Zed: Leveraging Data Types to Process Eclectic Data

Amy Ousterhout[†], Steve McCanne[★], Henri Dubois-Ferrier[◇],
Silvery Fu[†], Sylvia Ratnasamy[†], Noah Treuhaft[★]
UC Berkeley[†] Brim Data[★] 12th Ave Labs[◇]

ABSTRACT

Data-processing systems increasingly face data that is *eclectic*—it spans many heterogeneous schemas and its schemas evolve over time. Unfortunately, existing approaches for processing and querying data are not ideal for eclectic data since they impose a trade-off between efficient querying and simplicity. We argue that this limitation stems from the very foundations of data processing: data models and their corresponding query languages. No existing approach—whether relational, document, or hybrid—is designed to enable ingesting, querying, and reasoning about heterogeneous types of data. In this paper we propose Zed, a new approach to data processing that centers around *data types*. Zed elevates data types to be first-class members of both the data model and query language, and by doing so offers a promising path towards easing the processing of eclectic data.

1 INTRODUCTION

The nature of data is changing, driven by growing applications such as IoT and monitoring as well as an increasing desire to relate data that spans multiple administrative domains. As a result, data today is *heterogeneous*—it spans many different schemas—and *evolving*—its schemas change frequently and sometimes unexpectedly. This data with diverse and dynamic schemas—that is, *eclectic data*—raises new challenges with ingesting, storing, and querying data. In particular, with eclectic data, the process of data discovery becomes a crucial part of the processing pipeline. When data does not conform to a small and well-known set of schemas, then the first step to processing the data is to understand the "structure" of a given data set: discovering what schemas are present, how often each occurs, how similar or dissimilar particular schemas are, and so forth. We use the term *data introspection* to refer to this process of exploring the structure of a dataset. Only once users understand the structure of their data can they take steps to transform, clean, or otherwise "prepare" the data for additional processing. In total these steps — spanning introspection and preparation — can be extremely time consuming, taking 80% or more of analysts' time [41].

The database community has long recognized this shift toward eclectic data and, over the last two decades, has vigorously debated different approaches to representing and processing diverse data [63, 68]. Today, the community has largely converged on two data models, which are known to have different strengths and weaknesses. The *relational model's* rigid structuring of data according to explicit schemas [38, 39] enables efficient querying and formats such as columnar [26, 56, 65] as well as data introspection [44]. It is the model of choice for analytics in relational databases [18, 20, 23, 24] and nested variants of it form the foundation of big data systems [1, 6, 31, 56, 73]. With the relational model, a schema is first defined for some entity (e.g., a relational table or Parquet file) and then values that conform to that schema are added to the entity. Consequently, ingesting eclectic data is challenging because the predefined schema must anticipate all possible variations of input data. On the other hand, the *document model's* self-describing data values make it trivial to mix heterogeneous data and to ingest data with never-before-seen schemas. Thus data sources commonly generate data in the document model (e.g., JSON), and document databases leverage this model to enable easy ingestion and querying of eclectic data [7, 15]. However, this approach sacrifices efficiency and clarity about what kind of data is present.

It is well-recognized that neither the relational nor document model is always best. As a result, several recent research efforts and industrial deployments attempt to combine these two and achieve the best of both. For example, some approaches such as AsterixDB can be configured to behave like either the relational model or the document model [30, 57], while others such as Snowflake or Lakehouses can store some data in each model [20, 31, 32, 40, 45, 52].

Unfortunately, as we will discuss (§2), even these combined approaches are far from ideal. First, users must contend with two data models: they must decide how to split or replicate their data across both models and a single query can typically leverage the benefits of only one model. Second, to achieve the benefits of explicit schemas, users still have to clean their data from the document into the relational model. This cleaning process is known to be complex and brittle [8, 41], especially without effective introspection tools to discover what kinds of data are present in the first place. And yet, thirdly, these systems do not make introspection over document data any easier. Thus users are stuck in a catch-22: in order to clean data they must be able to introspect over it, yet in today's systems, rich introspection is only possible after the data has been cleaned.

We believe that these approaches fall short because they are *hybrid* rather than truly *unified*. They co-implement both models, but a given piece of data can only benefit from one model at a time. Taking a step back, we wondered if this sacrifice is truly necessary, or if a solution could be found by addressing the problem at a lower layer. Perhaps it is time, yet again, to rethink the foundations of data processing: the data model and its corresponding query language. We argue that eclectic data would benefit from a *new approach to unification* in which a single data model can simultaneously provide the benefits of both explicit schemas (efficient analytics, ease of introspection) and mixed heterogeneous data (seamless ingestion, query results that span heterogeneous data).

We start with the observation that what fundamentally enables both efficient analytics and data introspection is knowing the fully

 13th Annual Conference on Innovative Data Systems Research (CIDR '23). January 8-11, 2023, Amsterdam, The Netherlands.

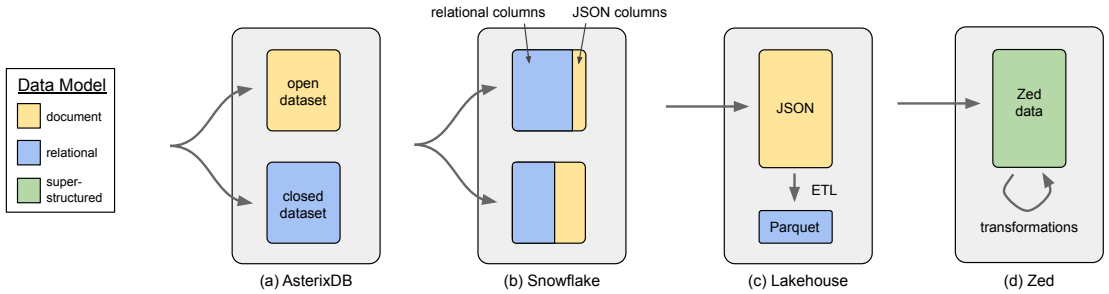

Figure 1: Three example hybrid approaches to data processing: (a) AsterixDB splits incoming data into open and closed datasets in separate files, (b) Snowflake adds columns of JSON data to each relational table, and (c) Lakehouses use ETL to clean a subset of JSON data into Parquet. (d) In Zed, all data (raw or transformed) is represented in the super-structured data model.

specified *type* of each piece of data. For example, this allows us to store data in formats that are organized by type (as in columnar formats) and to query for information about what types of data are present. The relational model has this explicit type information, but is overly rigid in how it uses it; it explicitly requires the set of allowable types (i.e., schemas) to be defined up front and only accepts data from these schemas. Unfortunately, attempts to relax this rigidity by integrating the flexibility of the document model have often introduced new problems. For example, some add new external components (e.g., schema registries [22]) that must be integrated with the data processing pipeline, while others incorporate partially typed data (e.g., Snowflake's `OBJECT` type [40]) and thus lose some of the benefits of type information.[1] This leads us to the goal of defining a data model in which every data value has an explicit, complete type and yet there are no constraints on which types are permitted to coexist (e.g., in the same file, table, or set of query results). In other words, our goal is *comprehensive yet flexible typing*. In addition, this data and its type information should be self-contained so that parsing data never requires coordination with an external registry; instead users can query both data and types in a single unified manner.

