Adaptive Schema Databases [*]

William Spoth[b], Bahareh Sadat Arab[i], Eric S. Chan[o], Dieter Gawlick[o],
Adel Ghoneimy[o], Boris Glavic[i], Beda Hammerschmidt[o], Oliver Kennedy[b],
Seokki Lee[i], Zhen Hua Liu[o], Xing Niu[i], Ying Yang[b]
b: University at Buffalo   i: Illinois Inst. Tech.   o: Oracle
{wmspoth|okennedy|yyang25}@buffalo.edu
{barab|slee195|xniu7}@hawk.iit.edu   bglavic@iit.edu
{eric.s.chan|dieter.gawlick|adel.ghoneimy|beda.hammerschmidt|zhen.liu}@oracle.com

ABSTRACT

The rigid schemas of classical relational databases help users in specifying queries and inform the storage organization of data. However, the advantages of schemas come at a high upfront cost through schema and ETL process design. In this work, we propose a new paradigm where the database system takes a more active role in schema development and data integration. We refer to this approach as *adaptive schema databases* (*ASDs*). An ASD ingests semi-structured or unstructured data directly using a pluggable combination of extraction and data integration techniques. Over time it discovers and adapts schemas for the ingested data using information provided by data integration and information extraction techniques, as well as from queries and user-feedback. In contrast to relational databases, ASDs maintain multiple *schema workspaces* that represent individualized views over the data, which are fine-tuned to the needs of a particular user or group of users. A novel aspect of ASDs is that probabilistic database techniques are used to encode ambiguity in automatically generated data extraction workflows and in generated schemas. ASDs can provide users with context-dependent feedback on the quality of a schema, both in terms of its ability to satisfy a user's queries, and the quality of the resulting answers. We outline our vision for ASDs, and present a proof of concept implementation as part of the Mimir probabilistic data curation system.

1. INTRODUCTION

Classical relational systems rely on schema-on-load, requiring analysts to design a schema upfront before posing any queries. The schema of a relational database serves both a navigational purpose (it exposes the structure of data for querying) as well as an organizational purpose (it informs storage layout of data). If raw data is available in unstructured or semi-structured form, then an ETL (i.e., Extract, Transform, and Load) process needs to be designed to translate the input data into relational form. Thus, classical relational systems require a lot of upfront investment. This makes them unattractive when upfront costs cannot be amortized, such as in workloads with rapidly evolving data or where individual elements of a schema are queried infrequently. Furthermore, in settings like data exploration, schema design simply takes too long to be practical.

Schema-on-query is an alternative approach popularized by NoSQL and Big Data systems that avoids the upfront investment in schema design by performing data extraction and integration at query-time. Using this approach to query semi-structured and unstructured data, we have to perform data integration tasks such as natural language processing (NLP), entity resolution, and schema matching on a per-query basis. Although it allows data to be queried immediately, this approach sacrifices the navigational and performance benefits of a schema. Furthermore, schema-on-query incentivizes task-specific curation efforts, leading to a proliferation of individualized lower-quality copies of data and to reduced productivity.

One significant benefit of schema-on-query is that queries often only access a subset of all available data. Thus, to answer a specific query, it may be sufficient to limit integration and extraction to only relevant parts of the data. Furthermore, there may be multiple "correct" relational representations of semi-structured data and what constitutes a correct schema may be highly application dependent. This implies that imposing a single flat relational schema will lead to schemas that are the lowest common denominator of the entire workload and not well-suited for *any* of the workload's queries. Consider a dataset with tweets and re-tweets. Some queries over a tweet relation may want to consider re-tweets as tweets while others may prefer to ignore them.

In this work, we propose ***adaptive schema databases*** (*ASDs*), a new paradigm that addresses the shortcomings of both the classical relational and the Big Data approaches mentioned above. ASDs enjoy the navigational and organizational benefits of a schema without incurring the upfront investment in schema and ETL process development. This is achieved by automating schema inference, information extraction, and integration to reduce the load on the user. Furthermore, instead of enforcing one global schema, ASDs build and adapt idiosyncratic schemas that are specialized to users' needs.

We propose the probabilistic framework shown in Figure 1 as a reference architecture for ASDs. When unstruc-

[*]Authors Listed in Alphabetical Order

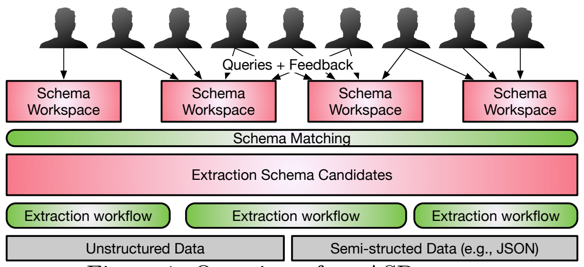

Figure 1: Overview of an ASD system

tured or semi-structured data are loaded into an ASD, this framework applies a sequence of data extraction and integration components that we refer to as an **extraction workflow** to compute possible relational schemas for this data. Any existing techniques for schema extraction or information integration can be used as long as they can expose ambiguity in a probabilistic form. For example, an entity resolution algorithm might identify two possible instances representing the same entity. Classically, the algorithm would include heuristics that resolve this uncertainty and allow it to produce a single deterministic output. In contrast, our approach requires that extraction workflow stages produce non-deterministic, probabilistic outputs instead of using heuristics. The final result of such an **extraction workflow** is a **set of candidate schemas** and a probability distribution describing the likelihood of each of these schemas. In ASDs, users create **schema workspaces** that represent individual views over the schema candidates created by the extraction workflow. The schema of a workspace is created incrementally based on queries asked by a user of the workspace. Outputs from the extraction workflow are dynamically imported into the workspace as they are used, or users may suggest new relations and attributes not readily available to the database. In the latter case, the ASD will apply schema matching and other data integration methods to determine how the new schema elements relate to the elements in the candidate schemas, and attempt to synthesize new relations or attributes to match. Similar to extraction workflows, the result of this step is probabilistic. Based on these probabilities and feedback provided by users through queries, ASDs can incrementally modify the extraction workflow and schema workspaces to correct errors, to improve their quality, to adapt to changing requirements, and to evolve schemas based on updates to input datasets. The use of feedback is made possible based on our previous work on probabilistic curation operators [35] and provenance [2]. By modelling schemas as views over a non-relational input dataset, we decouple data representation from content. Thus, we gain flexibility in **storage organization** — for a given schema we may choose not to materialize anything, we may fully materialize the schema, or materialize selectively based on access patterns.

