Towards Auto-Generated Data Systems

Alvin Cheung
University of California, Berkeley
akcheung@cs.berkeley.edu

Maaz Bin Safeer Ahmad
Adobe Research
mahmad@adobe.com

Brandon Haynes
Microsoft Research
brandon.haynes@microsoft.com

Chanwut Kittivorawong
University of California, Berkeley
chanwutk@berkeley.edu

Shadaj Laddad
University of California, Berkeley
shadaj@berkeley.edu

Xiaoxuan Liu
University of California, Berkeley
xiaoxuan_liu@berkeley.edu

Chenglong Wang
Microsoft Research
chenglong.wang@microsoft.com

Cong Yan
Microsoft Research
cong.yan@microsoft.com

ABSTRACT

After decades of progress, database management systems (DBMSs) are now the backbones of many data applications that we interact with on a daily basis. Yet, with the emergence of new data types and hardware, building and optimizing new data systems remain as difficult as the heyday of relational databases. In this paper, we summarize our work towards automating the building and optimization of data systems. Drawing from our own experience, we further argue that any automation technique must address three aspects: user specification, code generation, and result validation. We conclude by discussing a case study using videos data processing, along with opportunities for future research towards designing data systems that are automatically generated.

PVLDB Reference Format:
Alvin Cheung, Maaz Bin Safeer Ahmad, Brandon Haynes, Chanwut Kittivorawong, Shadaj Laddad, Xiaoxuan Liu, Chenglong Wang, and Cong Yan. Towards Auto-Generated Data Systems. PVLDB, 16(12): 4116 - 4129, 2023.
doi:10.14778/3611540.3611635

1 INTRODUCTION

Since the dawn of the relational data model, designing and implementing data systems has been a challenging and resource-intensive task. Not only do developers need to design a user-facing or programming interface that is intuitive for users to manipulate the stored data, devising an efficient implementation of the data model and query language is also crucial in order for the system to be usable.

The recent plethora of new data types (e.g., videos captured on phones, high-dimensional embeddings from machine learning models), new hardware architectures (e.g., GPU and other accelerators), and new data-intensive applications (e.g., training deep learning models) has exacerbated this problem. To illustrate, the initial prototype of Postgres took multiple students and three years to implement [53], and even a lightweight system such as SQLite took months before its first prototype was released [54]. Likewise, commercial implementations take even more resources to construct [55]. In fact, our own experience in building data processing systems for video data has been no better — as we will discuss in Sec. 7, it took us multiple years to design and implement a prototype system for storing and manipulating geospatial videos, and modifying our design has been a time-consuming task.

Given the recent advances in using machine learning to aid in programming tasks, it is tempting to apply such techniques to (semi-)automate the building of new data systems. In fact, such endeavor can be traced all the way back to query-by-example that was developed during the early days of SQL [74], where the idea was to develop a graphical interface for users to provide a subset of data that they would like to retrieve from the database, and the tool would find a query execution plan that performs the desired computation. Since then, various techniques have been developed, ranging from how users specify their intention to the algorithms that are used to find efficient query execution plans.

In this paper, we summarize the work we have performed in the past few years on the various aspects of automating data systems implementation. We believe that while recent advances in machine learning, programming systems, and user interface research can automate many tasks in the design and implementation of data systems, we still have a long way to go before such techniques can be applied to automatically design and implement data systems, with numerous research opportunities ahead. Learning from our experience, we argue that in order to leverage such techniques in an effective manner to build data systems, three aspects must be addressed: 1) how developers specify their intentions in terms of what they want to build, 2) algorithms to be used for generating the implementation given user specifications and, 3) mechanisms to validate the implementations generated for correctness, and refine them needed. We hope that identifying the key elements in building automated data systems, we can help the research community focus on future efforts across the different aspects.

In the following, we first describe each of these aspects in detail. We then illustrate examples of each aspect using our prior work across a number of data systems-related application domains: constructing end-user interfaces to data systems, building database-backed applications, and finding execution plans for queries. We

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611635

conclude by describing our ongoing work on building a database system and programming framework for video analytics, and future challenges that lie ahead.

2 ASPECTS OF AUTOMATION

To automate the design and implementation of data systems, we argue that three key aspects must be addressed as we outline below.

2.1 User Intention

Before we can generate implementations, we must design means to capture what the end user would like to build. This aspect is arguably the most important one in automation, as failing to understand what the user plans to build directly impacts the usefulness of the generated implementation. Researchers have designed "direct" interfaces that allow users to provide or manipulate (part of) the program to be generated (e.g., by providing the skeleton of a SQL query that includes only the tables to be scanned but not the predicates), or "indirect" ones where users provide inputs using natural language and the tool will translate such input into its own internal representation.

User familiarity. Ideally, there should be minimal friction from the user's perspective in using the interface to incorporate automation into their data processing workflows. The declarative nature of query languages makes it easy to design declarative interfaces for automation. For instance, classical programming-by-example (PBE) techniques, where users provide examples of input data and the corresponding output, fall within this category of declarative interfaces. Furthermore, recent advances in natural language processing and generative models allow users to specify their intentions using human languages, for instance as code comments in the case of Copilot [23] that can be incorporated into a code developer's existing development workflow, or as a natural language "question and answer" interface such as ChatGPT [41].

Refinement. No interface is perfect and humans make mistakes. A well-designed interface to capture user intention should include means for the user to alter their original input, either due to errors in the generated implementation or user's changing their mind. Classical query interfaces (e.g., SQL) are not designed to be "correctable" and users must start from the beginning if they are not satisfied with the previous results returned. Recent development in data exploration interfaces have incorporated ways for users to change their previous inputs, such as using a "drag and drop" interface to alter previously provided intentions [1]. Refinements can also be done in the case of natural language interfaces by users providing a new utterance that updates their previous intention.

Extracting intentions. The goal of a user interface is to capture user's intention. As such, there should be a simple way to extract the captured intention from it. Doing so in an effective manner requires designing an internal representation for user intentions, which serves as a bridge between the user input interface and the generation algorithm. Such representations include logical formulas (e.g., constraints describing what the generated implementation should and should not return), formal representations (e.g., lambda calculus in the case of semantic parsing [70]), or a combination of representations (e.g., code skeleton provided by the user along with constraints for the implementation to be generated expressed using assertions). For indirect user interfaces (e.g., natural language), it is important to have an effective translation mechanism that can convert user's input into the tool's internal representation, as errors in translation will generate incorrect implementations.

2.2 Code Generation

Given user intentions, the next step is to design algorithms that can be used to generate implementations. Such algorithms can be categorized as *search algorithms*, where the algorithm searches for code implementations that satisfy the inputs or constraints provided by the user, or as *optimization algorithms*, where a skeletal or partial implementation has been provided, and the goal is to tune the implementation such that it is optimal with respect to user-provided metrics.

There has been much work done in both categories. For code search algorithms, this includes code enumeration algorithms given a search space of programs (which can be expressed using a grammar, for instance) [74], to techniques based on version space algebra [37] where rules that are specific to the application domain are designed to explore the space of programs in a systematic manner, or utilize solvers to guide the search [52]. On the other hand, for optimization algorithms ranging from dynamic programming (e.g., the Selinger optimizer for join order [48]) to integer programming have been used, together with recent work on using machine learning models to find the optimal setting for database engine parameters [4].

2.3 Validation

Any generated implementation should be validated. Unfortunately, this is often an overlooked part of the generation workflow. At the minimum, there should be means to check the generated implementation against the user-provided specifications and see if the user specification is satisfied. Validation methods include testing the generated implementation against the provided specification, using model checkers and theorem provers to assert the validity of the generated and the original unoptimized or incomplete implementations. While it may be difficult to validate the generated implementations formally (e.g., the user specification was provided using natural language), mechanisms can be put in place to allow users examine and debug the generated implementation, or refine it if necessary.

3 APPLICATION DOMAINS

We now illustrate the three aspects mentioned above using a number of application domains. We first describe the domains before going into the details.

3.1 End-User Data Programming

While SQL is the de facto language for manipulating relational data, authoring SQL queries can be difficult due to their highly expressive constructs. In Scythe [60], we use the programming-by-example paradigm to help users write SQL queries, using a similar interface from online help forums such as Stack Overflow. In Sickle [72], a follow-up work to Scythe, we further examine the task of authoring *analytical* SQL queries (e.g., those with complex aggregations or

user-defined analytical functions). In Sickle, we explored using a combination of programming-by-example and partial queries provided by the user, along with an interface design that enables users to refine their intentions as necessary. Similarly, in Falx [62] we proposed a similar programming-by-example interface to auto-generate visualizations from relational data.