The key to achieving comprehensive yet flexible typing is to change the way that we associate type information with data values. Instead of associating a single schema with each file or table, which makes it difficult to store or process heterogeneous data together, we propose a new *data type abstraction* that is associated directly with individual data values. This enables each value to have an explicit data type (as in the relational model) but also allows different values that are processed together to have different types (as in the document model). We call the resulting data model the **super-structured data model**, because it subsumes the structures of the relational and document data models. Data is organized as a sequence of typed values; when all values have the same type, this model is equivalent to the relational model.

Realizing this approach requires care in designing the type abstraction. We propose an abstraction with the following properties:

- Types are *associated with individual values*, rather than with a collection such as a table or file.
- Types are *complete* - we observe that partial types such as `OBJECT` or `JSON`, or allowing a value to include extra fields beyond those specified by its type, prevent the full benefits of types.
- Types are *first class* - users can query for data types and the query results—which contain types—are returned in the same data model. Users can refer to types by name and data formats can assign numeric type IDs for efficient storage and querying.
- Type definitions are *inlined* - this enables data sources to define new types on the fly without out-of-band coordination or additional burden relative to writing JSON data.

However, a new data model alone cannot provide the introspection and unification that we seek; we also require a corresponding query language that can exploit the full power of this data model. While the data model encodes per-value type information, the role of the query language is to expose this type information to users. Users should be able to query *by* type (e.g., returning all data of a specified type), or *for* type (e.g., returning the type of a data value). In short, types must be *first-class members* of the query language as well. This enables rich introspection and is crucial for truly subsuming both the document and relational models. Querying by type allows a user to extract all values with a given schema; the returned data corresponds to a table in the relational model and can be processed as such.

In this paper we propose **Zed**, a new approach to data processing that is centered around these data types. Zed includes a new super-structured data model (§3.1) and query language (§3.2). In addition, Zed's type abstraction allows data to be represented in the format most suitable for the task at hand: columnar for analytics, human-readable for debugging, etc. Because all data is typed, converting it between formats within the family is lossless and fully automated. Thus Zed includes a "family of data formats" (§3.3). We will show how Zed overcomes the challenges of existing hybrid approaches and unifies the document and relational models in a new way, *embodying both at the same time* (§4), and describe directions for future research (§5).

2 THE LIMITATIONS OF HYBRID

As pointed out by prior work, the relational model's rigid schemas and the document model's lack of schemas each create significant challenges for users [27, 30, 40, 68]. *Hybrid* approaches have emerged in response, attempting to combine the two models to achieve the best of both.[2] There are many such hybrid systems, including research approaches [30, 45, 52, 57] and industry examples [19, 20, 31, 32, 40]. These approaches can simplify multi-model

[1] We refer to types that are not fully specified as *partial types*, and their fully-specified counterparts as *complete types*.

[2] Note that when we refer to "hybrid systems," we refer to the subset of "multi-model systems" [55] that relates to the relational and document data models.

deployments, but no consensus has emerged regarding which is best, and each still faces significant challenges.

In this section, we illustrate the limitations of hybrid systems during ingestion (§2.1), querying (§2.2), and data introspection (§2.3). For the sake of concreteness, we focus on three specific systems; other hybrid systems often face similar limitations. Figure 1 shows the three example systems and configurations.

AsterixDB [30]. AsterixDB's data model is a superset of JSON that also includes a schema language. It supports both "open Datasets" of heterogeneous document-model data and relational-like "closed Datasets" of homogeneous data. All data can be queried with AsterixDB's AQL or with SQL++ [57]. In our example configuration, data with known schema is validated and stored in "closed Datasets" while the rest is stored in "open Datasets".

Snowflake [40]. Snowflake extends traditional relational data warehouses with support for semi-structured data. This is done primarily via columns of type `OBJECT` that can store JSON documents, as well as corresponding extensions to SQL.[3] In our configuration, all data is stored in Snowflake tables, with the heterogeneous, nested, and dynamic portions stored in such columns of JSON.

Lakehouse [31, 32]. Lakehouses store all data (whether raw or processed) in a data lake and query it using SQL or dedicated libraries. A lakehouse deployment might store all incoming data directly in the lake and use ETL to clean a subset of the data into Parquet files [6] for efficient analytics.[4]

2.1 Data Ingestion

Data sources today often generate data in a schemaless format such as JSON [11]. When sources have eclectic data they often prefer JSON over alternatives such as Avro [1] or Protocol Buffers [21] because it enables heterogeneous data to coexist in a file and it avoids the upfront complexity of defining and agreeing on schemas. When JSON data arrives at a storage layer for ingestion, hybrid approaches face two main challenges.

Cleaning data into the relational model. In order for data to fully benefit from schemas (for data introspection and efficient queries and storage), it must be converted from JSON into the relational model. The challenges of this are well-known: data must be matched to and validated against the relevant schema and any non-conforming data must be cleaned or dropped [8]. Automated approaches to this such as Fivetran [9] or Informatica [10] can be error-prone and brittle in the face of changing schemas. While this problem first arose with purely relational systems, hybrid systems also suffer from it since they store some data in the relational model.

Which model for which data? Users of hybrid systems face the challenge of deciding which data model to use for which parts of their data. Users of AsterixDB must decide which Datatypes are stable enough to be able to leverage closed Datasets; users of Snowflake must decide which fields of their data are eclectic enough to warrant `JSON` columns and when data is different enough to warrant a separate table altogether; and users of Lakehouses must decide which data to ETL into Parquet. These decisions impact how users will be able to query their data (§2.2), and if users want to convert data between models later, they may encounter the challenges and delays of cleaning their data, as described above.

2.2 Querying Data

Different data models provide different benefits during querying. However, in hybrid approaches, most data is only stored in one model or the other; this is the case for all data in our AsterixDB and Snowflake configurations, and the JSON data that isn't ETLed into Parquet in the Lakehouse. This data can typically only benefit from the properties of the data model that it is stored in.