Concretely, this paper makes the following contributions:

- We introduce our vision of ASDs, which enable access to unstructured and semi-structured data through personalized relational schemas.
- We show how ASDs leverage information extraction and data integration to automatically infer and adapt schemas based on evidence provided by these components, by queries, and through user feedback.
- We show how ASDs enable adaptive task-specific "personalized schemas" through schema workspaces which are probabilistic relational views over semistructured or unstructured input datasets.
- We illustrate how ASDs communicate potential sources of error and low-quality data, and how this communication enables analysts to provide feedback.
- We present a proof of concept implementation of ASDs based on the *Mimir* [29] data curation system.
- We demonstrate through experiments that the instrumentation required to embed information extraction into an ASD has minimal overhead.

2. EXTRACTION AND DISCOVERY

An ASD allows users to pose relational queries over the content of semi-structured and unstructured datasets. We call the steps taken to transform an input dataset into relational form an **extraction workflow**. For example, one possible extraction workflow is to first employ *natural language processing* (NLP) to extract semi-structured data (e.g., RDF triples) from an unstructured input, and then shred the semi-structured data into a relational form. The user can then ask queries against the resultant relational dataset. Such a workflow frequently relies on heuristics to create seemingly deterministic outputs, obscuring the possibility that the heuristics may choose incorrectly. In an ASD, one or more modular information extraction components instead produce a *set* of possible ways to shred the raw data with associated probabilities. This is achieved by exposing ambiguity arising in the components of an extraction workflow. Any NLP, information retrieval, and data integration algorithm may be used as an information extraction component, as long as the ambiguity in its heuristic choices can be exposed. The set of schema candidates are then used to seed the development of schemas individualized for a particular purpose and/or user. The ASD's goal is to figure out which of these candidates is the correct one for the analyst's current requirements, to communicate any potential sources of error, and to adapt itself as those requirements change.

Extraction Schema Candidates. When a collection of unstructured or semi-structured datasets D is loaded into an ASD, then information extraction and integration techniques are automatically applied to extract relational content and compute candidate schemas for the extracted information. The choice of techniques is based on the input data type (JSON, CSV, natural language text, etc...). We associate with this data a **schema candidate set** $\mathcal{C}_{ext} = (\mathbf{S_{ext}}, P_{ext})$ where $\mathbf{S_{ext}}$ is a set of candidate schemas and P_{ext} is a probability distribution over these schemas. We use S_{max} referred to as the **best guess schema** to denote $\arg\max_{S \in \mathbf{S_{ext}}}(P(S))$, i.e., the most likely schema from the set of candidate schemas. Similar data models have been studied extensively in probabilistic databases [16], allowing us to adapt existing work on probabilistic query processing [31], while still supporting a variety of techniques for information extraction [12], natural language processing [13], data integration [14, 15, 21], and more.

EXAMPLE 1. *As a running example throughout the paper, consider a JSON document (a fragment is shown below) that stores a college's enrollment. Assume that for every graduate student we store name and degree, but only for some students there is a record of the number of credits achieved so far. For undergraduates we only store a name, although several undergraduates were accidentally stored with a de-*

Student

Name
Alice
Bob
Carol
Dave

(a) P = 0.19

Student

Name	Deg
Alice	PhD
Bob	MS
Carol	(null)
Dave	U

(b) P = 0.27

Undergrad

Name
Carol
Dave

Grad

Name
Alice
Bob

(c) P = 0.22

Undergrad

Name	Deg
Carol	(null)
Dave	U

Grad

Name	Deg	Credits
Alice	PhD	10
Bob	MS	(null)

(d) P = 0.32

Figure 2: Extracted Schema Candidate Set and Data

gree. A semi-structured to relational mapper may extract schema candidates as shown in Figure 2.

```
{"grad":{"students":[
  {name:"Alice",deg:"PhD",credits:"10"},
  {name:"Bob",deg:"MS"}, ...]},
 "undergrad":{"students":[
  {name:"Carol"},{name:"Dave",deg:"U"}, ...]}}
```

Querying Extracted Data. We would like to expose to users an initial schema that allows D to be queried (i.e., the best guess schema S_{max}), while at the same time acknowledging that this schema may be inappropriate for the analyst, incorrect for her current task, or simply outright wrong. Manifestations of extraction errors appear in three forms: (1) A query incompatible with S_{max}, (2) An update with data that violates S_{max}, or (3) An extraction error resulting in the wrong data being presented to the user. The first two errors are overt and, thus easy to detect automatically. In both cases, the primary challenge is to help the user to determine whether the operation was correct, and if necessary, to repair the schema accordingly. Here, the distribution P_{ext} serves as a metric for schema suggestions. Given a query (resp., update or insert) Q, the goal is to compute $\arg\max_{S \in \mathbf{S_{ext}} \wedge S \vDash Q}(P(S))$, where the $S \vDash Q$ denotes compatibility between schema and query, i.e., the schema contains the relations and attributes postulated by the query. While S_{max} has the highest probability of all schema candidates in $\mathbf{S_{ext}}$, that does not imply that it has the highest probability with respect to the schema elements mentioned in the query. Thus, we use S_{max} as a generic best guess to enable the user to express queries at first, but then adapt the best schema over time. Note that the personalized schemas we introduce in the next section even allow queries to postulate new relations and attributes.