3.2 Building Database-Backed Applications

Database-backed web applications (DBWAs) underpin numerous daily online interactions, spanning from banking services to social networks. These DBWAs follow a three-tiered architecture that comprises a presentation tier (the view), an application tier, and a storage tier, each carrying out specific functions. The view, executed by the web browser, interacts with the application tier residing on a server, while the storage tier oversees the management of database queries and persistent data. This architecture opens up numerous optimization opportunities. For instance, historically, different tiers have been optimized independently, Coco [34] explores application semantics to enhance database query optimization. Similarly, Hyperloop [66] demonstrates a view-driven optimization approach. It enables developers to assign priorities to different webpage elements, and uses the analysis results to optimize the entire webpage. This approach illustrates how a nuanced understanding of DBWAs can lead to substantial performance improvement.

3.3 Query Compilation

Generating efficient query execution plans has been one of the core data management research topics for decades. While the historical focus has been on compiling relational query plans, recent efforts have focused on plan generation for non-relational queries (e.g., JSON document stores, time-series databases, etc). In Casper [2, 3], we utilized program synthesis to automatically generate distributed query execution plans in Spark and Hadoop. Starting from sequential query plans to be executed on a single-node provided by the user, Casper searches for possible plans expressed using Spark and Hadoop that implements the same functionality but executed in a distributed fashion.

Besides the execution plan, choosing the appropriate data structures to maintain data in memory also impacts the efficiency of query execution. Towards that end, we have looked into auto-generating cost-effective data structures in Chestnut [47, 64] and Katara [33], where we use combinations of classical in-memory data structures (e.g., lists and maps) along with Conflict-free Replicated Data Types (CRDTs) [50] to speed up distributed query processing.

As a preview, Table 1 summarizes the domains mentioned above, along with how the three aspects of automation are addressed in each of the scenarios. We will discuss each in detail in subsequent sections.

4 CAPTURING USER INTENTION

We first discuss our prior approaches in designing different means for users to specify their intentions, using the application domains mentioned above as an illustration.

4.1 Programming-by-Example

SQL is a highly expressive language for querying relational databases. Unfortunately, to complete practical database querying or analysis tasks, users not only need to master advanced table operators (e.g., outer-join, grouping, partition) but also need to learn how to combine them to form idioms like "arg-max," "greatest-n-per-group," and "moving average." Similarly, when users need to create data visualizations, they face the challenges of reshaping the input data into "tidy format" using data transformation libraries like tidyverse [56] and pandas [43], which is observed in prior work as a significant barrier for visualization authoring.

So, without programming, how can inexperienced users express their database querying, analysis, and visualization intents? Our solution is *programming by example*: instead of directly writing a program p to solve their task, the user demonstrates the task by providing an example input-output (I/O) pair to ask a program synthesizer to find a program p that matches the I/O examples, i.e., $p(I) = O$. Because the user only needs to know the expected output of their task on a small input as opposed to how the output should be computed in the given language (say SQL), I/O examples are commonly employed by inexperienced end-users / non-programmers in online forums like Stack Overflow to illustrate data manipulation objectives. In the past few years, we developed three programming-by-example tools to democratize database querying and data visualization.

Relational Query. Scythe [59, 60] is a programming-by-example system for database querying. Given a list of input tables $\bar{T}_{in}$ and an output table T_{out}, our goal is to construct a query q in SQL that includes projection, filtering, join, group, aggregate, union such that $q(\bar{T}_{in}) = T_{out}$. For example, a user has an input table with columns id, date, item, price, and they want to retrieve entries in T_1 that have the highest sale price for each item. To use Scythe, the user demonstrates the idea with the following example input T_1 and its corresponding output T_{out}.

T_1

id	date	item	price
1	12/24	1	10
2	12/25	1	30
3	12/24	2	20
4	12/25	2	10
5	12/27	3	50

T_{out}

c_0	c_1	c_2	c_3
1	12/25	1	30
3	12/24	2	50

With this example, *Scythe* synthesizes the "arg-max" SQL query "`Select * from T1 where T1.price = (Select max(t.price) from T1 as t where t.item = T1.item)`" to solve the user's task. As we will describe more in detail later, the synthesis process is achieved by the abstract-guided enumerative search algorithm, where *Scythe* efficiently enumerates and prunes programs based on SQL grammar to search for queries that are consistent with the user examples.

Analytical Computation. In retrieval tasks, I/O examples are effective for expressing the tasks because the output example preserves the structure information of the query. However, in analytical tasks like moving average and aggregation, asking users to provide the output example is not as ideal. On one hand, because analytical computation always involves groups of values, it requires users to manually find all the values that belong to a specific group and compute their aggregated results; on the other hand, the final output "hides" the computation process, which makes it harder

Table 1: Application domains and aspects of automation discussed in this paper

Domain	User Intention	Code Generation	Validation
End-user data programming [60, 62, 72]	Programming-by-example	Solver-based program synthesis	Consistency with user examples
Database-backed web applications [34, 66]	Unoptimized application code	Rule-based code analysis	Semantic equivalence with input code
Query compilation [3, 17, 64]	Unoptimized application code	Solver-based program synthesis	Theorem prover

for the synthesizer to automatically identify the correct operators needed for the query. To address this problem, we revised the basic programming-by-example specification into a new specification "programming-by-computation-demonstration," where the user demonstrates the computation process of how output rows are derived using values from the input table (via expressions) as shown in Fig. 1. Our tool, named *Sickle* [72], can then take advantage of the computation traces to synthesize analytical SQL queries to solve the querying task.

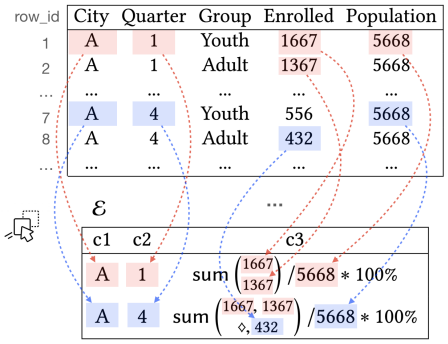

Figure 1: Using *Sickle*, the user demonstrates the analytical task of calculating the moving average of enrolled participants per city divided by population.

Visualization by Example. When it comes to data visualization, instead of asking the user to provide the input-output tables, we ask users to provide input data T_{in} and a *partial visualization* as the output example V_{ex}. The visualization example consists of a subset of visual elements (i.e., bars in a histogram, and points in a scatterplot) that should be included in the final visualization. Our tool, *Falx* [61, 62], synthesizes a data transformation program p_t and a visualization script p_v such that the final visualization contains all elements provided by the user, i.e., $V_{ex} \subseteq p_v(p_t(T_{in}))$. Fig. 2 shows how a user can use *Falx* to create a visualization that compares New York and San Francisco temperature.

4.2 Using a Familiar Programming Interface

Besides programming-by-examples, asking users to provide an unoptimized or incomplete implementation is another means to capture user intent, especially for users with programming skills. We next discuss two cases where we have applied such techniques to capture user intent.

Translating Sequential Code to MapReduce. MapReduce [20], a popular paradigm for developing data-intensive applications, has

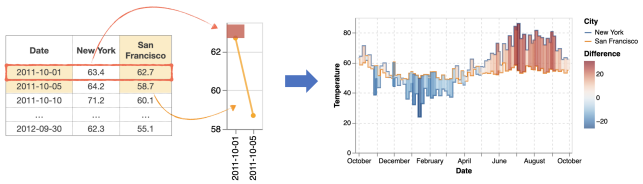

Figure 2: Using *Falx*, the user demonstrates how they plan to visualize two data points in the input data to compare New York and San Francisco temperature. *Falx* generalizes the user demonstration to a full visualization that shows all rows in the input.

```
@Summary(
  m = map(reduce(map(mat, λm1), λr), λm2)
  λm1 : (i, j, v) → {(i, v)}
  λr : (v1, v2) → v1 + v2
  λm2 : (k, v) → {(k, v/cols)})
int[] rwm(int[][] mat, int rows, int cols) {
    int[] m = new int[rows];
    for (int i = 0; i < rows; i++) {
        int sum = 0;
        for (int j = 0; j < cols; j++) {
            sum += mat[i][j];
        }
        m[i] = sum / cols;
    }
    return m;
}
```

(a) Input: Sequential Java Code

```
RDD rwm(RDD mat, int rows, int cols) {
  Spark.broadcast(cols);
  RDD m = mat.mapToPair(e -> Tuple(e.i, e.v));
  m = m.reduceByKey((v1, v2) -> (v1 + v2));
  m = m.mapValues(v -> (v / cols));
  return m;}
```

(b) Output: Apache Spark Code

Figure 3: Translation of the row-wise mean benchmark to MapReduce (Spark) using Casper.

varied and highly efficient implementations [6–8, 44]. All these implementations expose an application programming interface (API) to developers. While the concrete syntax differs slightly across the different APIs, they all require developers to organize their computation into map and reduce stages in order to leverage their optimizations.