For example, the relational model's schemas enable introspection queries that explore what kinds of data are present [44] and high-performance querying over efficient formats [6, 40, 56, 65]. For data stored only in the document model, these benefits are not available. A rich body of literature attempts to infer schemas over this kind of data [2, 29, 34–37, 40, 42, 53, 54, 59, 62, 70] but these heuristics can be inaccurate.[5] For example, human intervention may be necessary to determine whether two values have the same schemas in document-model data in AsterixDB or the Lakehouse.

On the other hand, the document model enables queries that can flexibly mix heterogeneous data [27, 30]. For example, consider a hypothetical query to sort all data in a large data system by time and return the five earliest records, effectively: `SELECT * FROM * ORDER BY time LIMIT 5`.[6] This query is straightforward to issue over JSON data, even if the data and results span multiple schemas. However, to issue this query over relational data, the rows from heterogeneous tables must somehow be combined. For example, the SQL `UNION` operator can be used to create a wide table that contains all the columns of all the heterogeneous tables and many `null` values; this is sometimes called an "uber table" whose "uber schema" is the union of all columns spanned by the data. A user can then issue the query above over this uber table and obtain the results in an uber table that is bloated with many `null`s. Notably, AsterixDB does not suffer from this problem. Even though data in a closed Dataset is by definition homogeneous, AsterixDB's data model is a superset of JSON, and AsterixDB query results can mix heterogeneous data, similar to JSON. In contrast, issuing this query across Snowflake tables with differing relational columns will trigger the same problem as in purely relational systems.

2.3 Data Introspection

With the proliferation of eclectic data, users are increasingly interested in *data introspection*. Users want to query data *for its schema*, for example to enumerate the schemas spanned by a dataset and how many records have each. Users also want to query data *by its schema* to select a subset of data with particular schemas for further exploration. Users may want to perform these tasks at any stage of processing: during ingestion, on stored data, or over query results.

The relational model does enable some forms of data introspection. For example, users can query `INFORMATION_SCHEMA` to list tables and their column names and types, or query external systems like Aurum [44]. However, the relational model and query languages for it lack support for referring to schemas holistically. Some query languages such as N1QL enable users to query by or for

[3] MySQL [12] and PostgreSQL [14] offer similar functionality with a `JSON` type.
[4] Parquet's data model is the relational model, extended to support nested data.
[5] Schema inference over data in property graphs or RDF faces similar challenges [17, 59].
[6] The syntax `FROM *` is not valid SQL but is used here to illustrate the challenges of processing heterogeneous data with the relational model.

field types (e.g., with `ISSTRING()` or `TYPE()` functions), but these only work for primitive types and return "object" or "json" for more complex data [15, 16, 24]. As a result, the relational model does not support queries by schema, because it can neither mix heterogeneous data together nor refer to schemas in queries. Introspection in the document model is even more challenging, because it lacks explicit schemas. In addition, inferring the schema of data or which data has the same schema can be inaccurate and brittle (§2.2).

Unfortunately, this puts users in a catch-22. In order to clean their data into the relational model, users need to first understand what kinds of data are present, but the limited tools for introspection that are available today are only available for data in the relational model. In short, it is difficult for users to introspect or clean their data until *after* it has already been cleaned into a pre-defined set of schemas. Furthermore, query languages for hybrid systems [19, 30, 40, 57] do not avoid this catch-22 because they do not provide a holistic way to refer to schemas in queries or in query results.

3 THE DESIGN OF ZED

Zed's key goal is to provide *comprehensive yet flexible typing* throughout data processing. Achieving this goal allows Zed to unify the document and relational models and enable new functionality for data introspection. Zed's key technique is a new *data type abstraction* that represents the structure of an individual data value; achieving comprehensive yet flexible typing requires careful design of this abstraction. We now describe Zed types and how they impact Zed's data model (§3.1), query language (§3.2), and formats (§3.3), as well as how Zed manages sets of types (§3.4).

3.1 Zed Data Model

We first describe the Zed data model and then highlight some of the unconventional design decisions behind it.

3.1.1 Data Model. The Zed data model consists of an ordered sequence of typed data values. Each value's type is either a primitive type (int32, string, bool, type, etc.), a complex type (record, array, set, map, union, etc.), a named type, or the null type. Many of these types have analogs in other data models, and we describe the significance of some of them below. For example, consider Zed records. A record consists of an ordered set of named values called "fields". Field names are simply strings and field values can be any Zed value, with the types described above. This enables nested data, as a record can contain arrays, sets, other records, etc. A row in a relational table, a record in Avro, or a JSON document could each be represented using a single Zed record.

For example, consider the following two records of IoT sensor data, represented in Zed's text-based format, ZSON:[7]

```
{ts:09:01:00(time),temp:68(int32)}
{ts:10:12:00(time),percent_humidity:43.7(float32)}
```

The first record has type `{ts:time,temp:int32}` while the second record has type `{ts:time, percent_humidity:float32}`. Despite having different types, these two records can be stored in the same file[8] because data types in Zed are **associated with individual data values**.

```
{ts:09:01:00(time),temp:68(int32)}(=temperature)
{ts:10:12:00(time),percent_humidity:43.7(float32)}(=humidity)
{ts:17:29:00(time),temp:71(int32)}(=temperature)
{ts:17:45:00(time),temp:80(int32),unit:"F"(string)}(=temperature)
{ts:18:02:00(time),temp:28(int32),unit:"C"(string)}(=temperature)
```

Figure 2: Example IoT data from temperature and humidity sensors, represented in text-based ZSON.

Types are **first-class members** of the Zed data model. This means that a type—even the type of a record—is a single entity, which can be represented as a Zed value whose type is *type*. As a result, a query for the type of a record returns the record's type in the same Zed data model, rather than a separate data model as in existing systems [44]. Zed also enables users to define *named types*, which bind a name to a type. For example, ZSON includes decorator syntax for defining named types, e.g., `(=temperature)` can be used to define a named `temperature` type:

```
{ts:09:01:00(time),temp:68(int32)}(=temperature)
```

The Zed query language enables users to refer to these named type values using angle brackets (§3.2). For example, a query can extract all records with type `{ts:time,temp:int32}` using the `<temperature>` syntax; this is the equivalent of a single relational `temperature` table.

