as a pre-determined set of "plans", one of which any solution program in $Programs(\sigma)$ must implement. The pruning strategies simply discard programs that clearly diverge from these "plans". Strategy (1) was motivated in Section 2.2, and Strategy (2) in Section 2.4.

Before formally developing these pruning strategies, we must first answer two main questions:

(1) Which subgraphs $G_q \subseteq G_{user}$ should we use for pruning?

(2) Given G_q and a skeleton σ, how do we construct the set of decompositions S_σ?

6.1.1   How to pick subgraphs $G_q \subseteq G_{user}$? The G_q that will help us prune the search space of programs are ones whose decompositions give us meaningful information. Consider the user graph G_{user} in Figure 2. A subgraph G consisting solely of nodes from an input table, like $G = $, is not useful because its decomposition is the trivial one: $\langle G, K_0, \ldots, K_0 \rangle$, where K_0 is the empty graph.

However, subgraphs of G_{user} which relate the input and output, like , will meaningfully decompose, and allow us to conduct the pruning steps described above. We formalize this intuition by introducing the concept of *query graphs*:

Definition 6.4 (Query). Given a user specification $(\vec{t}_{in}, t_{o_{part}}, G_{user})$, a *query graph* G_q is a subgraph of G_{user} with at least one node corresponding to every input and the (partial) output, i.e.

$$\forall(t \in \vec{t}_{in} \vee t = t_{o_{part}}).\ N(G_q) \cap N(TableAbstraction(t)) \neq \emptyset$$

Additionally, a query must contain at least one path from an input node to an output node. This ensures that queries represent a meaningful fragment of the relationship between input and output.

A *unit query*, like has *exactly* one node corresponding to every input and *exactly* one node corresponding to output. A *compound query* is simply a non-unit query.

6.1.2   How to determine possible decompositions of G_q given a skeleton σ? In constructing S_σ, the set of decompositions of G_q for a skeleton σ, there is a clear tradeoff between the effort spent building S_σ and the pruning power it gives Gauss. We could, for instance, simply let S_σ be the set of all decompositions for G_q, regardless of the skeleton σ or the user-provided input/partial output. While this is easy to pre-compute, it would not allow Gauss to do any pruning.

For optimal pruning power, S_σ should only contain decompositions that are realized by a concrete $P \in Programs(\sigma)$ for the current input and output. However computing those decompositions precisely requires enumerating all programs in $Programs(\sigma)$. That is, of course, equivalent to solving the synthesis problem itself and thus does not help in pruning. Instead of either of these extremes, we would like to hit the sweet spot: the decompositions in S_σ are not *unrealizable* and can be computed independently of the synthesis problem at hand.

Consider the decomposition $\langle$, $\rangle$ for . It is a meaningless decomposition as no program in our table transformation domain would ever be able to realize it. This is because of the way we define and implement our component abstractions: in them, a computation edge like MEAN will always end at a computation node ❶. So, the first element of the decomposition can never appear in a component abstraction.

More concretely, a decomposition $\langle G_1, \ldots, G_k \rangle$ is *unrealizable* with respect to a skeleton σ if for all graphs G_i *cannot* ever occur in the component abstraction of $C_i(\sigma)$ (regardless of the input tables and constant arguments). For now, assume we have an oracle O that offers a function $Witnessed(G, O, C)$ that checks this property for us:

Definition 6.5 ($Witnessed(G, O, C)$). $Witnessed(G, O, C)$ returns true if there exist inputs $\vec{i}$ and constant arguments $\vec{c}$ such that $o = C(\vec{i}, \vec{c})$ and $G \subseteq CompAbstraction(C, \vec{i}, \vec{c}, o)$, or false otherwise.
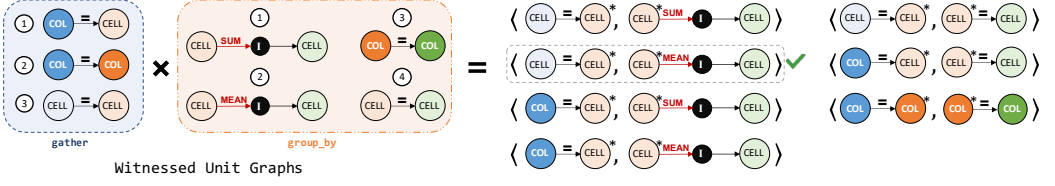
Fig. 9. Constructing decompositions with respect to skeleton ($v_1 = \texttt{gather}(t_1, \vec{\Box}_1); v_2 = \texttt{group\_by}(v_1, \vec{\Box}_2)$)

This allows us to formalize the notion of a *unrealizable* decomposition:

Definition 6.6 (Unrealizable Decomposition). We say that a decomposition $\langle G_1, \ldots, G_k \rangle$ of graph G is unrealizable with respect to a skeleton σ if there exists j such that $\neg Witnessed(G_j, O, C_j(\sigma))$.

We define the oracle O and the details of this *Witnessed* function in Section 6.6.

We define the set of possible decompositions of G_q given a skeleton σ as simply the set of all decompositions that are *not* unrealizable with respect to σ. We denote this set as *AllDecompositions*(G_q, σ).

We briefly and informally describe the construction of *AllDecompositions*(G_q, σ) when G_q is a unit query. The case where G_q is a compound query is similar and the full algorithm is explained in Appendix A. When G_q is a unit query, the construction involves two steps:

(1) Build a set of unit graphs S_i such that for each graph G in S_i, *Witnessed*$(G, O, C_i(\sigma))$ is true.
(2) Collect all sequences $\langle G_1, \ldots, G_k \rangle$—where $\forall i. G_i \in S_i$—which are decompositions of G_q.

For example, say we want to compute all decompositions of $G_q = $ with respect to the skeleton $\sigma = (v_1 = \texttt{gather}(t_1, \vec{\Box}_1); v_2 = \texttt{group\_by}(v_1, \vec{\Box}_2))$. In Step (1), we exhaustively enumerate all unit graphs for which *Witnessed* is true, for each of the components of σ. The unit graphs for gather and group_by are shown in Figure 9 in the blue and orange boxes respectively. In Step (2), we do a *combinatorial search* for all decompositions of G_q by assembling the unit graphs into a sequence. This results in 7 possible sequences, shown in Figure 9. This assembly ensures that the output nodes of the unit graphs for gather align exactly with the input nodes of group_by. Only one of these 7 sequences is in fact a decomposition of G_q; it is highlighted in Figure 9.

Armed with this concept of decompositions for query graphs, we are now ready to develop the overall enumerative synthesis algorithm, which uses decompositions extensively for pruning.

6.2 Overall Algorithm

Algorithm 1 outlines the GAUSS algorithm. It proceeds as follows. First, GAUSS extracts the set of unit queries from G_{user} in Q (Line 1). It then prepares a list of skeletons to explore, where each skeleton σ satisfies the property that *AllDecompositions*$(G_q, \sigma) \neq \emptyset$ for all unit-queries $G_q \in Q$ (Line 2). This is a direct instantiation of the first pruning strategy discussed in Section 6.1. Then, the outer loop at Line 3 iterates over the possible skeletons while the inner call to ENUMERATE (Line 5) searches for a solution program with that skeleton.

Before GAUSS enumerates programs with a particular skeleton σ, it calls FEASIBLE (line 4) to perform more checks to determine whether *Programs*(σ) can possibly contain a solution. If not, it *prunes* away the part of the search space corresponding to *Programs*(σ).

Finally, if the call to ENUMERATE fails to find a solution with skeleton σ, GAUSS attempts to identify a *small* subset of queries $Q_\kappa \subseteq Q$ (Line 8) that capture the root cause of this failure. The graph union of these queries, G_κ, is a subgraph of G_{user} not contained in the graph abstraction of *any* program in *Programs*(σ). GAUSS then keeps track of this subgraph G_κ to help prune future skeletons earlier (Line 9). A concrete example of this learning was discussed in Section 2.3.

Algorithm 1 Return a program P satisfying user spec $(\vec{t_{in}}, t_{o_{part}}, G_{user})$, or $\bot$ if no such P exists.