While exposing optimization via an API shields application developers from the complexities of distributed computing, legacy applications must re-organize the computation using the target framework's API in order to leverage such frameworks. This requires not only expertise but also a significant expenditure of time and effort. Further, each code rewrite or algorithm reformulation opens another opportunity to introduce bugs. Casper [2, 3] is a tool that alleviates these issues by automatically translating sequential Java code into semantically equivalent MapReduce programs. Rather than relying on rules to translate different code patterns, Casper is inspired by prior work on cost-based query optimization [49] and program synthesis (to be discussed in Sec. 5.2), which consider compilation to be a dynamic search problem.

To illustrate, given a sequential Java program, such as the row-wise mean function in Fig. 3(a), Casper first annotates the program with a *program summary* that helps with the translation to MapReduce. The program summary describes how the output of the code fragment (in this case `m`) can be computed using a high-level intermediate representation (IR) of a series of *map* and *reduce* with transformer functions on the input data (i.e., `mat`), as shown on lines 1 to 5 of Fig. 3(a). While the summary is not executable code, it accurately describes the semantics of the input code using an abstraction that is easy to translate to the concrete syntax of the target MapReduce framework. This is shown in Fig. 3(b) where the *map* and *reduce* primitives from our IR are translated to the appropriate Spark API calls.

Casper infers program summaries through program synthesis and verification. Casper's IR for program summaries is designed to succinctly express computations in the MapReduce paradigm, allowing program synthesis to discover complex algorithms found in real code. Furthermore, Casper accelerates the search process by incrementally expanding the space of candidates considered during search and uses a domain-specific cost model to bias search towards performant MapReduce translations. We provide more details on Casper's incremental grammar generation in Sec. 5.3.

Synthesizing Replicated Data Types. Besides finding execution plans, choosing the appropriate data structures is another aspect that affects execution efficiency. For developers who only have experience writing software for single-node systems, reasoning about the failure conditions and asynchrony in a distributed system can be a daunting task. Recent innovations like CRDTs [50], a class of data structures that can be replicated in an eventually consistent manner, offer a more friendly object-oriented interface. But they have complex correctness guarantees that make it hard for developers to craft their own domain-specific types.

With program synthesis, this challenge turns into an opportunity! The same correctness guarantees can be encoded as constraints on the synthesis problem, with existing single-threaded logic used as a reference to verify the behavior of the distributed implementation. We apply this approach in Katara [33], a system that automatically synthesizes CRDTs that are observationally equivalent to sequential data types.

By providing an existing data structure implementation in C/C++, such as a Boolean toggle or set, and a lightweight annotation that specifies how concurrent mutations should be ordered, Katara can automatically perform end-to-end synthesis to derive a provably equivalent CRDT. For example, in Figure 4, Katara takes a sequential implementation of a shopping cart (which supports insertions and removals) along with a tie-breaking *opOrder* which specifies how concurrent operations should be ordered. Using just this information, Katara can automatically synthesize a provably-correct CRDT implementation that can be used as a drop-in replacement.

Input: Sequential C Datatype and Operation Order

```
set* init_state() { return set_create(); }

set* next_state(set* state, int add, int v) {
  if (add == 1) return set_insert(state, v);
  else return set_remove(state, v);
}

int query(set* state, int v) {
  return set_contains(state, v);
}
```

$$opOrder(o_1, o_2)\text{: } o_{1,add} = 1 \vee o_{2,add} \neq 1$$

Output: Synthesized CRDT Design

crdt *ShoppingCart*
  initialState: $(\{\}, \{\})$
  merge $(a_1, a_2), (b_1, b_2)$
    return $(a_1 \cup b_1, a_2 \cup b_2)$
  operation $(s, add, value)$
    return $merge(s,$ **if** $add = 1$ **then**
      $(\{value\}, \{\})$
    else
      $(\{\}, \{value\})$
    $)$
  query $((s_1, s_2), value)$
    return $value \in (s_1 \setminus s_2)$

Figure 4: A user-provided sequential reference and the CRDT design synthesized by Katara.

To generate the implementation, Katara uses program synthesis as the code generation mechanism, based on the sequential, single-node implementation. We will discuss the details in Sec. 5.2.

4.3 Program Analysis

In applications that interact with databases, the SQL queries are often not created directly by users or issued stand-alone; instead, they are a sequence of queries automatically generated by the application code. One such example is web applications developed via object-relational mapping (ORM) frameworks like Django [21] and Rails [45], where users write code in a programming language such as Python or Ruby, and the ORM translates such code into SQL.

The following shows an example snippet from a project-management web application, which uses an ORM library to retrieve projects with non-empty associated todos with condition *done==false*. These projects are then rendered in a webpage returned to the user. In such applications, the application code is the user specification, where it describes (1) how the data is generated for the application database, (2) what data they want to retrieve from the database, and (3) how to make use of such retrieved data. Therefore, program

analysis can help understand the application developer's intent and optimize the application queries accordingly. We illustrate how this can be done through three systems, PowerStation, Panorama and Coco.

```
# code to retrieve data from the database
user = User.where(:id=param[:id])
projects = user.projects
todo_projects = projects.select{|p|
↪  p.todos.where(done=>false).count>0}
# code to generate a webpage to render unfinished projects
<% for p in todo_projects %>
<a href=..%=p.id%..><%=p.name%></a>
```

Listing 1: An example snippet that generates a web page showing a user's unfinished projects

PowerStation [65, 67, 68] improves commonly-seen inefficient queries in database applications that use ORM frameworks to generate SQL queries. These queries are often not inefficient by themselves, but because of how the query result is used. For example, an automatically-generated SQL query `SELECT * FROM table` retrieves the entire table while the application only uses the first tuple, it can be optimized by adding a `LIMIT 1` clause to retrieve the only tuple being used. Powerstation performs program analysis on 12 popular open-source applications to extract 9 common anti-patterns which produce inefficient queries and generate patches for 6 of these anti-patterns. Such anti-patterns widely exist: PowerStation identifies 1221 inefficiency issues in these applications, and automatically generates patches for 730 of them, achieving performance gain up to 10×.

While PowerStation improves the application without changing its semantics, Panorama [69] offers the user alternative choices in the application design that can achieve much greater performance gain. For example, a forum application shows a list of discussion posts on its main webpage. The developer may render all the posts at once and let the user scroll down to browse them, which may make this webpage slow under a large number of posts. In this case, the developer may want to consider paginating the webpage to show a fixed number of posts and let the user browse other posts by clicking "next page" button. We build Panorama to automate this process. It again uses program analysis to extract the intention from the user-provided code, with the goal of understanding the cost of queries issued to generate the webpage as well as the cost to process the query result. A heatmap for the webpage is then rendered to let developers easily understand which component incurs the highest computational cost to render. It will also recommend alternative designs like pagination and automatically refactors the code if the developer accepts the new design. Our user study shows that using Panorama's recommended designs would not affect the user experience (occasionally even improving it) while achieving speedup up to 38× in rendering.

As a third illustration, Coco [34] is a system that automatically extracts both application and database constraints from the source code. Database constraints are explicitly defined in the migration files [40] for applications built using ORM frameworks, and they alter the database schema over time as the application evolves. Application constraints are semantically embedded in the application code during model class definition. Listing 2 gives an example of application constraint. Lines 1-4 define the `Member` class and lines 5-7 create and save a `Member` object. Line 4 utilizes Rails' built-in validation API and is called whenever the object is saved to the database, as shown in line 7. Rails executes the validation on Line 4 by executing a query to determine if a user with the same project already exists in the member table and raises an error if so. Here, the validation function implicitly defines a data constraint that *given a project, the users belonging to the project are unique*. Yet, it is only defined in the application code but not specified as part of the database schema, as developers can write arbitrary code in the validation function, and not all of them can be easily translated to SQL constraints. Because of its flexibility, ease of use, and error management capabilities, constraints are often defined in the application rather than the database. Coco uses both database and application constraints to enhance query performance. It works by treating the application source code as specification, analyzing the code by building an abstract syntax tree (AST) for each file, and performing pattern-matching on the AST nodes. Using this analysis, Coco extracts an average of 289 constraints for six evaluated applications, and subsequently uses these extracted constraints to improve query execution by over 2× to be discussed in Sec. 5.

```
# Member Class definition
class Member
    belongs_to :user, :project
    validates_uniqueness_of :user_id, :scope => :project_id
# Create a Member object and save it to the database
member = Member.new(user_id=1, project_id=2)
member.save
```

Listing 2: Redmine code with an implicit data constraint.

5 CODE GENERATION

The goal of extracting user intentions is to use them to generate code. We highlight the different approaches we have devised in prior work in automating the code generation process.

