Sinew: A SQL System for Multi-Structured Data

Daniel Tahara
Yale University
daniel.tahara@yale.edu

Thaddeus Diamond
Hadapt
thaddeus@hadapt.com

Daniel J. Abadi
Yale University and Hadapt
dna@cs.yale.edu

ABSTRACT

As applications are becoming increasingly dynamic, the notion that a schema can be created in advance for an application and remain relatively stable is becoming increasingly unrealistic. This has pushed application developers away from traditional relational database systems and away from the SQL interface, despite their many well-established benefits. Instead, developers often prefer self-describing data models such as JSON, and NoSQL systems designed specifically for their relaxed semantics.

In this paper, we discuss the design of a system that enables developers to continue to represent their data using self-describing formats without moving away from SQL and traditional relational database systems. Our system stores arbitrary documents of key-value pairs inside physical and virtual columns of a traditional relational database system, and adds a layer above the database system that automatically provides a dynamic relational view to the user against which fully standard SQL queries can be issued. We demonstrate that our design can achieve an order of magnitude improvement in performance over alternative solutions, including existing relational database JSON extensions, MongoDB, and shredding systems that store flattened key-value data inside a relational database.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

Keywords

dynamic schema; SQL; RDBMS; NoSQL; JSON; MongoDB

1. INTRODUCTION

A major appeal of NoSQL database systems such as MongoDB, CouchDB, Riak, Cassandra, or HBase as the storage backend for modern applications is their flexibility to load, store, and access data without having to define a schema ahead of time. By removing this up-front data management effort, developers can more quickly get their application up and running without having to worry in advance about which attributes will exist in their datasets or about their domains, types, and dependencies.

Some argue that this 'immediate gratification' style of application development will result in long-term problems of code maintenance, sharing, and performance. However, proponents of the NoSQL approach argue that for rapidly evolving datasets, the costs of maintaining a schema are simply too high. Whichever side of the debate one falls on, in practice, the amount of production data represented using key-value and other semi-structured data formats is increasing, and as the volume and strategic importance of this data increases, so too does the requirement to analyze it.

Some NoSQL database systems support primitives that enable the stored data to be analyzed. For example, MongoDB currently provides a series of aggregation primitives as well as a proprietary MapReduce framework to analyze data, whereas other NoSQL databases, such as Cassandra and HBase, connect to Hadoop directly, leveraging Hadoop MapReduce and other execution frameworks such as Apache Tez to analyze data.

Unfortunately there are significant drawbacks to using any of the options provided by the NoSQL databases. Local primitives are a step away from the SQL standard, which renders a large number of third party analysis and business intelligence tools (such as SAP Business Objects, IBM Cognos, Microstrategy, and Tableau) unusable. Meanwhile, while there are several projects in the Hadoop ecosystem that provide a SQL interface to data stored in Hadoop (such as Hadapt, Hive, and Impala), they require the user to create a schema before the data can be analyzed via SQL, which eliminates a major reason why the NoSQL database was used in the first place. In some cases, a schema can be added after the fact fairly easily, but in many other cases significant processing, ETL, and cleaning work must be performed in order to make the data fit into a usable schema.

This paper describes the design of a layer above a traditional relational database system that enables standard SQL-compliant queries to be issued over *multi-structured* data (relational, key-value, or other types of semi-structured data) without having to define a schema at any point in the analytics pipeline. The basic idea is to give the user a logical view of a universal relation [13, 15] where a logical table exists that contains one column for each unique key that exists in the dataset. Nested data is flattened into sepa-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2376-5/14/06...$15.00
http://dx.doi.org/10.1145/2588555.2612183.

rate columns, which can result in the logical table having potentially hundreds or thousands of columns.

Since physically storing data in this way is impractical, the physical representation differs from the logical representation. Data is stored in a relational database system (RDBMS), but only a subset of the logical columns are materialized as actual columns in the RDBMS, while the remaining columns are serialized into a single binary column in the database system. These columns are then transparently rendered to the user by internally extracting the corresponding values from the serialized data.

The primary contribution of our work is a complete system architecture that enables a practical implementation of this vision. This architecture contains the following components: a relational storage layer, catalog, schema analyzer, column materializer, loader, query rewriter, and inverted text index. This paper describes the design, implementation, and interaction of these components.

We also build a prototype of our proposed system, and compare it against several alternatives for storing and analyzing multi-structured data: (1) MongoDB (2) a JSON extension of Postgres and (3) storing key-value pairs in entity-attribute-value 'triple' format in a table in an RDBMS. We find that our prototype consistently outperforms these alternatives (often by as much as an order of magnitude) while providing a more standard SQL interface.

2. RELATED WORK

Analytical systems that do not require a user to define a schema when data is loaded into the system generally fall into two categories: (1) systems that are specialized for non-relational data models and that do not have a SQL interface (2) more general systems that offer a SQL or SQL-like interface to the data at the cost of requiring the user to define the structure of the underlying data before data can be queried.

In exchange for programming convenience or flexibility, systems in the first category often require that the consumer of the data sacrifice performance on some subset of processing tasks and/or learn a custom query language. For example, Jaql offers a modular data processing platform through the use of higher-order functions, but despite its extensibility, it is not optimized for performing relational operations [6]. MongoDB, although it accepts any data representable as JSON, requires that the user learn its JavaScript-based query language, which does not natively support operations such as relational joins[1].

In the second category are systems that allow a user to define a schema on arbitrary, external data and query that data using a relational query engine. Most database systems support this functionality through external tables, with foreign data wrappers transforming data from a non-relational format to a relational format as it is read [17]. These external tables can also be indexed to improve performance or transparently loaded into the database system [4, 2]. Many SQL projects within the Hadoop ecosystem (such as Hive and Impala) use a similar concept, providing a relational query engine without a separate storage module. Data remains in Hadoop's filesystem (HDFS), and the user registers a schema for stored HDFS data with the SQL-on-Hadoop solution. After registering this schema, the user may issue queries, and data is read from HDFS and processed by the SQL-on-Hadoop solution according to this schema.

Although these systems are more flexible than systems that require a user to define a schema at load time, they are still limited by the requirement that, in order to issue SQL queries, the user must pre-define a target schema over which to execute those queries. Sinew drops this requirement, and automatically presents a logical view of the data to the user based on data rather than user input.

Google Tenzing [10], Google Dremel [16], and Apache Drill[2] offer the ability to query data through SQL without first defining a schema. Tenzing, which is a SQL-on-MapReduce system similar to Hive, infers a relational schema from the underlying data but can only do so for flat structures that can be trivially mapped into a relational schema. In contrast, Dremel and Drill (and Sinew) support nested data. However, the design of Drill and Dremel differ from Sinew in that Drill and Dremel are only query execution engines, designed purely for analysis. In contrast, Sinew is designed as an extension of a traditional RDBMS, adding support for semi-structured and other key-value data on top of existing relational support. With this design, Sinew is able to support transactional updates, storage-integrated access-control, and read/write concurrency control. Furthermore, since it integrates with an RDBMS, Sinew can also benefit from the RDBMS's statistics gathering and cost-based query optimization capabilities. This makes Sinew similar to Pathfinder [8], a processing stack designed to convert from XML and XQuery to relational tuples and SQL, but Sinew differs in that it is more broadly purposed to support any form of multi-structured data and explicitly attempts to provide a SQL interface to that data.

In addition to transparently providing a fully-featured relational interface to multi-structured data, Sinew introduces a novel approach to storing multi-structured data inside an RDBMS. One common approach, historically used by XML databases, is to create a 'shredder' that transforms documents into RDBMS records [5, 7, 14], often relying on some variation on the edge model in order to map documents into relational tuples [5]. However, even when the underlying relational schema is optimized for relational primitives such as projection and selection (e.g. by partitioning data into tables by attributes [14]), the performance of the systems is limited by the fact that reconstructing any single record requires multiple joins.