Type definitions in Zed are stored **inline** with the data. When a new type first appears in a Zed file, the type definition is stored inline at that point in the file. The `temperature` record above shows how Zed does this in its text-based format; we describe how Zed efficiently encodes type definitions in its binary formats in Section 3.3. A consequence of inlined types is that type definitions are locally scoped. Thus Zed must provide a way to reliably interpret and merge types that are defined in different scopes; we describe Zed's approach in Section 3.4.

Finally, each type in Zed specifies the **complete** and potentially nested structure of data values with that type. The Zed data model deliberately omits types that provide only partial type information such as `OBJECT`, `JSON`, or `any`, as they conflict with the goals of the data model. In Zed, types are "closed," meaning that a record of type `T` never contains extra fields beyond those defined by `T`. Any Zed value is nullable, e.g., a record of type `T` can omit fields of `T` by explicitly setting them to `null`; this avoids an exponential explosion in the number of defined types when different records omit different combinations of fields. While in many cases it is useful to enforce constraints on which fields may be `null`, our view is that such policies should be specified and enforced at a higher layer (e.g., in an ingestion pipeline or specified in a query) rather than by the data model.

Complete types may appear to be at odds with Zed's goal of flexibility. For example, suppose a temperature sensor outputs `temperature` records with two fields at first, but then adds a third `unit` field to `temperature` so that it can output data as either Fahrenheit or Celsius, as shown in Figure 2; this is the classic "schema evolution" problem. Is this valid data, or will Zed throw an error? This is in fact valid Zed data; the data itself defines the

[7] In practice, the `time` type in Zed represents native epoch 64-bit nanosecond times and ZSON represents them in ISO format (e.g., 2023-01-01T09:01:00Z). For simplicity, we abbreviate these time values in the example ZSON data shown in this paper.

[8] In this paper we use the term "file" as a simplification; a sequence of Zed values may exist in a file but also in a cloud object, a stream sent over HTTP, etc., and in general, the sequence need not be seekable.

named types and the Zed runtime must understand that a named type's binding may evolve within a data stream. In this example, line 4 changes the definition of the named `<temperature>` type to `{ts:time,temp:int32,unit:string}` from this point in the file onward.[9] This ability to change the binding between a name such as `temperature` and the type it refers to ensures that named types are always valid, and that arbitrary Zed data can be appended to a file regardless of what other named types already exist in it. This is crucial for providing complete flexibility.

3.1.2 Design Decisions. Zed differs from existing data models in how it relates types to data values. In relational databases or Parquet files, a type is defined for an entire table or file and all records it contains share the same type. This achieves comprehensive types but not flexibility. In contrast, in the document model, each record is self describing and has its own implied type, and there is no way to specify that multiple records have the same type. This achieves flexibility but not comprehensive typing. By decoupling the type assignment from the data's organization into files, Zed can achieve comprehensive types and flexibility simultaneously.

Some existing approaches define types in schema registries, resulting in globally scoped types; a type defined in a registry may be used by any program in that administrative domain [22]. However, globally scoped types are at odds with flexibility, because ensuring compliance with a schema registry may restrict what kinds of data users can write. Scoping named types locally to a file and inlining their type definitions gives data sources the flexibility to define arbitrary new types on the fly and ensures that data consumers always have the required type information to decode data.

Zed also takes an unconventional approach regarding what a type definition specifies. Some existing data models flexibly accommodate eclectic data by supporting partial types such as `JSON` or `OBJECT` [12, 14, 40], or by allowing "open" data types, where a record may contain extra fields beyond those specified by its type's type definition [13, 25, 30]. However, incomplete type information makes it challenging to leverage efficient formats such as columnar [40] and makes the results of queries about types less useful. We also observe that union types can handle cases where a single field might reasonably assume one of a few different types (e.g., an ID field that can be an int or string), and heterogeneity of type structure can be accommodated by making it easy to define new types. As a result, Zed takes a different approach based on complete types, as in Parquet and Avro, but differs from Parquet and Avro by allowing heterogeneously typed sequences of values.

Finally, while prior work has studied ways of representing relation names within data [50, 61, 71, 72], we are not aware of any existing data model that can express an entire schema or type as a single value within serialized data.

3.2 Zed Query Language

The Zed query language aims to support the same core functionalities as existing query languages for relational and document data, and to also provide new functionality by exposing type information to users so that they can query by and for types. Zed supports both of these—querying data and querying types—within the same language. We will briefly sketch Zed's query language here; a complete description is beyond the scope of this paper.

Dataflow Operators	`cut`, `drop`, `head`, `join`, `put`, `rename`, `search`, `sort`, `tail`, `uniq`, `where`
Functions	`abs`, **`cast`**, `ceil`, `floor`, **`is`**, `log`, `lower`, **`nameof`**, `sqrt`, **`typeof`**, **`typeunder`**, `upper`
Aggregate Functions	`and`, `any`, `avg`, `count`, `max`, `min`, `or`, `sum`

Table 1: Some of the operators and functions in the Zed query language. Bolded functions use types or type names as an input or output.

The Zed language borrows heavily from existing query languages such as SQL, jq, and Lorel [27, 46], as well as traditional UNIX shells. Similar to jq and Lorel, the Zed language can issue queries across heterogeneous data values and tolerate missing fields. For example, a query for `avg(temp)` over the data in Figure 2 will quietly skip the non-matching `humidity` record and return the average of the `temp` fields in the other four records. The Zed language also supports many standard SQL operators such as `join` and `where` and common UNIX commands such as `head` and `sort`.

A key novelty of the Zed language is that it takes well-known features of existing programming languages—type introspection and first-class types—and applies them to a query language. *Type introspection* is the ability to obtain the type of an individual data value. Python supports type introspection with a `type` function, and Zed enables it with a `typeof()` operator. This is a necessary building block to enable rich queries about types, and is feasible to implement because the Zed data model associates a type with every individual data value, even records. For example, a query for the type of the first record in Figure 2 returns `<temperature={ts:time, temp:int32}>`. In contrast, existing query languages do not enable full type introspection. The `type` operator in jq, `TYPE` in N1QL [16], and `typeof` in JavaScript return strings such as "number" or "boolean" instead of a dedicated type type, and they do not capture the complete structure of complex types, simply returning "object" when issued over a JSON-version of each of the records in Figure 2.

Programming languages have leveraged *first-class types* since the 1960s [60, 69], and there is some debate over exactly what properties are required in order for an element to be considered "first class" in a programming language. Here we highlight four key first-class properties that types have in the Zed language:

- Types can be returned from functions: `typeof()` returns a type
- Types can be arguments to functions: `is(<temperature>)`
- Types can be tested for equality: `typeof(this)==<temperature>`[10]
- Support for type literals: `type temperature= {ts:time,temp:int32}`

These first-class types enable Zed to support rich data introspection, as we will show in Section 4.4.