5.1 Rule-Based Search

A simple search algorithm is a rule-based search, which defines the patterns to look for in the original program as well as rules to produce new program for each pattern. In Chestnut [47, 64], we explore patterns in the schema and the queries in an application to find better data structures to store the data based on such patterns. For example, in the TPC-H benchmark, the *l*ineitem table is often inner-join or semi-joined with the *o*rders table on *o*_orderkey to filter the orders where each order can map to several lineitems. Chestnut accelerates such queries by using a nested data structure that stores a list of *l*ineitems (storing a subset of projected fields) inside each *o*rders tuple. Such data structure essentially pre-computes the joins while avoiding redundancy as compared to fully materialized join results. Chestnut uses a rule-based search that produces nested data structures according to foreign keys and join patterns and only searches for nestings that can be used to answer queries (rather than trying all possible data structures given the large search space), and chooses the optimal one based on the estimated query cost. It achieves significant performance improvement by speeding up queries to 42× for applications built using ORMs, which typically store results using nested objects and are often inefficient.

On the other hand, Coco leverages detected constraints to enhance query performance by rewriting queries. It uses the constraints it has extracted to enumerate feasible rewrites, such as removing DISTINCT. Subsequently, it filters out inefficient rewrites based on estimated costs. To guarantee that the rewritten queries are semantically equivalent to the original, Coco uses both testing and formal verification, to be discussed in Sec. 6. This process results in a lookup table composed of pairs of original and optimized queries. As the application runs, this table is used online to efficiently rewrite queries, thereby enhancing performance. Fig. 5 shows the total number of queries rewritten by Coco after analyzing six open-source Ruby on Rails applications, as well as the number of rewrites after each step. In total, Coco's constraint-driven optimizer improves the performance of 2,492 queries, 118 of which have over 2× speedup.

Coco utilizes a rule-based search mechanism to list potential rewrites, with each rewrite drawing upon specific types of constraints. As such, Coco enumerates potential rewrites exclusively when the relevant constraints are present. In practical terms, Coco extracts all columns employed in a query and cross-references them with the extracted constraints. For each identified constraint, Coco applies the associated potential transformations. For instance, for the query in Listing 3, Coco extracts the used columns `members.user_id`, `users.id`, `users.status`, and `members.project_id`. Coco then determines that `users.id` and each pair of (`members.user_id`, `members.project_id`) is unique. It then removes the DISTINCT keyword and add `LIMIT 1` to generate three candidate rewrites as shown in lines 14–16. Coco's modular design makes it easy to add new types of rewrite rules. Users can simply add a new semantic query rewrite rule to the search space by providing the utilized constraints and associated enumeration.

```
1  -- Original Query
2  SELECT DISTINCT users.* from users
3  INNER JOIN members ON members.user_id = users.id
4  WHERE users.status = $1 AND (members.project_id = $2)
5
6  -- Used columns
7  members.user_id, users.id, users.status, members.project_id
8
9  -- Extracted constraints on used columns
10 Uniqueness: (members.user_id, members.project_id) pair is unique
11 Uniqueness: users.id is unique
12
13 -- Candidate rewritten templates:
14 1. SELECT users.* from users INNER JOIN members ON members.user_id =
   ↪  users.id WHERE users.status =  $1 AND (members.project_id = $2)
   ↪  -- remove DISTINCT
15 2. SELECT DISTINCT users.* from users INNER JOIN members ON
   ↪  members.user_id = users.id WHERE users.status =  $1 AND
   ↪  (members.project_id = $2) LIMIT 1  -- add LIMIT 1
16 3. SELECT users.* from users INNER JOIN members ON members.user_id =
   ↪  users.id WHERE users.status =  $1 AND (members.project_id = $2)
   ↪  LIMIT 1  -- remove DISTINCT and add LIMIT 1
```

Listing 3: Heuristic-guided rewrite for a Redmine query. Redmine is an open-source application built using the Ruby on Rails ORM framework.

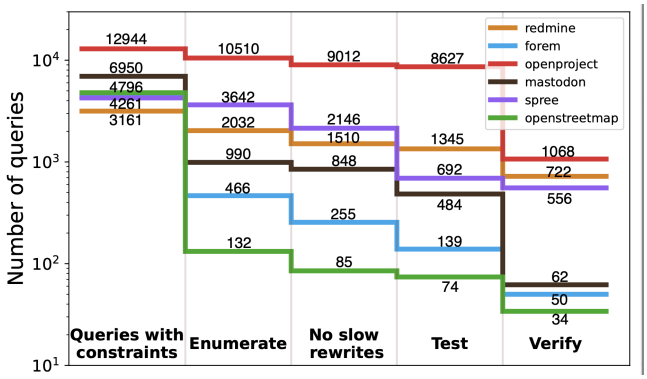

Figure 5: Number of queries after each of Coco's analysis step. Queries with constraints: queries that can potentially be rewritten by leveraging the constraints extracted by Coco. Enumerate: possible query plans that are enumerated by Coco's rule-based search algorithm given the extracted constraints. No slow rewrites: queries that can be sped up after being rewritten. Coco determines the potential speedups using the database's cost estimator. Test: rewritten queries that return the same results as the original by running tests on a small test database. Verify: once the rewrites pass the test phase, Coco then uses the Cosette solver in an attempt to formally verify their equivalence. Each line represents a different application, and the final numbers are the number of queries that not only achieve a speedup but are also proved to be semantically equivalent to the original query.

5.2 Program Synthesis

Abstraction-guided Program Synthesis. The workhorse behind programming-by-example systems is the *abstraction-guided program synthesis algorithm*, a technique that leverages abstract semantics of the underlying language to effectively prune incorrect program skeletons during top-down enumerative search to speedup the synthesis process. This technique powers relational query and data transformation program synthesis in *Scythe*, *Falx* and *Sickle* described in Sec. 4.1.

Fig. 6 shows how *Falx* synthesizes a data transformation p from input tables T_1, T_2 and partial output example T. Unlike simple top-down enumerative search algorithms that would fully expand the search tree until all leaf nodes are concrete programs, *Falx*'s abstraction guided synthesis algorithm allows it to prune program sketches (i.e., programs with uninstantiated parameters annotated as holes "□") before they are fully expanded. This allows the synthesizer to prune the full sub-tree without paying the cost of fully expanding the tree and evaluating all programs (which is exponential to the number of holes in the sketch). The reason that the synthesizer can prune such program sketches is that while the program is not yet fully instantiated, we can derive properties of the output table T_{out} based on known parameters (i.e., properties that all programs in this sub-tree share), and if these properties conflict with the synthesis objective, we can confidently prune the program sketch.

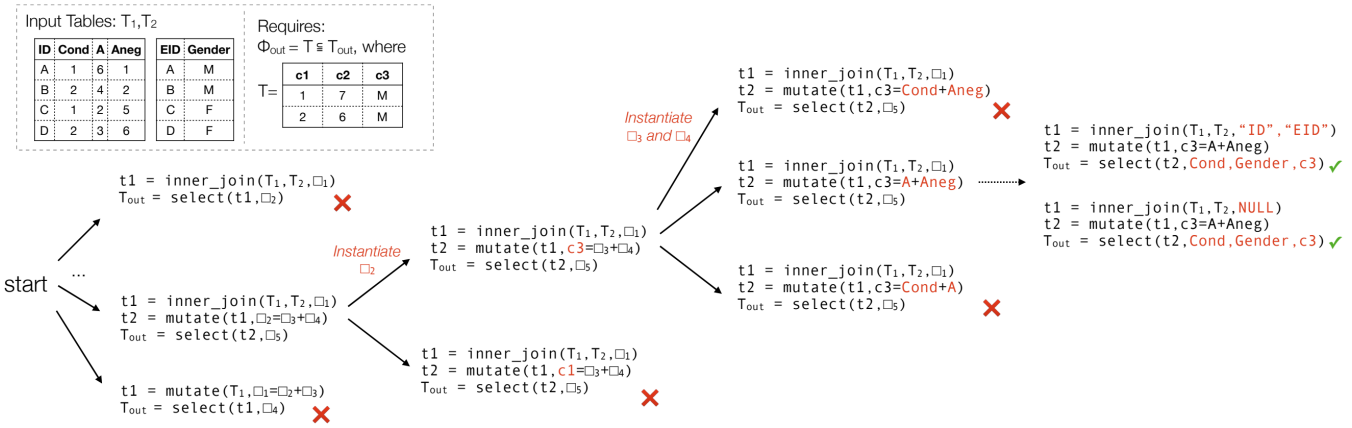

Figure 6: Given input tables T_1, T_2 and partial output table T, *Falx* synthesizes a data transformation program p composed of R tidyverse operators whose output T_{out} satisfies the objective Φ_{out}. At each step, the synthesis algorithm first picks a known variable and expands it (new values expanded at each step are labeled in red), then it evaluates each program sketch using abstract semantics of the table transformation language and prunes it if the evaluation process results in conflicts.

For example, given the program `t1 = inner_join(T1, T2, _); T_out = select(t1, _)` [1] in Fig. 6, while the synthesizer does not yet know the possible join predicates and projection parameters, it can infer the properties of this sketch that the program cannot generate new values that are not in T_1 and T_2. Given that, the synthesizer compares the property with the output example T: the partial output T contains new value 7 that is not in T_1 or T_2, and thus the program sketch is incorrect and can be pruned away. These properties are organized systematically as *abstract semantics* of the language, allowing the synthesizer to recursively derive such properties given any composition of the partially instantiated operators.