More recently, work examining the storage of large datasets generated by e-commerce and other online applications has suggested using a row-per-object data mapping. This mapping requires that objects be flattened into tuples corresponding to every possible key in the collection and uses wide (i.e. having many columns) tables for storing such data [1, 11]. Often, however, the data in question contain a significant number of sparse keys [3, 18], so the wide-table approach requires an RDBMS for which null values do not cause excessive space utilization or performance reduction. Column-oriented RDBMSs satisfy this criteria, but they run into difficulty in reconstructing nested objects because the objects themselves are not explicitly represented, only the sets of keys that appear in them. We will discuss the sparsity problem more extensively (and present a solution) in Section 3.1.1.

[1] http://docs.mongodb.org/manual/reference/

[2] https://cwiki.apache.org/confluence/x/sDnVAQ

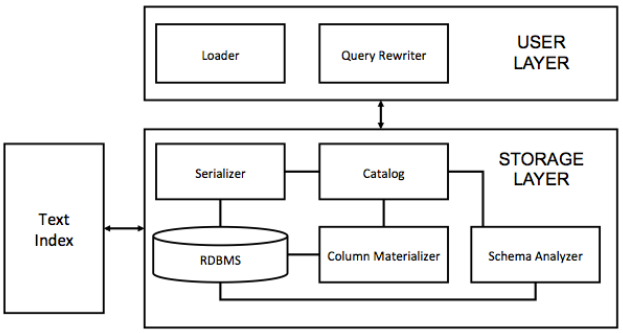


Figure 1: System Architecture

3. SYSTEM ARCHITECTURE

Fundamentally, Sinew differs from previous work in that it is a system that can both query and manipulate multi-structured data without requiring the user to define a schema at any point in the query process. Furthermore, it offers performant reads and writes by leveraging existing database functionality and delegating query optimization and execution to an underlying RDBMS whenever possible.

Although we rely heavily on an RDBMS in our design, we constrain the design to not require changes to the RDBMS code. This greatly expands the applicability of Sinew, allowing it to be used in concert with most existing relational database systems (preferably object-relational systems) instead of requiring a transition to a new database system. Sinew therefore should be thought of as a layer that sits above a database system.

The architecture of Sinew is shown in Figure 1. At the highest level, the system comprises the following components, which will be detailed in the remainder of this section:

- An RDBMS with a 'hybrid' physical schema (Section 3.1.1)
- A catalog (Section 3.1.2)
- A database schema analyzer (Section 3.1.3)
- A column materializer (Section 3.1.4)
- A loader (Section 3.2.1)
- A query rewriter (Section 3.2.2)
- An (optional) text index (Section 4.3)

For ease of discussion we will assume that data is input to Sinew in JSON format. In general, any data that is represented as a combination of required, optional, nested, and repeated fields is supported, even if the types vary across fields of the same name.

3.1 Storage Layer

3.1.1 Hybrid Schema

Given a collection of input data, which takes the form of documents comprising a set of potentially nested key-value pairs, Sinew automatically generates a logical, user-facing view and maintains it separately from the underlying schema of the data in RDBMS relations. We first discuss the logical view (an evolving schema that is created based on the data loaded thus far, against which the user can issue SQL queries) and then discuss its physical manifestation.

In the logical view, Sinew presents a universal relation where each document corresponds to one row. Each unique top-level key found in any document is exposed to the client as a column in the universal schema with the traditional


```
{
  "url":"www.sample-site.com",
  "hits": 22,
  "avg_site_visit": 128.5,
  "country": "pl"
}
```

```
{
  "date":"8/19/13",
  "ip":"123.45.67.89",
  "url":"www.sample-site2.com",
  "hits": 15,
  "owner": "John P. Smith"
}
```


Figure 2: Example key-value data

url	hits	avg_site_visit	country	date	ip	owner
www.sample-site.com	22	128.5	pl			
www.sample-site2.com	15			8/19/13	123.45.67.89	John P. Smith

Figure 3: User view of data from Figure 2

modes of access provided by SQL. Therefore, for each key-value pair from a document, the value will be (logically) stored in the row corresponding to the document and the (logical) column corresponding to the key. If the document contains a nested object, its subkeys are also referenceable as distinct columns using a dot-delimited name with the subkey preceded by the key of the parent object. As with other database primitives such as strings and integers, the nested object remains referenceable by the original key. (Arrays are less straightforward and are discussed in Section 4.2).

For example, given the dataset given in Figure 2, the user would have the view shown in Figure 3, and the following query:

```
SELECT url FROM webrequests WHERE hits > 20;
```

would return the set of all values associated with appearances of the key 'url' in objects with more than 20 'hits'.

There are two ways of storing attributes of the logical schema in the RDBMS, and we use the terms *physical* and *virtual* to describe the two cases. A physical column is any column in the logical view that is also stored as a physical column in the database system. In contrast, a virtual column is a column in the logical view that does not exist as a physical column in the database system; instead, it is accessed by runtime extraction from a serialized representation (described in Section 4.1) of the raw key-value pairs.

Accordingly, there are two extremes for mapping the logical schema to a physical schema: (1) storing all columns as physical columns or (2) storing all columns virtually. In (1), we have a physical database schema identical to the logical schema described above (i.e. a wide table containing one column for every unique key that exists in the data set), whereas in (2), we have a single-column table with the key-value pairs for each object serialized (as text or binary) and stored in that column (one object per row).

The all-physical column option offers a simpler system design but can run into problems handling datasets with many, potentially sparse attributes or those with large nested objects. For example, row-oriented RDBMSs allocate at least 1 bit of storage space for each schema attribute for every row in the table. This space is reserved either in the tuple header or in the body of the tuple with the rest of the data. This preallocated space per attribute (whether or not a non-null value for the attribute exists) can lead to storage bloat for sparse data, which can significantly degrade performance.

To better understand the problems of an all-physical approach, consider the present version of the Twitter API[3].

[3]https://dev.twitter.com/docs/platform-objects/tweets

#	Query
1	SELECT DISTINCT “user.id” FROM tweets;
2	SELECT SUM(retweet_count) FROM tweets GROUP BY “user.id”;
3	SELECT “user.id” FROM tweets t1, deletes d1, deletes d2 WHERE t1.id_str = d1.“delete.status.id_str” AND d1.“delete.status.user_id” = d2.“delete.status.user_id” AND t1.“user.lang” = ‘msa’;
4	SELECT t1.“user.screen_name”, t2.“user.screen_name” FROM tweets t1, tweets t2, tweets t3 WHERE t1.“user.screen_name” = t3.“user.screen_name” AND t1.“user.screen_name” = t2.in_reply_to_screen_name AND t2.“user.screen_name” = t3.in_reply_to_screen_name;

Table 1: Twitter Queries

The API specifies that tweets have 13 nullable, top-level attributes, which expand into 23 keys when fully flattened. Adding in nested user objects (which can optionally contain a tweet), hashtags, symbols, urls, user mentions, and media, the flattened version of the original tweet can contain upwards of 150 optional attributes. If we attempted to store this representation in InnoDB, a popular storage engine for MySQL, this would amount to at least 300 bytes of additional storage overhead per record (InnoDB headers include 2 bytes per attribute[4]). For a minimal tweet (just text with no additional entities or metadata), this header overhead can actually be larger than the size of the data itself. Even in RDBMSs with efficient NULL representations (such as Postgres, which uses a bitmap to indicate the presence of NULLs), there is a non-negligible system cost to keeping track of sparse data. Therefore, as a practical limitation, and to simplify the design of the catalog, most row-oriented RDBMSs place a hard limit on the number of columns that a schema may declare.