The Zed query language is inspired by the dataflow pipeline pattern of traditional UNIX shells. It operates over a sequence of Zed values that can be piped from one operator to the next, though Zed flowgraphs can also split and merge the processing pipeline in the form of a directed-acyclic graph. Table 1 shows examples

[9] Alternatively, the runtime could automatically convert the conflicting types into a named sum type, represented in Zed by a named union of the underlying unique types. We leave exploring this approach to future work.

[10] The identifier `this` refers to input data values one-by-one, so a query for values with `typeof(this)==<temperature>` returns all `<temperature>` records.

of the different components of the Zed query language. Dataflow operators take in and output a sequence of Zed values. Functions appear in expressions (e.g., `3 + sqrt(x)`) and take in zero or more Zed values and output a single Zed value. Finally, aggregate functions take in a sequence of input values, are evaluated by the `summarize` operator, and produce an aggregated value.

Many Zed operators and functions leverage type information. For instance, the function `is` tests if a value is of a specified type (e.g., `is(<temperature>)`). The `nameof` function returns the string name of a value's type. The `typeunder` function returns the underlying type of a value, capturing its structure but omitting name information for named types (e.g., `typeunder` of the first record in Figure 2 is `<ts:time, temp:int32>`). Note that Zed allows queries to refer to types either with their name (using the `nameof` and `typeof` functions), or by their structure (using the `typeunder` function). This is useful for disambiguating between values that have the same type name but different structures (or vice versa), as with the different `<temperature>` types shown in Figure 2.

3.3 Zed Format Family

Prior work has shown that no single format is best for all use cases [66, 67]. Thus, Zed provides a family of three data formats: a text-based format and two binary formats. Zed also supports indexes over its binary formats, where the indexes themselves are represented in one of the binary Zed formats. Data can be converted between these formats with no loss of information or human involvement because all three implement the Zed data model, including its comprehensive typing and ordering of values and fields. This is similar to converting between row-based and columnar relational data [28, 33] and contrasts with converting data between the document and relational models.

A key challenge in designing Zed's data formats is encoding type definitions in a space-efficient way. In Zed's text-based format ZSON, each value has a type that is implied by its textual structure as in JSON, but additional type information can be specified inline with each individual value. For example, in Figure 2, `temp` values are decorated with `(int32)` to indicate that they are 32-bit integers rather than the implied integer type of 64-bit integers. As a result of these decorations, a ZSON file will typically be larger than a JSON file containing the same data. In contrast, Zed's binary formats encode type definitions efficiently, once per unique type, as each type is encountered. When a new data type first appears in a sequence, it is assigned a numeric ID and both the ID and type definition are encoded in the stream. All subsequent values with the same type are encoded with the numeric type ID, rather than repeating the complete type definition. As a result, the amount of space required for type definitions in a binary Zed file scales linearly with the number of distinct types rather than the number of records. In practice, these type definitions comprise a tiny overhead compared to the data. Briefly, Zed's formats are:

ZSON: ZSON is a text-based format, similar to JSON but with support for types, as shown in Figure 2. Moreover, ZSON is a superset of JSON, which makes compatibility with JSON-based legacy systems simple and easy. In our implementation, we have not tried to optimize ZSON to make it particularly performant as performance-critical computations are always carried out in the efficient (and information-equivalent) binary formats described below.

ZNG: ZNG is a binary row-based format, somewhat like Avro [1] but with support for heterogeneous values within the same file. ZNG interleaves encoded values and encoded type definitions. Each type definition binds the next available numeric type ID to a specific user-defined type. Values are encoded by first encoding the type ID. Then the value itself is encoded using a tag-encoding approach which consists of a tag specifying the value's length and then the value itself, similar to Protocol Buffers [21]. While formats like JSON are challenging to parse efficiently [51, 58], ZNG can be parsed quickly because each value's type and length are completely specified by its encoding. This enables a parser to skip records or parts of records whose fields are not relevant to a query.

VNG: Vector ZNG (VNG) is a binary format which generalizes existing columnar formats [5, 6, 56] to support heterogeneous values. VNG arranges values into "vectors", which are a generalization of columns in existing approaches.[11] Each VNG file includes the vectors data section, a metadata section of reassembly maps, and a trailer defining the section boundaries. The *data section* encodes the vectors of data, where each is a vector of primitive-type values corresponding to leaf elements of a hierarchical data type (e.g., all the `unit` values in Figure 2). The *metadata section* enumerates all the types in the file and for each type it describes where to locate the vectors of data within the data section. The *trailer* includes the size of each section and additional metadata. All three sections are encoded (re)using ZNG, contrasting with Parquet which relies on an additional format (Thrift) for encoding type definitions and other metadata.

3.4 Zed Type Contexts

Each sequence of Zed values defines a set of types. Thus Zed requires a way to efficiently manage the set of types that are present within a single sequence and also to relate these sets of types across different sequences. For example, within a single sequence, Zed needs to track the relationship between types and type IDs, and across sequences, the Zed runtime needs to identify which values have the same type.

Thus Zed introduces the *Zed type context*, an abstraction that represents the set of types that are present in a file, stream, etc. A type context maps each defined type to a reference to each such type (e.g., a numeric type ID). A type context is explicitly embedded in ZNG and VNG files, and the Zed runtime builds up its own internal representation of the type context as it reads in Zed data. Within a single sequence of Zed values, the type context enables the Zed runtime to efficiently serialize a value by prefixing its type ID and deserialize subsequent values by their type IDs.

The Zed type context also provides a way to relate types across different sequences. When multiple sequences of values have different type contexts, the type references may conflict. For example, in one data stream, the type `{x:int64,y:int64}` might have type ID 33 and in another stream it might have type ID 42; this is because types are defined inline rather than in a global schema registry. These conflicts must be resolved in order to correctly query data by type across multiple files. While relational systems are designed to compare two schemas at once, e.g., to determine if two tables

[11] We use the term "vector" instead of "column" since the values are not columns of a table but rather vectors of primitive values where each vector corresponds to a leaf value in a hierarchical type.

can be UNIONed, they do not address the challenge of efficiently determining the relationships between two sets of independently defined types.