Verified Lifting. Katara uses verified lifting [10, 14, 29, 51] to search for CRDT implementations. Verified lifting internally uses program synthesis as the code generation mechanism. On top of lifting, Katara introduces several new components to the lifting pipeline. Because synthesizing a CRDT requires not only defining the implementations of operations and queries on the datatype, but specifying how the state of the data structure is represented internally, Katara must search through a space of potential storage types. We leverage the composition of join-semilattices, which define how instances of a CRDT can be merged, to define a grammar of state structures that we perform an enumerative search over.

But the challenges don't end there! Unlike past verified lifting work where the correctness conditions rely on equality of outputs when given the same inputs, CRDTs separate interactions into mutating operations and queries, and thus correctness relies on *observational* equivalence. When the original datatype and the CRDT are fed the same sequence of operations, and then are queried for a piece of that state, then the result of the query must be the same. Katara takes the approach of synthesizing inductive invariants to verify correctness over unbounded interactions.

But performing this synthesis can be extremely expensive, sometimes taking multiple hours on some benchmarks. To tackle this, Katara applies another novel technique in the verified lifting space: using multiple phases of synthesis to enumerate candidates according to bounded verification and only synthesizing inductive invariants after using the first phase to prune the search space. Altogether, Katara is able to synthesize CRDT equivalents to many classic data structures in minutes, and comes up with CRDT implementations that have better performance characteristics than human-designed ones in existing literature. Katara demonstrates how program synthesis can go beyond *translating* code to new domains, instead *extending* it with new capabilities.

[1]The select operator in R tidyverse implements projection, unlike in relational algebra.

5.3 Incremental Search

The fundamental hurdle faced by search-based methods is scalability. As the space of candidate solutions grows larger, exhaustively searching for a correct solution can become prohibitively expensive. This problem is further exacerbated when the goal is to not merely find a correct solution but rather an optimal solution. An important tactic for scaling up search is to incrementally expand the space of candidate solutions considered during search. For example, if the search space is encoded as a context-free grammar, we can expand the space of candidate solutions by (1) adding new production rules to the grammar, and (2) increasing the number of times that each product rule can be expanded.

There are two benefits to this approach. First, since search time for a valid solution is proportional to the size of the search space, we can often find valid solutions quickly for less complex problems that do not require a very expressive grammar to solve. Second, as larger grammars generate longer and more complex solutions, the discovered solutions are likely to be more expensive computationally. Hence, incremental grammar generation can be a way to bias search towards more desirable candidates, such as those with higher performance.

The idea of incremental search was used in Casper, where we partition the space of program summaries into different *grammar classes*, where each class is defined based on a number of syntactical

features: (1) the number of Map/Reduce operations, (2) the number of emit statements in each *map* stage, (3) the size of key-value pairs emitted in each stage, as inferred from the types of the key and value, and (4) the length of expressions (e.g., $x + y$ is an expression of length 2 while $x+y+z$ has a length of 3). All of these features are implemented by altering the production rules in the search space grammar. A grammar hierarchy is created such that the set of all program summaries that are expressible in a grammar class G_i is a syntactic subset of those expressible in a higher level class, i.e., G_j where $j > i$. Our approach was quite effective in pruning down the search space and reduced search time by at least one order of magnitude as evaluated using our benchmark set.

6 REASONING ABOUT CORRECTNESS

Validating the generated code is an important part of any auto-generated system. While testing the generated implementation against the user-provided inputs might suffice in situations where test cases are available, formal reasoning represents another validation mechanism. In this section, we illustrate this using Cosette [15, 17, 18], an equivalence checker for SQL. While the problem of determining equivalence for arbitrary SQL queries is undecidable [58], Cosette focuses on fragments of SQL queries that commonly arise in practice, including conjunctive queries, correlated queries, queries with outer joins, and queries with restricted uses of aggregates.

As an equivalence checker, Cosette takes in two SQL queries and determines if they are equivalent, i.e., for two given queries if they will always return the same results for all possible input relations. Cosette leverages recent advances in both automated constraint solving and theorem proving, and returns a counterexample (in terms of input relations) if two queries are inequivalent, or a proof of equivalence otherwise. As shown in Fig. 7, Cosette consists of two parts: a proof search engine that translates the input queries into a formal representation called U-semiring expressions, and leverages a proof assistant to find a proof that shows that the two input queries are semantically equivalent; and a counterexample search engine that converts the queries into constraints, and uses a constraint solver to find *counterexample relations*, i.e., instances of input relations to the two queries such that, when executed, the two queries will return different results, hence proving that they are inequivalent.

Cosette internally uses a combination of constraint solving and theorem proving to search for equivalence proofs and inequivalent counterexamples. As such provers do not come with an understanding of SQL, our first task is to model the semantics of relations and queries.

6.1 Proof Search

In searching for proofs, our goal is to show that the two input queries always return the same results for all input relations. While we can model relations as sets and queries as iterations over sets, it would be challenging to search for proofs under such modeling, as doing so will require proving equivalence of iterations which can become highly undecidable even for simple iterations.

For proof search, Cosette instead models relations and queries based on K-relations, which was first proposed by Green et al. [25]. Under K-relations, a relation is modeled as a function that maps

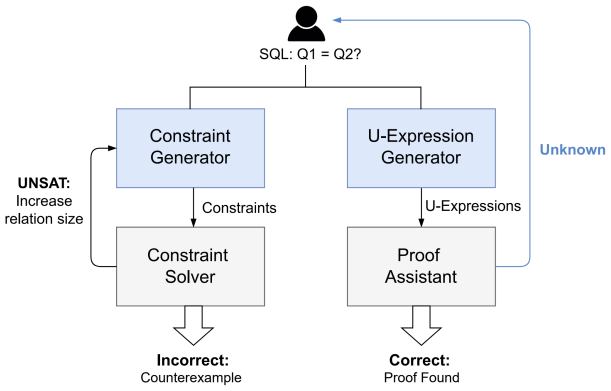

Figure 7: Architecture of Cosette

tuples to a commutative semiring, $\mathbf{K} = (K, 0, 1, +, \times)$. In other words, a K-relation, R, is a function:

$$[\![R]\!] : \texttt{Tuple}(\sigma) \rightarrow K$$

with finite support. Here, $\texttt{Tuple}(\sigma)$ denotes the (possibly infinite) set of tuples of type σ, and $[\![R]\!](t)$ represents the multiplicity of t in relation R. For example, a relation under SQL's standard bag semantics is an $\mathbb{N}$-relation (where $\mathbb{N}$ is the semiring of natural numbers), and a relation under set semantics is a $\mathbb{B}$-relation (where $\mathbb{B}$ is the semiring of Booleans). All relational operators and SQL queries can be expressed in terms of semiring operations; for example:

$$[\![\texttt{SELECT * FROM } R\ x,\ S\ y]\!] = \lambda\ (t_1, t_2)\ .\ [\![R]\!](t_1) \times [\![S]\!](t_2)$$

$$[\![\texttt{SELECT * FROM } R \texttt{ WHERE } a > 10]\!] = \lambda\ t\ .\ [t.a > 10] \times [\![R]\!](t)$$

$$[\![\texttt{SELECT } a \texttt{ FROM } R]\!] = \lambda\ t.\ \sum_{t' : \texttt{Tuple}(\sigma)} [t'.a = t] \times [\![R]\!](t')$$

For any predicate b, we denote $[b]$ the element of the semiring defined by $[b] = 1$ if the predicate holds, and $[b] = 0$ otherwise.

To extend K-relations for *automatically* proving SQL equivalences under database integrity constraints, we develop a novel algebraic structure called U-semiring, as the semiring in K-relations [16]. A U-semiring extends a semiring with a few new operators and a minimal set of axioms, each of which is a simple identity.[2] With these additions, SQL equivalences can be reasoned by checking the equivalences of their corresponding U-expressions using only these axioms expressed in U-semiring identities. Cosette implements this semantics by translating the input queries into U-semiring expressions, along with a rule-based search engine to find equivalent proofs for the input queries.

6.2 Searching for Counter-Examples

Besides searching for equivalent proofs, Cosette also looks for counterexamples that can show that two SQL queries are inequivalent. It does so using bounded verification with a constraint solver, where it models a table as a list of tuples of symbolic values, where each tuple models a row in the table together with its multiplicity in the table. For example, given a table schema $(id, price, sales)$, Cosette creates the following symbolic table:

[2] An *axiom* is a logical sentence, such as $\forall x\ .\ R(x) \Rightarrow S(x)$.

id	*price*	*sales*	*multiplicity*
s_{11}	s_{12}	s_{13}	m_1
s_{21}	s_{22}	s_{23}	m_2

In this table, the symbolic values s_{ij} are symbolic integers that represent any possible integers, and m_i is a symbolic non-negative integer that represents how many times the tuple would appear in the actual concrete table. The choice of explicitly representing multiplicity is to concisely represent larger tables with less symbolic values to reduce computation overhead. This symbolic table directs Cosette to search the space of all tables with at most two distinct tuples to check for possible distinguishing inputs that can disambiguate candidate queries.