Column-oriented database systems do not have the same storage bloat problem for wide, sparse data [1] and can flatten data to a larger degree than row-stores. However, a fully flattened physical representation is often not optimal. Referring again to the Twitter API, a common query pattern might be to retrieve the ‘user’ who posted a given tweet (Twitter nests an entire user object as an attribute of a tweet). In a fully flattened representation, parent objects of nested keys no longer exist explicitly; rather, they are the result of merging all columns whose attribute names are an extension of the desired parent object. In order to return the parent object, the system must first compute the proper set of columns and then pay the cost of merging them together. For nested keys that are frequently accessed together, it is better to store them as a single collection than as individual elements.

Given the sparsity overhead of the “all-physical-column” approach, one may be tempted to go to the other extreme, the “all-virtual-column” approach described above. Although the cost of key-value extraction from the column containing the serialized data can be kept small (see Section 4.1 and Appendix B), storing all attributes serialized within a single column degrades the ability of the RDBMS optimizer to produce efficient query plans. This is because, given our stipulation of not modifying underlying database code, the optimizer cannot maintain statistics on an attribute-level. As far as the optimizer is concerned, virtual columns do not exist.

[4]http://dev.mysql.com/doc/refman/5.5/en/innodb-table-and-index.html#innodb-physical-record

#	Column	With Virtual Column	With Physical Column
1	user.id	HashAggregate	Unique
2	user.id	HashAggregate	GroupAggregate
3	user.lang	1. Merge join: d1 = d2 2. Filter 3. Merge join: t1 = d1	1. Filter 2. Merge join: t1 = d1 3. Merge join: d1 = d2
4	user.screen_name	1. Merge join: t2 = t3 2. Merge join: t1 = t2	1. Merge join: t2 = t3 2. Hash join: t1 = t3

Table 2: Effect of Virtual Columns on Query Plans

_id	key_name	key_type
1	url	text
2	hits	integer
3	avg_site_visit	real
4	country	text
5	ip	text
6	owner	text

(a)

_id	count	materialized	dirty
1	2	t	f
2	2	t	f
3	1	f	t
4	1	f	t
5	1	f	t
6	1	f	t

(b)

Figure 4: Example Catalog

To demonstrate the potential differences in query plans, we performed the queries listed in Table 1 over a set of 10 million tweets from Twitter. Each tweet has the attributes described above, the sparsity of which vary between less than 1% all the way up to 100%. (See Section 6 for our experimental setup). The query plans generated by the optimizer in both conditions are presented in Table 2. These plans contain differences in the operators used for the UNIQUE and GROUP BY queries, and also differences in the JOIN order for both join queries. The differences can be attributed to the fact that the optimizer assumes a fixed selectivity for queries over virtual columns (200 rows out of 10 million in these experiments). In cases when the selectivity is in fact much lower (i.e. more tuples match a given predicate), the resulting query plan will be suboptimal, which can have significant performance implications depending on the dataset and system configuration. For example, the self-join saw an order of magnitude improvement when querying over a physical column versus over a virtual one, with an originally 50-minute query completing in just over 4 minutes.

In order to take advantage of the performance benefits of the column-per-attribute mapping as well as the space efficiency and extensibility of single-column serialization, we opt for a combination of the two mappings. Under our *hybrid schema*, we create columns for some attributes and store the remainder in a special serialized column that we now refer to as the *column reservoir*. This allows us to attain the benefits of leveraging physical columns in the database system when they are most helpful, while keeping the sparsest and least frequently accessed keys as virtual columns.

3.1.2 Catalog

In order to maintain a correct mapping between the logical and physical schemas and facilitate optimizations in the rest of the system, Sinew carefully documents attribute names, types, and methods of storage (physical or virtual column). This metadata is kept in a *catalog*, which records the following information:

- What keys have been observed
- Key type information that has been derived from the data
- The number of occurrences of each key

- Whether the column is physical or virtual
- A 'dirty' flag (discussed in Section 3.1.4)

In practice, this catalog is divided into two parts. The first, as shown in the example in Figure 4(a) contains a global list of attributes appearing in any document across all relations as a set of *id, name, type* triples. This global table aids data serialization (described in Section 4.1) by serving as the dictionary that maps every attribute to an ID, thereby providing a compact key representation whenever a particular attribute must be referred to inside the storage layer. The second part of the catalog (see Figure 4(b)) is maintained on a per-table basis (instead of globally across tables) and contains the rest of the information mentioned above.

With this information, Sinew is able to identify both the logical schema and current physical schema, enabling the query transformer to resolve references to virtual columns into statements that match the underlying RDBMS schema. The statistics (both the number of unique keys and number of occurrences of each key), are used by the schema analyzer (described in Section 3.1.3) to dynamically make decisions about which columns to materialize and which to leave in the column reservoir. We discuss this interaction below.

3.1.3 Schema Analyzer

In order to adapt to evolving data models and query patterns, a schema analyzer periodically evaluates the current storage schema defined in the catalog in order to decide the proper distribution of physical and virtual columns. The primary goal in selecting which columns to materialize as physical columns is to minimize the overall system cost of materialization and associated system overhead of maintaining tables with many attributes, while maximizing the corresponding increase in system performance.

Dense (i.e. frequently appearing) attributes and those with a cardinality that significantly differs from the RDBMS optimizer's default assumption are good candidates for materialization. A simple threshold is used for both cases. Attributes with a density above the first threshold or with a cardinality difference above the second threshold are materialized as physical columns, while the remaining attributes are left as virtual columns.

As new data is loaded into the system, the density and cardinality characteristics of columns may change. Therefore, the schema analyzer also checks already materialized columns to see if they are still above threshold. If not, they are marked for dematerialization.

3.1.4 Column Materializer

The column materializer is responsible for maintaining the dynamic physical schema by moving data from the column reservoir to physical columns (or vice-versa). Our goal in the design of the materializer is for it to be a background process that is running only when there are spare resources available in the system. A critical requirement necessary to achieve this goal is for materialization to be an incremental process that can stop when other queries are running and pick up where it left off when they finish and resources become free. Therefore, our design does not force a column to be materialized in entirety—rather, some values for a key may exist in the reservoir while others exist in the corresponding physical column. We call such a column *dirty* and ensure that the *dirty bit* in the catalog is set for that column in this situation. When the dirty bit is set, both the physical column and the reservoir must be checked for values for a particular key for any query that accesses that key (this is done via the COALESCE function—see Section 3.2.2).

The materializer works as follows. Whenever the schema analyzer decides to turn a virtual column into a physical column or vice-versa, it sets the dirty bit for that column to true inside the catalog in anticipation of data movement. Periodically, the materializer polls the catalog for columns marked as dirty, and for any such column, it checks the catalog to see if the column is now supposed to be physical or virtual (this determines the direction of data movement). Then, it iterates row-by-row, and for any row where it finds data in the reservoir when it is supposed to be in a physical column (or vice-versa) it performs an atomic update of that row (and only that row) to move the value to its correct location. The materializer and loader are not allowed to run concurrently (which we implement via a latch in the catalog), so when the iteration reaches the end of the table, it can be guaranteed that all data is now in its correct location. The materializer then sets the dirty bit to false, and the process is complete.

As mentioned above, the important feature of this design is that although each row-update is atomic, the entire materialization process is not. At any point, the materializer can be stopped and queries processed against the partially materialized column. These queries run slightly slower than queries against non-dirty columns, due to the need to add the COALESCE function to query processing. The precise slowdown is dependent on the how the underlying database system implements COALESCE. In our PostgreSQL-based implementation (see Section 5), we observed a maximum slowdown of 10% for queries that access columns that must be coalesced. For disk-bandwidth limited workloads, we observed no slowdown at all.