$\underline{\text{SYNTHESIZE}}(\vec{i}, o_{part}, G_{user})$

1: $Q \leftarrow ExtractUnitQueries(G_{user})$
2: $S \leftarrow \{\sigma \mid$ length of $\sigma \leq$ MaxLength and $\forall G_q \in Q. \, AllDecompositions(G_q, \sigma) \neq \emptyset\}$
3: **for each** $\sigma \in S$ **do**
4:     **if** FEASIBLE$(\sigma, Q, \vec{i}, o_{part}, G_{user})$ **then**
5:         $P, G_\kappa \leftarrow$ ENUMERATE$(\sigma, Q, \vec{i}, o_{part}, G_{user})$
6:         **if** $P \neq \bot$ **then**
7:             **return** P
8:         **else if** G_κ is not empty **then**
9:             $Q \leftarrow Q \cup \{G_\kappa\}$
10: **return** $\bot$

6.3 Enumeration

Algorithm 2 For skeleton σ, queries Q, and spec $(\vec{t_{in}}, t_{o_{part}}, G_{user})$, return solution $P \in Programs(\sigma)$ if it exists, else the graph union of the smallest set of queries capturing the conflict.

$\underline{\text{ENUMERATE}}(\sigma, Q, \vec{t_{in}}, t_{o_{part}}, G_{user})$

1: $k \leftarrow length(\sigma); P \leftarrow \sigma; d \leftarrow 1; \mathcal{F} \leftarrow \varnothing$
2: **while** $d > 0$ **do**
3:     $P' \leftarrow FillHoles(P, d)$
4:     **if** $P' = \bot$ **then**
5:         $d \leftarrow d - 1$
6:         $Backtrack(P, d - 1)$
7:         **continue**
8:     $\mathcal{F}_{P'} \leftarrow \{G_q \in Q$ and P does not realize any decomp. in $\in AllDecompositions(G_q, \sigma)\}$
9:     **if** $\mathcal{F}_{P'} \neq \varnothing$ **then**   ▷ Failure to realize decompositions
10:         $\mathcal{F} \leftarrow \mathcal{F} \cup \{\mathcal{F}_{P'}\}$
11:     **else if** $d < k$ **then**   ▷ More holes to fill
12:         $d \leftarrow d + 1; P \leftarrow P'$
13:     **else if** P' satisfies $(\vec{t_{in}}, t_{o_{part}}, G_{user})$ **then**   ▷ Solution found
14:         **return** $P', \varnothing$
15:     **else**
16:         $\mathcal{F} \leftarrow \mathcal{F} \cup \{\varnothing\}$
17: $Q_\kappa \leftarrow$ smallest subset of Q such that $\forall s \in \mathcal{F}. \, s \cap Q_f \neq \varnothing$ and $\varnothing$ if no such subset exists.
18: **return** $\bot, \overset{graph}{\bigcup}_{G_q \in Q_\kappa} G_q$

Algorithm 2 outlines the ENUMERATE procedure, used by GAUSS to populate program arguments. The loop at Line 2 enumerates *partial programs* by *filling* the holes $\vec{\square} = \langle \vec{\square}_1, \ldots, \vec{\square}_k \rangle$ one-by-one, via the *FillHoles* function. The variable d captures the *depth* to which the partial program P has been filled. In Lines 8 and 9, we prune any partial program which does not realize any of the available decompositions for a query G_q. This is the second pruning step motivated in Section 6.1; we gave a concrete example in Section 2.4.

If no solution is found, ENUMERATE computes the smallest set of queries, Q_κ such that no program or partial program explored was able to realize an available decomposition for *at least one* of the

queries (Line 17). ENUMERATE returns the graph union of Q_κ (Line 18), which essentially captures the root cause of failure of enumeration. This is the first step of the learning strategy described in Section 2.3. For the skeleton $\sigma = (v_1 = \text{group\_by}(t_1, \vec{\Box}_1); v_2 = \text{gather}(v_1, \vec{\Box}_2))$, this union of conflict queries corresponds to the following subgraph of G_{user} from Figure 2:
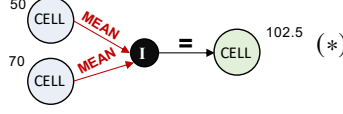
This graph embodies the fact that no programs for this skeleton were able to involve two cells in the same row (50 and 70) in an aggregation. We use this information in the FEASIBLE function to filter out other skeletons suffering from the same mistake.

6.4 The FEASIBLE Check

Algorithm 3 Given a skeleton σ, queries Q, and user-specification $(\vec{t}_{in}, o_{part}, G_{part})$, return **false** if $Programs(\sigma)$ is guaranteed to not contain a solution else **true**.

$\underline{\text{FEASIBLE}}(\sigma, Q, \vec{t}_{in}, t_{o_{part}}, G_{user})$
1: **for each** $G_q \in Q$ **do**
2:      **if** $AllDecompositions(G_q, \sigma) = \varnothing$ **then return false**
3:      **for each** $d \in AllDecompositions(G_q, \sigma)$ **do**
4:          **if** $Strengthening(d, \sigma)$ is *inconsistent* with G_{user} **then**
5:   **return false**
6: **return true**

Algorithm 3 outlines FEASIBLE, used by GAUSS to prune skeletons before enumerating arguments. The first check at Line 2 checks if decompositions are available for every query derived from the user spec G_{user}. Although a similar check is performed in the main algorithm at Line 2 in Algorithm 1, it is repeated because the set of queries can be updated in the main loop of the algorithm. The reasoning behind this check is the same as was outlined for the first pruning step of Section 6.1.

The second check in Line 4 of Algorithm 3 uses the *strengthening* of a decomposition and checks its consistency against the user graph. The next section formalizes the notion of strengthening with respect to a skeleton. We motivated it in Section 2.3: the idea of adding additional nodes and edges to the conflict graph to determine the conditions under which it must occur in the final graph abstraction. At a high-level, FEASIBLE checks in Line 4 if the nodes and edges introduced in the strengthening pertaining to the inputs and final output are consistent with the user provided graph. If they are not, the skeleton can be skipped.

For example, the strengthening of the decomposition with respect to the skeleton $\sigma = (v_1 = \text{group\_by}(t_1, \vec{\Box}_1); v_2 = \text{gather}(v_1, \vec{\Box}_2))$ of the conflict graph on the left of Figure 5d, is shown on the right of Figure 5d. This strengthened decomposition captures the fact that cells being aggregated by the first component group_by must be in the same column for any program in $Programs(\sigma)$ to produce the correct aggregation. However, this is inconsistent with the user intent graph G_{user} from Figure 2 because the two cells corresponding to table values 50 and 70 are actually in different columns. Thus, FEASIBLE can safely conclude that the skeleton cannot contain a solution, and returns false.

6.5 Strengthening Decompositions

Before formalizing the idea of strengthening, we motivate it with a simple example. Consider a skeleton containing a single component group_by, i.e. $\sigma = (v_1 = \text{group\_by}(t_1, \vec{\square}))$. Suppose there exists a program $P \in Programs(\sigma)$ which when executed on some input realizes the decomposition $\langle$ $\rangle$ for a query graph . What more can we say about P? Are there any other decompositions it is guaranteed to realize? We can in fact prove that P must also realize the following decomposition:
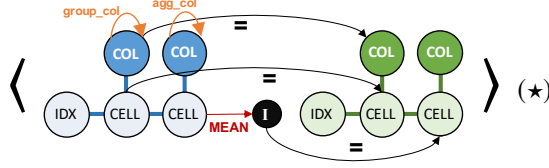
This is because of the properties of tables and the group_by function. Specifically (a) the table structure guarantees that every cell node must have a corresponding column and row node. And (b) every group_by operation must have at least one grouping column and the cell in that column in the same row as the cell being aggregated must be *equal* in value to an output cell in the same row as the cell holding the result of aggregation. Additionally, the columns being grouped and aggregated are designated by self-edges. More formally, we define strengthenings of decompositions as follows:

Definition 6.7 (Strengthening($\langle G_1, \ldots, G_k \rangle, \sigma$)). A strengthening of a decomposition $d = \langle G_1, \ldots, G_k \rangle$ with respect to σ is defined as the decomposition $d' = \langle G'_1, \ldots, G'_k \rangle$ such that:

$$\forall P \in Programs(\sigma) \text{ and inputs } \vec{i}. \; Realizes(d, P, \vec{i}) \implies Realizes(d', P, \vec{i})$$

The strengthening operation thus captures additional relationships that must be present between the nodes of the graphs involved in the decomposition. GAUSS exploits strengthened decompositions to more aggressively prune the search space.