Detecting extraction errors is harder and typically only possible once issues with the returned query result are discovered. Rather, such errors are most often detected as a result of inconsistencies observed while the analyst explores her data. Thus, the goal of an ASD is to make the process of detecting and repairing extraction errors as seamless as possible. Our approach is based on pay-as-you-go or on-demand approaches to curation [21, 29, 34], and is the focus of Sections 4 and 6 below.

3. ADAPTIVE, PERSONALIZED SCHEMAS

An ASD maintains a set of **schema workspaces** $\mathcal{W} = \{W_1, \ldots, W_n\}$. Each workspace W_i has an associated mapped context $\mathcal{C}_i = (S_i, \mathcal{M}_i, P_i)$ where S_i is a schema, $\mathcal{M}_i$ is a set of possible schema matchings [4], each between the elements of S_i and one $S \in \mathbf{S_{ext}}$, and P_i assigns probabilities to these matches. In the future we will lift the restriction to schema matches and allow the relationship between the extracted schema and the schema of a workspace to be more complex than that (e.g., expressed as a schema mapping [15]). Users may maintain their own personal schema workspace or share workspaces within a group of users that have common interests (e.g., a business analyst workspace for the sales data of a company). We plan to provide version control style features for the schemas of workspaces including access to data through past schema versions and importing of schema elements from one workspace into another. Recent work on schema evolution [9] and schema versioning demonstrates that it is possible to maintain multiple versions of schemas in parallel where data is only stored according to one of these schemas. Specifically, we plan to extend our own work on flexible versioning of data [30] and flexible schema extensions for SQL [24].

Importing Schema Elements. Initially, the schema of a workspace is created empty. When posing a query over an extracted dataset, the user can refer to elements from schema S_{max}, schema S_i, or new relations and attributes that do not occur in either. References of the first two types are resolved by applying the extraction workflow to compute the instances for these schema elements (and potentially mapping the data based on the matches in $\mathcal{M}_i$). If a query references schema elements from S_{max}, then these schema elements are added to the current workspace schema plus one-to-one matches with probability 1 between these elements in S_{max} and S_i are added to the matching $\mathcal{M}_i$.

A query may also remain agnostic to the specific schema elements it requires, and instead declaratively provide query goals in terms of higher-order logical primitives. That is, the user postulates the existence of schema elements (which is expressible in second-order logic) and part of their structure (e.g., attributes that are referenced) without having to fully qualify them. This constrains the schema workspace, but still allows the workspace to adapt and evolve over time. A concrete example of this idea can be found in the flexible schema data extensions for SQL [24] (FSD-SQL). FSD-SQL allows query authors to remain agnostic to the exact physical structure of inter-attribute relationships, automatically adapting the query structure as needed.

Example 2. *Consider the following FSD-SQL query, which returns all students in the PhD program:*

```
SELECT name FROM Grad
WHERE json_exists(deg == 'PhD')
```

The data initially shows that each student has only one degree — The best schema is one in which there is a 1-to-1 mapping between student and degree and `deg` *is an element of the Grad relation. Thus, the above query is equivalent to the classical SQL query:*

```
SELECT name FROM Grad WHERE deg = 'PhD'
```

However, let's say that the data also contains the following student, registered for both programs.

```
{ name: "Eve", deg: ["MS", "PhD"] }
```

With this new data, it may be appropriate to represent the degree field by a many-to-one relationship, and the equivalent query becomes a more complex primary-key to foreign-key join:

```
SELECT g.name FROM Grad g WHERE EXISTS (
  SELECT * FROM GradDeg d
  WHERE g.id = d.id AND d.deg = 'PhD' )
```

A language construct like `json_exists` remains agnostic to which underlying representation is used, creating a query that is more resilient to schema evolution and supporting a broader range of possible schemas.

Probabilistic Semantics of ASD Queries. Note that ASD queries are inherently probabilistic, as the result varies depending on the distribution of possible extractions. However, we do not have to overwhelm the user with full probabilistic query semantics. Instead, we apply the approach from [29,35] to return a deterministic best guess result based on S_i and expose uncertainty through user interface cues [23] and through human-readable explanations generated on-demand.

Example 3. *Continuing with our running example, assume a user operating in workspace W_1 would like to retrieve all the names of students based on the enrollment JSON document. One option the user can take is to query the relations exposed by the best guess schema S_{max}. For instance, one way to express this query over the schema in Figure 2 is:*

```
SELECT name FROM Undergrad UNION
SELECT name FROM Grad
```

To process this query, the ASD would run the extraction workflow to create the relational content of the Undergrad *and* Grad *relations (it would be sufficient to create the projections of these relations on* name *only). The query is then evaluated over these extracted data. As a side-effect, by accessing these schema elements, the user declares interest in them and they are added to the schema workspace. Note that only accessed attributes are added to the workspace. If the workspace's schema was empty before, the resulting schema would be $S_1 = \{\mathsf{Undergrad(name)}, \mathsf{Grad(name)}\}$. Additionally, in $\mathcal{M}_1$ these elements are matched with their counterpart in S_{max}. If afterwards the user retrieves the degree of a graduate student then the* Grad *relation's schema would become* (name, deg)*.*

Declaring New Schema Elements. So far we have only discussed the case where a query refers to existing schema elements (either in the user schema or the extracted schema). If a query uses schema elements that are so far unknown, then this is interpreted as a request by the user to add these schema elements to the schema workspace. It is the responsibility of the ASD to determine how schema elements in the extracted schema are related to these new elements. Any existing schema matching (and mapping discovery) approach could be used for this purpose. For instance, we could complement schema matching with schema mapping discovery [7, 33] to establish more expressive relationships between schema elements. Based on such matches we can then rewrite the user's query to access only relations from the extracted schema using query rewriting with views (a common technique from virtual data integration [17]) or materialize its content using data exchange techniques [15]. Again we take a probabilistic view by storing all possible matches with associated probabilities and choosing the matches with the highest probability for the given query.