3.2 User Layer

3.2.1 Loader

A bulk load is completed in two steps, serialization and insertion. In the first step, the loader parses each document to ensure that its syntax is valid and then serializes it into the format described in Section 4.1. As the serialization takes place, the loader aggregates information about the presence, type, and sparsity of the keys appearing in the dataset and adds that information to the catalog. More precisely, for every key-value pair that is loaded, the loader infers the data type and looks up the resulting key and type (the combination of which we call an *attribute*) in the catalog to get its attribute ID. If the attribute does not exist in the catalog, the serializer inserts it into the catalog, receives a newly generated ID, and serializes the key-value pair into the column reservoir along with the rest of the data. Thus, virtual columns for new keys are created automatically at load time during serialization, and the cost of adding a new attribute to the schema is just the cost to insert the new attribute into the catalog during serialization the first time it appears in the dataset (an invisible cost to the user).

On insertion, all of the serialized data gets placed into the column reservoir regardless of the current schema of the underlying physical relation. Sinew then sets the dirty flag in the catalog to true for all affected columns, and their data are eventually moved to the appropriate physical columns when the column materializer notices the dirty bit and ma-

terializes the newly loaded data. This design decision is motivated by the desire to keep the system components as modular as possible. By always loading into the column reservoir, the loader does not need to be aware of the physical schema, and does not need to interact with the schema analyzer and column materializer components of the system.

3.2.2 *Query Rewriter*

Sinew's hybrid storage solution necessitates that queries over the user-facing, logical schema be transformed to match the underlying physical schema. Therefore, Sinew has a query rewriter that modifies queries before sending them to the storage layer for execution. Specifically, after converting a given query into an abstract syntax tree, the rewriter validates all column references against the information in the catalog. Any column reference that cannot be resolved, whether because it refers to a virtual column or because it refers to a dirty, physical column, gets rewritten. For example, given the query:

```
SELECT url, owner
FROM webrequests
WHERE ip IS NOT NULL;
```

the reference to the virtual column, 'owner,' will be transformed to a function that extracts the key from the column reservoir based on the serialization format chosen by the system (we give a sample implementation of one such function in Section 4.1):

```
SELECT url, extract_key_txt(data, 'owner')
FROM webrequests
WHERE ip IS NOT NULL;
```

In the case when 'owner' is dirty (i.e. not fully materialized), the column reference will be transformed instead as a SQL COALESCE over the physical column and key extraction:

```
SELECT url, COALESCE(owner, extract_key_txt(data, 'owner'))
FROM webrequests
WHERE ip IS NOT NULL;
```

In addition to the desired key, the extraction function takes a type argument (the above is syntactic sugar for passing 'text' as an argument), which is determined dynamically by the query rewriter based on type constraints present in the semantics of the original query. The extraction function then applies to only those values of the correct type. This behavior allows Sinew to elegantly handle situations where the same key corresponds to values of multiple types—rather than throwing an exception for type mismatches (e.g. if the value is an argument to a function that expects an integer), it will instead selectively extract the integer values and return NULL for strings, booleans, or values of other types. On the other hand, in the common case where the expected type of an attribute cannot be determined from the query semantics (e.g. the case of a projection), the function will simply return the value downcast to a string type.

4. ENHANCEMENTS

4.1 Custom Serialization Format

There are a number of options for storing serialized object data, but most are not well suited to performing common RDBMS operations. One approach is to keep the data in its original string form and store it a single text field. This makes loading trivial (no transformation needs to be performed prior to the load), but manipulating the data is expensive because the system must convert it to a logical representation before performing any computation.

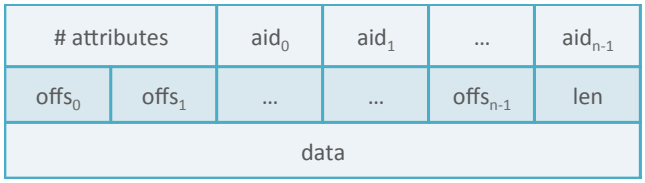


Figure 5: Serialization Format

An alternative approach is to use a serialization format such as Apache Avro or Google Protocol Buffers, which represent objects as blocks of binary, rather than text. In general, both formats eliminate the syntactic redundancy of human-readable text representations and offer faster iteration over the individual keys, in particular by memoizing the schema of the serialized data. However, both formats, like JSON, are 'sequential', meaning that random reads on the original data are not supported. In order to extract a single key from a given datum, the application must either (1) deserialize the entire datum into a logical form and then dereference the appropriate attribute or (2) read through the serialized datum one attribute at a time until reaching the desired attribute or the end of the datum (if the attribute does not exist).

Since attribute extraction (projection in relational algebra) is an extremely common operation for SQL queries, we decided to use a custom serialization format instead of either of the above, which are focused more on data transfer and platform independence than analytics. In particular, our format reduces the cost of extracting attributes stored in a serialized object by allowing random reads on the data. We explore the exact performance differences between Sinew's format, Avro, and Protocol Buffers in Appendix A.

Much like a standard RDBMS tuple, our serialization format has a header that includes object metadata and a body that holds the actual data. Specifically, the header, the structure of which is shown in Figure 5, is composed of an integer indicating the number of attributes present in the record, followed by a sorted sequence of integers corresponding to the attribute IDs (as specified by the catalog) of the keys present. After the list of attribute IDs is a second series of integers indicating the byte offset of each attribute's value within the data. The body contains the binary representation of the actual data.

By separating the document structure from its data, our serialization format enables Sinew to quickly locate a key or identify its absence without reading the entirety of the serialized data. To find a key, Sinew simply needs to check the list of attribute IDs, and if the search ends up empty, it concludes that the key does not exist in the document. If it finds the key, it will look up the offset in the list of offsets, and jump to the correct location in the data to retrieve the typed value. We chose to separate the list of keys from the list of offsets (rather than including offset information right next to the key) in order to maximize cache locality for binary searches for attribute IDs within the header.

Key extraction is straightforward. Given a desired key, the extraction module retrieves the corresponding attribute ID and type information from the dictionary in the catalog. For each record, it then performs a binary search on the attribute ID list in the header and, if the ID is present, retrieves its

offset from the list that follows the attribute IDs. With the offset information combined with the offset information of the next attribute, the module can compute the attribute length and retrieve the value in its appropriate type. The cost of doing a read is $O(\log n)$ in the number of attributes present in a given datum, since it does a binary search in the header for a reference to the offset and length of the desired value. Because of this, it will perform significantly better than the other aforementioned serialization formats which have a worst-case cost of $O(n)$. The attribute search also has better constant factors due to the cache benefits of storing all attribute IDs consecutively.

It is worth noting that column-oriented serialization formats such as RCFiles or Parquet permit random access and could therefore be used to serialize the data in the column reservoir. However, given the hybrid design of our storage solution where some attributes are stored in physical columns and others serialized in a reservoir column, the reservoir should match the orientation of the physical columns. Hence, if Sinew's underlying database system is a column-store, RCFiles or Parquet may be used instead of our custom serialization format. However, if the underlying database system is a row-store, a column-oriented data format does not integrate well with a per-row reservoir, and our custom, object-centric serialization format should be used.

4.2 Nested Objects and Arrays

In our hybrid storage schema, nested objects and arrays can cause performance bottlenecks even if they are materialized since the contents of the collections remain opaque to the optimizer. We explore a few techniques for improving the physical representation of nested collections in this section.