Algorithm 4 Fixed-Point Iteration for Strengthening using oracle O.
Assume σ expands to $(v_1 = C_1(\vec{p_1}, \vec{\square}_1); \ldots; v_k = C_k(\vec{p_k}, \vec{\square}_k))$

$\underline{\text{STRENGTHENING}}(\langle G_1, \ldots, G_k \rangle, \sigma)$
1: $\langle G'_1, \ldots, G'_k \rangle \leftarrow$ copy of $\langle G_1, \ldots, G_k \rangle$
2: **while** $\exists j. \; G'_j \neq Strengthen(G'_j, O, C_j)$ **do**
3: $\quad G'_j \leftarrow Strengthen(G'_j, O, C_j)$
4: $\quad$ **for each** (i, ent) such that G'_i shares nodes with entity *ent* with G'_j **do**
5: $\quad\quad G_{ent} \leftarrow$ subgraph induced in G'_j by nodes in G'_j with entity *ent*
6: $\quad\quad G'_i \leftarrow G'_i \cup G_{ent}$
7: **return** $\langle G'_1, \ldots, G'_k \rangle$

6.5.1 Fixed-point Computation of Strengthening. The computation of *Strengthening*($\langle G_1, \ldots, G_k \rangle, \sigma$) relies again on the concept of an oracle O, which we discussed before.

Definition 6.8 (Strengthen(G, O, C)). *Strengthen*(G, O, C) returns the largest graph G' such that (a) $G \subseteq G'$ (b) for any component abstraction of C for any arbitrary input, for all isomorphisms G_s of G in the abstraction, there will be an isomorphism G'_s of G' in the abstraction such that $G_s \subseteq G'_s$.

Roughly, the graph result of *Strengthen*(G, O, C) captures additional nodes and edges that must be present for any occurrence of G in a component abstraction of C for any input. The graph in the strengthened decomposition (★) above is in fact the result of applying *Strengthen* on

 for the component group_by.

Algorithm 4 gives the algorithm for computing *Strengthening*$(\langle G_1, \ldots, G_k \rangle, \sigma)$ for a decomposition $\langle G_1, \ldots, G_k \rangle$ with respect to skeleton σ. We walk through the algorithm with the following running example. Suppose, given the query graph below on the left, we want to strengthen its decomposition, on the right, with respect to the skeleton $\sigma = (v_1 = \text{group\_by}(t_1, \vec{\square}_1); v_2 = \text{gather}(v_1, \vec{\square}_2))$:
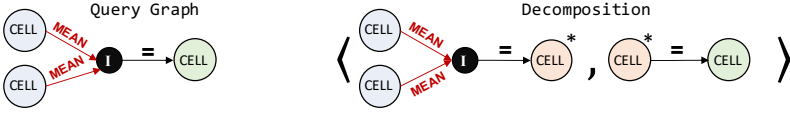
In Lines 2-3, we pick a graph in the decomposition *Strengthen*. Suppose we pick the first graph (the one for group_by). We *Strengthen* and update it to the graph on the left below. It captures the property that *"Aggregated cells should be in the same column. Their group cells must be in the same row and must be equal"*. In Lines 4-6, we update the rest of the graphs in the decomposition with the newly added nodes and edges, if they originally shared nodes with the same entity. We do the update for the graph corresponding to gather because the input to gather is the output of group_by. Hence we add the orange nodes and their edges to obtain the decomposition below on the right:
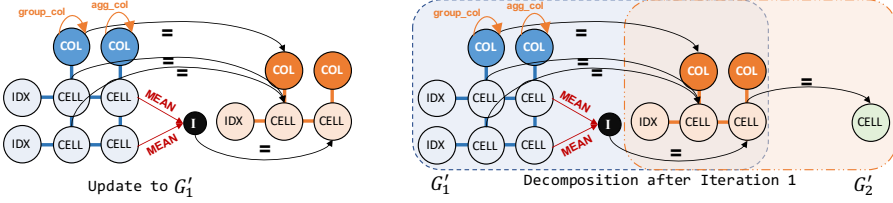
In the second iteration, we pick the second graph and apply *Strengthen* to obtain the graph on the left below. This one captures the property that all cells have a row and column. This time, we do not need to update the first graph as no new nodes have been added for the input to gather. The algorithm finishes and the final decomposition is shown on the right below:
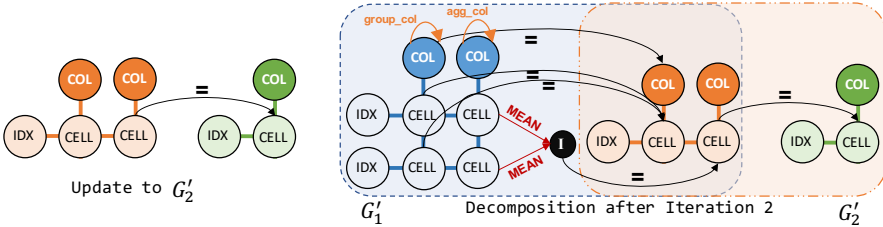
Use of Strengthening in ENUMERATE. Although omitted from Algorithm 2 for brevity, the *FillHoles* internally utilizes the strengthenings of decompositions to reduce the number of possibilities for holes. For example, suppose we have the skeleton $\sigma = (v_1 = \text{group\_by}(t_1, by = \square_1, col = \square_2(\square_3))$ and the query graph is in Q. The strengthening of the decomposition of the query graph, shown at the top of this section, contains the self-edges with labels group_col and agg_col. These self-edges in turn help prune the argument space and by quickly filtering down the possibilities for the grouping and aggregating columns. Overall, this internal use of strengthening greatly reduces the number of possibilities explored.

6.6 The Oracle

We now formalize the concept of an *oracle*, used throughout this section to develop Gauss's algorithm using the concept of *component examples*.

6.6.1 Component Examples. An *example* for a component C is a tuple $(C, \vec{i}, \vec{c}, o)$, where $o = C(\vec{i}, \vec{c})$. Essentially, an example is a set of arbitrary input tables and constant arguments and the component output on those inputs. An oracle O is simply a set of examples which contains at least one example for each component C in our table transformation domain.

If O contains n examples, i.e. $|O| = n$, we say it is of size n. We use O_∞ to denote the oracle (of infinite size) containing every possible example. Gauss's algorithm uses two operations involving the oracle, namely *Witnessed* and *Strengthen*. We slightly modify the original definitions (Definitions 6.5 and 6.8) to use examples explicitly:

Definition 6.9 (Witnessed(G, O, C)). *Witnessed*(G, O, C) returns true if there exists an example $(C, \vec{i}, \vec{c}, o) \in O$ such that $G \subsetneq CompAbstraction(C, \vec{i}, \vec{c}, o)$ and false otherwise.

Definition 6.10 (Strengthen(G, O, C)). *Strengthen*(G, O, C) returns the largest G' s.t. (a) $G \subseteq G'$ and (b) for all examples $(C, \vec{i}, \vec{c}, o) \in O$, and isomorphisms G_s of $G \subseteq CompAbstraction(C, \vec{i}, \vec{c}, o)$, there will be an isomorphism G'_s of G' in $CompAbstraction(C, \vec{i}, \vec{c}, o)$ such that $G_s \subseteq G'_s$.

The reader may notice that the results of these functions are sensitive to the number of examples used. This relates directly with the issues of soundness and completeness in the next section.

6.7 Soundness and Completeness

Gauss's algorithm is trivially sound as it only returns a program if it satisfies the specification. Completeness, on the other hand, depends on the soundness of the pruning strategies. That is, programs pruned must be guaranteed to not solve the specification. The soundness of our pruning strategies has already been discussed in the previous sections, but that discussion assumes that the results of the *Witnessed* and *Strengthen* operations are *correct*, i.e., the oracle used is O_∞. But in practice, our oracle O is finite, containing $|O| = n$ examples.

Fortunately, we can prove that for a finite set of graphs, there exists a finite oracle such that it is *behaviorally equivalent* to O_∞ when it comes to the results of the *Witnessed* and *Strengthen* functions. Let us define behavioral equivalence first:

Definition 6.11 (Behavioral Equivalence). We say that O_1 is *behaviorally equivalent* to O_2 with respect to a (possibly infinite) set of graphs S_G, if *Witnessed*$(G, O_1, C) = $ *Witnessed*(G, O_2, C) and *Strengthen*$(G, O_1, C) = $ *Strengthen*(G, O_2, C) for all graphs $G \in S_G$ and for all components C.