Given the symbolic table, Cosette then proceeds to generate constraints over these symbolic values based on input queries. Cosette first translates input queries q_1, q_2 into a pair of Rosette [57] programs p_1, p_2, and then generates an assertion $p_1(T) == p_2(T)$ asking Rosette to either prove it within the bound of given symbolic input T or generate a counter-example. Rosette would further compile the assertion into SMT constraints and asks Z3 to instantiate all symbolic values defined in the table, and returns the counterexample to the user if one is found.

7 A CASE STUDY: DATA MANAGEMENT FOR SPATIAL-TEMPORAL VIDEOS

We now describe a case study that highlights the challenges involved in designing data systems for a new domain in our prior work, and how having automatic generation techniques can help with design exploration and implementation.

7.1 Virtual & Augmented Reality Video Data

Managing virtual and augmented reality video (respectively VR and AR video) data at scale is a formidable challenge. While all video applications tend to be data-intensive, VR and AR video applications are especially so, typically requiring data transfer of terabytes per second [35]. Despite this challenge, developers often work with VR and AR data as if it were ordinary, two-dimensional (2D) video. This result in brittle applications that intermix application logic with 2D video plumbing (e.g., spherical projection [12, 46], encoding idiosyncrasies [19, 36])

To address this impedance, we introduced LightDB [27], a system that treats all types of VR and AR video in a logically unified manner. LightDB enables developers to reason about VR and AR video as a declarative, first-class construct, and frees them from needing to reason about storage and format irrelevancies such as compression codec, resolution, or stereoscopic representation.

To do so, LightDB introduces a logical abstraction called a *temporal light field* (TLF). A TLF captures the degrees of freedom (i.e., spatial, rotational, and temporal) available to a viewer in VR or AR space. LightDB then exposes an algebra and declarative query language (*VRQL*) which developers use to express their operations over TLFs. In addition to decoupling application intent from low-level plumbing, this enables the LightDB query optimizer to identify the most performant way in which to execute the application. To further increase opportunities for optimization, LightDB additionally supports various indexes (e.g., a tile index similar to [19]) that may be defined on TLFs.

As LightDB introduces a new data model, we had to design a new algebra and API, and implement them from scratch. In the end, we designed a query algebra comprising of a minimal set of ~20 non-blocking operations on TLFs that are composed to express rich functionality that includes a number of modern VR and AR applications we describe in [27]. Each operation is a unary or n-ary function from one or more input TLFs to an output TLF. The algebra includes **selection operators** that capture a region of space or resample at a particular resolution; **partitioning operations** that enable subqueries to operate on partitions independently, **merge operators** that combine TLFs; **transformations** that modify, translate, or rotate TLFs; and **storage operators** that load or store TLFs to (from) disk.

Developers use VRQL to express VR and AR applications. Functions in VRQL correspond to the LightDB algebra. For example, consider the application described in [28], where a developer wishes to partition a viewer's perspective into regions and render areas a viewer is likely to look in higher quality. Using VRQL, a developer might write the C++ query[3] shown in the Listing 4.

```
1  Scan("input_tlf")
2    >> Partition(Time, 1)
3    >> Partition(Theta, π / 2)
4    >> Partition(Phi, π / 4)
5    >> Subquery([](auto& partition) {
6          return Encode(partition, is_important(partition)
7              ? Quality::High : Quality::Low) })
8    >> Store("rtp://...");
```

Listing 4: A VRQL query to partition the viewer's perspective and render the important ones in higher quality.

This query reads an input TLF and partitions it into one-second segments and regions of size $(\frac{\pi}{2}, \frac{\pi}{4})$. Next, the subquery operator is applied to each partition, which changes the quality to that given by a user-defined function that predicts a user's future orientation (e.g., by extrapolating the direction of a viewer's head motion and encoding the projected region in high quality). Finally, the result is streamed to a viewer's headset using the RTP protocol. We describe further real-world applications in [27].

Collectively, the LightDB algebra and query language enable developers to express modern VR and AR applications using up to 97% fewer lines of code [27]. These applications are then executed using the LightDB architecture, query optimizer, and reference implementation, resulting in up to a 4× speedup compared to variants expressed in common 2D video processing frameworks such as FFmpeg [9] and OpenCV [42].

7.2 Geospatial Video Data

Geospatial videos are videos in which the location and time that they are shot are known. Specifically, the cameras that shoot such videos contain geospatial metadata, such as their location, rotation, and intrinsic [71] over time. Joining with the geospatial metadata, objects (e.g., cars, pedestrians) within geospatial videos also inherit geospatial properties. As a result, users analyze geospatial videos through manipulating and querying for such objects. However, the lack of programming frameworks and data management systems

[3]This example is drawn with minor modifications from [27]. Additionally, as is typical in functional languages, VRQL uses $g(\alpha) \gg f(\beta)$ as shorthand for $f(g(\alpha), \beta)$.

for geospatial videos has made it challenging for end users to specify their workflows, let alone run them efficiently.

Consider a data journalist writing an article on car crashes statistics and would like to examine the footage collected on traffic cameras to look for a specific behavior, e.g., two cars crashing at an intersection. Given today's technology, they will either watch all the footage themselves, or concatenate a myriad of machine learning models to track objects in the videos [11], estimate the objects' depth [24] from the cameras that shot the videos, and combine the outputs (tracks and depth) from the models using the cameras' geospatial metadata to estimate the objects' geospatial locations. After that, they will need to join the objects' geospatial locations with road network information (say stored in a geospatial database [73]) to find the cars that are at an intersection. Finally, they use a general programming language to loop through each car to find an event where two of them face each other at the same period of time. Sadly, the former is simply infeasible given the amount of video data collected, while the latter is use-case specific, error-prone, and reliant on users' deep programming expertise that many do not possess.

Given LightDB, we next attempted to design a programming framework specifically targeted to geospatial videos. The result is Spatialyze [30], a system for geospatial video analytics that bridges video processing and geospatial data analytics. Spatialyze comes with a conceptual data model where users create and ingest videos into a "world," and users interact with the world by specifying objects (e.g., cars) and scenarios (e.g., cars crossing an intersection) of interest via Spatialyze's domain-specific language (DSL) embedded in Python called S-Flow.

```
1  w = World()
2  world.addGeogConstructs(RoadNetwork('road-network/boston-seaport/'))
3  world.addVideo(GeospatialVideo(Video('v0.mp4'), Camera('c0.json')))
4  world.addVideo(GeospatialVideo(Video('v1.mp4'), Camera('c1.json')))
5  obj1, obj2, intersection = (world.object(), world.object(),
6                              world.geogConstruct(type='intersection'))
7  world.filter(
8    (obj1.type == 'car') & (obj2.type == 'car') &
9    (distance(obj1, obj2) < 10) &
10   contains(intersection, (obj1, obj2)) &
11   headingDiff(obj1, obj2, between=[135, 225])  )
12 world.saveVideos('output_videos/', addBoundingBoxes=True)
```

Listing 5: Geospatial video analytic workflow with Spatialyze

With Spatialyze's conceptual data model, we introduce a concept of *World*, a virtual geospatial environment. *World* is the outer-most building-block that contains 1) *Geographic Constructs*, non-moving constructs, such as roads or intersections; 2) *Cameras*, configurations (extrinsics and intrinsics [71] over time) of the cameras that shot the input videos; and 3) *Movable Objects*, objects tracked from the input videos, e.g., cars.

With the conceptual data model, we show S-Flow, a DSL that allows the users to interact with their geospatial and video data. We propose a *build-filter-observe* programming paradigm for users to construct geospatial video analytic workflow as shown in Listing 5. First, in lines 1-4, users *build* a world, by integrating video data with the geospatial metadata. Second, in lines 5-11, they *filter* for the video parts of interest, by describing relationships between *Movable Objects* (cars) and their surrounding *Geospatial Constructs* (intersection). Finally, in lines 5, they *observe* the filtered world, in this case, by saving all the filtered video parts into video files for further examination.

Users now spend less time watching videos as they only need to watch the relevant ones, with objects of interest highlighted. Analyzing videos together with geospatial metadata, S-Flow provides a declarative interface where users need not specify "how" to find the video snippets of interest; instead, they only need to describe "what" their video snippets of interest look like, and Spatialyze will find such video snippets accordingly.

7.3 Lessons Learned

Aside from users' challenges in analyzing and storing geospatial videos, we also faced several challenges while designing and building LightDB and Spatialyze to support such tasks.