Zed resolves this conflict by mapping input type contexts into a shared output type context and relabeling each value with its type ID in the shared type context. This mapping process is efficient and involves a simple table lookup per typed value. In the example above, the shared type context might assign type ID 35 to type `{x:int64,y:int64}` so the first stream merely needs a map from 33 to 35 and the second stream a map from 42 to 35. Once these type-context maps have been constructed, Zed can adjust each value's type ID to its new type ID in the shared context. These adjustments do not require re-encoding values themselves, because the Zed formats do not embed type IDs within value encodings (e.g., in ZNG they are prefixed). In this way, the output stream has a single consistent type context for all the values in it.

The Zed runtime constructs these type-context maps on the fly. When the runtime encounters a new type in an input stream, it first determines the type's canonical type representation. The canonical type representation is an efficient and unique binary representation of the type, which is not dependent on type IDs. Then, the runtime uses the canonical type representation to index into a hash table that maps each type to its type ID in the shared type context (35 in the example above). Finally, the runtime adds an entry to the type-context map mapping the type ID in the input context to the type ID in the shared context.

Since a type context is built up by sequentially scanning a sequence of values, it would seem that the Zed runtime must process a stream from the beginning in order to properly decode the types in use. This creates a challenge for use cases that might want to seek into a specific location in a large file or for stream processing where data is continuous (e.g., event data arriving over Kafka [4]). To address this, the ZNG format supports type-context reset markers. These are analogous to resynchronization markers in video codecs and simply zero out the type context and restart the process for emitting type definitions inline. A ZNG encoder is free to reset the type context at any point in a stream, and a ZNG decoder can thus seek to any reset marker and initiate decoding. Adding reset markers adds some overhead but we have found that in practice the overhead is a tiny fraction of the encoded data; moreover, the overhead can be controlled by adjusting the frequency of resets.

4 DATA PROCESSING WITH ZED

Here we describe Zed's implementation and how data generation, querying, and introspection work in a Zed-based data-processing ecosystem.

4.1 Zed Implementation

We are in the midst of building out the Zed architecture. It currently consists of 120K lines of code (LOC) including the Zed system and extensive tests. The majority of the code is written in Go. Our codebase includes the query language definition, compiler, and command line tool `zq` (73K LOC), the formats ZNG, VNG, ZSON, and indexes (12K LOC), and code for reading and writing data in Zed's formats and in other formats such as Parquet, NDJSON, and CSV (12K LOC). Zed is available open source at https://github.com/brimdata/zed.

Zed users can store data in files directly on the file system or in a Zed data lake. When users store their data in files, they can parse, search, and analyze it using Zed's command-line tools (e.g., `zq`). In addition, Zed supports a Zed lake in which data is organized into data "pools". Zed enables transactional reads and writes over data in the Zed lake using an immutable transaction log, as in Iceberg [3] or Delta Lake [31], though Zed stores its transactions in a Git-like commit history supporting branching, merging, time travel, and data lineage. Users can query Zed lake data using command-line tools, the Zui application, or APIs in Go and Python.

Thousands of active users employ Zed at desktop scale, e.g., to analyze heterogeneous network logs or to ingest data from legacy servers that use heterogeneous data formats. The Zed lake portion of Zed is under ongoing development and is currently used in production by a number of early-adopter data teams, e.g., for database ETL and security workflows.

4.2 Data Generation and Ingestion with Zed

In a Zed-based data-processing ecosystem, data sources may generate data in a number of common formats (e.g., JSON, CSV, Parquet, or Avro). Alternatively, if a source is Zed aware, it can generate data in any of the native Zed formats and can exploit the full richness of the Zed data model. For example, a source can define named types to facilitate the relationship between specific data structures in a native programming language and the named type in Zed. Named type definitions require specifying a name for the type as well as the underlying type (often a record type comprising field names and field types). Leveraging named types is typically not burdensome for data sources because these types are already defined by in-memory data structures in the source system and a client library can directly marshal these data structures into serialized Zed. When a source wants to define a new type or redefine a named type, it simply starts writing data with that new type, as shown in Figure 2. Data sources may forgo named types altogether and this is very common; in this case writing Zed data is very similar to writing JSON data.

When data arrives at the storage layer, users can store it in files or in a Zed data lake. In either case, data can be stored as both ZNG or VNG, and Zed can also build indexes over it. How the data is split across different formats may impact query performance but has no impact on the types of queries that can be issued, the way queries are expressed in the query language, or what query results look like.

Compared to generating and ingesting data in the document model, using Zed is no more difficult. Compared to ingesting data into relational or hybrid systems, using Zed is much easier. Zed users might choose to transform data on ingest, but they are never required to predefine schemas or to clean data to conform to any particular structure before it can be stored.

Users who are unable to modify their data sources to output data in one of Zed's native formats can still benefit from Zed. As mentioned above, Zed can consume data in a number of popular formats and automatically convert the input to native Zed formats on the fly. Zed can also convert outputs of queries to any desired format. As such, users can query data in any supported format using Zed's query language and convert the output as desired, incurring a performance cost to perform format conversion. In addition, if users

```
{
    ts: 18:47:00 (time),
    triggered: true (bool),
    loc: {floor: 3 (int32), room: "robotics lab" (string)} (=location)
} (=motion)
{
    ts: 19:03:00 (time),
    triggered: true (bool),
    loc: {floor: 2 (int32), room: "kitchen" (string)} (=location)
} (=motion)
{
    ts: 19:04:00 (time),
    volume: 9.5 (float32),
    loc: {floor: 4 (int32), room: "media lab" (string)} (=location)
} (=sound)
```

Figure 3: Nested IoT data output by motion and sound sensors in different rooms, represented in ZSON.

would like to *shape* their data into a specific set of data types, Zed's type abstraction and extensive support for type conversions and data transformations all provide an excellent foundation compared to existing systems; we leave an in-depth exploration of this shaping functionality to future work.

4.3 Querying Data in Zed

All Zed data, regardless of the format that it is stored in, can benefit from both the document model's flexibility and the relational model's types. Zed supports queries that are typically issued over data in the document model today, because they involve mixing heterogeneous data. For example, Zed supports the query from Section 2.2, which sorts heterogeneously typed values by time and returns the first five. We can issue this query using the command-line tool `zq`:

```
zq "sort ts | head 5" sensor_data.zng
```

Figure 2 shows example results for this query. Zed can also search data, for example performing a full-text search over the nested data shown in Figure 3:

```
zq "search 'lab'" nested_sensor_data.zng
```

This query searches all string fields in the heterogeneous data, including the nested `room` field, and returns the first and third records. To make search easy, keyword searches are part of the Zed language and the `search` operator can be omitted when the search terms are unambiguous. For example, the query string above can be shortened to just `"lab"`. In addition, search keywords can be mixed with Boolean predicates, e.g., `"lab and ts > 19:00"`.