We can prove the following:

THEOREM 6.12. *If a set of graphs S_G is finite, there exists O of finite size $n \in \mathbb{N}$ such that O is behaviorally equivalent to O_∞ for S_G.*

PROOF. Assuming a finite S_G, we can construct such an O as follows. Let us consider the case for a single component C; we can then repeat this construction procedure for the finite number of other components. For every graph G in S_G, if *Witnessed*(G, O_∞, C) is true, there is some example in O_∞ witnessing it. So, we add this example to O. Since S_G is finite, say $|S_G| = k$, this adds at most k examples to O. Definition 6.10 implies that adding examples to O can only cause a (monotonic) *reduction* in the number of nodes and edges added by *Strengthen*(G, O, C) to G. Suppose *Strengthen*(G, O, C) adds n nodes and m edges while *Strengthen*(G, O_∞, C) adds n' nodes and m' edges to G respectively. We need to add examples to O to match the behavior; let us pick the

examples that cause a reduction of at least 1 in the number of nodes and edges added by *Strengthen*. In the worst case, each example will only remove one node or edge, and we will need to add $(n - n') + (m - m')$ examples to O to get identical behavior on S_G. So, to match O_∞'s behavior, we have added at most $k + (n - n') + (m - m')$ to O for each component; this proves that O is finite. □

In our implementation for the domain of table transformations, the set of possible node and edge labels is finite, and we set an upper limit of 2 on the number of inputs for each component. Consequently, the number of possible unit queries is fixed. Further, we only track compound queries at Line 9 in Algorithm 1 if it is obtained by merging *at most* 2 unit queries. Thus the set of graphs for which *Witnessed* and *Strengthen* is invoked is finite, and by Theorem 6.12, there exists a finite oracle containing n examples with which we can guarantee completeness. The problem of determining it still remains however. We determine n empirically by generating random examples till the results of *Witnessed* and *Strengthen* stabilize. We found that 100 random examples for each component were enough for our domain. One can also use a set of benchmark problems with known solutions to determine n which allows the algorithm to return the correct solution.

7 USER INTERFACE IMPLEMENTATION

We also provide a UI frontend to Gauss that helps users transparently create the intent graph as they interact with the UI to construct a partial output. As shown in Figure 3, the user is presented with interactive widgets for the input tables and an empty, editable space for constructing the partial output. The user can simply copy-paste values from the input to the output or use any of the primitive operations exposed by the UI. These operations can be accessed via right-clicking on any arbitrary selection of cells as shown in Figure 3b. Upon selection of the operation, the result is copied into the clipboard using which the user can paste the value in the partial output. Note that directly editing the partial output is not permitted by the UI. This ensures that values in the partial output are some function of the input cells or columns. Once the user is satisfied with the partial output provided, they can click Synthesize to retrieve solutions from Gauss. If there are multiple solutions, the user can cycle through them in the UI.

If operations need to be chained together, such as taking the ratio of two sums, the UI provides a scratch space to store intermediate computations before pasting the result into the partial output. Additional examples capturing various aspects of the UI are provided in the appendix.

8 EVALUATION

In this section, we present a comprehensive evaluation of Gauss's algorithm along two dimensions:

What is the upper-limit on the pruning power of graph-based reasoning? Given access to a *total* specification i.e. an input/output example and a user intent graph capturing *all* relationships between the input and output, how does Gauss compare against state-of-the-art pruning-based synthesizers for table transformations that accept the example alone? We evaluate this by measuring the synthesis runtimes and number of program candidates explored by each tool. As Gauss can exploit the user intent graph, we expect it to take significantly less time to synthesize a solution and explore far fewer candidates than the baselines.

Can user intent graphs reduce the size of output specifications? We also evaluate whether user intent graphs can help reduce the burden of specifying a complete output. We minimize the number of output elements–and related user intent graph nodes—in our specifications (Section 8.2), until Gauss can no longer find a solution. This allow us to measure the *achievable* reduction in specification size that user intent graphs enable.
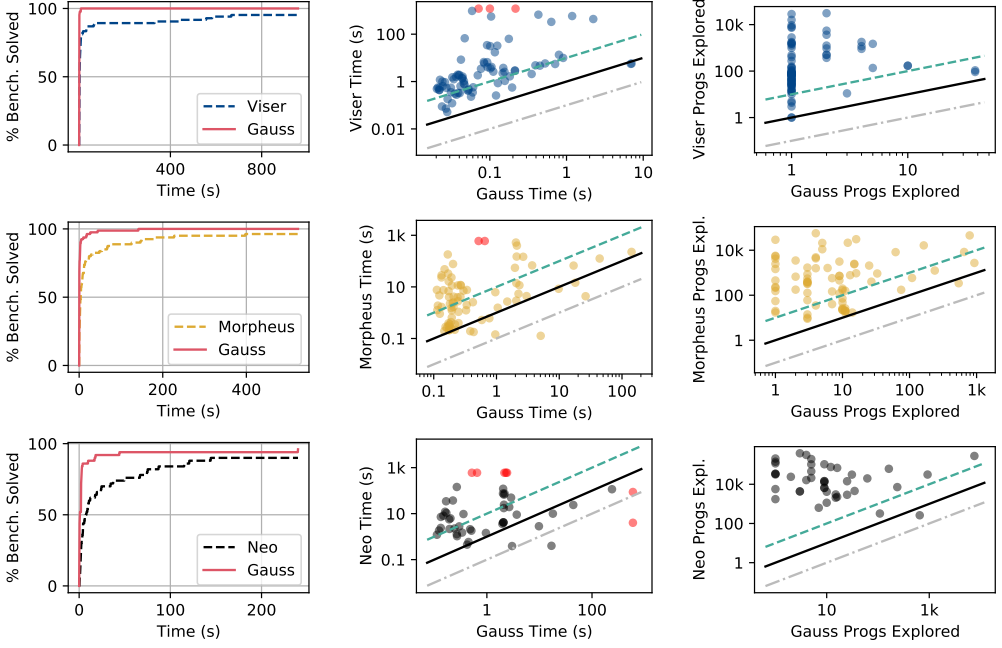
(a) Time to solve all benchmarks.   (b) Time to solve per benchmark.   (c) Progs. explored per benchmark.

Fig. 10. Comparison to Viser, Morpheus and Neo. Red dots in (b) indicate timeouts. In (b) and (c), dots above black line indicate that Gauss is better, and dots above teal dotted line indicate that Gauss is 10x better.

8.1 Baselines and Benchmarks and Hardware

We compare Gauss against three synthesizers - Morpheus [Feng et al. 2017], Neo [Feng et al. 2017] and Viser [Wang et al. 2019]. Our benchmark suite contains the 80 benchmarks used in Morpheus [Feng et al. 2017], 50 benchmarks from Neo and 84 benchmarks from Viser [Wang et al. 2019]. Note that Viser couples the problems of synthesis of table transformation and plotting programs. So, we extract the table outputs inferred during the first step of their algorithm as the output spec for our benchmarks. We also discarded Viser benchmarks that were not solvable by Viser due to the lack of expressiveness of its supported operations. Morpheus and Neo benchmarks are *harder* than Viser benchmarks in that their ground-truth solutions use 3-5 API function calls while those for Viser benchmarks use 1-2 function calls.

The pandas operations Gauss supports are disjoint from those targeted in AutoPandas [Bavishi et al. 2019a], so we cannot directly compare to it. Since all three tools support slightly differing sets of operations and thus cannot solve a portion of each other's benchmarks, we instantiate and compare Gauss against them individually. All experiments are performed on a 16-core Intel i9-9900K @ 3.6Ghz machine with 64 GB RAM running Ubuntu 18.04. We use the publicly available implementations for all three tools.

8.2 Obtaining Total Specifications for Gauss

A total specification consists of (1) an example containing input and full output tables and (2) a user intent graph capturing all relationships between the input and output. This corresponds to a scenario where a user uses our UI to provide the complete output table. For our experiments,

we obtain the user intent graph programmatically by simply using the graph abstraction of the ground-truth program. Recall that our graph abstraction does not retain any information about the functions in the program or their arguments, and is thus suitable for modeling this scenario.

8.3 Pruning Power

First, we aim to quantitatively assess the pruning capabilities of graph-based reasoning in Gauss. We measure this by comparing the synthesis times and number of programs explored by Gauss (specification includes user intent graphs) and our baselines (specification is only input-output examples). We use a timeout of 10 minutes for Morpheus and Neo benchmarks and 20 minutes for Viser, both twice the number used in the respective papers. Figure 10 shows the results.

Figure 10a compares the synthesis times of Gauss to each of the baselines. The x-axis shows the time-budget and the y-axis shows the number of benchmarks that can be solved within that budget. We see that (a) Gauss is able to solve all 80 Morpheus benchmarks with a 2-minute budget while Morpheus solves 78 with a 10-minute budget, (b) Gauss solves 48 out of 50 Neo benchmarks with a budget of 10 minutes, with 47 in under a minute, while Neo solves 45 with a 10-minute budget and (c) Gauss solves all 84 Viser benchmarks with a budget of 10 seconds while Viser only solves 81 with a 20-min. budget. Note that we ignore the time Viser spent synthesizing plotting programs.