Building Infrastructure. First of all, geospatial video analytics as a task is a combination of two different but related analytic tasks, analyzing videos and analyzing geospatial data. Designing a system for each of them already has its own set of challenges, let alone integrating them together. We chose to design and build Spatialyze around tracked objects from videos because tracked objects also have spatial-temporal properties that can be joined with *Geographic Constructs* temporally and *Cameras* spatial-temporally. With the tracked objects as a new data type, we build our *Movable Objects* query engine on top of an existing geospatial query engine [73]. Still, we spent a countless amount of time in manually designing efficient join queries when users query for multiple objects. Our development time for building this infrastructure can be cut short with automated synthesizing (Sec. 5.2) of such queries, and thus, reduces the whole development time of Spatialyze.

API Design. After making a decision on designing a system around object tracks, the next challenge is designing a programming interface for the user. This programming interface serves as an abstraction for users to interact with the tracked objects and their geospatial data. LightDB proposed its own light-field centric data model and API, and Spatialyze uses the *World* metaphor inspired by VisualWorldDB [26] and Apperception [22], where *World* as a simple virtual environment that contains object tracks from videos and cameras that shot the videos. It took us 3 years (and at least one PhD worth of time) to design these different interfaces and prototype them for related use cases. Unfortunately, we are still not done yet, as new use cases keep appearing and they require API modifications. Some of the techniques described in Sec. 4 will be useful in reducing the amount of time needed to explore different API designs.

Users Interaction. Even with the simplified interface presented in Spatialyze, users still need to have basic programming and data processing skills. As future work, since we have S-Flow as a declarative DSL for interacting with geospatial video data, we can apply the programming-by-example techniques mentioned in Sec. 4.1 to allow users to search for videos given a set of similar video examples as input. Then, the system can automatically synthesize a S-Flow program (Sec. 5.2) that searches for the videos of users' interest based on the given example videos.

8 FUTURE WORK

We now highlight further challenges in generating data system implementations, and other application domains beyond those described earlier where techniques discussed in this paper can be applied.

Extending Data Structure Search. Much of the early work in verified lifting has focused on synthesizing *logic* in new domains that compute identical results to a reference implementation. But projects like Katara have demonstrated that the same overarching goals are also applicable to stateful programs that other logic can interact with over time. Our work on synthesizing CRDTs shows that the correctness conditions in such domains can be encoded for automated verification, and bounded verification enables rapid pruning of the much larger search space of internal data representations. In future work, we believe there are ample opportunities to apply similar approaches in other stateful domains, such as synthesizing parallel data structures and distributed actor programs.

Applying Equivalence Saturation. The power of verified lifting and program synthesis comes from its ability to generate code that applies novel techniques that do not exist in legacy software. But classic enumerative techniques for synthesizing this code can be expensive and produce results that can be difficult to explain. An exciting alternative for automated code generation is to apply equality saturation through e-graphs [38, 39, 63] to transform the source program through local rewrite rules. These rules can be independently verified and composed to enable complex transformations. For example, our early work shows how this technique can discover the classic semi-naive join algorithm for Datalog by composing primitive rewrites [31]. We believe that combining e-graphs with classic synthesis techniques will enable a rich space of efficient and easy-to-verify compilers.

Leveraging Generative Models. Recent advances in generative models (e.g., large language models) have made them attractive as code generators. However, such models typically do not have any means to validate the generated implementation against the input, and furthermore, the generated implementation can be difficult to modify or refine if it does not fit the user's intentions. It will be interesting to design techniques that enable users to manually interact with the generated implementation, or design user-aided algorithms to modify the generated implementation in case of error.

Verified Lifting for Distributed Systems. Developing systems that efficiently make use of elastic cloud compute is a challenging task. Declarative approaches that leverage the perspective of databases, a la Dedalus [5], offer a unique opportunity to transform the process of distributing a program into a *query optimization problem*. Declarative languages make properties of the program such as monotonicity easy to identify and leverage for optimizations. But this comes at a cost: it is difficult for developers to rewrite their existing systems into such languages. In the Hydro project [13], we are applying verified lifting to tackle this problem by *lifting* existing code into declarative languages. For example, we are developing techniques to automatically replicate the CRDTs synthesized by Katara and analyze queries over them to determine the consistency levels which can be guaranteed [32]. In future work, we hope to expand this to more general-purpose programs, leveraging techniques like CRDTs to automatically distribute stateful tasks.

9 CONCLUSION

With advances in machine learning and other areas of automated programming, more and more portions of data systems will inevitably be auto-generated in the near future. We argue that designing techniques for automation requires one to be aware of three key aspects: means for users to express their intentions, algorithms for code generation, and mechanisms to validate the generated code. We illustrate these three aspects using our prior work, along with open research challenges towards fully-automated generation of data systems.

ACKNOWLEDGEMENT

This work is supported in part by the National Science Foundation through grants IIS-1955488, IIS-2027575, DOE awards DE-SC0016260 and DE-SC0021982, ARO award W911NF2110339, ONR award N00014-21-1-2724, and generous gifts from our industry partners.

We would like to express our immense gratitude towards the VLDB Awards Committee for their recognition of our work. Alvin in particular is humbled to have the honor to learn from the many students and mentors who have made the work described in this paper possible.

REFERENCES

[1] Azza Abouzied, Joseph M. Hellerstein, and Avi Silberschatz. 2012. DataPlay: interactive tweaking and example-driven correction of graphical database queries. In *The 25th Annual ACM Symposium on User Interface Software and Technology, UIST '12, Cambridge, MA, USA, October 7-10, 2012*, Rob Miller, Hrvoje Benko, and Celine Latulipe (Eds.). ACM, 207–218.

[2] Maaz Bin Safeer Ahmad and Alvin Cheung. 2017. Optimizing Data-Intensive Applications Automatically By Leveraging Parallel Data Processing Frameworks. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 1675–1678.

[3] Maaz Bin Safeer Ahmad and Alvin Cheung. 2018. Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. 1205–1220.

[4] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 1009–1024.

[5] Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. 2011. Dedalus: Datalog in Time and Space. In *Datalog Reloaded*, Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 262–281.

[6] Apache Flink team. [n.d.]. Apache Flink Project. https://flink.apache.org.

[7] Apache Hadoop 2023. http://hadoop.apache.org. Accessed on: 2023-08-08.

[8] Apache Spark 2023. https://spark.apache.org. Accessed on: 2023-08-08.

[9] Fabrice Bellard. [n.d.]. FFmpeg. https://ffmpeg.org.

[10] Sahil Bhatia, Sumer Kohli, Sanjit A. Seshia, and Alvin Cheung. 2023. Building Code Transpilers for Domain-Specific Languages Using Program Synthesis (Experience Paper). In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States (LIPIcs)*, Vol. 263. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 38:1–38:30.

[11] Mikel Broström. 2022. Real-time multi-camera multi-object tracker using YOLOv5 and StrongSORT with OSNet. https://github.com/mikel-brostrom/Yolov5_StrongSORT_OSNet.

[12] Chip Brown. 2017. Bringing pixels front and center in VR video. https://www.blog.google/products/google-vr/bringing-pixels-front-and-center-vr-video.

[13] Alvin Cheung, Natacha Crooks, Joseph M Hellerstein, and Mae Milano. 2021. New directions in cloud programming. *The Conference on Innovative Data Systems Research (CIDR)*, 14.

[14] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. In *PLDI*. ACM, 3–14.

[15] Shumo Chu, Daniel Li, Chenglong Wang, Alvin Cheung, and Dan Suciu. 2017. Demonstration of the Cosette Automated SQL Prover. In *SIGMOD Conference*. ACM, 1591–1594.

[16] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries. *PVLDB* 11, 11 (2018), 1482–1495.

[17] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *CIDR*. www.cidrdb.org.

[18] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: proving query rewrites with univalent SQL semantics. In *PLDI*. ACM, 510–524.

[19] Maureen Daum, Brandon Haynes, Dong He, Amrita Mazumdar, and Magdalena Balazinska. 2021. TASM: A Tile-Based Storage Manager for Video Analytics. In *ICDE*. 1775–1786.

[20] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113.

[21] Django Software Foundation. [n.d.]. *Django*. https://djangoproject.com

[22] Yongming Ge, Vanessa Lin, Maureen Daum, Brandon Haynes, Alvin Cheung, and Magdalena Balazinska. 2021. Demonstration of Apperception: A Database Management System for Geospatial Video Data. *Proc. VLDB Endow.* 14, 12 (jul 2021), 2767–2770.

[23] Github. [n.d.]. Github Copilot. https://github.com/features/copilot.

[24] Clément Godard, Oisin Mac Aodha, Michael Firman, and Gabriel Brostow. 2019. Digging Into Self-Supervised Monocular Depth Estimation. arXiv:1806.01260 [cs.CV]

[25] Todd J. Green, Gregory Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *PODS*. 31–40.