Example 4. *Assume that a user would like to find names of students without having to figure out which relations in S_{max} store student information. A user may ask:*

```
SELECT name FROM Student
```

Since relation Student *occurs in neither S_1 nor S_{max}, the ASD would run a schema matcher to determine which elements from S_{max} match with* Student *and its attribute* name*, for instance by probabilistically combining the* name *attribute of* Grad *and* Undergrad *as in the query from Example 3.*

In the example above, three Student(name) relations could reasonably be extracted from the dataset: One with just graduate students, one with just undergraduates, and one with both. Although it may be possible to heuristically select one of the available extraction options, it is virtually impossible for a single heuristic to cover all use cases. Instead, ASDs use heuristics only as a starting point for schema definitions. An ASD decouples its information extraction heuristics from the space of possible extractions that could be emitted. In the next section, we present how these uncertain heuristic choices can be validated or corrected as needed in a pay-as-you-go manner [21, 29]. Note that we can use any existing schema matching algorithm to create schema matching $\mathcal{M}_i$ as long as it can be modified to expose probabilities. As we have demonstrated in previous work this assumption is reasonable — Mimir [29] already supports a simple probabilistic schema matching operator.

4. EXPLANATIONS AND FEEDBACK

Allowing multiple schemas to co-exist simultaneously opens up opportunities for ambiguity to enter into an analyst's interaction with the database. To minimize confusion, it is critical that the analyst be given insight into how the ASD is presently interpreting the data.

Our approach to communicating the ASD's decisions leverages our previous work on: (1) explaining results of probabilistic queries and data curation operators in Mimir [23,29, 35], and (2) provenance frameworks for database queries [2, 30]. We discuss the details of these systems along with our proof of concept implementation in Section 6. An ASD must be able to: (1) Warn the analyst when ambiguity could impact her interaction, (2) Explain the ambiguity, (3) Evaluate the magnitude of the ambiguity's potential impact, and (4) Assist the analyst in resolving the ambiguity. In this section, we explore how an ASD can achieve each of these goals in the context of three forms of interaction between the ASD and the outside world: Schema, Data, and Update.

Schema Interactions. Schema interactions are those that take place between the analyst and the ASD as she composes queries and explores the relations available in her present workspace. Recall that referencing a relation that does not exist in the workspace and extraction schema S_{max} does not necessarily constitute a problem since this triggers the ASD to add this relation to the workspace and figure out which relations in $\mathbf{S_{ext}}$ it could be matched with. However, it may be the case that no feasible match can be found. This either means that the user is asking for data that is simply not present in the dataset D or that errors in the extraction workflow or matching caused the ASD to miss the correct match. To explain the failure, we may provide the user with a summary of why matching with $\mathbf{S_{ext}}$ failed. For example, there are no relations with similar names in $\mathbf{S_{ext}}$.

Data Interactions. Data interactions happen when the ASD produces query results. Here, ambiguity can typically not be detected by static analysis and is often hidden behind multiple layers of aggregation and projection. For example, the query in Example 4 can have three distinct responses, depending on which relations from the extraction schema are matched against Student relation in the workspace that the analyst is currently using. At this level un-intrusive interface cues [23] are critical for alerting the analyst to the possibility that the results she is seeing may not be what she had in mind. The Mimir system uses a form of attribute granularity provenance [29, 35] to track the effects of sources of ambiguity on the output of queries. In addition to flagging potentially ambiguous query result cells and rows (e.g., attributes computed based on a schema match that is uncertain), Mimir allows users to explore the effects of ambiguity through both human-readable explanations of their causes and statistical precision measures like standard deviation and confidence scores. Linking results to sources of ambiguity also makes it easier for the analyst to provide feedback that resolves the ambiguity. In Section 6 we show how we leverage Mimir to streamline data interactions in our prototype ASD.

Update Interactions. Finally, update interactions take place when the ASD receives new data, or changes to existing data. In comparison to the other two cases, there may not be an analyst directly involved in an update interaction like a nightly bulk data import. Thus, the ASD must be able to communicate the ambiguity arising from update interactions to analysts indirectly. The main problem with updates is that the extraction schema candidates $\mathbf{S_{ext}}$ and its probability distribution P_{ext} may get out of sync with the data it is describing. For example, when extracting JSON schemas, Oracle's DataGuides [25] transparently upgrade the type of a primitive-valued object to a singleton array if necessary for compatibility. Workspaces that have already imported mappings to the object expect it to be a primitive value.

However, rather than blocking an insertion or update which does not conform with the extraction schema outright, the ASD will represent schema mismatches as missing values when data is accessed through the out of sync schema. Alternatively, we can attempt to resolve data errors with a probabilistic repair. For example, an array of primitive values can be coerced into a primitive value by the probabilistic repair-key operation [1], allowing us to once again leverage probabilistic data curation systems like Mimir for explanations and feedback. However, because of the possibility that the schema is incorrect, feedback on these curation steps includes an additional two options for the analyst. In addition to repairing the potential data error, the analyst can choose to upgrade her workspace's schema to match the changes, or may choose to checkpoint her workspace and ignore new updates.