Figure 10b shows a per-benchmark comparison of synthesis time. Dots above black solid line on the figure are benchmarks where Gauss is faster than the respective tool, those above the teal dashed lines are benchmarks indicate at least 10× speedups for Gauss, and those under dotted-dashed gray line indicate 10× slowdowns by Gauss. Red dots along the horizontal and verical axes represent timeouts for the baselines and Gauss respectively. We find that Gauss is faster than Morpheus and Neo on most benchmarks (69/80 and 37/50 respectively), and at least 10× faster on 35/80 and 20/50 benchmarks respectively. This is significant as Morpheus is written in C++ and Neo is written in Java and both parallelize their search by program depth, while Gauss is a sequential program written entirely in Python. Gauss is also faster than Viser on all but 2 benchmarks, and over 10× faster on 57 benchmarks.

Figure 10c shows a per-benchmark comparison of number of candidates explored. This includes all partial and complete programs encountered during the search. The results reveal the root cause of Gauss's performance improvements: the additional graph specifications enable Gauss to explore significantly fewer candidate programs than all baselines. For many Viser benchmarks, Gauss only needs to explore one or two candidates before finding a solution. The effect is more pronounced for Neo as its set of benchmarks are the hardest in terms of the size of the solution program. Thus, the extra graph information helps Gauss prune away most of the search space.

On average across all benchmarks, Gauss prunes 76% of partial programs encountered in Enumerate (Section 6.3) and 15% of skeletons in Feasible (Section 6.4).

Overall, the results show that user intent graphs enable orders-of-magnitude reductions in the search space compared to methods using only input-output examples as spec.

8.4 Specification Minimization

We quantitatively assess whether user intent graphs allow for a substantial reduction in the size of I/O examples provided. In particular, we try to find the smallest output tables (and corresponding user intent graphs) that still allow Gauss to synthesize the correct programs.

For this experiment, we only consider benchmarks from Viser [Wang et al. 2019] since it was designed to work on partial input-output examples. For each benchmark, we manually minimized the output and the user intent graph, while ensuring that Gauss still returns the correct solution. In particular, we kept only the output elements (cells, column names, and row indices) which were representative of the transformation. We correspondingly reduced the user intent graph to a graph
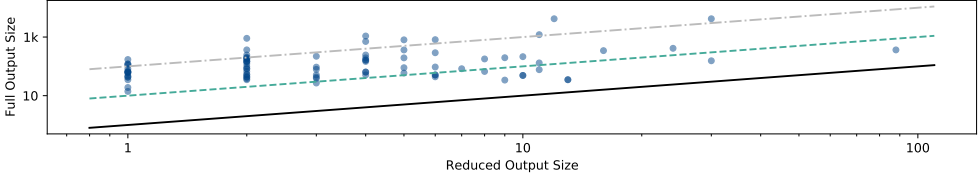
Fig. 11. Maximal reduction in number of output nodes such that Gauss still synthesizes the correct program. Dots above the green line and grey line indicate that reduction is more than 10x and 100x respectively.

that includes the full input, but only the nodes associated with these output elements. This process enables us to mimic a user inputting only these key output elements in our prototype UI.

Figure 11 shows the results. The dots correspond to benchmarks. The x-axis corresponds to the number of nodes in the reduced output while the y-axis is the number of nodes in the full output. The dots above the green line and grey line are benchmarks where the reduction is more than 10× and 100× respectively. Note that both the axes are *log scale*.

Gauss synthesizes the correct program even when provided orders-of-magnitude less information about the output. On 87% of benchmarks, it is able to find the correct solution with 10 or fewer output nodes; on 19% it finds the solution with only a single output node! The impact on runtime is minimal. In fact, Gauss finds the solution 1.2× faster on average with partial outputs.

Overall, we find that the maximal reduction of output size—while retaining Gauss's ability to find the solution—is 33× on average. Although the reduction obtained by users in a real deployment of our UI would likely be less, these results suggest that capturing user intent would reduce the burden of output specification.

9   DISCUSSION

Characterization of Graphs in Gauss. The graph abstractions for programs in Gauss can be seen as data-flow graphs since they capture relationships between the inputs and outputs. The algorithm does not place any restriction on edge or node labels, and hence the graphs can model any relationship between the input and output.

Necessity of a User Interface. Gauss returns a solution program if its abstraction contains the user intent graph as a subgraph. This inherently places the restriction that the user intent graphs need to use the same collection of node and edge labels, and capture computation in the same way, as the graph abstraction. Our use of a a UI neatly solves this problem because the UI automatically translates user interaction to a user intent graph. It also frees the user from needing to understand the internals of Gauss.

Ease of Use. Although we provide a UI in this paper, we do not explicitly evaluate its usability. Thus our empirical results, where we programmatically generate the user intent graph, may not reflect real-world usage. For example, although our UI supports construction of partial outputs in cases where chaining multiple computations is required—such as taking the ratio of two sums—, it may not be easy to use. Nonetheless, we feel intelligent UI design that allows users to input formulas for such cases can resolve this problem.

Multiple Possible Representations. There may be multiple possible ways to construct the partial output using our UI. For example, a user could explicitly compute average by first taking a sum and then dividing by the count. Our current implementation does not handle this case. This could be handled by either implementing rewrite rules for graphs in the UI to represent the operation as a mean, or incorporating such alternatives into the component abstractions. We leave this for future work.

10 RELATED WORK

Component-Based Synthesis. In component-based synthesis, a small user-provided set of components are combined to synthesize the target program. Similar to Sketch [Solar-Lezama et al. 2006], Gascón et al. [2017]; Jha et al. [2010] have the end-user place a *syntactic bias* on the search space by specifying the set of components needed. Feng et al. [2017] have the developer of the synthesis engine *instantiate* the engine with a set of components, similar to Gauss.

Abstractions for Program Synthesis. Program abstractions have long been used to combat the inherent search space explosion in program synthesis. Blaze [Wang et al. 2017b] and Atlas [Wang et al. 2018] use abstract semantics of the DSL components to construct a compact representation of all candidate programs to reason about them simultaneously. Atlas learns these abstract semantics from a separate training set of I/O examples. Synquid [Polikarpova et al. 2016] and Morpheus [Feng et al. 2017] prune invalid programs using refinement-types and first-order logic specifications respectively. Morpheus uses linear relationships between table attributes as specifications for components. Singh and Solar-Lezama [2011] use I/O examples in the abstract domain of shapes, termed *storyboards*, to synthesize low-level data-structure manipulations. The shapes discard irrelevant details about the data-structure.

Gauss combines the story-board and component-level specifications approach. It accepts input-output examples along with graph-based specifications of intent, and searches the space of programs efficiently using graph abstractions of components provided by the developer.

Graphs for Program Synthesis. AutoPandas [Bavishi et al. 2019a] is a synthesis engine based on deep learning that uses graphs to represent table transformations. These graphs are automatically inferred from the I/O example, discarding most information about values in the graph except for equality relationships. Our graph specifications are user-provided, and capture a lot more information. Additionally AutoPandas relies on a huge training set of examples to train its graph neural network, Gauss performs well with only 100 examples per function.

Synthesis using Divide-and-Conquer. The divide-and-conquer paradigm forms the basis for a large body of literature in program synthesis powering a variety of application domains [Bavishi et al. 2019b; Feser et al. 2015; Gulwani 2011; Polozov and Gulwani 2015]. Gauss falls in this bucket as well as it decomposes the problem of satisfying the graph specification by dividing it into *queries*.

Rich Input Modalities and Interaction. The *lossy* nature of I/O examples has led to a series of works that augment this form of specification with additional information from the end-user. Scythe [Wang et al. 2017a], expects a bag of constants to be used in the target SQL query. Mars [Chen et al. 2019] exploits keywords from natural language descriptions and short snippets from forums such as StackOverflow. Raza et al. [2015] accept natural language descriptions for the sub-tasks, solutions for which are combined to form the final program. Systems such as Trifacta [Rattenbury et al. 2017] track interactions of the user with the spreadsheet to gather more information about their intent. Gauss's graph abstraction shares the fundamental goal of capturing extra relationships between the inputs and output.

Recent works also take the orthogonal route of using multiple rounds of interaction with the end-user. Systems such as Bastani et al. [2019], Peleg et al. [2018] and Wrex [Drosos et al. 2020] allow users to inspect the produced program and refine the specification. Viser [Wang et al. 2019] allows users to provide more elements of the target visualization. Recently, Peleg et al. [2019] proposed a model of interaction using predicate abstraction which formalizes termination conditions for interactive synthesis. Similar interactions could be modeled with graphs—by adding nodes/edges to direct the synthesizer to the desired part of the search space.