[26] Brandon Haynes, Maureen Daum, Amrita Mazumdar, Magdalena Balazinska, Alvin Cheung, and Luis Ceze. 2020. VisualWorldDB: A DBMS for the Visual World. In *Conference on Innovative Data Systems Research*.

[27] Brandon Haynes, Amrita Mazumdar, Armin Alaghi, Magdalena Balazinska, Luis Ceze, and Alvin Cheung. 2018. LightDB: A DBMS for Virtual Reality Video. *VLDB* 11, 10 (2018), 1192–1205.

[28] Brandon Haynes, Artem Minyaylov, Magdalena Balazinska, Luis Ceze, and Alvin Cheung. 2017. VisualCloud Demonstration: A DBMS for Virtual Reality. In *SIGMOD*. 1615–1618.

[29] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified lifting of stencil computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 711–726.

[30] Chanwut Kittivorawong, Yongming Ge, Yousef Helal, and Alvin Cheung. 2023. Spatialyze: A Geospatial Video Analytics System with Spatial-Aware Optimizations. arXiv:2308.03276 [cs.DB]

[31] Shadaj Laddad, Conor Power, Tyler Hou, Alvin Cheung, and Joseph M. Hellerstein. 2023. Optimizing Stateful Dataflow with Local Rewrites. arXiv:2306.10585 [cs.PL]

[32] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, Natacha Crooks, and Joseph M. Hellerstein. 2022. Keep CALM and CRDT On. *Proc. VLDB Endow.* 16, 4 (dec 2022), 856–863. https://doi.org/10.14778/3574245.3574268

[33] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M. Hellerstein. 2022. Katara: Synthesizing CRDTs with Verified Lifting. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 173 (oct 2022), 29 pages. https://doi.org/10.1145/3563336

[34] Xiaoxuan Liu, Shuxian Wang, Mengzhu Sun, Sicheng Pan, Ge Li, Siddharth Jha, Cong Yan, Junwen Yang, Shan Lu, and Alvin Cheung. 2023. Leveraging Application Data Constraints to Optimize Database-Backed Web Applications. *Proc. VLDB Endow.* 16, 6 (apr 2023), 1208–1221.

[35] Tim Milliron, Chrissy Szczupak, and Orin Green. 2017. Hallelujah: The World's First Lytro VR Experience. In *SIGGRAPH*. Article 7, 2 pages.

[36] Kiran M. Misra, C. Andrew Segall, Michael Horowitz, Shilin Xu, Arild Fuldseth, and Minhua Zhou. 2013. An Overview of Tiles in HEVC. *Journal of Selected Topics in Signal Processing* 7, 6 (2013), 969–977.

[37] Tom M. Mitchell. 1982. Generalization as Search. *Artif. Intell.* 18, 2 (1982), 203–226.

[38] Charles Gregory Nelson. 1980. *Techniques for Program Verification*. Ph.D. Dissertation. Stanford, CA, USA. AAI8011683.

[39] Robert Nieuwenhuis and Albert Oliveras. 2005. Proof-Producing Congruence Closure. In *Proceedings of the 16th International Conference on Term Rewriting and Applications* (Nara, Japan) *(RTA'05)*. Springer-Verlag, Berlin, Heidelberg, 453–468. https://doi.org/10.1007/978-3-540-32033-3_33

[40] Ruby on Rails Team. [n.d.]. Active Record Migrations. https://guides.rubyonrails.org/active_record_migrations.html.

[41] OpenAI. [n.d.]. ChatGPT. https://chat.openai.com/.

[42] OpenCV. [n.d.]. Open Source Computer Vision Library. https://opencv.org.

[43] Pandas Development Community. [n.d.]. Python Data Analysis Library. http://pandas.pydata.org.

[44] Spiros Papadimitriou and Jimeng Sun. 2008. DisCo: Distributed Co-clustering with Map-Reduce: A Case Study Towards Petabyte-Scale End-to-End Mining. In *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining (ICDM '08)*. IEEE Computer Society, Washington, DC, USA, 512–521.

[45] Ruby on Rails Team. [n.d.]. *Ruby on Rails*. https://rubyonrails.org/

[46] David Salomon. 2011. *The Computer Graphics Manual*. Springer.

[47] Mingwei Samuel, Cong Yan, and Alvin Cheung. 2020. Demonstration of Chestnut: An In-memory Data Layout Designer for Database Applications. In *ACM SIGMOD Conference*.

[48] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System *(SIGMOD '79)*. 23–34.

[49] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '79)*. ACM, New York, NY, USA, 23–34.

[50] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. 386–400.

[51] Anirudh Sivaraman, Mihai Budiu, Alvin Cheung, Changhoon Kim, Steve Licking, George Varghese, Hari Balakrishnan, Mohammad Alizadeh, and Nick McKeown. 2015. Packet Transactions: A Programming Model for Data-Plane Algorithms at Hardware Speed. *CoRR* abs/1512.05023 (2015).

[52] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*. 404–415.

[53] Michael Stonebraker, Lawrence A. Rowe, and Michael Hirohama. 1990. The Implementation of Postgres. *IEEE Trans. Knowl. Data Eng.* 2, 1 (1990), 125–142.

[54] SQLite Team. [n.d.]. History Of SQLite Releases. https://www.sqlite.org/chronology.html.

[55] Iain Thomson. 2011. Google opens BigQuery for cloud analytics. https://www.theregister.com/2011/11/14/google_bigquery_cloud_analytics/.

[56] Tidyverse [n.d.]. Tidyverse: R packages for data science. https://www.tidyverse.org.

[57] Emina Torlak and Rastislav Bodik. 2014. A lightweight symbolic virtual machine for solver-aided host languages. *ACM SIGPLAN Notices* 49, 6 (2014), 530–541.

[58] Boris Trakhtenbrot. 1950. Impossibility of an algorithm for the decision problem in finite classes. *D. Akad. Nauk USSR* 70, 1 (1950), 569–572.

[59] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017. Interactive Query Synthesis from Input-Output Examples. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 1631–1634.

[60] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *PLDI*. 452–466.

[61] Chenglong Wang, Yu Feng, Rastislav Bodík, Alvin Cheung, and Isil Dillig. 2020. Visualization by example. *Proc. ACM Program. Lang.* 4, POPL (2020), 49:1–49:28.

[62] Chenglong Wang, Yu Feng, Rastislav Bodík, Isil Dillig, Alvin Cheung, and Amy J. Ko. 2021. Falx: Synthesis-Powered Visualization Authoring. In *CHI '21: CHI Conference on Human Factors in Computing Systems, Virtual Event / Yokohama, Japan, May 8-13, 2021*. ACM, 106:1–106:15.

[63] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (jan 2021), 29 pages. https://doi.org/10.1145/3434304

[64] Cong Yan and Alvin Cheung. 2019. Generating Application-specific Data Layouts for In-memory Databases. *PVLDB* 12, 11 (2019), 1513–1525.

[65] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. 2017. Understanding Database Performance Inefficiencies in Real-world Web Applications. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*. 1299–1308.

[66] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. 2020. View-Driven Optimization of Database-Backed Web Applications.. In *CIDR*.

[67] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. 2018. How *not* to structure your database-backed web applications: a study of performance bugs in the wild. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018*. 800–810.

[68] Junwen Yang, Cong Yan, Pranav Subramaniam, Shan Lu, and Alvin Cheung. 2018. PowerStation: automatically detecting and fixing inefficiencies of database-backed web applications in IDE. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. 884–887.

[69] Junwen Yang, Cong Yan, Chengcheng Wan, Shan Lu, and Alvin Cheung. 2019. View-centric performance optimization for database-backed web applications. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 994–1004.

[70] Luke S. Zettlemoyer and Michael Collins. 2005. Learning to Map Sentences to Logical Form: Structured Classification with Probabilistic Categorial Grammars. In *UAI '05, Proceedings of the 21st Conference in Uncertainty in Artificial Intelligence, Edinburgh, Scotland, July 26-29, 2005*. AUAI Press, 658–666.

[71] Zhengyou Zhang. 2014. *Camera Parameters (Intrinsic, Extrinsic)*. Springer US, Boston, MA, 81–85. https://doi.org/10.1007/978-0-387-31439-6_152

[72] Xiangyu Zhou, Rastislav Bodík, Alvin Cheung, and Chenglong Wang. 2017. Synthesizing analytical SQL queries from computation demonstration. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. ACM, 168–182.

[73] Esteban Zimányi, Mahmoud Sakr, and Arthur Lesuisse. 2020. MobilityDB: A Mobility Database Based on PostgreSQL and PostGIS. *ACM Trans. Database Syst.* 45, 4, Article 19 (dec 2020), 42 pages. https://doi.org/10.1145/3406534

[74] Moshé M. Zloof. 1975. Query-by-Example: the Invocation and Definition of Tables and Forms. In *Proceedings of the International Conference on Very Large Data Bases, September 22-24, 1975, Framingham, Massachusetts, USA*. ACM, 1–24.