The ASD can also adjust the information extractor to adapt the schema candidate set $\mathcal{C}_{ext}$ and its probability distribution. However, this change could invalidate the matches of an existing workspace. For instance, consider an extracted schema that contains a relation Student(name,credits). If subsequent updates to the dataset insert many students without credits, then eventually the relation Student(name,credits) should be replaced with Student(name). If a workspace schema contains an attribute matched to Student.credits, then this attribute is no longer matched with any attribute from the extraction schema. When explaining missing values to the user, we plan to highlight their cause, whether they are the result of data that can not be cast to the current schema, or of an orphaned workspace attribute (it is no longer matched to any attribute in $\mathcal{C}_{ext}$).

User Feedback. We envision letting the user provide various kind of feedback about errors in query results such as marking sets of invalid attribute values and unexpected rows. Based on provenance we can then back-propagate this information to interpret these as feedback on matches and extraction workflow decisions. To determine a precise method for determining the best fixes based on such information is an interesting avenue for future work. Continuing with our running example, assume that the user when postulating the existence of a `Student` relation was only interested in the names of graduate students. If the ASD has matched the student relation against both `Undergrad` and `Grad`, then the result will also include undergraduates. To indicate that there is a problem, the user can mark some of the undergraduate names in the result as erroneous. The ASD can back-propagate these markers to the inputs using provenance (e.g., see [6, 8]). In the example, these back-propagated markers all annotate data that is in the result based on the match between `Student` and `Undergrad`. Thus, they provide evidence against this schema match and the ASD may decide to remove the match.

5. ADAPTIVE ORGANIZATION

A classical RDBMS determines the physical organization of data based on its schema. This leads to excellent query performance and good storage utilization, because the schema information can be exploited to choose a beneficial physical design. For instance, using this approach it is not necessary to store schema information with each row, since all rows share the same structure and may be interpreted in the same way. However, this approach has the disadvantage that even small schema changes may necessitate physical reorganization of large amounts of data. Conversely, storing data in its native semi-structured or unstructured form does not enforce a fixed schema for the data and, thus, no physical reorganization is needed when the schema evolves. However, this flexibility comes at the cost of reduced query performance as the raw data needs to be decoded and transformed at runtime, and at the cost of higher storage requirements.

5.1 Materializing Personalized Schemas

From a user's perspective, data in a personalized schema is relational, i.e., SQL queries are treated as queries that access relational views. Under the hood, when a query accesses unstructured or semi-structured data, the data is transparently transformed into relational form, e.g., through an SQL-standard flatting operation for semi-structured data such as `JSON_TABLE()`. Thus, ASDs effectively decouple the data format used for storage (the unstructured or semi-structured input datasets) from the data format used for querying (in the format defined by a schema workspace). Technically, it would be sufficient to just store the input datasets and generate data according to a workspace schema at query time. This corresponds to the second method mentioned above. Alternatively, we could materialize data according to a workspace schema eagerly after each change to the schema. This is the approach taken by traditional data warehousing. In ASDs any (partial) relational workspace schema is essentially a view over the semi- or unstructured input datasets.

Thus, we have the freedom to materialize such a view just-in-time when it is requested or drop it if it is no longer deemed beneficial. Existing adaptive physical design and caching techniques [5, 11, 18, 27, 36] can be utilized to make such decisions. For example, these "views" can be materialized as in-memory data structures [25] or be stored as disk-resident materialized views and indexes as illustrated in [24].

Having this flexibility enables an ASD to adapt to usage patterns. During periods of frequent data evolution, materialized schemas require additional efforts to be kept up to date with the evolving schema and data. If this effort exceeds the performance benefit for queries, then these structures should be dropped. In ASDs this is not a problem, since these structures can be re-created from the input datasets at any point in time. In other words, in ASDs, the schema-based storage approach is merely an optional cache that we can fully control.

5.2 Shared Materializations

Since ASDs will host multiple schema snapshots from multiple workspaces over the same raw input data, it is advantageous to extensively share materializations across schemas. The schemas discovered by ASDs can be used to guide performance tuning by caching content of relations that are used frequently and are stable (their schema and content has not been modified recently). In this regard, it will be important to determine differences among schema snapshots from the same and from different workspaces to identify opportunities for sharing and to update cached data through incremental materialized view maintenance techniques. Additionally, we can leverage techniques from revision control systems, such as, copy-on-write, storing change deltas, and materializing at large change boundaries, to organize the shared physical cache for schema snapshots.

Furthermore, we can cache the outputs of data extraction or projections over this output to benefit multiple workspace schema elements. This leads to interesting optimization problems because of the sharing of elements, a problem analogous to the view selection problem [27]. The difference to other automated approaches for tuning physical design is that the sharing of such elements by workspace schemas is encoded in their matchings, which simplifies the identification of sharing opportunities. It remains to be seen whether this information can be exploited to devise caching strategies that are fine tuned for ASDs.

6. PROOF OF CONCEPT

We now outline a proof of concept ASD implementation, leveraging the Mimir [29, 34] system for its provenance and feedback harvesting capabilities. For this preliminary implementation, we chose to implement a probabilistic version of the normalizing relational data extractor for JSON data recently proposed by DiScala and Abadi [12]. This extractor uses heuristics based on functional dependencies (FDs) between the objects' attributes to create a narrow, normalized relational schema for the input data. We use this extractor as a proof of concept extraction workflow in our prototype.