REFERENCES

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. DeepCoder: Learning to Write Programs. *CoRR* abs/1611.01989 (2016). arXiv:1611.01989 http://arxiv.org/abs/1611.01989

Osbert Bastani, Xingjun Zhang, and Armando Solar-Lezama. 2019. Synthesizing Queries via Interactive Sketching. *ArXiv* abs/1912.12659 (2019), 1–15.

Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019a. AutoPandas: Neural-Backed Generators for Program Synthesis. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 168 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360594

Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. 2019b. Phoenix: Automated Data-Driven Synthesis of Repairs for Static Analysis Violations. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, New York, NY, USA, 613–624. https://doi.org/10.1145/3338906.3338952

Yanju Chen, Ruben Martins, and Yu Feng. 2019. Maximal Multi-Layer Specification Synthesis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, New York, NY, USA, 602–612. https://doi.org/10.1145/3338906.3338951

Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3313831.3376442

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program Synthesis Using Conflict-Driven Learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 420–435. https://doi.org/10.1145/3192366.3192382

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-Based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 422–436. https://doi.org/10.1145/3062341.3062351

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

Adrià Gascón, Ashish Tiwari, Brent Carmer, and Umang Mathur. 2017. Look for the Proof to Find the Program: Decorated-Component-Based Program Synthesis. In *Proceedings of the 29th International Conference in Computer Aided Verification.* p, a, 86–103. https://doi.org/10.1007/978-3-319-63390-9_5

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. ACM, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided Component-based Program Synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. ACM, New York, NY, USA, 215–224. https://doi.org/10.1145/1806799.1806833

Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 542–553. https://doi.org/10.1145/2594291.2594333

Hila Peleg, Shachar Itzhaky, Sharon Shoham, and Eran Yahav. 2019. Programming by predicates: a formal model for interactive synthesis. *Acta Informatica* 1, 57 (08 2019). https://doi.org/10.1007/s00236-019-00340-y

Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming Not Only by Example. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 1114–1124. https://doi.org/10.1145/3180155.3180189

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. ACM, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. ACM, New York, NY, USA, 107–126. https://doi.org/10.1145/2814270.2814310

Tye Rattenbury, Joseph M. Hellerstein, Jeffrey Heer, Sean Kandel, and Connor Carreras. 2017. *Principles of Data Wrangling: Practical Techniques for Data Preparation* (1st ed.). O'Reilly Media, Inc.

Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2015. Compositional Program Synthesis from Natural Language and Examples. In *Proceedings of the 24th International Conference on Artificial Intelligence* (Buenos Aires, Argentina) *(IJCAI'15)*. AAAI Press, Buenos Aires, Argentina, 792–800.

Rishabh Singh and Armando Solar-Lezama. 2011. Synthesizing Data Structure Manipulations from Storyboards. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. ACM, New York, NY, USA, 289–299. https://doi.org/10.1145/2025113.2025153

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. ACM, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017a. Synthesizing Highly Expressive SQL Queries from Input-Output Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 452–466. https://doi.org/10.1145/3062341.3062365

Chenglong Wang, Yu Feng, Rastislav Bodik, Alvin Cheung, and Isil Dillig. 2019. Visualization by Example. *Proc. ACM Program. Lang.* 4, POPL, Article 49 (Dec. 2019), 28 pages. https://doi.org/10.1145/3371117

Xinyu Wang, Greg Anderson, Isil Dillig, and K. L. McMillan. 2018. Learning Abstractions for Program Synthesis. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 407–426.

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017b. Program Synthesis Using Abstraction Refinement. *Proc. ACM Program. Lang.* 2, POPL, Article 63 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158151

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Brussels, Belgium, 1–10.
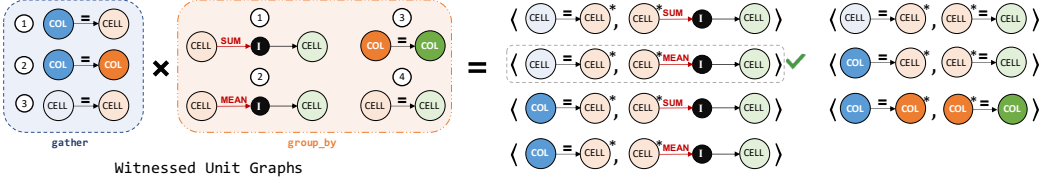
Fig. 12. Constructing decompositions with respect to skeleton ($v_1 = \texttt{gather}(t_1, \vec{\Box}_1); v_2 = \texttt{group\_by}(v_1, \vec{\Box}_2)$)

A   CONSTRUCTION OF DECOMPOSITIONS

**Algorithm 5** Construction of *AllDecompositions*(G_q, σ) using oracle O. Assume σ is of length k.

1: **procedure** *AllDecompositions*(G_q, σ)
2:     **if** G_q is a unit query **then**
3:         Construct $\langle S_1, \ldots, S_k \rangle$ such that $S_i = \{G \mid G$ is a unit graph $\wedge$ *Witnessed*$(G, O, C_i(\sigma)\}$
4:         $D \leftarrow \{\langle G_1, \ldots, G_k \rangle \mid \forall i.\ G_i \in S_i \wedge \langle G_1, \ldots, G_k \rangle$ is a decomposition of $G_q\}$
5:     **else**
6: $D \leftarrow$ *merge AllDecompositions*(G_u, σ) for unit queries G_u within G_q
7:     **return** D

Let us step through the algorithm on an example. Say we want to compute all decompositions of the query graph $G_q =$ with respect to skeleton $\sigma = (v_1 = \texttt{gather}(t_1, \vec{\Box}_1); v_2 = \texttt{group\_by}(v_1, \vec{\Box}_2))$. Since G_q is a unit query, we first exhaustively enumerate all unit graphs that are witnessed by the components of σ (Line 3). The unit graphs for gather and group_by are shown in Figure 12 in the blue and orange boxes respectively. We then do a combinatorial search for all valid decompositions of G_q (Line 4) by assembling the unit graphs into a sequence. This results in 7 possible sequences, as shown in Figure 12. This assembly essentially makes sure that the output nodes of the unit graphs for gather align exactly with the input nodes of group_by. Only one of these seven is a valid decomposition for G_q and is highlighted in Figure 12.

Consider the compound query G_q below:
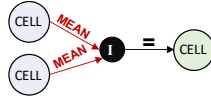
To obtain decompositions for G_q with respect to skeleton $\sigma = (v_1 = \texttt{gather}(t_1, \vec{\Box}_1); v_2 = \texttt{group\_by}(v_1, \vec{\Box}_2))$, we first get the decompositions for the constituent unit queries within G_q. There is just the unit query . We then search over all "merges" of these unit query decompositions (Line 6). A merge is essentially component-wise union of the unit query decompositions, but in which different nodes of the constituent unit queries can be merged. This results in two possible decompositions that are shown below.
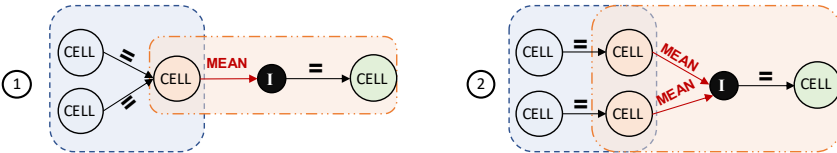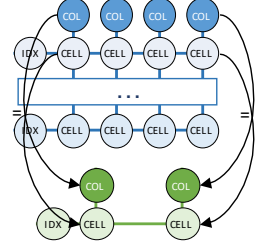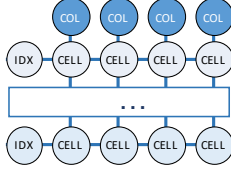
(a) Given the input data frames, the UI creates the input part of the graph, with structural edges. Some rows are omitted from the graph for space.

(b) After a user directly copy-pastes values ("id1" and "A" in the first column; "y" and "9" in the third column) from the input to the output, the UI adds output nodes linked to the input.
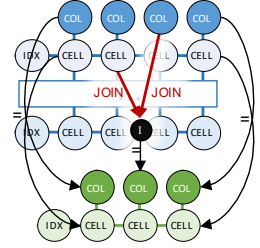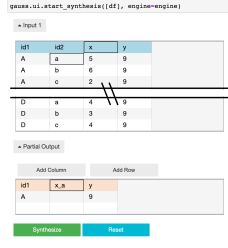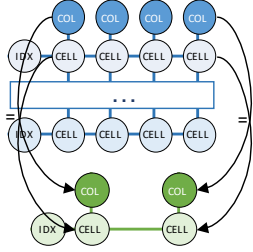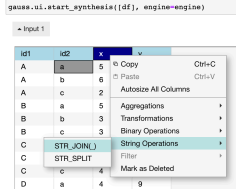
(c) The UI provides a menu of functions that can be computed on a data selection. After the user clicks "STR_JOIN", the UI copies the resulting concatenated string to the clipboard.