For nested objects, there are a few alternatives. Although Sinew will catalog the sub-attributes of any nested object that is materialized and mark them for materialization if necessary, a fully flattened data representation is not necessarily optimal (as discussed in Section 3.1.1). Therefore, while Sinew defaults to a single table containing one document per row (using a universal relation schema), the system does allow some relaxation of this extreme. If there are logical groups that can be formed within the set of documents (e.g. in the case of nested objects), the user can specify that these be put in separate tables and joined together at query time.

In the case of arrays, the user can opt for one of a variety of options depending on the significance of the array syntax (e.g. unordered set, ordered list, etc.). By default, the system stores the array as an RDBMS array datatype, but if the number of elements in the array is fixed (and small), it can instead store each position in the array as a separate column (as suggested by Deutsch et. al. [12]). This mechanism can offer significant performance improvements for array containment and other predicates, since the predicates reduce to trivial filters over the external table.

Alternatively, if the array is intended to be an unordered collection or if it comprises a list of nested objects, the user can specify that the array elements be stored in a separate table as tuples of the form *parent object id, index, element*. Maintaining a separate table not only decreases the complexity of cataloging, but also ensures that Sinew maintains aggregate statistics on the collection of array elements rather than segmenting those statistics by position in the array. Furthermore, when the array is a collection of nested objects, the 'element' can be divided into separate columns, one for each attribute within the nested object. For situations in which these nested objects are homogeneous, this helps Sinew to create a more optimal physical schema and in turn, offer better query performance.

4.3 Inverted Index and Text Search

Since most implementations of text indexes include mechanisms to offer performant range queries, partial matching, and fuzzy matching, we can further enhance Sinew's performance and the expressivity of its queries by including an external text index over the data stored in the RDBMS.

Inverted indexes are particularly useful for queries over virtual columns. At a high level, an inverted text index tokenizes the input data and compiles a vector of terms together with a list of IDs corresponding to the records that contain that term. Additionally, it can give the option of faceting its term vectors by strongly typed *fields*. Sinew leverages this functionality by associating a field with each attribute in its catalog and rewriting predicates over virtual columns into queries of the text index. The results of the search (a set of matching record IDs) can then be applied as a filter over the original relation.

Although our main motivation for incorporating inverted indexes was to speed up evaluation of queries containing standard SQL WHERE clause predicates on virtual columns, Sinew also uses them to support text search over the entire data set. Users can search for text that may appear in any column (physical or virtual), and Sinew leverages the inverted indexes to find the set of rows that contain the text.

Not only does the full text search capability enable Sinew to offer a more expressive set of predicates on semi-structured and relational data, but it also allows Sinew to handle completely unstructured data alongside that data by simply storing the unstructured data in a generic text column and providing access to it through the text indexes. However, because unstructured data has no analogy for relational attributes (unlike semi-structured data, whose keys can correspond to attributes), the interface to this unstructured data is not 'pure' SQL; rather, Sinew includes a special function which can be invoked in the WHERE clause of a SQL statement and takes two parameters: (1) the keys over which the search should be performed ('*' means all keys) and (2) the search string.

A sample query is shown below:

```
SELECT *
FROM webrequests
WHERE matches('*', "full text query or regex");
```

5. IMPLEMENTATION

Although Sinew is RDBMS-agnostic, we use Postgres as the underlying RDBMS for our experiments, since it has a history of usage in the literature and is therefore a good reference point for a comparative evaluation. Furthermore, as mentioned in Section 3.1.1, Postgres's efficient handling of null values[5] makes it particularly well-suited for the task of storing sparse data. Each tuple stored by Postgres varies in size based on the attributes actually present, rather than

[5]Postgres uses a bitmap to indicate which attributes are null for a given tuple, so a null value occupies just a single bit rather than the entire width of a column.

being stored in a fixed, pre-allocated block based on the schema.

Both the schema analyzer and column materializer are implemented as Postgres background processes. The management of the background process is delegated entirely to the Postgres server backend, which simplifies the implementation but does not impact functionality.

The data serialization is implemented through a set of user-defined functions (UDFs) to convert to and from JSON, as well as functions to extract an individual value corresponding to a given key (see Section 3.2.2). Implementing the serialization using UDFs does not impose a high performance cost (we quantify this cost in Appendix B) and allows Sinew to push down query logic completely into the RDBMS. Although the ability to define UDFs is a feature of most modern RDBMSs, some systems do not support UDFs. For those systems, Sinew can perform serialization and key extraction completely outside of the RDBMS (at reduced performance). Therefore, the use of UDFs is an implementation detail and does not reduce the generality of our overall design.

For the external text index, we use Apache Solr, a search index that is highly optimized for full text search and data discovery. The Solr API is exposed to Postgres using a UDF that returns the Postgres row IDs corresponding to the records that match a given Solr query in a specific table. The result set is then applied as a filter on the original table as part of the execution of the query.

6. EXPERIMENTS

Our experiments compare performance of Sinew versus popular NoSQL solutions and RDBMS-based alternatives for querying and manipulating multi-structured data. We ran all 11 queries from the NoBench NoSQL benchmark suite developed at the University of Wisconsin [9][6]. We chose this suite as opposed to developing our own benchmark because it comprises a fairly diverse analytics workload, and is therefore a good reference point for future systems. However, NoBench does not include update tasks, so we added a random update task to the NoBench suite (see Section 6.6) in order to evaluate the full read-write capabilities of the benchmarked systems. For our experiments, we used NoBench to generate two datasets of 16 million and 64 million records (10GB and 40GB, respectively). Each record has approximately fifteen keys, ten of which are randomly selected from a pool of 1000 possible keys, and the remainder of which are either a string, integer, boolean, nested array, or nested document. Two dynamically typed columns, dyn1 and dyn2, take either a string, integer, or boolean value based on a distribution determined during data generation.

Our benchmarking system has a 3.6 GHz, quad-core, Intel Xeon E5-1620 processor with 32 GB of memory and 128 GB of solid-state storage. We observed read speeds of 250-300MB/s. We executed each of the 12 queries (11 from NoBench plus the update task) 4 times and took the average of the results. All queries were performed with warmed caches to simulate a typical analytics environment where analysis is run continuously; therefore, queries over the small data set (16 million records) are never I/O bottlenecked (since the data fits entirely in memory). The larger data set (64 million records) is larger than memory, and queries over this dataset can potentially be I/O limited if the I/O cost of bringing in the table from storage outweighs the processing CPU cost.

[6]The full set of queries can be found on page 9 of the extended paper: http://pages.cs.wisc.edu/~chasseur/argo-long.pdf.

6.1 Benchmarked Systems

We evaluated the performance of Sinew versus three alternatives: MongoDB, a shredding system using the Entity-Attribute-Value model, and Postgres with JSON support.

Sinew: Our experimental version of Sinew is built on top of Postgres version 9.3 beta 2, and our installation preserves the default configuration parameters, including a 128 MB limit on shared memory usage. As described in Section 5, we installed the remaining features of Sinew as Postgres extensions. Although our integration of Solr into Sinew is complete, and Sinew therefore supports the full range of text indexing described in Section 4.3, we chose not to use text indexes for this benchmark since we are primarily interested in the evaluating queries that do not involve full text search.

The column materialization policy was simple: a column was marked for materialization if it was present in at least 60% of objects and had a cardinality greater than 200. This policy resulted in materialization for str1, num, nested_array, nested_object (itself a serialized data column), and thousandth. The other ten keys, including the dynamic and sparse keys remained as virtual columns.

MongoDB: MongoDB is the most popular NoSQL database system[7], deployed in hundreds of well-known production systems[8]. For our benchmarks we ran MongoDB 2.4.7 with a default configuration. Notably, MongoDB does not restrict its memory usage, so it was not uncommon to see upwards of 90% memory usage during execution of our queries.