6.1 Deterministic Extraction

The DiScala extractor [12] runs on collections of JSON objects with a shared schema. The first step in the extraction process is to create a functional dependency (FD) graph from the objects. The nodes in a dependency graph represent attributes and an edge from attribute A to attribute B denotes that a functional dependency $A \rightarrow B$ approximately holds. The objects are first flattened, discarding nesting structure and decomposing nested arrays, resulting in a single, very wide and very sparse relation. The extractor creates a FD graph for this relation using a fuzzy FD detection algorithm [19], originally proposed by Ilyas et. al., that keeps it resilient to minor data errors. Any subtree of this graph can serve as a relation, with the root of the tree being the relation's key (since it implies all attributes in this subtree). At this point, the original, deterministic DiScala extractor heuristically selects one set of relations upfront to use as a *final* target schema.

In a second pass, the extractor attempts to establish correlations between the domains of pairs of attributes. Relations with keys drawn from overlapping domains become candidates for being merged together. Once two potentially overlapping relations are discovered, the DiScala extractor uses constraints given by FDs to identify potential mappings between the relation attributes.

6.2 Non-Deterministic Extraction

The DiScala extractor makes three heuristic decisions: (1) Which relations to define from the FD graph, (2) Which relations to merge together, and (3) How to combine the schemas of merged relations. In an ASD, such heuristics are expected to serve only as a rough, first draft of the schema, and not as a final, correct representation of the data. Errors in the first of these heuristics appear in the visible schema, and as discussed in Section 2 are easy to detect and resolve. The remaining heuristics can cause data errors that are not visible until the user begins to issue queries. As a result, the primary focus of our proof of concept is on the latter two heuristics.

Concretely, our prototype ASD generalizes the DiScala extractor by providing: (1) Provenance services, allowing users to quickly assess the impact of the extractor's heuristics on query results, (2) Sensitivity analysis, allowing users to evaluate the magnitude of that impact, and (3) Easy feedback, allowing users to easily repair mistakes in the extractor's heuristics. We leverage a modular data curation system called Mimir [29, 34, 35] that encodes heuristic decisions through a generalization [22] of a common encoding of ambiguous and incomplete data called C-Tables [16, 20]. A relation in Mimir may include placeholders, or *variables* that stand in for heuristic choices. During normal query evaluation, variables are replaced according to the heuristic. However, the terms themselves allow the use of program analysis as a form of provenance, making it possible to communicate the presence and magnitude of heuristic ambiguity in query results, and also allow users to easily override heuristic choices.

EXAMPLE 5. *Consider the task of mapping a source relation $R(A, B)$ to a target relation $R'(C)$. Mimir's schema matcher uses a query of the form:*

```
SELECT CASE {C}
         WHEN 'A' THEN A
         WHEN 'B' THEN B
         ELSE NULL
       END AS C
FROM R
```

where {C} denotes a variable with domain {'A', 'B', NULL} that selects between the possible input columns, or declares

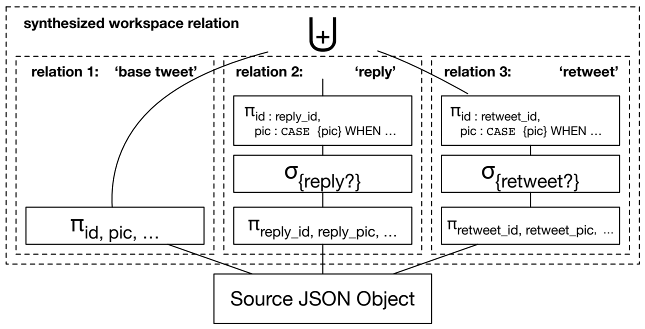
synthesized workspace relation
$\uplus$
relation 1: 'base tweet'
relation 2: 'reply'
relation 3: 'retweet'
$\pi_{id : reply\_id, pic : CASE \{pic\} WHEN ...}$
$\sigma_{\{reply?\}}$
$\pi_{reply\_id, reply\_pic, ...}$
$\pi_{id : retweet\_id, pic : CASE \{pic\} WHEN ...}$
$\sigma_{\{retweet?\}}$
$\pi_{retweet\_id, retweet\_pic, ...}$
$\pi_{id, pic, ...}$
Source JSON Object

Figure 3: Structure of an extracted relation's merged view

that there is no match.[1] *The resulting C-Table includes a labeled null for each output row that can take the values of* `R.A`, `R.B`, *or* `NULL`, *depending on how the model assigns variable values. Consider this example instance:*

R	A	B
	1	2
	3	4

R'	C
	`{C}`='A' ? 1 : (`{C}`='B' ? 2 : `NULL`)
	`{C}`='A' ? 3 : (`{C}`='B' ? 4 : `NULL`)

Conceptually, every value of $R'.C$ is expressed as deferred computation (a future). A valuation for $\{C\}$ allows R' to be resolved into a classical relation. Conversely, program analysis on the future links each value of $R'.C$, as well as any derived values back to the choice of how to populate C.

Heuristic data transformations, or *lenses*, in Mimir consist of two components. First, the lens defines a view query that serves as a proxy for the transformed data with variables standing in for deferred heuristic choices. Second, a model component abstractly encodes a joint probability distribution for each variable through two operations: (1) Compute the most likely value of the variable, and (2) Draw a random sample from the distribution. Additionally, the model is required to provide a human-readable explanation of the ambiguity that the variable captures.

For this proof of concept we adopt an interaction model where the ASD dynamically synthesizes workspace relations by extending one extracted relation (the primary) with data from extracted relations containing similar data (the secondaries). Figure 3 illustrates the structure of one such view query: The primary and all possible secondaries are initialized by a projection on the source data. A single-variable selection predicate (i.e., `{relation?}`) reduces the set of secondaries included in the view. Second, a schema matching projection as illustrated in Example 5 adapts the schema of each secondary to that of the primary.