(d) After pasting the result of a computation to the output, in addition to adding the output nodes, the UI adds an intermediate node, linked to the output via equality and to the inputs via relation edges.

Fig. 13. Walkthrough of how the UI creates a graph specification capturing intent while the user constructs the output.

B   USER INTERFACE EXAMPLES

As discussed in Section 2, we do not expect users to directly provide the graph specification of intent to Gauss. We created a prototype UI that creates the graph specification under the hood as a user creates its input output example. This prototype is embedded as a Jupyter notebook plugin. This UI is embedded as a Jupyter notebook plugin; an interactive version of the UI is available at the link: https://github.com/gaussoopsla2021anonymous/gaussoopsla2021anonymous. This section walks through how a user might interact with the UI to solve some table transformation problems.

B.1   Example from StackOverflow

First, we look at the problem posed in the StackOverflow post 62280527[4]. This problem can be solved in the Morpheus subset of tidyr and dplyr. The original dataframe the user provides has two id columns and two variable columns (see left of Figure 13a). One of the variables, x, depends on both id1 and id2, while the second, y, only depends on on id1. The user want to create a wider table, so that all variables are dependent only on id1—by creating new columns that combine the id2-dependent variable, x, with the different values of id2.

Figure 13 shows how the user could use our prototype UI to construct the output example in this case, as well as how that UI constructs a user intent graph specification for Gauss. The user first

---

[4]https://stackoverflow.com/questions/62280527/

loads up the synthesis engine with their input dataframes. As illustrated in Figure 13a, at this point, the UI adds the nodes corresponding to all input elements to its graph specification, and links them together with basic structural edges. Each cell is linked via indexing edges to its row index and column index, as described in Section 4. The prototype UI does not allow the user to directly modify row indices, but nodes for row indices are added to the graph specification to provide GAUSS with structural relationships between the cell nodes.

Then, the user copies over a few values that are identical in the input and the output (Figure 13b). In particular, the user copies over the column header "id1" and the first value "A" of the column to the first column of the output. Then, the user copies the column header "y" and the first value in that column, "9", to the third column of the output. After each paste into the output, the UI adds nodes to the output, and links these via equality edges to the nodes that were copied in the input. The right-hand side of Figure 13b shows the graph specification after pasting both ("id1", "A") and ("y", "9") to the output.

For output elements that are not direct copies of input values, but rather functions of input values, the UI gives the user a menu of possible computations. After selecting the input elements to the computation, the user right-clicks the selection to get a menu of computations. In this case, the user wants to concatenate the variable x with the first value in the id2 column, so they choose "String Operations » STR_JOIN(_)". After the user clicks this option, the UI copies the result ("x_a" in this case) to the clipboard. As shown in Figure 13d, when the user pastes the value to the second column header in the output, behind-the-scenes, the UI adds an intermediate node representing the result of the computation to the graph specification. The UI then links this intermediate node to the input nodes used in the computation via the appropriate relation edges and to the output node the user pasted into via an equality edge.

At this point, the user could paste more values, but GAUSS actually has enough information to synthesize the transformation. About 7 seconds after the user clicks "Synthesize", the UI outputs the results of synthesis: a set of completed outputs and the program performing that transformation on the input. In this case, the first output-program pair output by GAUSS is the correct solution in the R library tidyr/dplyr:

```
out_1 = gather(input, "var", "val", "x")
out_2 = unite(out_1, "newvar", "var", "id2")
out_3 = spread(out_2, "newvar", "val" )
```

The StackOverflow answer is equivalent, but uses a newly-added (Sep 2019) API function instead, `pivot_wider`.

B.2 Chaining of Operators

In the last section, we saw how the UI could be used to find solutions to table transformations where one operation was performed on the input table data. The UI can also be used to solve table transformation problems where operations are "chained", i.e. operations are performed on results of other operations.

Figure 14 shows how the UI can be used to demonstrate, and find the solution to, such transformations. The transformation at hand assumes three columns: Name, the name of the competitor, ToD, the time of day at which a score was a achieved, and Score, the score achieved by the competitor at the time. The goal is to get a single-column output in the end that consists of one entry per name, listing the name and the maximum score that participant achieved, i.e. "Alex_950" if Alex achieved a high score of 950.

In order to demonstrate this transformation, the user first loads the input table into the UI (Figure 14a). Then they select all the score for a certain participant (Alex, in this case), and choose the operation "Aggregations » MAX" from the right-click menu (Figure 14b). After clicking on this
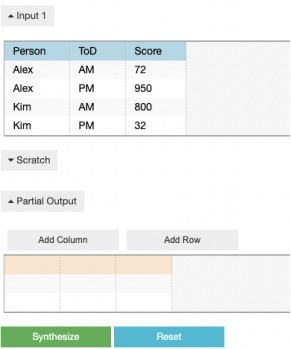
(a) The UI loaded with the input dataframe.
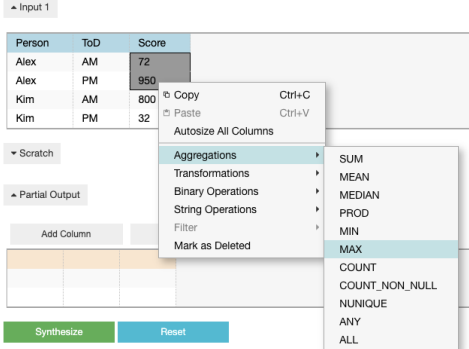
(b) The user selects two the MAX operation over two cells with the same name.
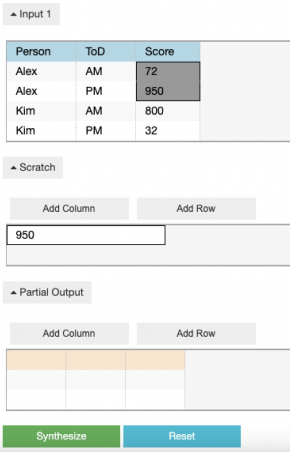
(c) The user pastes the results of the MAX operation into scratch space.
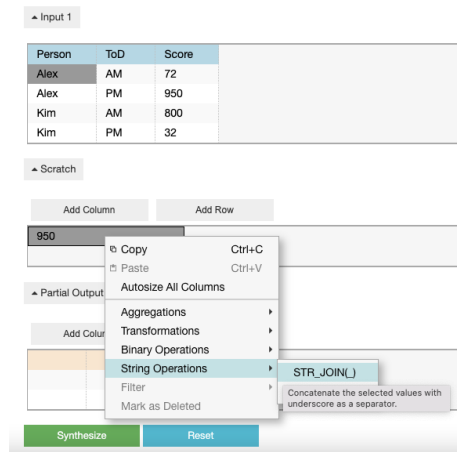
(d) The user selects values from the input and the scratch space for the STR_JOIN operation.
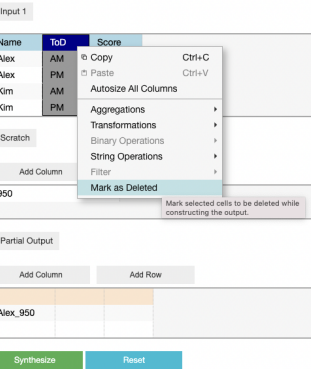
(e) After pasting the results of the STR_JOIN operation to the output, the user marks the ToD column as deleted.
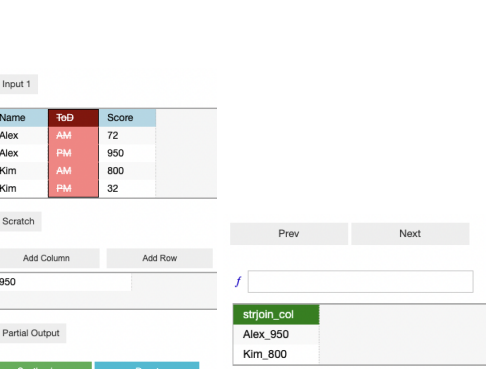
(f) In less than 10 seconds, GAUSS provides a solution that semantically match the user input (right).

Fig. 14. A run-through of user interaction in which the user uses the scratch space in order to chain operations.