Zed can also leverage types for efficient analytics, as is commonly done with relational data today. For example, we can issue an analytics query over the data shown in Figure 3, formatted in columnar VNG:

```
zq "count() by loc.floor" nested_sensor_data.vng
```

This query counts the number of occurrences of each floor in the nested `location` records.

Though Zed's formats can mix heterogeneous data, it's still possible to organize data separately by type, either within a file or by using multiple files. This enables all of the performance optimizations of existing relational systems, while providing the flexibility during ingestion and querying of document-based systems. Thus Zed subsumes both approaches at the same time.

```
$ zq -f table "count() by typeof(this)" sensor_data.zng
typeof                                          count
<temperature={ts:time,temp:int32}>              452
<humidity={ts:time,percent_humidity:float32}>   82
<temperature={ts:time,temp:int32,unit:string}> 239
```

Figure 4: A data introspection query to count the number of records of each type in a ZNG file of IoT sensor data.

4.4 Data Introspection in Zed

Zed enables two main forms of introspection. First, users can *query for types* to view what types are present in a dataset. Figure 4 shows an example of counting the number of values with each type in an IoT dataset and outputting the results in a tabular format (with `-f table`). Second, Zed supports *querying by type*. For example, querying for records with `typeof(this)==<humidity>` selects all records with type `humidity`, effectively extracting a relational `humidity` table out of a collection of heterogeneous records. This enables Zed to provide the same functionality as relational systems, and to select groups of records with related types for further transformations or querying.

These data introspection operations seem quite simple, but are actually quite powerful, especially when applied to large datasets with hundreds of complex nested data types. Because all Zed data is typed, these operations work at any stage of processing—when data is generated, during ingestion, while stored, or during querying—thereby avoiding the catch-22 of existing relational and hybrid approaches. Because type information is encoded by data sources, the number of distinct types is much smaller than if you treated each combination of JSON fields as a distinct type, and more accurate than if you relied on schema inference. And finally, because types are first-class members of both the query language and data model, all of these introspection queries reuse the same query language and data model that are used by regular queries over data (§4.3).

Note that Zed does not eliminate the problem of data cleaning. As prior work has pointed out, a key part of data cleaning is resolving *semantic heterogeneity*, for example to determine if a `wages` field means the same thing in one dataset as another [64, 68]. Zed still requires that users resolve such questions. However, Zed's introspection capabilities can significantly ease data cleaning by providing visibility into the set of types present in a dataset. This enables users to focus on resolving the semantic heterogeneity between different types, rather than trying to infer which records correspond to the same type in the first place.

5 RESEARCH QUESTIONS

Several research questions must be answered in order to realize the full benefits of Zed:

How can Zed optimize query performance? Unlike existing databases and data-processing systems that leverage the relational model, Zed has not yet benefited from extensive engineering effort to optimize query performance. Nonetheless, when querying homogeneous data, Zed should be able to leverage many of the same techniques that enable high performance when querying existing relational databases and data-processing systems such as Spark [73], because Zed has comprehensive type information for all data values. In contrast, Zed raises new questions about how to optimize query performance over *heterogeneous but typed* data. What are the fundamental overheads of querying heterogeneous data compared

to homogeneous data? What techniques can Zed employ—either in its data formats or query engine—to leverage type information to reduce the overheads of querying heterogeneous data?

What type-based operators and functions should Zed support to ease data introspection, shaping, and cleaning? Exposing type information in a query language opens up new opportunities for easing the processing of eclectic data. Because existing systems lack a holistic data type abstraction, introspection, shaping, and cleaning rules are typically specified on a column-by-column basis today [47, 48]. For example, to `UNION` relational tables `A` and `B`, a user must first ensure that their schemas match by identifying each column that differs between `A` and `B` and individually dropping or adding columns or casting their types. Instead, Zed's data types enable a `shape` operator that abstracts away all this column-by-column logic, automatically identifying any differences in the two types and adding, removing, and casting columns as necessary so that the two types match:

```
zq "shape(this, <B>)" data_A.zng
```

This is one example of how a function can leverage type information to ease data shaping, but more exploration is necessary to develop a complete language for introspection, shaping, and cleaning.

How should type information be leveraged in a complete Zed data-processing system? In the simplest Zed deployment, all ingested data is written as either ZNG or VNG and users directly query this data. However, there are opportunities to improve upon this model. For example, Zed could cache type information such as the set of types and their frequencies so that data introspection queries could avoid scanning all the data. In addition, Zed could extend existing work that decides which data format to leverage for each individual query [28, 33, 43, 49, 66] with policies that incorporate type information.

6 CONCLUSION

Dozens of different data models and query languages have been proposed over the last 50 years, ranging from hierarchical and graphical approaches to semantic and object-oriented approaches [68]. Despite this, handling eclectic data is still challenging today, with popular solutions cobbling together a mixture of the relational and document models (§2). With Zed we explore a different approach, embodying both the document and relational approaches *at the same time for the same data*. Zed is under active development and many interesting research questions remain, but we believe that Zed offers a promising path towards simplifying and easing the processing of eclectic data.

ACKNOWLEDGEMENTS

We thank the entire Brim team for their work developing Zed. We also thank Joseph M. Hellerstein, Jamie Brandon, Garrison Hess, the CIDR reviewers, and our user community for their helpful feedback.