Our adapted DiScala extractor interfaces with this structure by providing models for relation- and schema-matching, respectively. We refer the reader to [12] for the details of how the extractor computes relation and attribute pairing strength. The best-guess operations for both models use the native DiScala selection heuristics, while samples are generated and weighted according to the pairing strength computed by the extractor. By embedding the DiScala extractor as a lens, we are able to leverage Mimir's program analysis capabilities to provide feedback. When a lens is queried, Mimir highlights result cells and rows that are ambiguous [23]. On-request, Mimir constructs human-readable explanations of why they are ambiguous, as well as statistical metrics that capture the magnitude of the potential result error. Crucially, this added functionality requires only lightweight instrumentation and compile-time query rewrites [29].

[1]Currently, the schema matcher assumes that one target attribute can only be matched against one source attribute.

	Time		
Dataset	**Precompute**	**ASD**	**Classic**
TwitterM	214s (1.34)	1.26s (0.06)	0.31s (0.04)
TwitterW	625s (36.9)	1.49s (0.05)	0.28s (0.0012)

Figure 4: Overhead of the provenance-aware extractor (average of the trail runs, standard deviations are in parenthesis)

6.3 Evaluation

Mimir and the prototype ASD are implemented in Scala 2.10.5 being run on the 64-bit Java HotSpot VM v1.8-b132. Mimir was used with SQLite as a backend. Tests were run multi-threaded on a 12x2.5 GHz Core Intel Xeon running Ubuntu 16.04.1 LTS. The primary goal of our experiments is to evaluate the overhead of our provenance-instrumented implementation of the DiScala extractor compared to the behavior of the classical extractor. We used a dataset, **TwitterM** consisting of 200 columns and **TwitterW** consisting of 400 columns of twitter JSON data. The extractor was run on 100,000 rows taken from Twitter's Firehose. Figure 4 shows the performance of the extractor. We show preprocessing time, the time necessary to compile and evaluate `SELECT * FROM workspace_relation`, and the same query against a table containing the output of the classical DiScala extractor. Note that both these queries return the same set of results, but the former is instrumented, allowing it to provide provenance and feedback capabilities. Runtime for the instrumented extractor is a factor of 2 larger than the original, but is dominated by fixed compile-time costs.

7. RELATED WORK

Information extraction and integration have been studied intensively. Several papers study schema matching [4], schema mapping [15,17], entity resolution [14,32] and mapping XML or JSON data into relational form [12]. Furthermore, there exists extensive work on discovering schemas [7, 33] and ontologies [26] from collections of data. We build upon this comprehensive body of work and leverage such techniques in ASDs. With JSON as a simplified semi-structured data model, data can be stored, indexed and queried without upfront schema definition. Liu et al. [25] have introduced the idea that a logical schema can be automatically derived from JSON data and be used to materialize query-friendly adaptive in-memory structures. Niu et al. [30] present a version graph model for supporting nonlinear, virtualized version histories for relational. Curino et al. [9] study how multiple schema versions can co-exist virtually in the context of schema evolution. However, they have not addressed automatic schema discovery and schema adaptation based on queries and updates. This paper takes these ideas one step further by introducing the notion of adaptive schema databases based on flexible schema management principles [24] and establishes a practical framework of probabilistic schema inference with user feedback and provenance tracking.

8. CONCLUSIONS AND FUTURE WORK

We presented our vision for *adaptive schema databases* (*ASD*s), a conceptual framework for querying unstructured and semi-structured data through iteratively refined, personalized schemas. We discussed how ASDs can be realized leveraging probabilistic query processing techniques and by incorporating extraction and integration into the DBMS. We outlined data models that allow the physical layout of an ASD to be adapted in response to workload changes. Finally, we presented our proof of concept implementation within the Mimir probabilistic data cleaning system and demonstrated its feasibility. This paper represents only the first step towards practical ASDs. Fully realizing ASDs will require the database community to address several challenges:

Discovery. Schemas serve a role in helping users explore and understand new data by providing an outline of the available information. In an ASD, in addition to contending with the set of tables (resp., attributes) that *do* exist in their workspace, users must also contend with the set of tables and attributes that *could* exist in their workspace. Enabling discovery [7, 28] will be critical for making ASDs practical.

Materialization. We have illustrated that ASDs can, in principle, perform well. However, supporting interactive, large-scale workloads will require a complete re-thinking of the database's physical layer. We expect research in this area to borrow heavily from existing work on adaptive data structures [18], as well as on more coarse-grained strategies for workload sharing [10].

Data Synthesis. Populating the space of possible schemas will also be a challenge. Existing techniques for schema extraction (e.g., [3, 12, 24]) are a start, and we anticipate substantial opportunities for contribution in this space. However, these approaches only populate a finite set of relation and attribute names. Ultimately it would be desirable to enable synthesis of new tables and attributes from existing data. New tables might be derived by merging (or filtering) existing ones according to ontological relationships. For example, given Giraffe, Dog, and Cat relations, we can synthesize a new Animal relation. New attributes might be derived as (probabilistic) transformations of existing data. This latter challenge is especially relevant for geospatial applications, where positional information might be expressed through GPS coordinates, political boundaries (e.g., counties or zip codes), addresses, street corners, or any of a wide array of descriptors.

Conflict Response. When an ASD receives an invalid query, it considers possible schema changes that could repair the query. It is possible that such changes will conflict with schema decisions made previously. Making arbitrary changes to the schema is undesirable, as these changes might break existing workloads. It will be necessary to help the user see and understand the implications of revising existing decisions. For example, one response might be versioning or tentatively branching the schema [10, 30]. Another direction might be to use log analysis strategies that help users assess the impact of schema revisions.