Entity-Attribute-Value (EAV): A common target for systems that shred XML, key-value, or other semi-structured data and store them in an RDBMS is the EAV model [3]. Under this model, each object is flattened into sets of individual key-value pairs, with the object id added in front of each key value pair to produce a series of *object id, key, value* triples (the object id is referred to as an 'entity' and the key as an 'attribute').

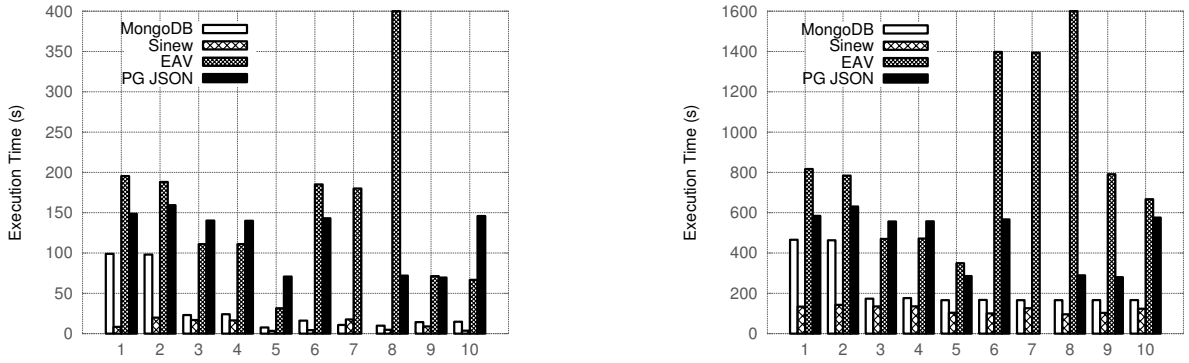
It is therefore fairly straightforward to store multi-structured data in a relational database using the EAV model [9]. By adding a mapping layer on top of an RDBMS, we can translate queries over specific attributes into queries over the underlying table, which, in the case of our implementation, is a 5-column relation of object id, key name, and key value (with one column for each primitive type, string, numerical, and boolean). As with Sinew, the EAV prototype runs on Postgres 9.3 beta 2 with the same system configuration.

Postgres JSON: Starting with Postgres 9.3, Postgres includes support for a JSON datatype, including built-in functions for performing key dereferences and other data manipulations. Given that this is built-in functionality and that we built Sinew on top of Postgres, we felt this was an important point of comparison to demonstrate what effect our architecture and optimizations would have on performance. Additionally, Postgres JSON is representative of commercial systems such as IBM DB2, which recently added support for JSON as an extension to its core RDBMS functionality[9]. The installation was identical to the installation for Sinew.

[7]http://db-engines.com/en/ranking

[8]http://www.mongodb.org/about/production-deployments/

[9]http://www.ibm.com/developerworks/data/library/techarticle/dm-1306nosqlforjson1



(a) 16 Million Records **(b) 64 Million Records**

Figure 6: NoBench Query Performance (Q1-Q10)

	16 million records		64 million records	
System	Load (s)	Size (GB)	Load (s)	Size (GB)
MongoDB	522.24	10.1	2170.13	40.9
Sinew	527.79	9.2	2155.12	33.0
EAV	1835.18	22.0	9910.87	87.0
PG JSON	284.11	10.2	1420.86	42.0
Original		10.5		38.1

Table 3: Load Time and Storage Size

6.2 Load and Data Size

Before executing the NoBench queries, we measured both load time and database size for each system. Our results are summarized in Table 3. Postgres JSON loads faster than the three other systems primarily because it only does simple syntax validation during the load process whereas each of the other three systems require some sort of data transformation. For MongoDB, this transformation is to BSON, for Sinew, to our custom serialization, and the EAV system, to the 5-column schema described above (which required an average of over 20 new tuples per record).

For data size, Sinew's data representation is the most compact, since the replacement of keys with key identifiers in the header serves as a type of dictionary encoding. Postgres JSON does not transform the JSON data, and so is approximately the same size as the input. MongoDB states in its specification that its BSON serialization may in fact increase data size because it adds additional type information into its serialization, and we observed this on our 64 million record data set. The EAV system is significantly larger than either of the previous three, since its representation requires one tuple per flattened key, with not only the key name and value, but also a reference to a parent object. For our two NoBench datasets of 16 and 64 million objects, this resulted in 360 million and 1.44 billion tuples in the underlying table.

It should be noted that neither Postgres nor MongoDB support traditional database compression without additional configuration. Therefore the data sizes reported in Table 3 are only the consequence of the data transformations and no further compression. It is reasonable to expect that compressing data would reduce the data size of all four systems.

6.3 Projections

Queries 1 through 4 are basic column projections over common top-level keys, common nested keys, and sparse keys. As Figure 6a shows, on the 16 million record dataset, Sinew outperforms both the Postgres JSON-based solution and the Entity-Attribute-Value data transformation by an order of magnitude. Postgres JSON stores JSON data as raw text. Therefore it must execute a significant amount of code in order to extract the projected attributes from the string representation, including parsing and string manipulation. In fact, the CPU cost of dereferencing a JSON key in the Postgres JSON implementation is so high that these projection queries (and the selection queries in the next section), which should be I/O bound due to their simplicity, are in fact CPU bound. This was verified when we performed the same query with cold caches and observed an identical execution time. In contrast, Sinew's binary representation of data (as described in Section 4.1 and further explored in Appendices A and B), is optimized for random attribute access and requires less extraction effort at query time.

The EAV system performs poorly because it adds a join on top of the original projection operation in order to reconstruct the objects from the set of flattened EAV tuples.

Sinew also performs these projection operations faster than MongoDB. The difference is an order of magnitude for queries 1 and 2 (projections over keys present in every object) and significant, but smaller for queries 3 and 4 (projections over sparse keys appearing in about 1% of objects). From these results, we draw two conclusions. First, despite the fact that BSON is a binary serialization, there is still a significant CPU cost to extracting an individual key or set of keys from a BSON object. For queries 1 and 2, this extraction cost must be paid for every object, but for queries 3 and 4, this cost must be paid only for the 1% of objects that actually contain the key. Second, checking whether or not a key exists in BSON is significantly faster than extracting the key (hence MongoDB's improved relative performance for projection over sparse columns), but is still slower than the equivalent operations over Sinew's storage.

The results are similar for the larger, 64 million record (40 GB) dataset, when the data can no longer fit into main memory. Although the speedups for Sinew are no longer an order of magnitude, it is clear that projection operations in the three other systems have significant CPU costs, while Sinew queries become I/O bound. Whereas the query time for Sinew increases by about a factor of 10, the others saw only an approximately linear increase in execution time relative to the number of additional records.

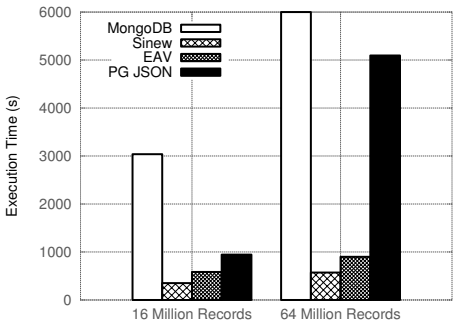


Figure 7: Join (NoBench Q11) Performance

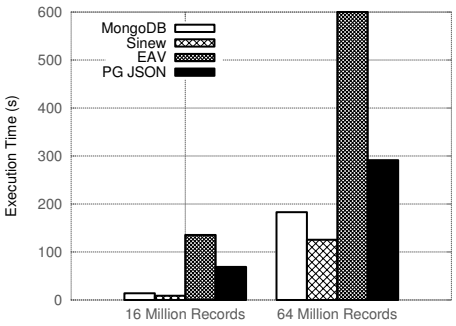


Figure 8: Random Update Performance

6.4 Selections

Queries 5 through 9 each select objects from the database on the basis of either an equality, range, or array containment predicate. Once again, we see similar performance differences among the systems, with more than an order of magnitude improvement in performance for Sinew and MongoDB when compared to the Postgres JSON and the EAV system and with Sinew outperforming MongoDB by between 40 and 75% (with one exception explained below).