REFERENCES

[1] Apache avro. https://avro.apache.org.
[2] Apache drill. https://drill.apache.org.
[3] Apache iceberg. https://iceberg.apache.org.
[4] Apache kafka. https://kafka.apache.org.
[5] Apache orc. https://orc.apache.org.
[6] Apache parquet. https://parquet.apache.org.
[7] Couchbase. https://www.couchbase.com.
[8] Features to simplify semi-structured data management in the databricks lakehouse. https://www.databricks.com/blog/2021/11/11/10-powerful-features-to-simplify-semi-structured-data-management-in-the-databricks-lakehouse.html.
[9] Fivetran. https://fivetran.com.
[10] Informatica. https://www.informatica.com.
[11] Introducing json. www.json.org.
[12] The json data type. https://dev.mysql.com/doc/refman/5.7/en/json.html.
[13] Json schema. https://json-schema.org.
[14] Json types. https://www.postgresql.org/docs/9.4/datatype-json.html.
[15] Mongodb. https://www.mongodb.com.
[16] N1ql. https://www.couchbase.com/products/n1ql.
[17] neo4j. https://neo4j.com/.
[18] Oracle database. https://www.oracle.com/database.
[19] Partiql. https://partiql.org/.
[20] Postgresql: The world's most advanced open source relational database. https://www.postgresql.org.
[21] Protocol buffers: Developer guide. https://developers.google.com/protocol-buffers/docs/overview.html.
[22] Schema registry overview. https://docs.confluent.io/platform/current/schema-registry/index.html.
[23] Sql server. https://www.microsoft.com/en-us/sql-server.
[24] Sqlite. https://www.sqlite.org/index.html.
[25] Xml schema. https://www.w3.org/standards/xml/schema.
[26] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD*, 2008.
[27] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The lorel query language for semistructured data. *International journal on digital libraries*, 1997.
[28] I. Alagiannis, S. Idreos, and A. Ailamaki. H2o: a hands-free adaptive store. In *SIGMOD*, 2014.
[29] W. Y. Alkowaileet, S. Alsubaiee, and M. J. Carey. An lsm-based tuple compaction framework for apache asterixdb. In *VLDB*, 2020.
[30] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, et al. Asterixdb: A scalable, open source bdms. In *VLDB*, 2014.
[31] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. van Hovell, A. Ionescu, A. Łuszczak, et al. Delta lake: high-performance acid table storage over cloud object stores. In *VLDB*, 2020.
[32] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia. Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics. In *CIDR*, 2021.
[33] J. Arulraj, A. Pavlo, and P. Menon. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *SIGMOD*, 2016.
[34] M.-A. Baazizi, D. Colazzo, G. Ghelli, and C. Sartiani. Counting types for massive json datasets. In *DBPL*, 2017.
[35] M.-A. Baazizi, D. Colazzo, G. Ghelli, and C. Sartiani. Parametric schema inference for massive json datasets. In *VLDB*, 2019.
[36] M.-A. Baazizi, H. B. Lahmar, D. Colazzo, G. Ghelli, and C. Sartiani. Schema inference for massive json datasets. In *EDBT*, 2017.
[37] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of concise dtds from xml data. In *VLDB*, 2006.
[38] E. F. Codd. *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc., 1990.
[39] E. F. Codd. A relational model of data for large shared data banks. In *Software pioneers*, pages 263–294. Springer, 2002.
[40] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, et al. The snowflake elastic data warehouse. In *SIGMOD*, 2016.
[41] D. Deng, R. C. Fernandez, Z. Abedjan, S. Wang, M. Stonebraker, A. K. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, and N. Tang. The data civilizer system. In *CIDR*, 2017.
[42] M. DiScala and D. J. Abadi. Automatic generation of normalized relational schemas from nested key-value data. In *SIGMOD*, 2016.
[43] J. Dittrich and A. Jindal. Towards a one size fits all database architecture. In *CIDR*, 2011.
[44] R. C. Fernandez, Z. Abedjan, F. Koko, G. Yuan, S. Madden, and M. Stonebraker. Aurum: A data discovery system. In *ICDE*, 2018.
[45] V. Gadepally, P. Chen, J. Duggan, A. Elmore, B. Haynes, J. Kepner, S. Madden, T. Mattson, and M. Stonebraker. The bigdawg polystore system and architecture. In *HPEC*, 2016.
[46] R. Goldman, J. McHugh, and J. Widom. From semistructured data to xml: Migrating the lore data model and query language. 1999.
[47] K. Herrmann, H. Voigt, A. Behrend, and W. Lehner. Codel–a relationally complete language for database evolution. In *East European Conference on Advances in Databases and Information Systems*, pages 63–76. Springer, 2015.
[48] K. Herrmann, H. Voigt, A. Behrend, J. Rausch, and W. Lehner. Living in parallel realities: Co-existing schema versions with a bidirectional database evolution language. In *SIGMOD*, 2017.
[49] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang, et al. Tidb: a raft-based htap database. In *VLDB*, 2020.

[50] L. V. Lakshmanan, F. Sadri, and I. N. Subramanian. Schemasql-a language for interoperability in relational multi-database systems. In *VLDB*, 1996.

[51] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann. Mison: a fast json parser for data analytics. In *VLDB*, 2017.

[52] H. Lim, Y. Han, and S. Babu. How to fit when no one size fits. In *CIDR*, 2013.

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

[54] 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.

[55] J. Lu and I. Holubová. Multi-model databases: a new journey to handle the variety of data. In *CSUR*, 2019.

[56] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. In *VLDB*, 2010.

[57] K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The sql++ query language: Configurable, unifying and semi-structured. *arXiv preprint arXiv:1405.3631*, 2014.

[58] S. Palkar, F. Abuzaid, P. Bailis, and M. Zaharia. Filter before you parse: Faster analytics on raw data with sparser. In *VLDB*, 2018.

[59] M.-D. Pham, L. Passing, O. Erling, and P. Boncz. Deriving an emergent relational schema from rdf data. In *WWW*, 2015.

[60] R. Popplestone. The design philosophy of pop-2. *Machine Intelligence*, 1968.

[61] K. A. Ross. Relations with relation names as arguments: Algebra and calculus. In *PODS*, 1992.

[62] W. Spoth, B. S. Arab, E. S. Chan, D. Gawlick, A. Ghoneimy, B. Glavic, B. Hammerschmidt, O. Kennedy, et al. Adaptive schema databases. In *CIDR*, 2017.

[63] M. Stonebraker. Background. *Readings in Database Systems*, 2015.

[64] M. Stonebraker. Chapter 12: A biased take on a moving target: Data integration. *Readings in Database Systems*, 2015.

[65] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, et al. C-store: A column-oriented dbms. In *VLDB*, 2005.

[66] M. Stonebraker, C. Bear, U. Çetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. Zdonik. One size fits all? part 2: Benchmarking results. In *CIDR*, 2007.

[67] M. Stonebraker and U. Çetintemel. "one size fits all" an idea whose time has come and gone. In *ICDE*. 2005.

[68] M. Stonebraker and J. Hellerstein. What goes around comes around. *Readings in database systems*, 2005.

[69] C. Strachey. Fundamental concepts in programming languages. *Higher-order and symbolic computation*, 2000.

[70] L. Wang, S. Zhang, J. Shi, L. Jiao, O. Hassanzadeh, J. Zou, and C. Wangz. Schema management for document stores. In *VLDB*, 2015.

[71] C. M. Wyss and E. L. Robertson. Relational languages for metadata integration. *TODS*, 2005.

[72] C. M. Wyss and F. I. Wyss. Extending relational query optimization to dynamic schemas for information integration in multidatabases. In *SIGMOD*, 2007.

[73] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.