1471 operation, the maximum score for Alex is copied to the clipboard. Since the user intends to use this
1472 result in further computation, they paste it to the scratch space in the UI (Figure 14c). The user
1473 can then select values from both the inputs and the scratch space and perform further operations
1474 on them; in Figure 14d we see they select both "Alex" from the input dataframe, as well as the
1475 computed maximum value as the arguments to the "STR_JOIN()" operation. They then paste the
1476 result of the string join into the output(Figure 14e. After clicking "Synthesize" for the first, the ToD
1477 column is present in the output, which does not match with the user's desired output. So, they user
1478 marks the ToD column as deleted (Figure 14e). When they click "Synthesize" again, they see the
1479 desired output table (right of Figure 14f), as well as a program that computes this output:

```
1480 out_1 = group_by(input, "Person")
1481 out_2 = summarise(out_1, "Person", max = max("Score"))
1482 out_3 = unite(out_1, "Person", "max", "strjoin_col")
```

1483 Note that the program correctly generalizes the demonstration by showing Kim's max score of 800
1484 in the output, even though that score was achieved at a different time of day than Alex's max score.
1485 Without the information of the max operation, several other completions might be possible for the
1486 user's illustrated partial output.

1487   While this example only used the scratchpad to store a single intermediate value, the scratchpad
1488 in the UI can be used to store multiple intermediate values, allowing arbitrary levels of operation
1489 chaining. The main restriction in practice to Gauss's ability to solve problems that require operator
1490 chaining is the depth of programs it explores.

C EXPERIENCE WITH REAL USERS

1493 In building our prototype Gauss UI, we did an informal study of the UI with two computer science
1494 graduate students. Both were unaffiliated with the Gauss project. The first student (henceforth
1495 referred to as Participant 1) had some experience using pandas to transform tables; the second
1496 student (henceforth refereed to as Participant 2) did not.

1497   We constructed a Jupyter notebook with 10 distinct problems for these participants to solve. Each
1498 problem specified a table transformation in natural language and with one input-output example.
1499 The goal was to find a program which could perform this transformation. In order to help the
1500 partcipants in this task, we gave the access to the Gauss UI. In particular, we provided a second
1501 input which the participants could use to build a user-intent annotated input-output example in
1502 the Gauss UI. Figure 15 shows this specification for the first problem of the ten. Note the natural
1503 language description and input-output example in the text, as well as runnable cells containing a
1504 different input table (first cell) and a command to load this input into the UI (second cell).

1505   First, we spent around 15 minutes going over the basic features of the UI. Then, we asked the
1506 participants the try to solve the 10 problems. For the first 5 problems, we asked them to only
1507 use the UI to solve the problems. That is, the participants had to load the second provided input
1508 into the Gauss synthesis UI, and construct a (partial) output from it that matched the specified
1509 transformation. For the second 5 problems, we told the participants they could either user the UI to
1510 find the solution program, or any other resources (e.g., a search engine), to write a program which
1511 matched the specification. The first 5 problems were somewhat easier than the second 5.

1512   Then, we let the participants try to solve the 10 problems, observing their progress, informally
1513 taking notes of any potential pain points, and monitoring whether they chose to use the UI for the
1514 last 5 problems. If the participants took more than 10 minutes to solve the problem, we stopped
1515 and moved on to the next problem, judging it as a timeout.

1516   Both participants were able to solve the first 5 problems with the UI alone. In fact, Participant 1
1517 solved all 10 problems with the UI, taking 1-3 minutes for each problem (including interaction with
1518 the UI, as well as validating the solution). On all 10 problems, they only provided partial outputs

Fig. 15. The specification for one of the problems we asked our participants to perform. It includes a natural language description of the transformation, along with an input-output example.

Fig. 16. Partial output provided by Participant 1 to solve Q1 (Fig. 15). The grey highlighted cells had to be added to the output for the solution to include the rows with value High (>80%) in column Var.

before coming to the correct solution. On 4 problems, they had to increase the expressiveness of these partial outputs before arriving at a solution: the first partial output they provided did not have enough information for GAUSS to find the correct solution. For instance, for the problem in Figure 15, they first provided a partial output that had the rows for Jan-16 with values Low and Medium in the range column, but they had to add a row with value High to get GAUSS to synthesize the full solution. Figure 16 shows the partial output provided, highlighting the additional elements that had to be added from the participant's first attempts. Overall, though, Participant 1 only provided 24% of the output in their partial output.

Participant 2 timed out on 2 of the 5 harder problems — they were unable to solve these problems with the UI and were unable to find external resources (i.e., via a search engine) to solve the problem. On the 8 problems they did solve, they used the UI exclusively, taking 1.5-5.5 minutes for each problem. Unlike Participant 1, Participant 2 provided full outputs on two of the problems (including problem 1, shown in Figure 15).

One of the problems that Participant 1 was able to solve but Participant 2 was unable to solve was Question 9, whose spec we provide in Figure 17a. In the problem, the input table lists the total number of items sold in different categories, as well as the number of items sold at different discount levels. The problem statement asks our participants to create a table which contains the number of items sold as a discount. Essentially, the problem can be solved either by subtracting the non-discounted items sold from total items sold, or summing the number of discounted items sold.

Participant 1 took the subtraction approach, illustrated in Figure 17b: the "143" in the output was produced by selected the Total cell and 0% cell in the TV & VIDEO row, selecting the "SUB" operation, and pasting this result to the output. This resulted in a correctly synthesized program.

Participant 2, on the other hand, took the summation approach, illustrated in Figure 17c: the "143" is produced by taking the sum of values in the 10%, 20%, and 30% columns. However, they

Q9

Description

Produce a table which gives, for each category of product, the total number of items sold, as well as those sold at discounts (i.e. not at 0%).

```
                    Category    0%   10%   20%   30%  TOTAL
0            HOME APPLIANCES   194   139   181   131    645
1                  COMPUTERS   174   118   173   103    568
2        CAMERAS & CAMCORDERS  110    94    83    68    355
3                 CELL PHONES   80    63    69    53    265
```

should become:

```
                    Category  TOTAL  DISCOUNTED
0            HOME APPLIANCES    645         451
1                  COMPUTERS    568         394
2        CAMERAS & CAMCORDERS   355         245
3                 CELL PHONES   265         185
```
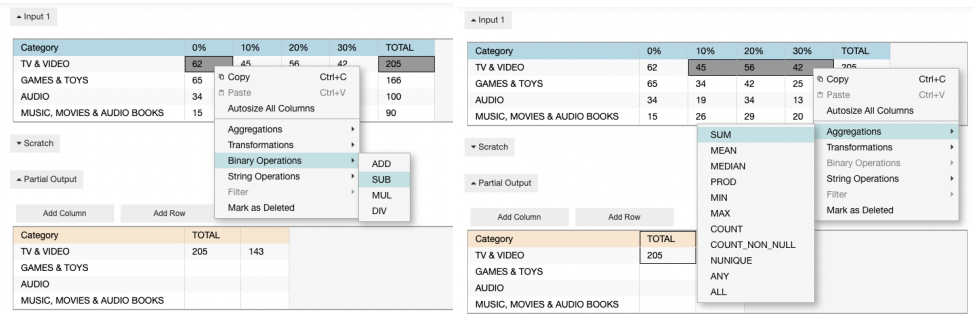
Exercise

Write a `pandas` program that performs the transformation on this input:

```
In [ ]:  input_table = pd.DataFrame({'Category': ['TV & VIDEO', 'GAMES & TOYS', '
                                     '0%': [ 62, 65, 34, 15], '10%': [ 45, 34
                                     '20%': [56, 42, 34, 29], '30%': [ 42, 25
                                     'TOTAL': [ 205, 166, 100, 90]})
         print(input_table)
```

```
In [ ]:  gauss.ui.start_synthesis([input_table])
```

(a) Specification for the problem.

(b) Participant 1 solved the problem using the "Binary Operations » SUB" operation.

(c) Participant 2 tried to solve the problem using the "Aggregation » SUM" operation.

Fig. 17. A problem from our case study to which only one participant was able to find a solution.

were unable to produce a partial output that enabled Gauss to find the correct solution. They then tried to use a search engine to figure out the pandas code that could perform the transformation, but failed. This puzzled us a the time, and several bugs were fixed after this case study with the participants. Currently, if either the method in Figure 17b or 17c is used to produce the illustrated partial outputs, Gauss successfully outputs the solution program.

Overall, this experience suggested that the UI approach to constructing user intent graphs was certainly viable. However, since one participant was not able to solve all the problems, it is clear that in practice users may not always perform demonstrations producing "good enough" user intent graphs. In-depth study of this problem will be key to the practicality of a Gauss-like approach in a deployed synthesis tool.