There are two interesting results for Query 7 (range predicate on the multi-typed key, 'dyn1'). First, for the smaller of the two datasets, MongoDB outperforms Sinew by about 40%. Whereas Postgres rewrites the BETWEEN predicate as two comparisons (>=, <) without precomputing the value and substituting the result into both comparisons, MongoDB appears to precompute the value before applying the comparison operators. This saves the cost of one deserialization per record. For the larger dataset, both Sinew and MongoDB are I/O bound, and since Sinew's data representation is about 25% more compact than BSON, it takes less time to read the data from disk and thus less time to perform the query.

The second notable aspect of Query 7 is that it cannot be executed in the Postgres JSON system, because extracting a key corresponding to multiple types is not valid within Postgres's grammar. That is, since the JSON extraction operator in Postgres returns a datum of the 'JSON' datatype rather than a string, integer, float, or boolean, the datum must be type-cast before being used in another function or operator. Since Postgres raises an error if it encounters a malformed string representation for a given type (e.g. 'twenty' for an integer), the query will never complete if a key maps to values of two or more distinct types (except for projection, which simply returns the result). Although it is technically possible to return the values in Postgres's generic JSON-text datatype and then apply a function to filter out values of the desired type after the fact, the operation not only requires additional user code, but also digging into the rewrite phase of Postgres's abstract syntax tree generation in order to provide the function information about the type expected.

Neither Query 8 nor Query 9 completed on the EAV system because each ran out of disk space when attempting to execute the query. The times shown are the points at which the system terminated execution.

6.5 Joins and Aggregations

Queries 10 and 11 evaluated the performance of a GROUP BY and JOIN operation across the four systems, respectively. The results for Query 10 are alongside the results for Queries 1-9 in Figures 6a and 6b, and the results for Query 11 are given separately (for reasons discussed below) in Figure 7. The results for Query 10 resemble the results for the previous 9 queries with the exception of Postgres JSON, which lags behind even the EAV system. Despite JSON being a built-in type, the Postgres optimizer has no understanding of, or statistics on, individual keys stored within the JSON objects. Therefore, when it produces a query plan for the GROUP BY, it miscalculates the selectivity of the GROUP BY predicate and produces a sub-optimal query plan (HashAggregate instead of a sort). Sinew avoids this pitfall by selectively and automatically materializing columns, which provides the optimizer with a more correct view of the overall data.

For Query 11, Sinew is again the fastest of the SQL options. However, unlike the previous queries, MongoDB lags far behind each of the other three systems and is an order of magnitude slower than Sinew. MongoDB has no native join support, so the query must be implemented in user code using a custom JavaScript extension combined with multiple explicitly defined intermediate collections. The execution is thus not only slow, but also uses a significant amount of disk. In the case of the 64 million record dataset, the query required so much intermediate storage that it could not complete on our benchmark systems. The EAV system also could not complete the join for lack of adequate disk space. As for Query 9 (Section 6.4), the times shown in the graph are the points at which the system terminated execution, which make it clear that even discounting the lack of disk space, MongoDB and the EAV system lag significantly behind Sinew's performance.

6.6 Updates

As mentioned above, we added a random update task to the NoBench benchmark in order to evaluate the full read-write capabilities of our system. In particular, we ran the following statement which affects approximately 1 in 10000 records and updates one of the sparse keys generated by NoBench:

```
UPDATE test
SET sparse_588 = 'DUMMY'
WHERE sparse_589 = 'GBRDCMBQGA======';
```

Figure 8 shows the results of this experiment. MongoDB does not provide transactional semantics, and therefore has to keep fewer guarantees relative to the PostgreSQL-based systems. Hence, we expected MongoDB to perform the best for this task. However, as explained above, MongoDB's predicate evaluation is 40% slower than Sinew's. The addi-

tional overhead of the predicate evaluation associated with this update task outweighed Sinew's overhead to maintain transactional guarantees, and therefore Sinew ended up outperforming MongoDB for this task.

Of the RDBMS-based solutions (which all share the same transactional overhead), the performance characteristics follow fairly naturally from the object mappings. Postgres JSON is slower than Sinew because, despite a fairly similar execution path, the CPU overhead of serializing and deserializing text-formatted JSON is large when compared to Sinew's customized key-value format. The EAV system lags even further behind because any query that touches multiple keys from a single record requires a self-join on the ID of the containing object.

6.7 Discussion

Although MongoDB offers high performance for single-object reads by object ID, it falls short in a number of other areas that severely limit its usefulness as an analytics solution. The lack of native join support can lead to a massive headache as user-generated joins take nearly an order of magnitude longer than an RDBMS join, and require large amounts of scratch space for intermediate results. Thus, despite its utility in systems needing high throughput write operations and single-key object lookups, it is not an ideal platform for analysis of that data.

Of the RDBMS-based alternatives to Sinew, each have significant drawbacks. The EAV system, despite its conceptual elegance, requires large amounts of extra logic in order to provide a transparent SQL experience to the user, and also requires more storage space and self joins during query execution (which reduces performance).

Postgres JSON requires no additional user logic, but in exchange, it has a number of deficiencies that prevent its usage as an analytics system for multi-structured data. Distinct keys that correspond to values of multiple types can lead to runtime exceptions. Array predicates are inexpressible since JSON array syntax and Postgres array syntax are mutually incompatible, and to our knowledge, Postgres does not provide a built-in mechanism for converting between the two (for our experiments, we used the approximate, but technically incorrect LIKE predicate over the text representation of the array). Although these deficiencies may be remedied with Postgres's recent announcement of jsonb (a novel binary format), a more systemic deficiency is the opaqueness of the JSON type to the optimizer, which renders the system incapable of producing efficient query plans without significant modifications to the Postgres optimizer.

7. CONCLUSION

In this paper, we have described the architecture and sample implementation of Sinew, a SQL system for storage and analytics of multi-structured data. In order to accommodate evolving data models, Sinew maintains a separate logical and physical schema, with its dynamic physical schema maintained by ongoing schema analysis and an invisible column materialization process. As a system built around an RDBMS, Sinew can take advantage of native database operations and decades of work optimizing complex queries such as joins, in addition to interacting transparently with structured data already stored in the RDBMS. We have built a prototype version of Sinew that outperforms a range of existing systems on both read and update tasks and demonstrates that Sinew offers a promising set of architectural principles.

8. ACKNOWLEDGMENTS

We would like to thank Torsten Grust, Patrick Toole, and the three anonymous reviewers for their thorough and insightful feedback. We would also like to thank Craig Chasseur at the University of Wisconsin-Madison for sharing his NoBench code. This work was sponsored by the NSF under grant IIS-0845643 and by a Sloan Research Fellowship.

9. REFERENCES

[1] D. J. Abadi. Column Stores for Wide and Sparse Data. In *Proc. of CIDR*, 2007.

[2] A. Abouzied, D. J. Abadi, and A. Silberschatz. Invisible Loading: Access-driven Data Transfer from Raw Files into Database Systems. In *Proc. of EDBT*, pages 1–10, 2013.

[3] R. Agrawal, A. Somani, and Y. Xu. Storage and Querying of E-Commerce Data. In *Proc. of VLDB*, 2001.