Ultimately, ASDs aim to provide navigational and organizational benefits of schemas without requiring the high upfront cost of classical schema design. If realized, ASDs have the potential to bridge the gap between relational databases and NoSQL, creating a far more user-friendly data exploration experience than either approach is capable of.

Acknowledgements: *This work was supported NPS Award #N00244-16-1-0022, NSF Award #1640864, and by gifts from Oracle. Opinions, findings and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Naval Postgraduate School, or Oracle.*

9. REFERENCES

[1] L. Antova, C. Koch, and D. Olteanu. $10^{(10^6)}$ worlds and beyond: Efficient representation and processing of incomplete information. *The VLDB Journal*, 18(5):1021–1040, Oct. 2009.

[2] B. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Reenactment for read-committed snapshot isolation. In *CIKM*, 2016.

[3] S. Balakrishnan, A. Y. Halevy, B. Harb, H. Lee, J. Madhavan, A. Rostamizadeh, W. Shen, K. Wilder, F. Wu, and C. Yu. Applying webtables in practice. In *CIDR*. www.cidrdb.org, 2015.

[4] P. A. Bernstein, J. Madhavan, and E. Rahm. Generic schema matching, ten years later. *PVLDB*, 4(11):695–701, 2011.

[5] N. Bruno and S. Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *SIGMOD*, 2005.

[6] P. Buneman, S. Khanna, and W.-C. Tan. On Propagation of Deletions and Annotations through Views. In *PODS*, pages 150–158, 2002.

[7] M. J. Cafarella, D. Suciu, and O. Etzioni. Navigating extracted data with schema discovery. In *WebDB*, 2007.

[8] G. Cong, W. Fan, F. Geerts, J. Li, and J. Luo. On the complexity of view update analysis and its application to annotation propagation. *TKDE*, (99):1–1, 2011.

[9] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo. Automating the database schema evolution process. *VLDB Journal*, 22(1):73–98, 2013.

[10] C. A. Curino, H. J. Moon, and C. Zaniolo. Graceful database schema evolution: The prism workbench. *pVLDB*, 1(1):761–772, Aug. 2008.

[11] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin. Automatic sql tuning in oracle 10g. In *VLDB*, 2004.

[12] M. DiScala and D. J. Abadi. Automatic generation of normalized relational schemas from nested key-value data. In *SIGMOD*, 2016.

[13] X. L. Dong, E. Gabrilovich, K. Murphy, V. Dang, W. Horn, C. Lugaresi, S. Sun, and W. Zhang. Knowledge-based trust: Estimating the trustworthiness of web sources. *pVLDB*, 8(9):938–949, 2015.

[14] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *TKDE*, 19(1):1–16, 2007.

[15] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. *TCS*, 336(1):89–124, 2005.

[16] T. Green and V. Tannen. Models for incomplete and probabilistic information. In *Current Trends in*

Database Technology, pages 278–296. 2006.

[17] A. Y. Halevy. Answering queries using views: A survey. *VLDBJ*, 10(4):270–294, 2001.

[18] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *SIGMOD*, 2009.

[19] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. Cords: Automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, 2004.

[20] T. Imieliński and W. Lipski, Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, Sept. 1984.

[21] S. R. Jeffery, M. J. Franklin, and A. Y. Halevy. Pay-as-you-go user feedback for dataspace systems. In *SIGMOD*, 2008.

[22] O. Kennedy and C. Koch. PIP: A database system for great and small expectations. In *ICDE*, 2010.

[23] P. Kumari, S. Achmiz, and O. Kennedy. Communicating data quality in on-demand curation. In *QDB*, 2016.

[24] Z. H. Liu and D. Gawlick. Management of flexible schema data in rdbmss-opportunities and limitations for nosql-. In *CIDR*, 2015.

[25] Z. H. Liu, B. Hammerschmidt, D. McMahon, Y. Liu, and H. J. Chang. Closing the functional and performance gap between SQL and NoSQL. In *SIGMOD*, 2016.

[26] A. Maedche and S. Staab. Learning ontologies for the semantic web. In *ICSW*, 2001.

[27] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. *SIGMOD Rec.*, 30(2):307–318, May 2001.

[28] A. Nandi and H. V. Jagadish. Assisted querying using instant-response interfaces. In *SIGMOD*, 2007.

[29] A. Nandi, Y. Yang, O. Kennedy, B. Glavic, R. Fehling, Z. H. Liu, and D. Gawlick. Mimir: Bringing ctables into practice. Technical report, The ArXiv, 2016.

[30] X. Niu, B. Arab, D. Gawlick, Z. H. Liu, V. Krishnaswamy, O. Kennedy, and B. Glavic. Provenance-aware versioned dataworkspaces. In *TaPP*, 2016.

[31] D. Suciu, D. Olteanu, C. Ré, and C. Koch. Probabilistic databases. *Synthesis Lectures on Data Management*, 3(2):1–180, 2011.

[32] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *PVLDB*, 5(11):1483–1494, 2012.

[33] K. Wang and H. Liu. Schema discovery for semistructured data. In *KDD*, volume 97, pages 271–274, 1997.

[34] Y. Yang. On-demand query result cleaning. In *VLDB PhD Workshop*, 2014.

[35] Y. Yang, N. Meneghetti, R. Fehling, Z. H. Liu, and O. Kennedy. Lenses: An on-demand approach to etl. *VLDB*, 8(12):1578–1589, 2015.

[36] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. Db2 design advisor: Integrated automatic physical database design. In *VLDB*, 2004.