[4] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. In *Proc. of SIGMOD*, pages 241–252, 2012.

[5] S. Amer-Yahia, F. Du, and J. Freire. A comprehensive solution to the XML-to-relational mapping problem. In *Proc. of WIDM*, pages 31–38, 2004.

[6] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C.-C. Kanne, F. Özcan, and E. J. Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. *PVLDB*, 4(12):1272–1283, 2011.

[7] T. Böhme and E. Rahm. Supporting Efficient Streaming and Insertion of XML Data in RDBMS. In *PROC. 3RD INT. WORKSHOP DATA INTEGRATION OVER THE WEB (DIWEB), 2004*, pages 70–81, 2004.

[8] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. Pathfinder: XQuery—the Relational Way. In *Proc. of VLDB*, pages 1322–1325, 2005.

[9] C. Chasseur, Y. Li, and J. M. Patel. Enabling JSON Document Stores in Relational Systems. In *Proc. of WebDB*, pages 1–6, 2013.

[10] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing: A SQL Implementation On The MapReduce Framework. In *Proc. of VLDB*, pages 1318–1327, 2011.

[11] E. Chu, J. Beckmann, and J. Naughton. The case for a wide-table approach to manage sparse relational data sets. In *Proc. of SIGMOD*, pages 821–832, 2007.

[12] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proc. of SIGMOD*, pages 431–442, 1999.

[13] R. Fagin, A. O. Mendelzon, and J. D. Ullman. A simplied universal relation assumption and its properties. *ACM Trans. Database Syst.*, 7(3):343–360, Sept. 1982.

[14] D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDBMS. *Bulletin of the Technical Committee on Data Engineering*, 22(3):27–34, 1999.

[15] D. Maier, J. D. Ullman, and M. Y. Vardi. On the Foundations of the Universal Relation Model. *ACM Trans. Database Syst.*, 9(2):283–308, June 1984.

[16] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive Analysis of Web-scale Datasets. *Commun. ACM*, 54(6):114–123, June 2011.

[17] J. Melton, J.-E. Michels, V. Josifovski, K. Kulkarni, P. Schwarz, and K. Zeidenstein. SQL and Management of External Data. *SIGMOD Rec.*, 30(1):70–77, Mar. 2001.

[18] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proc. of VLDB*, pages 302–314, 1999.

Task	Sinew	Protocol Buffers	Avro	Original
Serialization (s)	39.83	83.68	394.24	
Deserialization	32.56	45.01	1101.26	
Extraction (1 key)	0.90	17.11	108.89	
Extraction (10 keys)	8.40	21.03	112.91	
Size (GB)	0.57	0.47	1.93	0.90

Table 4: Comparison of Serialization Formats

APPENDIX

A. SERIALIZATION COMPARISON

In Section 4.1, we highlighted a number of shortcomings of existing serialization formats, and in this section, we verify our claims by comparing Sinew's serialization format against two increasingly popular serialization formats, Avro and Protocol Buffers. We compare their performance for serialization, deserialization (reassembling the input string), single key extraction, and multiple key extraction. For completeness, we also compare the size of the serialized data without any compression algorithms applied to the serialization output. We performed our experiments on a dataset of 1.6 million NoBench objects (using the same data and configuration as the benchmarks in Section 6).

Our results are shown in Table 4. In brief, Sinew's data format outperforms both Avro and Protocol Buffers on all tasks except size, where Protocol Buffers achieve a slightly smaller data representation due to more aggressive bit-packing.

Avro's poor performance is explained by the fact that Avro, unlike Protocol Buffers, has no primitive notion of 'optional' attributes. Instead, Avro relies on unions to represent optional attributes (e.g. [NULL, int] would represent an optional integer value). This requires that Avro store NULLs explicitly (since it expects a value for every key), which bloats its serialization size and destroys performance.

Protocol Buffers, on the other hand, present a more viable alternative to Sinew's format, but they also fall short on the tasks of deserialization and key extraction. In the case of deserialization, we see that Sinew outperforms Protocol Buffers by approximately 50%. We can attribute this to the fact that whereas Sinew performs all of its operations directly on the serialized binary data, Protocol Buffers operate on an intermediate logical representation of their data.

Although one could easily modify Protocol Buffers to perform deserialization directly from the physical, binary object, their performance on key extractions is fundamentally bounded by the fact that random attribute reads are not possible due to the sequential nature of their format. Although Protocol Buffers store attributes in a particular order (so they can 'short-circuit' a lookup of a non-existent key once the deserializer has passed the key's expected location), they still must traverse keys until this point serially. On the other hand, Sinew's format is hyper-optimized for random key reads, since it includes a per-record header with attribute IDs and offset information. This is the same reason why we see the relative performance gap fall as we extract a greater number of keys at any given time (1 vs 10). Whereas Sinew's performance is linear in the number of keys extracted (until a threshold is reached where it is more performant to use an intermediate logical representation similar to Protocol Buffers), Protocol Buffers have already paid the up-front cost of reading the binary, and further key extractions are a simple matter of a single pointer lookup.

Query	Virtual	Physical
SELECT "user.id" FROM tweets;	14.40	13.57
SELECT * FROM tweets WHERE "user.lang" = 'en';	63.59	63.37
SELECT * FROM tweets ORDER BY "user.friends_count" DESC;	74.59	73.55

Table 5: Virtual vs Physical Column Performance

B. VIRTUAL COLUMN OVERHEAD

One concern of serializing all virtual columns into a single column reservoir is that extracting relevant data from the reservoir may be more expensive than extracting data from physical columns. We therefore compared the performance of queries over data stored in virtual columns using our serialization format against the same queries over the same data, where the relevant attributes for the queries are stored in physical DBMS columns instead of as key-value pairs inside the column reservoir (the non-relevant attributes are still stored in the reservoir). As in Section 3.1.1, our benchmark dataset comprised 10 million tweets and our benchmark system was the same as in our experiments in Section 6.

Our results, summarized in Table 5, show that our object serialization introduces very little execution overhead. For each query, we saw less than a 5% reduction in performance when the query involved a reference to a virtual column instead of an equivalent physical column. For the query involving a single projection, the costs for processing the query are identical (whether "user.id" is stored in a physical column or virtual column) except for actually retrieving the value of the "user.id" attribute for each row that is scanned. If "user.id" is stored as a physical column, it requires one memory dereference to locate the attribute data within the tuple. If it is instead stored as a virtual column (i.e. serialized in the column reservoir), it requires one memory dereference to locate the attribute corresponding to the column reservoir, followed by a (cache efficient) search within the header to find the attribute and its offset, and one memory reference to locate the attribute value within the object reservoir. Thus, accessing a virtual column involves just one additional memory dereference, a binary search within the header, and the function call overhead of calling the extract UDF described in Section 3.2.2. These additional costs are small relative to the shared fixed costs of query processing (e.g. row iteration and query result collection).

As the fixed costs of query processing increase, the relative overhead of accessing data from virtual columns instead of physical columns decreases, since virtual column extraction is amortized over an increased execution time. Hence, the performance difference for the selection and ORDER BY queries was smaller (<2%) than that of column projection.

Although this experiment indicates that the overhead of virtual column extraction is small (especially relative to the rest of query processing), this overhead is still noticeable, and increases with an increased number of attribute extractions per query. Furthermore, recall from Section 3.1.1 that a large issue with storing data in virtual columns is that attribute statistics are hidden from the underlying database system (given our requirement that we not modify the underlying DBMS code) and this can result in poor optimization of queries and reduced performance. Thus, while the small overheard of our custom serialization format is promising, a hybrid architecture that materializes certain attributes (see Section 3.1.4) into physical columns is still necessary.